Domain-Specific Hardware/Architecture for Emerging Applications

by

Zhehong Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical and Computer Engineering)
in the University of Michigan
2022

Doctoral Committee:

Professor David Blaauw, Chair
Assistant Professor Ronald Dreslinski
Assistant Professor Hun-Seok Kim
Professor Dennis Sylvester

Zhehong Wang

zhehongw@umich.edu

ORCID iD: 0000-0001-5112-639X

Dedication

This dissertation is dedicated to my lifelong best friends: Hao Wu, Haoyu Song, Haoze Shen, Jushang Shen and Zhiyuan Wang.

Acknowledgements

My sincere appreciation to:

My family

My advisor

My fellow colleagues

And everyone who supported me through this long voyage.

Table of Contents

List of Tables

List of Figures

Abstract

Technology scaling has driven the development of the computing industry during the past 50 years. However, as soon as we reach the power and memory wall, the impact of Moore's Law started to wear away. The lack of "free" performance gain by simply scaling the technology implies that architecture and circuit designers will have to make the most of the potential of available technology nodes to face the challenge, requiring significant effort to develop innovative architectures and circuits. One solution is to trade off the programmability and flexibility of current microprocessors for a more optimized data and control flow of a specific application, and thus, the concept of domain-specific hardware came about. Though not a brand-new concept, as epitomized by graphics processors, highly computation-hungry Machine Learning (ML) applications, which have thrived in recent years, have benefited greatly from it with respect to both performance and energy.

This thesis presents three different domain-specific solutions for various emerging applications, including DNA sequencing, ML, and post quantum cryptography/homomorphic encryption, each of which employs different optimization schemes.

The first application-specific solution demonstrates a seed-extension accelerator for next-generation sequencing in 55nm process technology with a recently proposed automata architecture. With an array of 25×25 custom-designed processing elements, it performs 2.46M reads/s, rendering a 1581x improvement in power efficiency compared to a system with dual-socket Xeon E5-2597 v3 server processors. The second prototype presents an RRAM and model compression-based DNN accelerator in 22nm process that features algorithm, architecture, and

circuit optimizations. It achieves 16 million 8bit (decompressed) on-chip weights with the 24Mb RRAM, eliminating the energy-consuming off-chip memory access. The last work proposes and implements an architecture for accelerating third-generation FHE with AWS cloud FPGAs. A novel unbalanced PSI protocol based on third-generation FHE, optimized for the proposed hardware architecture, is introduced. The measurement results show that the proposed accelerator achieves >21× performance improvement compared to a software implementation for various crucial subroutines of third-generation FHE and the proposed PSI.

Chapter 1. Introduction

For the past 50 years, the development of microprocessors has continued to push the limits of the computer industry, powering a marvelous, digitized world. As shown in Figure 1.1 [1], the frequency and single-threaded performance of microprocessors have grown by roughly 1000x during this time, following the improvement of the semiconductor technology guided by Moore's Law, represented by the exponential increase in the number of transistors. However, this seemingly rocketing pace started to drop off at the beginning of the 21st century. The truth is that technology scaling accounted for 40% of the performance advance of microprocessors in the past decades, while microarchitecture innovation only contributed to 17% [2]. By simply scaling the transistors, computers could get almost "free" performance gain; thus, relatively little advances in

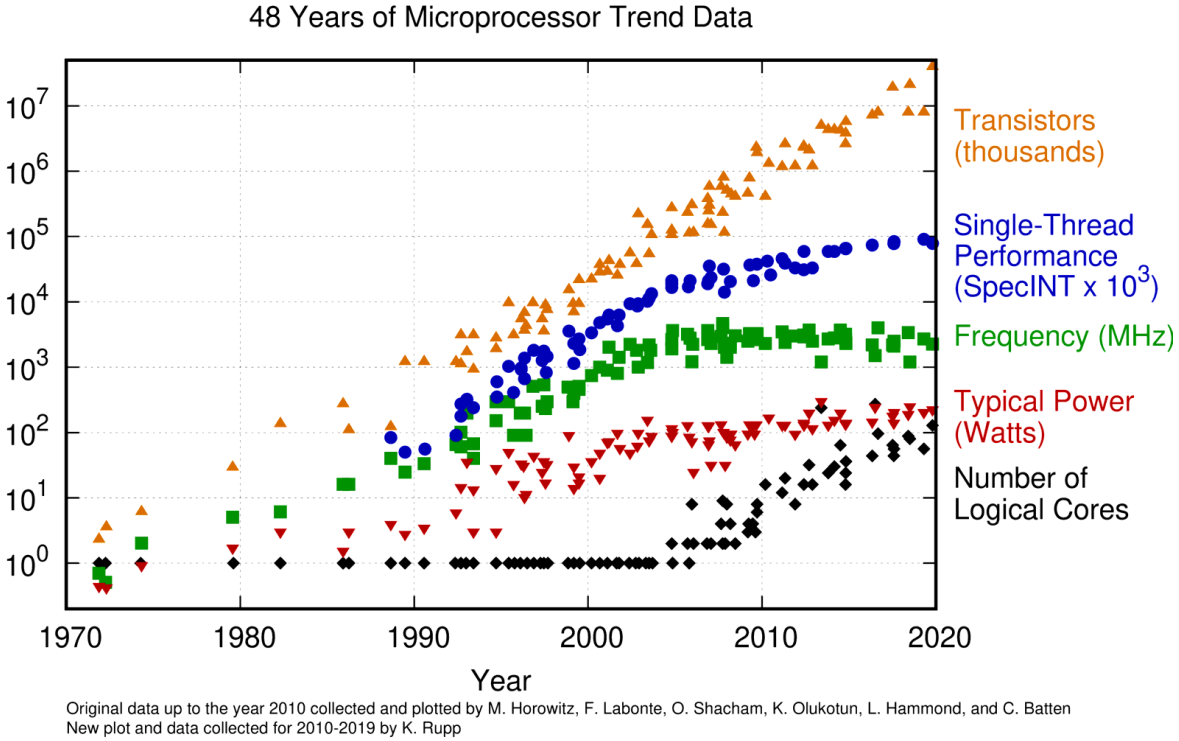

Figure 1.1 48 Years of Microprocessor Trend [1].

microarchitecture and circuits design were needed. It has even been called by some computer architects "the dark era of computer architecture." But with the decline of Moore's Law, innovations in microarchitectures and circuits are playing an increasingly pivotal role in the development of the computing industry.

One of the most important aspects of microprocessor design that demands new architectures and circuits is the power budget, including thermal limitation. Internet of Things (IoT) devices, or even mobile devices, constantly run into problems with power budgets because of shrinking form factors. Furthermore, as suggested by Dennard's Scaling, power density remains unchanged as technology scales, so cooling systems also limit the system power. Parallelism can help overcome these issues. By replacing one unit of a function with N units of the same function, all N units can operate at 1/N frequency while maintaining the overall performance. Furthermore, the voltage can be scaled down accordingly to get quadratic energy savings, or linear power saving, at the expense of a linear area increase [3]. Thus, as soon as microprocessors reach the cooling limit, multicore architectures emerged (Figure 1.1).

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3.1pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1.1pJ
32 bit	3.7pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

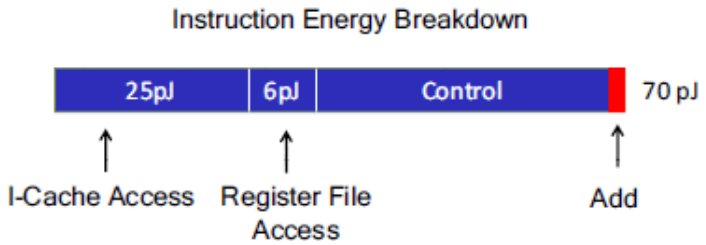

Figure 1.2 Rough Energy Costs for Various Operation in 45nm 0.9V [2].

However, this is just the tip of the iceberg. Modern CPUs typically employ complex instruction scheduling schemes to boost the instruction per clock (IPC), such as out-of-order (OoO) execution and branch prediction, which introduce high control overhead. In fact, even in an in-order processor, the actual operation only accounts for a marginal portion of the total energy cost of one instruction, a few pJ vs. 70 pJ, in 45nm process, as listed in Figure 1.2 [4]. The principal reason lies in the fact that microprocessors are highly programmable and flexible so that they can perform any algorithm. However, it is not always necessary to have such flexibility in an actual application where high-level operation can be abstracted. For example, graphics processors are generally more efficient for graphical applications, not only because of the enormously large number of parallel processing cores but also because the graphics pipeline is so mature that the hardware designers can use the prior knowledge to simplify the control logic and optimize the data flow, thus reducing any wasted energy. Therefore, dedicated hardware functional units for frequently invoked complex functions, which are called domain-specific hardware [5], help with the overall efficiency, which is attested to by the various domain-specific accelerators for Machine Learning (ML) [6]-[10] that are proposed to cope with the computation slack and lower power efficiency of deploying the algorithm on microprocessors.

Domain-specific hardware calls not only for optimized control flow but also for accommodated memory architecture since memory access energy also overshadows the actual computation energy cost (Figure 1.2). Cache hierarchy has been widely utilized in microprocessors to buffer the speed gap between the processor and the main memory by prefetching the required data to reduce the overall latency, following the principle of locality. However, in many applications, it is the memory bandwidth that renders the bottleneck. For instance, in modern graphics cards, the memory system is architected for high bandwidth to meet the massive

parallelism. The latest AMD Radeon 6000 series graphics card can push the bandwidth of the GDDR6 system to up to 512GB/s [11], while, in a typical desktop computer, 4 channels of DDR4-3200 deliver merely ~100 GB/s. Furthermore, graphics processors also hold a large pool of on-chip buffer for immediate results, thus reducing the write-back transaction to the main memory and improving the overall efficiency. In recent ML accelerators, optimized data flows were also introduced to reduce DRAM access by maximizing data reusage. Attempts in optimizing memory architecture by innovative memory systems were also made. The researchers in [12] replaced the DRAM in an ML system with a 3D-stacked SRAM using wireless communication and successfully reduced the overall memory access latency and power.

Advances in memory architecture are also promoted by new memory technology. The primary memory components in current computing systems are SRAM, DRAM, and, in some cases, flash. These charge-based memories suffer from the reduced capability to hold sufficient charge to maintain the information as technology scales, resulting in reliability issues. Next-generation storage solutions have emerged in recent years, such as MRAM (magnetoresistive) [13][14], RRAM (resistive) [15] and PCRAM (phase change) [16]. These advanced memories rely on a physical phenomenon to store information and are thus less impacted by technology scaling. Furthermore, the non-volatility and high speed also offer the potential for innovated memory circuits and architectures. Several accelerators, following the in-memory-computing concept, employed these non-volatile memories for better efficiency in some low precision scenarios [17][18].

Together, the above principles combined with some other recently proposed approaches, are employed in domain-specific solutions for three emerging applications, DNA sequencing, ML, and post quantum/homomorphic encryption, with each area demanding a unique optimization.

1.1 Next-Generation Sequencing for Whole-Genome Sequencing

The completion of the Human Genome Project (HGP) [19] has triggered immense interest in the application of Whole-Genome Sequencing (WGS), driving the development of Next-Generation Sequencing (NGS) technology to reduce the cost. Since the advance of NGS, the production cost of whole-human genome sequencing has plummeted by 10,000x from $10 million to $1000 in the last decade [20], as shown in Figure 1.3. This has led to wide use of DNA testing in both research and clinical diagnosis, creating more personalized patient treatments [21]. For example, genetic tests are now commonly used to predict the effectiveness of specific breast cancer treatments for patients [22]. Furthermore, identifying somatic mutations in the human genome sheds light on the evolution of human cancers, information that can be leveraged to prevent these cancers in individuals [23]. Likewise, large-volume genome testing across diverse samples can provide a better understanding of the cause of Alzheimer disease [24]. As technology speeds up WGS, its use will likely become a standard clinical practice, as prevalent as a blood test, in the coming decade.

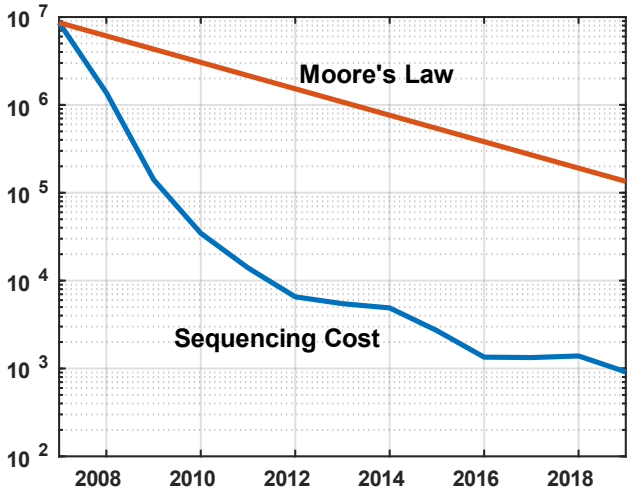

Figure 1.3 Sequencing Cost Plummeting V.S. Moore's Law [20].

The fast reduction of sequencing cost that benefits the ubiquitous application of WGS stems from the rapid improvement of the NGS technology. Compared to the HGP, which required 15 years to sequence the first human genome, NGS systems from Illumina can sequence over 45 human genomes in a single day [25], rendering a 200,000x speedup. As shown in Figure 1.4, a typical current-generation DNA sequencing pipeline is composed of roughly two main steps. First, the input genome is split into short DNA fragments, which are sequenced in a DNA sequencer backed by NGS technology to produce billions of reads (short sections of the input genome sequence). Then, the reads are passed into a reference-guided secondary analysis, where the sequences are assembled into the original genome order by aligning to a previously sequenced genome, for example, the one obtained from the HGP. The secondary analysis is far from a simple string comparison in the sense that the input genome does not necessarily perfectly match the

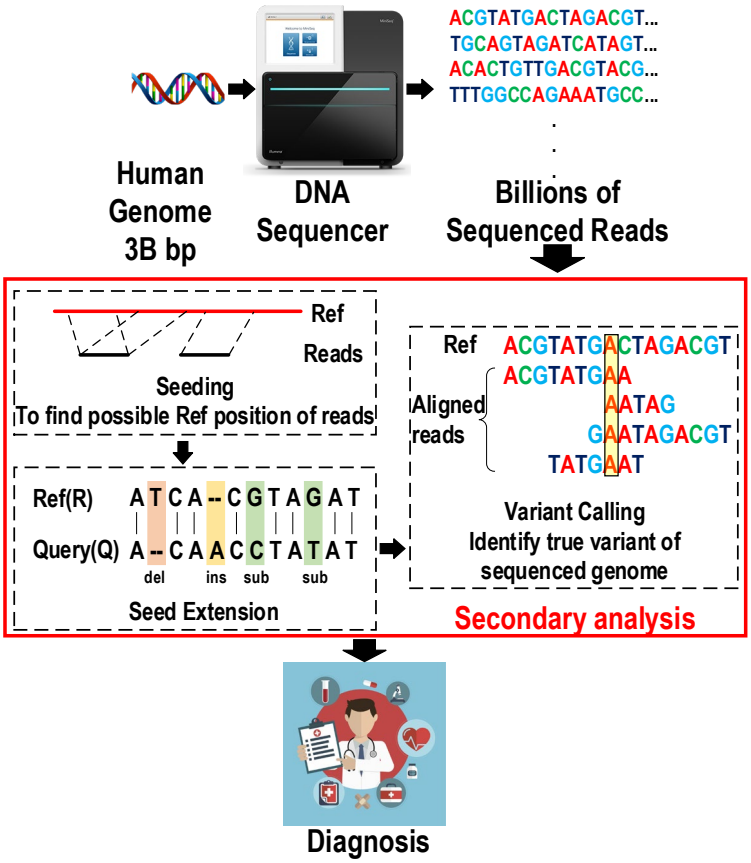

Figure 1.4 Reference Guided Sequence Analysis Pipeline with Seed Extension Highlighted.

referenced genome due to DNA mutations, requiring approximate string alignment. Furthermore, the produced reads are not necessarily an exact copy of the input genome due to the potential errors introduced during the DNA sequencing process, further complicating the task. Altogether, for each sequenced human genome, on average, 396 GB of data must be processed through the complex secondary analysis [26], which not only poses a significant computational challenge to current general-purpose computing systems but also represents a critical bottleneck in the sequencing pipeline since the improvement of the DNA sequencer enabled by NGS technology is far outpacing Moore's law. At such a rate, this computation bottleneck is projected to dominate the total cost and processing time of sequencing, becoming a key limiting factor in the growth of this important medical technology.

1.2 Deep Neural Network Accelerator

Deep neural network (DNN) algorithms, first introduced in the early 1960s [27], are the cornerstone of modern artificial intelligence (AI) because they achieve unprecedented accuracy on various computer vision and machine translation tasks. The next wave in the AI revolution is the deployment of these deep neural networks on mobile platforms to perform challenging tasks under real-world constraints. However, existing hardware and infrastructure cannot provide satisfying performance and energy efficiency for emerging deep-learning-based applications because of their excessive computation and large memory footprints in state-of-the-art DNN models. For object recognition with the ImageNet dataset [28], these DNN models [29][30][31] typically comprise more than 10 million parameters and require more than 10 GOP per inference, which translates to more than 50 MB on-chip storage and 300 GOPS throughput for real-time 30 fps operation. They consume >100 W of power with general-purpose graphics processing units (GPGPUs), which cannot be integrated on mobile platforms due to their excessive power consumption and form

factor. Therefore, there is a growing demand for high-performance, energy-efficient, and re-configurable DNN processors for mobile and embedded AI applications [32]-[42].

To address the aforementioned challenges, various approaches targeting at both optimizing machine learning (ML) algorithms and efficient hardware designs have been proposed to reduce the complexity of the DNN inference and to improve the energy efficiency, thereby maintaining accuracy for applications.

References [32][33] propose to re-architect the neural network models and leverage efficient building blocks to reduce both the model size and the number of multiply-and-accumulate (MAC) operations. However, despite the dramatic complexity reduction, these approaches create new DNN layers with novel memory-access and challenging computation requirements that are not well-optimized with existing hardware [42]. Alternative approaches such as [34][35] reduce the model complexity with pruning, quantization, entropy coding, and/or low-rank approximation of weights. However, their real-time energy-savings and performance gains are limited because of the inefficiency of running unstructured sparse models on the hardware. For example, [83] reports >1 W power consumption for real-time inference using compressed DNNs.

In parallel with improving DNN models, many digital ASICs [36]-[42] were proposed recently to accelerate deep learning on mobile platforms. Various optimization techniques are explored in these designs, including dataflow optimizations [12][36][37], precision reduction [37][38][39], sparsity awareness [40][41], bit-serial operation [37], etc. Combining these techniques onto silicon implementations, state-of-the-art DNN processors achieve more than 100 GOPS performance and ~2 TOPS/W efficiency during inference.

However, as shown in Figure 1.5, most of these digital ASICs adopt a DRAM-NPU (neural processing unit)-style processing architecture for loading and computing DNN models [84]. The

weights and input activations are transferred on chip for processing while computed output activations (OA) are transferred back to the large off-chip DRAM for temporary storage. While the processing on the NPU is extensively optimized through various techniques [37]-[41] (Figure 1.5), transferring data on/off the NPU to the DRAM becomes a major bottleneck in the overall system because of the frequent and extremely high-energy data access to external DRAMs. In fact, transferring a byte from DRAM consumes >3000× more power than performing an 8-bit MAC calculation [36]. To relieve this problem, [12][36][37] proposes to integrate dedicated weight and activation buffers and optimize the dataflow to reduce the data transfer to external DRAMs. Additionally, [39] proposes to leverage data compression technique to reduce the bandwidth to the DRAM. These methods significantly reduce the data access overhead to the DRAM but do not completely solve the problem.

To reduce the off-chip data/parameter accesses, a few prior designs [12][38] attempt to store all parameters on chip. However, [38] suffers from very limited on-chip memory capacity (only ~100 kB of weights are stored), which is insufficient to support large applications with >10 M weights. The design [12] achieves high capacity (7.68 MB on-chip weights and 96 MB SRAM stack) at the expense of high system power (3.3 W) due to the large SRAM stack and inductive inter-die communication.

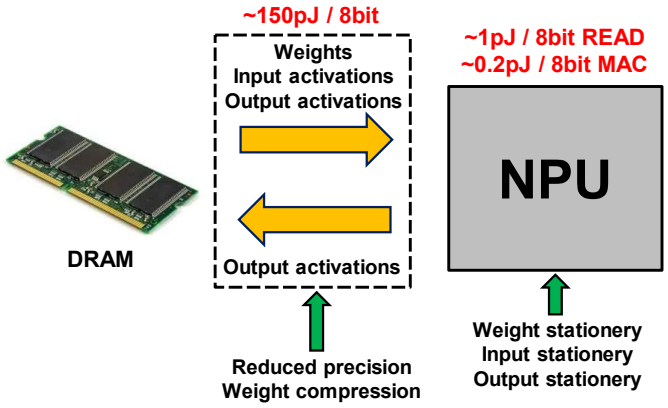

Figure 1.5 Conventional System-Level Dataflow of NPU.

1.3 Lattice-Based Post-Quantum Cryptography and Homomorphic Encryption

For the past decades, RSA encryption/public key encryption [43] has been successfully securing internet transactions and has become the foundation of the Information Era. The hardness of RSA encryption lies in the hardness of decomposing large numbers into prime factors. To date, there is still no known efficient algorithm for solving it on classical computers. However, following the advance of the conception of the Quantum Computer, an efficient algorithm has been proposed [44], which seems to anticipate the end of all classical number-theory-based encryptions in the future as quantum computers develop at a steady pace[45]. Thus, various researches of post-quantum cryptography that are resistant to quantum attacks have been proposed, including approaches that are lattice based, code based, multivariate based, etc. Among them, lattice-based cryptography, with its well-studied hardness, is the most widely adopted. Of the Round 3 finalists of the NIST Post-Quantum Cryptography Competition, 5 out 7 candidates are lattice based [45].

A lattice is an abstract mathematical structure that can be represented by a vector space that is formed by integer linear combinations of a set of bases. A 2-D example is shown in Figure 1.6 with two bases b1 and b2. A lattice problem, expressed in linear algebra form, states that given a matrix $\boldsymbol{A}$ and a vector $\boldsymbol{m}$, and let $\boldsymbol{b} = \boldsymbol{A} \times \boldsymbol{m} + \boldsymbol{e}$, where $\boldsymbol{e}$ is a vector of random noise, it is hard to recover $\boldsymbol{m}$ solely from $\boldsymbol{A}$ and $\boldsymbol{b}$. Obviously, without the added noise $\boldsymbol{e}$, $\boldsymbol{m}$ can be easily

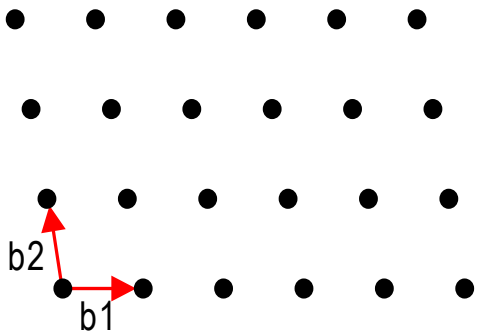

Figure 1.6 A 2-D Lattice.

recovered by multiplying $\boldsymbol{b}$ with the inverse of $\boldsymbol{A}$. But with a decent amount of noise, it has proved to be hard even for quantum computers. This is known as the Learning with Errors (LWE) problem, which can be reduced to a lattice problem within polynomial time [47].

This clear algebraic construction of Lattice Cryptography brings a rich pool of mathematical structure that facilitates fully homomorphic encryption (FHE). Homomorphic encryption (HE) is basically an encryption scheme that allows operations on encrypted data. More precisely, for a give function $f(x)$, a homomorphic encryption scheme satisfies $f(Enc(m)) = Enc(f(m))$. If it is homomorphic to any function, it is called full homomorphic.

Exploration of HE started following the advent of RSA encryption due to its multiplicative homomorphism. However, no scheme with fully homomorphic capability was devised until 2009, when Gentry proposed a general FHE framework [48] that all the subsequent works follow. It has been proven that from a Boolean circuit model perspective of computation, if an encryption scheme is homomorphic to its own decryption function followed by a universal logic gate, then it is homomorphic to any function. The operation that fulfills this property by transforming a partially HE or Leveled HE (LHE) into an FHE is called bootstrapping or recryption. Based on this idea, Gentry also presented a concrete construction based on an ideal lattice and sparse subset sum problem, but the software implementation reports around 30 minutes per bootstrapping [49]. Since then, various schemes have been proposed looking for more efficient implementation. Among them, the most well-known are BGV [50], BFV [51][52] and CKKS [53]. These second-generation schemes differ from Gentry's approach in the underlying hard problem: they use a Ring Learning with Errors (RLWE) problem for its better-studied hardness analysis and efficiency fulfilled by SIMD-styled operation [54]. BGV and BFV schemes operate on polynomial functions of an integer number and have shared much in common during their development since the first

publication, whereas CKKS works with complex numbers as plaintext space. Several open-source libraries that implement these schemes are accessible, including PALISADE [55], HElib [56] and SEAL [57]. These implementations can potentially reduce the recryption time to minutes depending on security parameters. Then after the GSW [58] scheme was proposed in 2013, two schemes, FHEW [59] and TFHE [60], were published as third-generation approaches. The third-generation schemes, compared to second-generation ones, focus on finding an efficient implementation of single-bit logic and a bootstrapping operation. Although, performance wise the third generation may not be superior to earlier schemes, since the amortized cost of the second generation equipped with SIMD-like [54] construction is estimated to be in the same order of magnitude as that of the third generation, it is well accepted for its simplicity and flexibility in terms of both concept and implementation. The reported recryption time of the third generation is generally around 0.1s to 1s.

1.4 Thesis Contribution and Organization

This work presents domain-specific accelerators for three emerging applications, namely, DNA sequencing, ML, and post quantum/homomorphic encryption. The remaining chapters are organized as described below.

In Chapter 2, an accelerator for seed extension, a critical and computationally intensive step in genome sequencing, is proposed. The accelerator, implementing a string-independent automata, consists of a triangular array of 25×25 custom-designed processing elements. It performs 2.46M reads/s, achieving a 1581x improvement in power efficiency and 165.5x smaller silicon footprint compared to a system with dual-socket Xeon E5-2697 v3 server processors.

Chapter 3 presents an energy-efficient deep neural network (DNN) accelerator with non-volatile embedded resistive random-access memory (RRAM) for mobile ML applications. This

DNN accelerator implements weight pruning, non-linear quantization, and Huffman encoding to store all weights on RRAM, enabling single-chip processing for large neural network models without external memory. A 4-core parallel and programmable architecture adapts to various neural network configurations with high utilization. We introduce a customized RRAM macro with a dynamic clamping offset-canceling sense amplifier (DCOCSA) that achieves sub-μA input offset. The on-chip decompression and memory error-resilient scheme enables 16 million (M) 8-bit (decompressed) weights on a single chip using 24 Mb RRAM. The proposed RRAM-DNN is the first digital DNN accelerator featuring 24 Mb RRAM as all-on-chip weight storage to eliminate energy-consuming off-chip memory accesses. The fabricated design performs the complete inference process of the ResNet-18 model while consuming 127.9 mW power in TSMC-22nm ULL CMOS. The RRAM-DNN accelerator achieves peak performance of 123 GOPs with 8-bit precision, exhibiting measured energy efficiency of 0.96 TOPs/W.

Chapter 4 presents the first accelerator architecture for third-generation FHE, targeting at the $RLWE \otimes RGSW$ operation, which is a fundamental function of both second-generation and third-generation FHE. By exploiting the asymmetric nature of the encryption, the architecture incorporates an asymmetric Inverse Number Theory Transform (INTT) module and Number Theory Transform (NTT) module, which are capable of maintaining high throughput with less resource usage while addressing different parameter sets. An extensive analysis of the architecture is included. A novel unbalanced PSI protocol that is based on third-generation FHE and is optimized for the proposed hardware architecture is proposed. The proposed PSI protocol makes the computation cost independent of the Sender's set size. We introduce several additional algorithm-architecture co-optimizations to reduce the computation and communication costs, rendering a practical application of the proposed PSI protocol. A prototype of the proposed

architecture is implemented with AWS cloud FPGA service. We develop all necessary high-level functions in C++ and benchmark the implemented architecture with different parameter sets. We make the SystemVerilog HDL code of the proposed accelerator and supporting software code publicly available at [136]. At last, we quantify and analyze the performance of the proposed hardware accelerator and PSI protocol. The measurements show over 21× performance improvement compared to a software implementation for various subroutines of the third-generation FHE and the proposed PSI.

Chapter 5 summarizes the main contributions of this thesis and discusses about some future directions of the projects.

Chapter 2. A 2.46M Reads/s Seed-Extension Accelerator for Next-Generation-Sequencing Using a String-Independent PE Array

2.1 Introduction

Before dealing with the bottlenecks mentioned in Chapter 1, it is worthwhile taking a close look at the secondary analysis (Figure 1.4). As mentioned before, the large number of reads, ~1.5B, produced by the DNA sequencer, are passed through the secondary analysis, which is further divided into three processing steps [61]. 1) In the seeding step, a set of possible locations where the read matches the reference is found by exactly matching small fragments (seeds) between the read and the reference. 2) The seed extension step evaluates these possible match locations by exploring approximate alignments between the read and the reference, including possible edits, to determine a final match location. 3) In the final step, the variant calling step, all the reads that are aligned to a particular base in the reference are evaluated to determine if a mutation occurred at that location.

Various software packages have been devised to handle these steps since early 21st century. Some of them cover the first two steps, such as BWA-MEM [62], Bowtie2 [63] and SOAP2 [64], while some focus on the third step but use the above libraries for the first two steps, like GATK [65] that incorporates BWA-MEM. There are also other generic string/sequence analysis libraries such as SeqAn [66] and SSW [67]. Among them, BWA-MEM has become a standard for the analysis pipeline and is advocated by GATK best practice guide.

BWA-MEM, as well as other libraries, adapts two algorithms, FM-Index [68], based on Burrows-Wheeler Transform (BWT) [69], and Smith-Waterman (SW) Algorithm [70], for the two steps, seeding and seed-extension respectively. GPU implementations [71][72] of these software packages and algorithms were released within the software community to cope with the computation bottleneck due to the increasing amount of data to be processed. However, acceleration for them gained little interest until recent years, following the tapering-off of Moore's Law. FPGA solutions were proposed at first for its versatility and better power efficiency compared to GPU systems, such as [73] for the seeding step and [74][75] for the seed-extension step. With the software stack getting stable, ASIC becomes appealing in terms of form factor and power efficiency. An ASIC designed for the seeding step was also presented in ISSCC 2018 [76] with 7.84 mm^2 die size and 135 mW of power, achieving ~1,000x and ~400x improvement in terms of power efficiency and area efficiency compared to GPU implementation. Yet little or no dedicated acceleration ASIC has been proposed to address other steps in DNA sequencing.

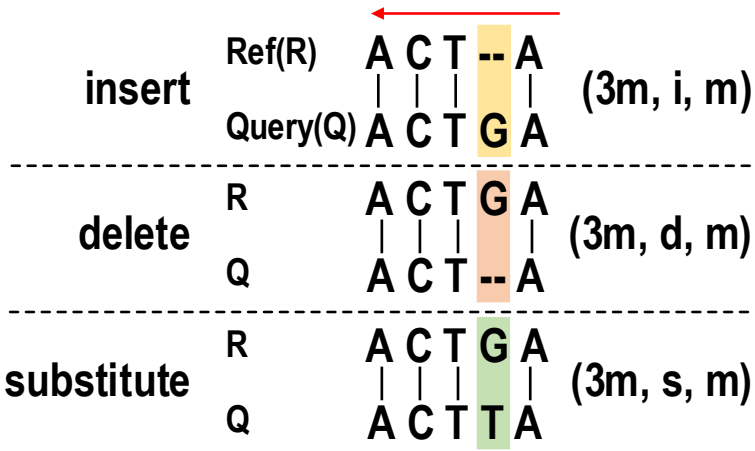

Figure 2.1 Levenshtein Edits.

In this project, we target at the seed-extension step, which requires a total of 14 billion alignments for each human genome, which takes ~5/~273 hours for 56-thread/single-thread workload on a server equipped with dual-socket Xeon E5-2697 v3 processors, using the optimized SeqAn library [77]. Seed extension aligns two DNA strings of ~100 bases: the read or query (Q)

and the portion of the reference (R) where the query is expected to align. However, as mentioned above, there can be mismatches between R and Q due to sequencing machine errors or mutations in an individual's DNA. Hence, approximate alignment is needed, allowing for the following Levenshtein edits as shown in Figure 2.1: insert (i), delete (d), and substitute (s). Figure 2.2 shows two of many possible alignments for an R and Q pair, each with an edit distance or score. The goal of seed extension is to find the alignments with the best score and to report the score and its associated strings of edits.

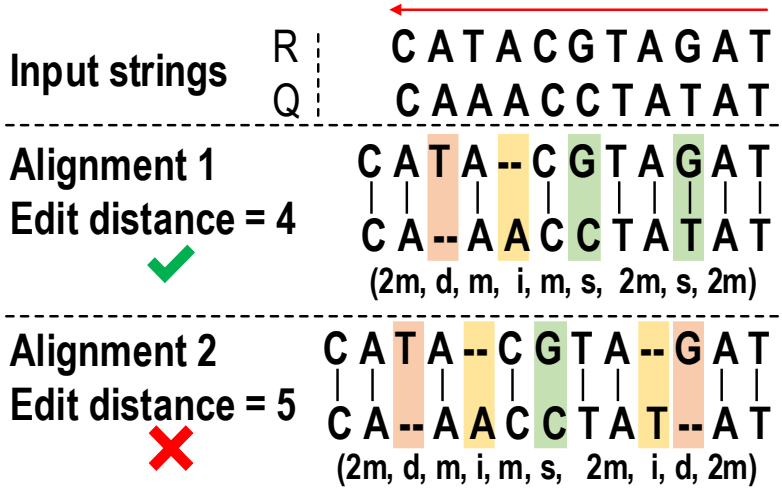

Figure 2.2 Example of Alignment.

We use a 25x25 triangular array of processing elements (PEs) that implements a string-independent automata algorithm for approximate string matching and also performs match score calculation and generation of the edit string. The proposed alignment accelerator, implemented in MIFS 55-nm DDC CMOS, operates at 670 MHz and achieves 2.46M reads per second with 8 mm^2 silicon area [78]. Marking, to our knowledge, the first seed-extension ASIC, it achieves a ~1581x power efficiency improvement and 165.5x smaller silicon footprint compared to deploying SeqAn library on a server with dual-socket Xeon E5-2697 v3 server processors [77], operating on the same Genome dataset and producing the same output.

2.2 Seed-Extension Algorithms

2.2.1 Smith-Waterman Algorithm

The canonical solution for the approximate string alignment problem is a dynamic programming (DP) algorithm called the Smith-Waterman Algorithm [70], which is adapted as a submodule of BWA-MEM and other software libraries. The basic version of the algorithm that only identifies the edit distance is illustrated in Figure 2.3.

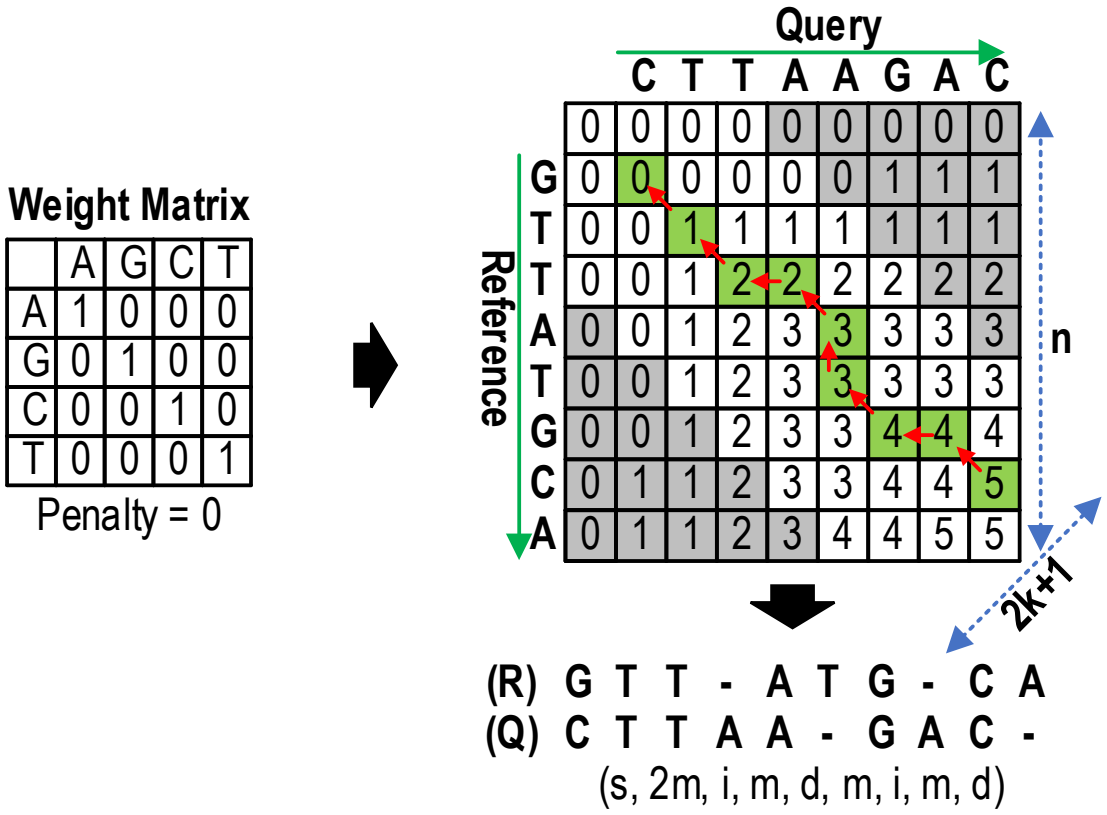

Figure 2.3 Basic Smith-Waterman Algorithm.

To align two input strings Query (Q) and Reference (R), first a DP matrix of size n^2 is initialized as all zeros, where n is the length of the input strings (assuming w.l.o.g. the input strings are of the same length). Then all the cells identified by coordinate (i, j), except for the ones in the first row and column, are filled by an alignment score calculated from the neighboring cells, according to Equation 2.1. The weight matrix denoted by $W(r_i, q_j)$ in Equation 2.1 assigns each

pair of the input characters in the input strings a score for match or mismatch. For example, here, the match score is set to 1 and mismatch score is set to 0. The penalty in the equation is reserved for the gap penalty calculation explained further in Section 2.3 and is set to 0 for simplicity in this example. Thus, Equation 2.1 is a rule of reward and penalty that accumulates the highest alignment score between the two input strings up to position (i, j).

$$E[i,j] = max\begin{cases} E[i-1,j-1] + W(r_i, q_j) \\ E[i,j-1] - penalty \\ E[i-1,j] - penalty \end{cases} \quad 2.1$$

In the meantime, each cell also keeps track of which one of the neighboring cells its score is calculated from. Finally, from the highest scores, the traces that hold the best scores are extracted backward. One of the best traces is shown in the example, which translates to an edit distance that equals 5.

The time and space complexity of the Smith-Waterman Algorithm is $O(n^2)$, where n is the maximum length of the input string pair, rendering a string-dependent space complexity and preventing efficient hardware implementation. In practice, one often focuses, dynamically or statically, on a band of fixed width along the diagonal of the DP matrix [79], leaving out the corners colored in gray in Figure 2.3. The parameter that sets the width of the band is the maximum target edit distance k. This banded DP matrix correctly generates output as long as the edit distance of the input string pair is no greater than k. This optimization potentially reduces the complexity to $O(kn)$ since k is much less than n in practice. But it is still string dependent.

2.2.2 String-Independent Automata

Another solution to sequence matching is based on Levenshtein Automata (LA). An LA is a finite state automaton that is defined on a sequence s and a number k. It can recognize the set of all sequences that are at most k edit distances away from the string s. For example, the state diagram of an LA, defined on string AGC and edit distance $k = 1$, is shown in Figure 2.4. Each state is denoted by n^e, which means that n characters consumed and e edit distance encountered so far. The state machine starts from state 0^0 and takes the characters of input sequence one by one. And it determines whether the input sequence is at most $k = 1$ edit distance away from the

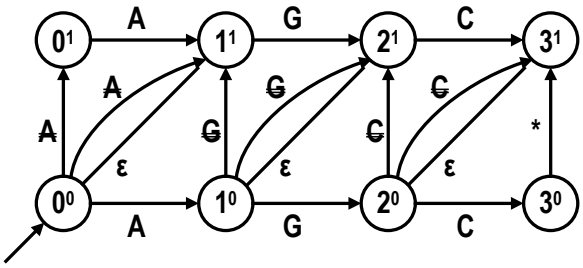

Figure 2.4 The State Diagram of an LA for Sequence s= AGC and Edit Distance k = 1.

sequence $s =$ AGC. Discussion about the finite state automaton theory is beyond the scope of this work, the reader is referred to [80] for further information. Although the complexity of LA is $O(kn)$, which is comparable to SW. It suffers from the fact that it is specific to a given sequence, as the LA in Figure 2.4 is only able to compare strings to AGC. Thus, LA is rarely utilized in practical sequencing software libraries.

While, our design adopts a recently proposed string-independent Levenshtein Automata algorithm [77]. The algorithm decouples the state machine from the specific sequence and has the advantage that varying length strings can be processed using the same matching hardware as long as the maximum edit distance remains fixed. Unlike the standard SW algorithm, where R/Q strings remain static and possible alignment paths are explored with an array of PEs, the R/Q strings of

arbitrary length are shifted through a state machine that simultaneously evaluates all possible alignments between the two strings.

The state machine for this algorithm consists of a 3D grid of tiny PEs, represented as circles in Figure 2.5, each performing a comparison of a base pair (BP), while the BPs of the input R/Q reads are supplied through the two shift registers, rightward and upward, one BP at a time. Starting from the PE at the bottom left, each PE reactivates itself when the BP matches, otherwise, it

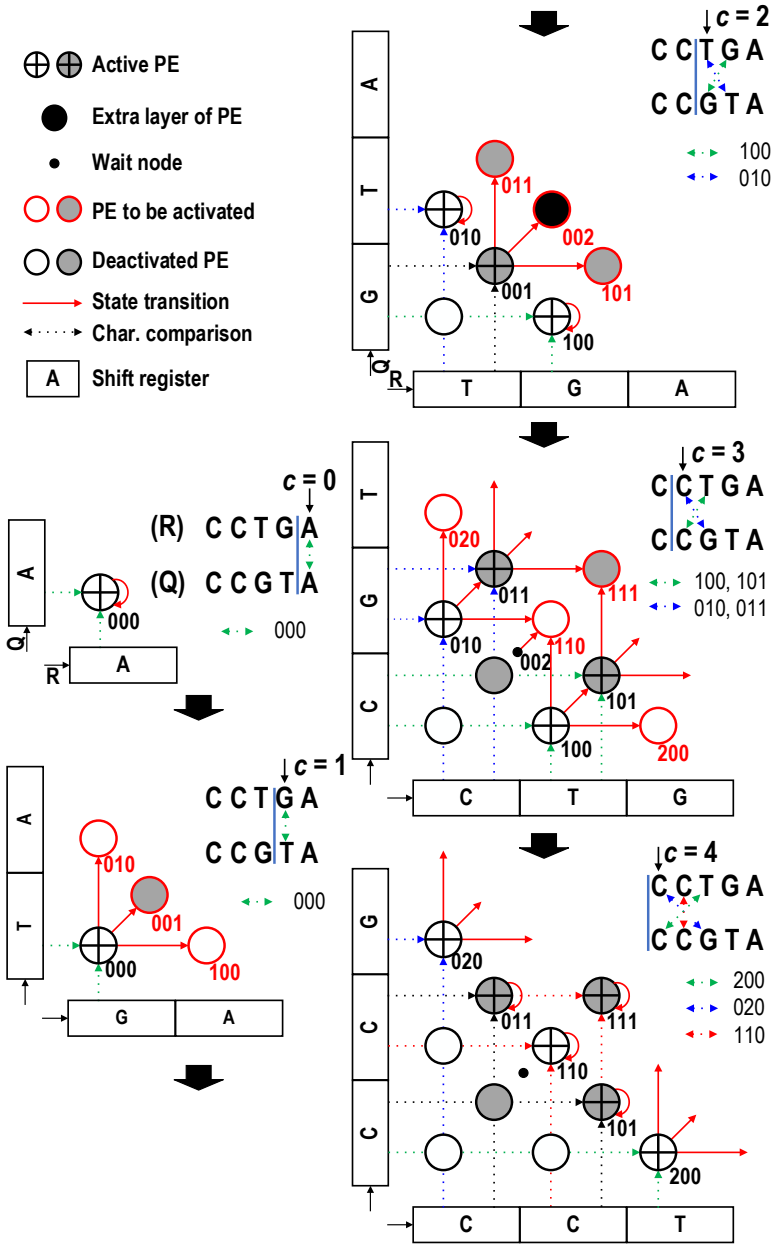

Figure 2.5 String Alignment Example.

simultaneously activates the neighboring PEs in a 3D fashion. The 3 dimensions represent insertion, deletion and substitution respectively, thus, each PE is uniquely assigned a state identifier (i, d, s) corresponding to the edit cost seen so far, e.g., PE120 corresponds to $1i + 2d + 0s$. So, the notations "PE" and "state" are referred interchangeably in the rest of the paper. In summary, the process propagates from PE000 diagonally into the 3 dimensions until the input R/Q reads are shifted out of the registers.

The example in Figure 2.5 performs as follows. Initially, in clock cycle zero ($c = 0$), only PE000 is activated and compares the first BP of the R/Q strings (A, A). For clarity, only the activated PEs are shown in the figure. Since the BP matches, the PE reactivates itself, indicating a match (m) edit string so far. In the next clock cycle ($c = 1$), the R/Q strings are shifted right/up, and PE000 compares BP (G, T). Since there is a mismatch, PE000 deactivates itself and instead activates its three neighbors, PE100, PE010 and PE001. PE100 evaluates possible alignment of R/Q after $1i$ and therefore uses the shifted base of R. Since it compares BP (G, G) in $c = 2$, it finds a match and reactivates itself for the next cycle. Similarly, PE010 represents $1d$ and compares BP (T, T) and reactivates. PE001 represents $1s$ and therefore looks at the unshifted bases of its inputs and compares BP (T, G), which is a mismatch. It therefore deactivates itself and activates PE101, PE011 and PE002.

In $c = 3$, PE100 and PE101 compare BP (T, C) and find a mismatch. While they both compare the same R/Q BP, PE100 represents $1i$ and PE101 represents $1i$ and $1s$; hence, their edit distances are not the same, preventing them from being merged. Similarly, PE010 and PE011 compare BP (C, G) and find a mismatch. PE002 represents 2s and compares BP (C, C), which is a match. However, if this pattern is followed, a full 3D grid of PEs develops, as shown by the large black circle in $c = 2$, which would result in $O(k^3)$ complexity. To reduce the complexity to

$O(k^2)$, instead, we use the observation that PE002 evaluates the same BP position in $c = 3$ as PE110 will evaluate in the next cycle, $c = 4$. In our example, since the R and Q strands are shifted right and up, respectively, in $c = 4$, PE110 also compares BP (C, C) in $c = 4$. Furthermore, in terms of edit distance, $i = d = s = 1$, so the edit distances represented by both PE002 and PE110 are equal ($1i + 1d = 2s$). Hence, they can be merged by replacing PE002 with a wait node that does nothing but activates PE110 in $c = 4$ (indicated by the small black dot in $c = 3\ and\ 4$). By offloading the evaluation of PE002 to PE110 in $c = 4$, a full 3D structure is collapsed into a 2D structure. Hence, in $c = 4$, PE110 finds a matched BP (C, C) and after reactivating itself, again finds the final match (C, C) in $c = 5$ (not shown in Figure 2.5), marking the optimal alignment of 2 edits. Note that several other PEs are also active in c = 4. However, they all have higher edit distances and are sub-optimal. Finally, the process ends after both of the input strings are shifted out of the shift registers.

Therefore, the resulting array has dimension $(k, k, 2)$. Compared to SW, the space complexity of the automata array, $O(k^2)$, is only quadratic in terms of the maximum edit distance k, making the size of the array much smaller and string independent. On the other hand, the process starts with R/Q shifted into the register and finishes with them shifted out, resulting in an additive linear time complexity $O(n + k)$ instead of multiplicative time complexity $O(kn)$ in the banded SW. Thus, the string-independent automata array renders a higher throughput for a given space.

2.3 Seed-Extension Accelerator

2.3.1 Implemented Architecture

The overall architecture of the accelerator and PE array is shown in Figure 2.6 (a). The PEs are extremely simple, consisting of only 6 gates, which OR incoming activations and activate self/neighbors depending on the comparison result. The BP comparisons are performed at the shift registers and are then passed diagonally to neighboring PEs since diagonal PEs use the same comparison result with a one-cycle delay as explained in Section II. This makes all communication local, allowing for very high-speed operation. The 25 × 25 triangular array can be decomposed

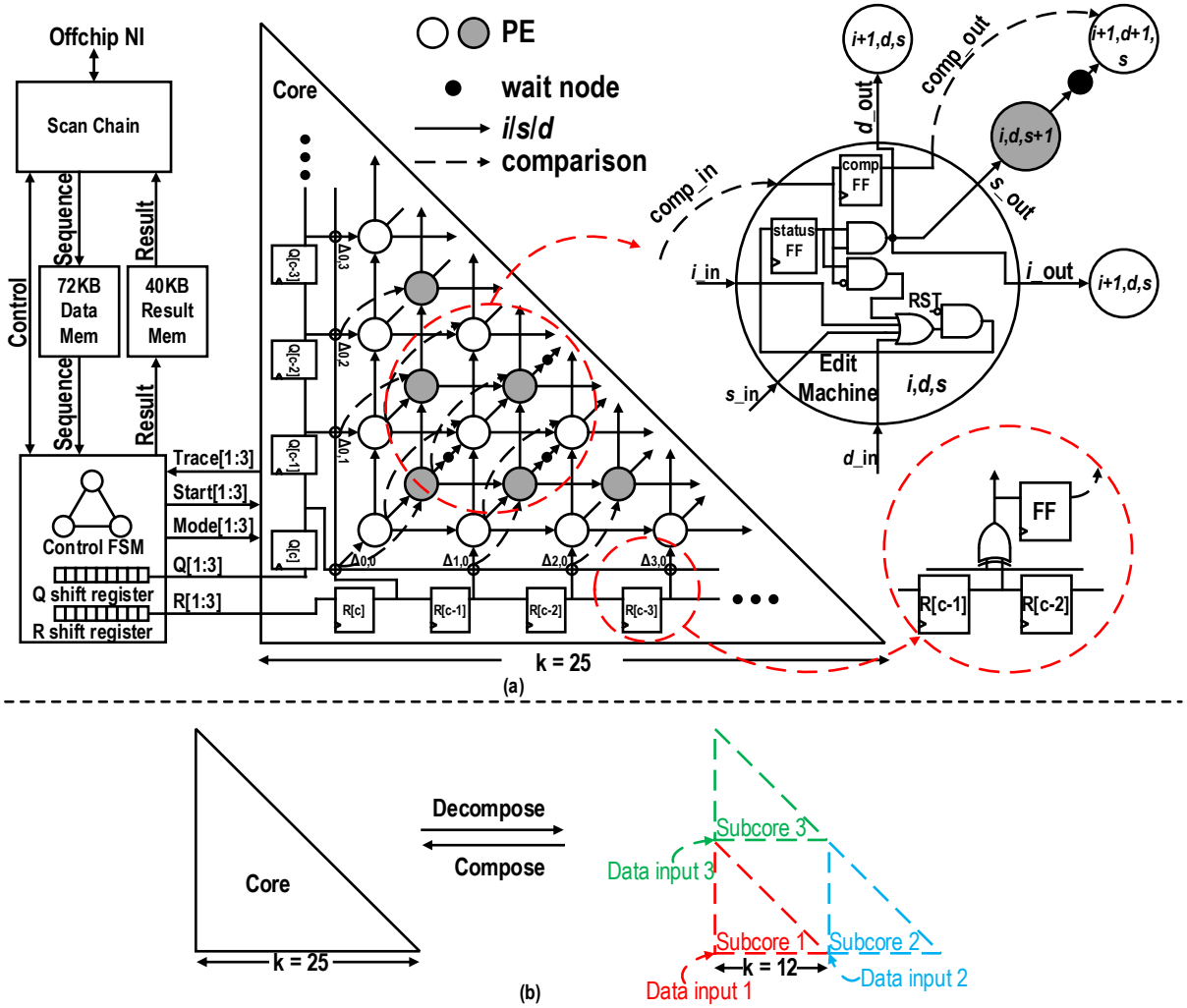

Figure 2.6 Overall Architecture of Test Chip and Details (a) and Composable Structure (b).

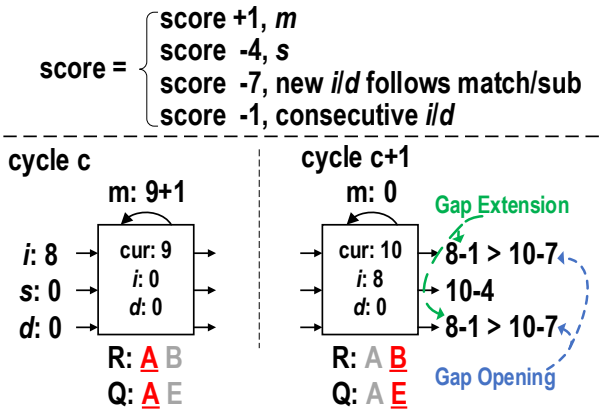

Figure 2.7 Score Scheme (Top) and Delayed Merging (Bottom).

into three smaller triangular arrays of 12×12 (Figure 2.6 (b)), enabling a trade-off between throughput and maximum edit distance, which is further discussed in Section 2.4.

2.3.2 Affine Gap Penalty and Score Machine

The standard seed-extension algorithm typically uses a more sophisticated scoring scheme in addition to edit distance, based on empirical statistics [81]. This includes the affine gap penalty used in standard BWA-MEM software [62], complying to the clipping heuristic. Accordingly, the scoring scheme, as shown in the top of Figure 2.7, is adopted in our design. The penalty scheme favors consecutive insertions and deletions (penalty -1) over new insertions and deletions (penalty -7), preventing merging confluence paths, which can occur with simple edit distance scoring.

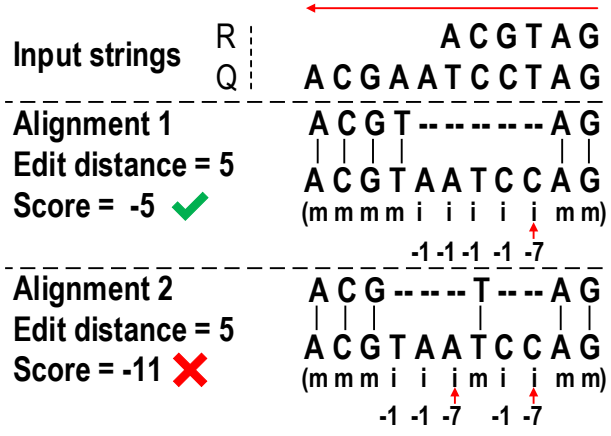

Figure 2.8 Affine Gap Scoring Example.

According to this rule, two sample alignments (Figure 2.8) have the same edit distance (5 insertions) but with different scores. Alignment 2 has two gaps separated by the matched T and T base pair in the middle, which imposes a doubled gap penalty compared to alignment 1 (two -7 penalties versus one), making the first alignment more desirable. To support the affine gap penalty, a delayed merge of two converging paths is required as in Figure 2.8. When two paths converge, they cannot merge immediately based on the current state and input scores alone since the future score depends on whether the path opens a new gap. In the example, although the incoming score from the preceding insertion edge, 8, is lower than the current score, $9 + 1$, the incoming score is not discarded immediately to merge the two paths. Instead, we latch it until a new gap opens in the next cycle to decide whether to take the incoming path and pass the higher score to the following PEs. Score calculation logic is introduced in addition to the basic PE to accommodate the

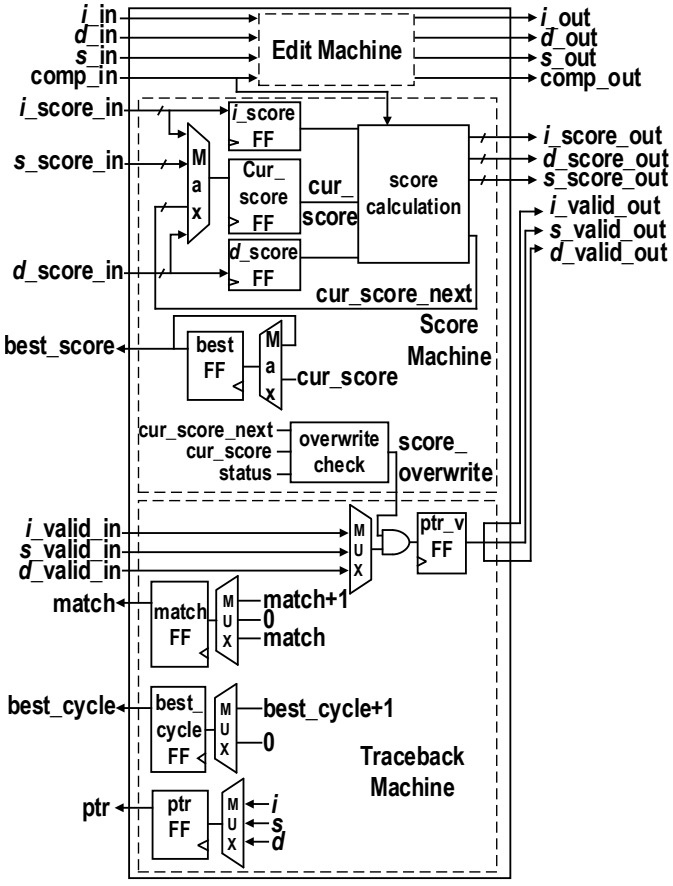

Figure 2.9 PE Augmented with Score Machine and Traceback Machine.

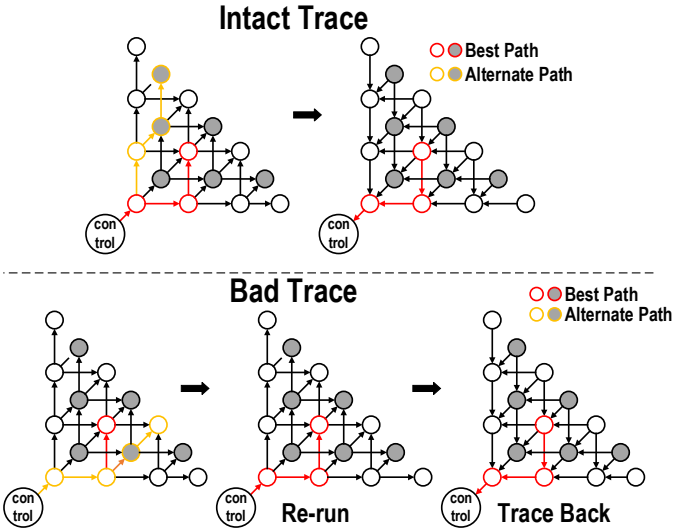

Figure 2.10 Traceback of Intact Trace (Top) and Bad Trace (Bottom).

aforementioned scheme, which passes the calculated score from PE to PE along with the state activations (Figure 2.9).

2.3.3 Collision Resolution and Traceback Machine

After the forward process of the input read pair as described above, the best trace is then shifted backward and collected by the controller to generate the final output (Figure 2.10, Top). This in-place trace back is supported by augmenting the edit machine with a traceback machine and a score machine as shown in Figure 2.9. The principle is to keep track of the best score and the incoming state pointer (i, d, s) that activates current state with that best score and also to count

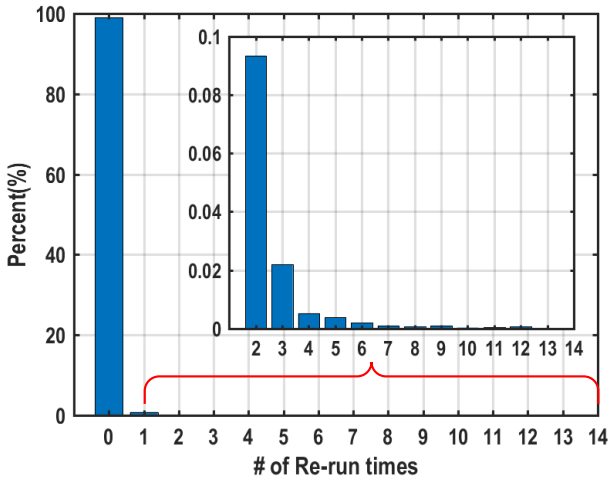

Figure 2.11 Distribution of Rr-run Times Across Dataset.

the matches that the current state meets before activating the next state. With this information, the traceback machine chains the winning states one-by-one during the backward propagation to form a complete winning path. However, during the forward process, it is possible for the correct pointer to be corrupted by a later, non-optimal path due to greedy affine gap scoring, thus breaking the backward trace of the best path. When such a collision is detected by the controller, the string is reprocessed up to the point when the broken state occurs and then traced back, as shown in Figure 2.10, Bottom. Although this requires reprocessing the string, in practice, this is rare, less than 1% of the reads (Figure 2.11), resulting in negligible performance degradation.

2.3.4 Complete Operating Sequence

Figure 2.12 shows the complete sequence of operation: after the full read pair is processed (phase I), each PE passes the maximum score backward in back-propagation mode, and the best score is retrieved from PE000 (phase II). The array controller also counts the number of cycles until the best score exits the array and then reverses the machine and propagates the best score into the array again for the same number of cycles (phase III), at which point the node that matches the

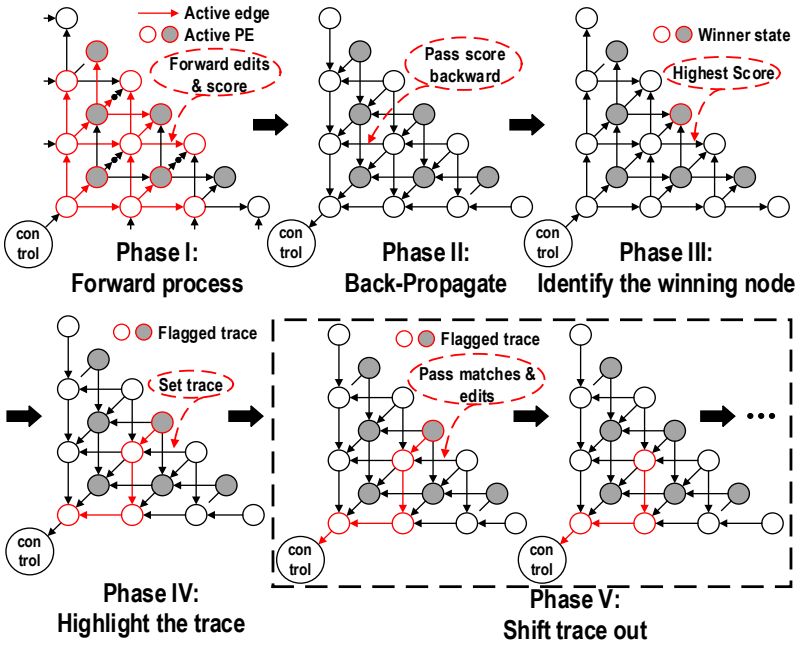

Figure 2.12 Process Sequence of the Proposed Accelerator.

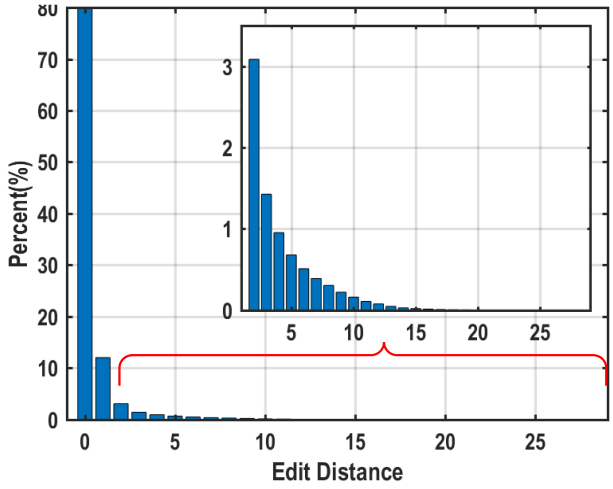

Figure 2.13 Edit Distance Across the Test Dataset.

best score self identifies. During forward processing, each PE also stores which of the incoming arcs (i, d, s) pass the best score. In phase IV, starting from the final winning state, the traceback pointers are connected in backward fashion until they reach PE000. Finally, in phase V, this backward trace is collected by shifting it back to PE000, revealing the edit string.

2.4 Measurement

A representative dataset, the Illumina Platinum Genomes ERR194147 dataset [82], is used to test our implementation. The distribution of edit distance of the reads in the dataset is shown in Figure 2.13, Left. From the distribution, 99.9967% of the reads are of edit distance no greater than 25, so with $k = 25$ for the PE array, 99.9967% of the reads are correctly processed. Since the

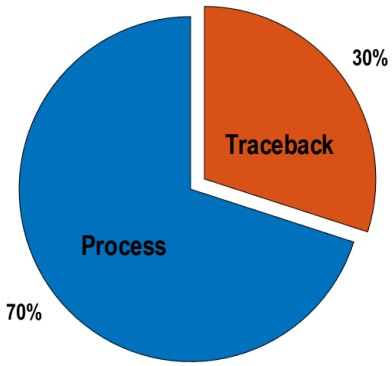

Figure 2.14 Average Time Breakdown of the Processing.

size of the array is quadratic in maximum edit distance supported, 25 is a good tradeoff point between accuracy and silicon area. For the decomposed mode, where $k = 12$, 99.7907% of the reads are correctly covered. A time breakdown of the processing on the dataset (Figure 2.14) shows

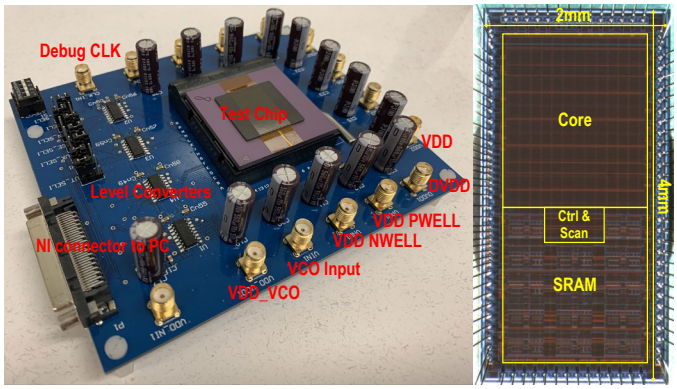

Figure 2.15 Test Board and Die Photo.

that 70% of the processing time is devoted to forward process and 30% is taken by the traceback phase.

Implemented in Mie Fujitsu 55-nm DDC technology, the test chip achieves 670 MHz core clock frequency at 0.9 V VDD and consumes 508 mW of power. Figure 2.15 shows the test board and die photo. As shown in the VDD scaling plot (Figure 2.16), the frequency of the test design can scale up to 834 MHz with 1.2 V VDD and 704 MHz with 0.2 V forward body bias at 0.9 V VDD. Figure 2.17 gives a frequency distribution across 6 different test chips.

To compare our work with the state-of-the-art, BWA-MEM is chosen as the ground truth to compare the output due to its popularity. But, because BWA-MEM also incorporates the seeding

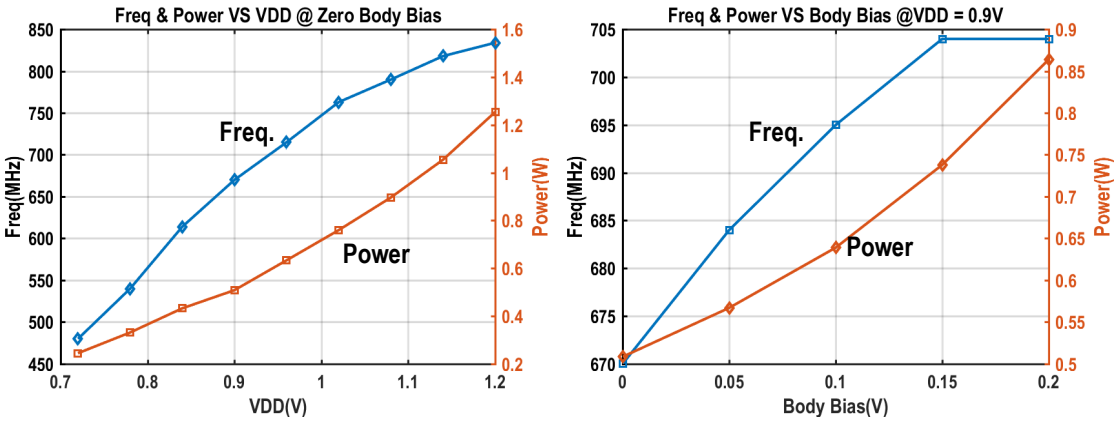

Figure 2.16 VDD Scaling and Body Bias Scaling Plot.

Table 2-1 Comparison to Other Work

	This work	CPU [66][77]	GPU [72]	[74]	[75]
Silicon Result	Yes	Software	Software	FPGA	FPGA
Technology	55nm DDC	22nm	28nm	40nm	40nm
Chip Size(mm^2)	8	662x2	551	N/A	N/A
Freq.(MHz)	670	2600	706	200	200
Power(W)	0.508	145x2	250	4.7	4.7
Throughput* (MRPS)	2.460	0.014	0.250	0.279	0.032
Power Efficiency (MRPS/W)	4.24	0.003	0.001	0.059	0.027
Area Efficiency** ($MRPS/mm^2$)	0.3075	0.000096	0.00011	N/A	N/A

*** Normalized to single core or single PE array**

**** Normalized by technology node**

step, the SeqAn library is implemented to study the performance of our work independent of the seeding step. The measured throughput is 2.46 million reads per second (MRPS) for the test dataset, and the results are identical to those obtained with BWA-MEM under the same settings of maximum edit distance k and scoring scheme. This marks a performance improvement of ~3x/~170x over SeqAn library deployed as 56-thread/single thread workload on a server with dual-socket Xeon E5-2697 v3 processors under the same configuration [66][67] and ~32,000x over SSW library on a 2GHz AMD processor [67]. It also achieves improvement of ~10x over a GPU implementation [72] or an FPGA implementation [74], all of which have a much larger silicon footprint, yielding a performance improvement normalized by area and technology of over 1000x. The power efficiency is 4.24 MRPS/W, also marking an over 1000x improvement over SeqAn on the CPU [77] and 70x improvement on the FPGA [74]. Table 2-1 summarizes the above comparison.

2.5 Conclusion

An accelerator for seed extension is presented. The accelerator consists of a triangular array of 25×25 custom-designed processing elements, implementing a string-independent automata.

Marking the first silicon implementation of the string-independent matching algorithm, it achieves 2.46M reads/s throughput and 4.24 MRPS/W power efficiency, providing 1581x power efficiency and over 1000x area efficiency improvement compared to deploying SeqAn library on dual-socket Xeon E5-2697 v3 server processors, while maintaining the same output as standard BWA-MEM library under the same configuration.

Publications related to the proposed accelerator can be found in [78] and [130].

Chapter 3. RRAM-DNN: An RRAM and Model-Compression Empowered All-Weights-on-Chip DNN Accelerator

3.1 Introduction

3.1.1 Prior Work and Limitations on Non-Volatile Memories

Although embedded Flash memory has been deployed in micro-controllers as non-volatile storage for code and data [85],[87], technology scaling poses a substantial challenge with regards to the use of such charge-based Flash, SRAM, and DRAM [86]. The reduced capacity to hold sufficient charge on the floating gate of Flash memory, the internal capacitive node of SRAM, and the cell capacitor of DRAM degrade the performance, reliability, and noise margin, limiting their applications. As possible solutions, emerging non-charge-based non-volatile memories have been proposed, such as RRAM [87]-[89], MRAM [90][91], and PCRAM [92]. Among them, RRAM is a promising candidate for wide adoption to ML/DNN applications as it has logic-process compatibility and a large on-off ratio between the high resistance state and low resistance state for potential multi-level operations [89]. Various DNN accelerators employing Computation-In-Memory (CIM) techniques on RRAM have been proposed [93][94]. However, due to limited computing precision, these CIM accelerators are not readily scalable to high-accuracy DNNs. And to date, there have been few designs that leverage RRAM's higher density and low standby power for all-on-chip parameter storage in large-scale digital DNN accelerators (versus a general-purpose non-volatile microcontroller [88]).

In this project, we present the first digital DNN accelerator featuring 24 Mb RRAM for all-on-chip weight storage to eliminate energy-consuming off-chip weight accesses, thereby reducing the overall system operating power. The design employs a 4-PE (processing element) architecture in 22nm ULL CMOS technology with 24×1 Mb custom-designed embedded RRAM banks. Using pre-compressed DNN models with an on-the-fly weight decompression mechanism, we achieve on average ~1.5 b/weight for AlexNet, 3.2b/weight for ResNet-18, resulting in a maximum total capacity of 16 M weights on chip. Highly parallelized and mesh-connected MAC arrays in the PE enable various workload mapping schemes to support DNN layers with different memory and compute characteristics. To reliably read and write to the RRAM, we propose a dynamic clamping offset-canceling sense amplifier (DCOCSA) that achieves sub-μA input-sensing offset and a Write-Verify scheme for reliable programing. Combined with a mesh-connected MAC array architecture and 8 Mb shared SRAM, the proposed DNN accelerator operates at 120 MHz at 0.8 V digital VDD, achieving 0.96 TOPS/W [95].

3.2 Overall Architecture

Figure 3.1 shows the overall architecture of the RRAM-DNN chip. The design consists of 4 PEs connected to a shared bus and a global shared memory. Each PE has its local memory for buffering the input/output activations, dedicated 6Mb RRAM memory banks for non-volatile parameter storage, MAC array units for highly parallelized processing, and instruction memory for controlling the layer functions. In the architecture, each PE has both read/write access to its

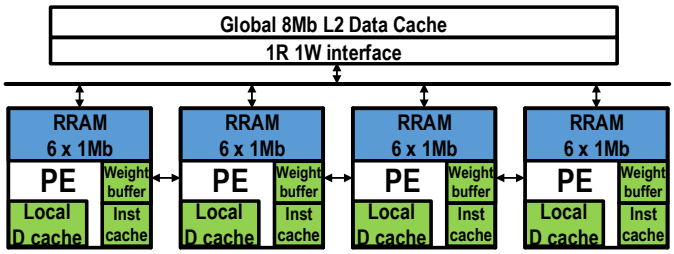

Figure 3.1 Overall Architecture of the RRAM-DNN Chip.

own local memory as well as read access to its neighboring PEs' local memories. The global shared memory is 8 Mb, and it supports parallel write and read access if the accesses are pre-partitioned to different memory banks. Due to the large chip size and the heterogenous memory hierarchy, different memories in the architecture have different access latencies. The local memories including the input and weight buffers achieve 1 cycle access latency. Accessing neighboring PE's memories and the global memory incur access latencies of 2 cycles and 4 cycles respectively. Moreover, the shared global memory coalesces multiple accesses by broadcasting data to all or a subset of 4 PEs when their read addresses are identical. In simulation, broadcasting data to coalesced requests results in ~4× latency reduction when multiple PEs are fetching the same input activation (IA) from the global shared memory.

During the execution of a layer function, a PE first loads a block of input activations from the global shared memory to its local memory following user-defined memory partitioning. The PE's neighbors can share its input activation because of the local connectivity between PEs. The PE then processes the layer function on the block of inputs with local stored weights. After all output activations are computed, the PE moves the output block back to the shared global memory. Each PE may process different data and execute different instructions, which can lead to a variable processing latency. Therefore, synchronization is necessary to ensure correct layer operations when the PEs are collaborating. The proposed design can be programed to synchronize all or a subset of 4 PEs.

3.2.1 Detailed Architecture of the Processing Element

Figure 3.2 details the design of a single PE. Inspired by [96], the PE architecture exploits parallelism and data reusability across different input dimensions to improve energy efficiency. Each PE has a mesh of 128 8-bit multiply/32-bit accumulate MAC units in 4 clusters (each with a grid of 4 × 8 MAC units). Each MAC also contains 32-bit flipflops to locally store processed partial sums. In total, 4 PEs have 512 MAC units on chip, enabling massive parallel processing for compute-intensive CNN operations. Moreover, each PE processes 4 input channels (IC), 4 output channels (OC), and 8 input activations in parallel to maximize the data reusability in the MAC array. Each PE has its own private 6 Mb RRAM for parameter storage. During the CNN operation, weights are first read from the RRAM, decompressed through the decompression engine, and transferred to small 2-bank, 4-kB interleaved weight buffers for frequent local accesses. The MAC array processing and weight decompression occur concurrently (pipelined) to maximize throughput. Accessing the small 4 KB weight memory provides 128 bit/cycle memory access bandwidth with high access energy efficiency. The 4-bank, 32 kB local buffer stores input and

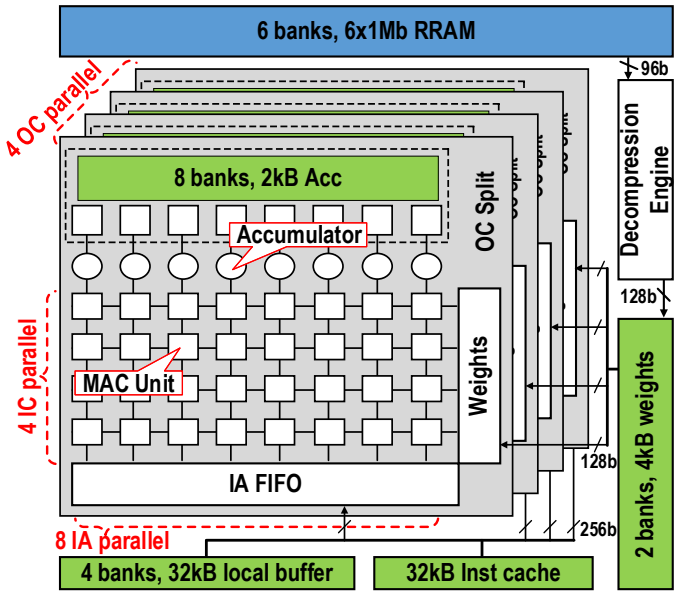

Figure 3.2 Detailed Architecture of the Processing Element.

output activation with 256 bit/cycle access bandwidth. The high data bandwidth from both the weight buffer and local buffers ensures the full utilization of the 128 MAC units.

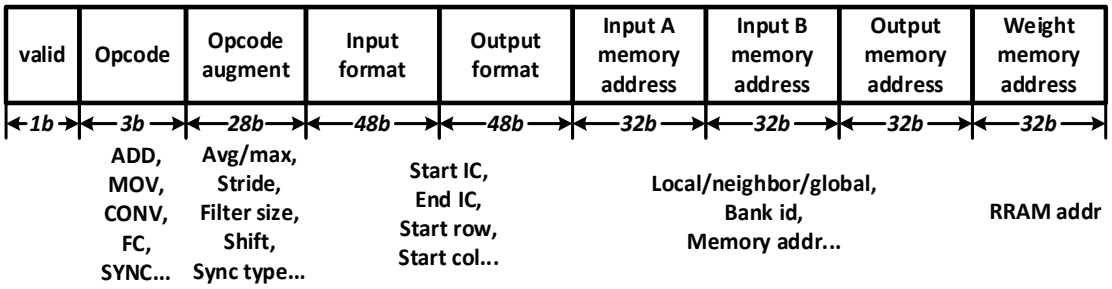

Figure 3.3 ISA of the Proposed RRAM-DNN Accelerator.

3.2.2 Instruction Set Architecture (ISA)

To control the processing of MAC units for hundreds of cycles without explicit instruction decoding in each cycle, 256-bit Very long Instruction Word (VLIW) instructions are used. Moreover, the instructions are stored in the 32 kB instruction memory of each PE so that it can be programmed independently to control the processing sequence and synchronization of the DNN algorithm if necessary. Offset (direct) addressing with respect to each PE's own base address is used in the ISA for arithmetic operations within a PE, including CONV, ADD and POOL etc., to reduce the bit-width of the instructions. Non-offset global direct addressing is used when the data are moved from/to the global memory. Figure 3.3 details the ISA of the proposed RRAM-DNN processor. The proposed ISA supports not only various layer functions such as convolution, pooling, matrix multiplication and ReLU, etc., but also flexible layer partition schemes such as the

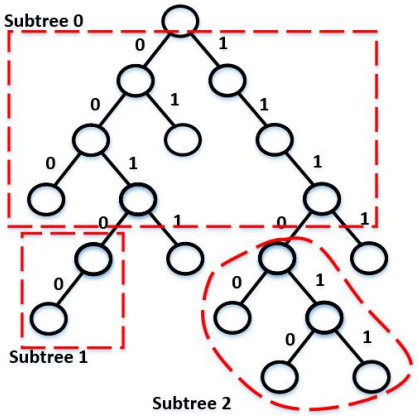

Figure 3.4 Parallel Huffman Decoder using Full Subtrees.

number of split input and output channels. Data concatenation and scaling can also be achieved through MOV (move), ADD (addition) instructions.

3.2.3 Decompression Engine

The weight compression algorithm is adopted from [34]. During the training, unimportant weights are pruned to zero and all non-zero weights are non-uniformly quantized to 64 levels. To compress each weight, we use the Huffman encoded weight value (one of 64 levels) as well as the run length of the non-zero weight position. This algorithm compresses each weight to bit on average with negligible accuracy degradation for ResNet-18 [31]. Each PE is equipped with a decompression engine to decode the compressed weights stored in the RRAM. Each decompression engine contains two programmable Huffman tables: one for weight values and the other for run-length positions. These tables share a parallel look-up table (LUT)-based decoder. Decompressing Huffman encoded weight values and run-length positions to meet the processing bandwidth of the PE is challenging. On the one hand, decompressing the Huffman encoded 96-bit in a single cycle requires a logic with very long critical paths (>10ns) due to inter-bit dependency in the compressed bit sequence. On the other hand, if the Huffman decoding was performed in

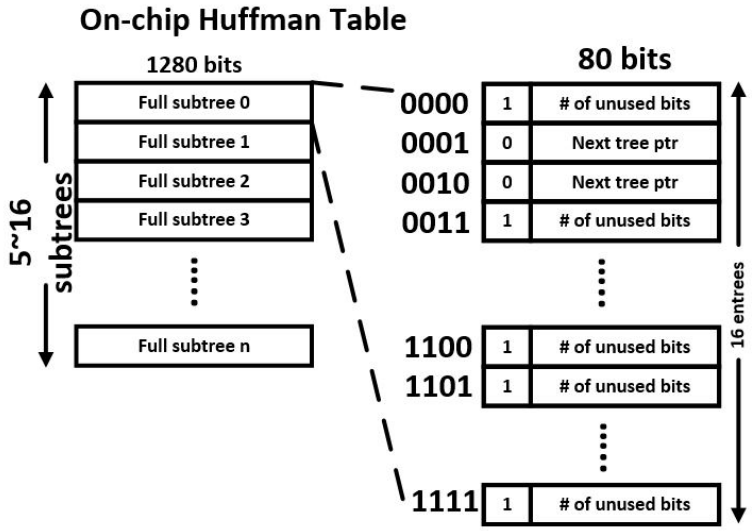

Figure 3.5 On-chip Huffman Table for Decompression.

series with single bit per cycle throughput, an entire weight packet would cost > 250 cycles to process. Decompression throughput needs to be balanced with the throughput of the MAC array which takes 72 cycles for processing 8 rows of 3x3 kernel. Therefore, instead of traversing a binary Huffman tree sequentially by advancing a single bit per clock cycle, we decode 4 bits in parallel to improve the performance (Figure 3.4) per cycle. This requires storing all possible 4-bit subtrees (Figure 3.5), which are stored in each PE and programmed through the PE programming interface. The critical path of decompressing 4 bits in parallel is 3 ns. Note that the layer-dependent non-uniform weight quantization and pruning requires reprogramming of these Huffman tables/trees for each DNN layer. We minimize the programming overhead by programming multiple PEs simultaneously when they share the same table.

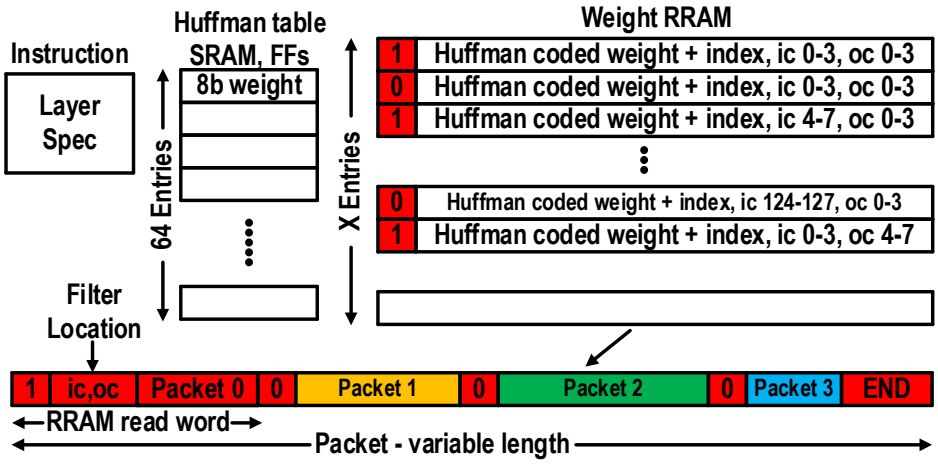

Figure 3.6 On-chip Compressed Weight Storage in RRAM.

3.2.4 RRAM Weight Storage and Static Error Resiliency

The compressed weights for convolutional layers are stored in the RRAM as packets shown in Figure 3.6. Each packet has a variable length (because each weight length is variable) and is split into multiple RRAM words. Each packet contains a layer specification, Huffman coded weight values, and run-length codes for 4 input and 4 output channels. The layer specification consists of the kernel offset and location for weights. We insert this specification information for every packet to make the system resilient to RRAM word errors. Since each weight and packet has

variable length, a single RRAM word error can cause catastrophic decompression failure for subsequent packets. The proposed packet specification enables faulty word mitigation by repeating the same packet (including the specification) twice if the first packet was written on a faulty RRAM word(s). In that case, the second packet overwrites the first faulty packet during the decompression process. We assume RRAM word error locations are static and identifiable before programming the chip.

3.3 Dataflow of Proposed RRAM-DNN

The proposed architecture and ISA support flexible mapping of heterogeneous DNNs for efficient hardware execution. This section discusses the various energy-efficient dataflows that are supported in the proposed architecture. The evaluations of different dataflows are performed with a python-based cycle accurate simulator, modeling the behavior of the designed 4-PE system. The simulator pre-allocates weights & activations onto the PEs and computes corresponding memory addresses based-on a given partitioning scheme. Then, the simulator profiles the chip behavior /

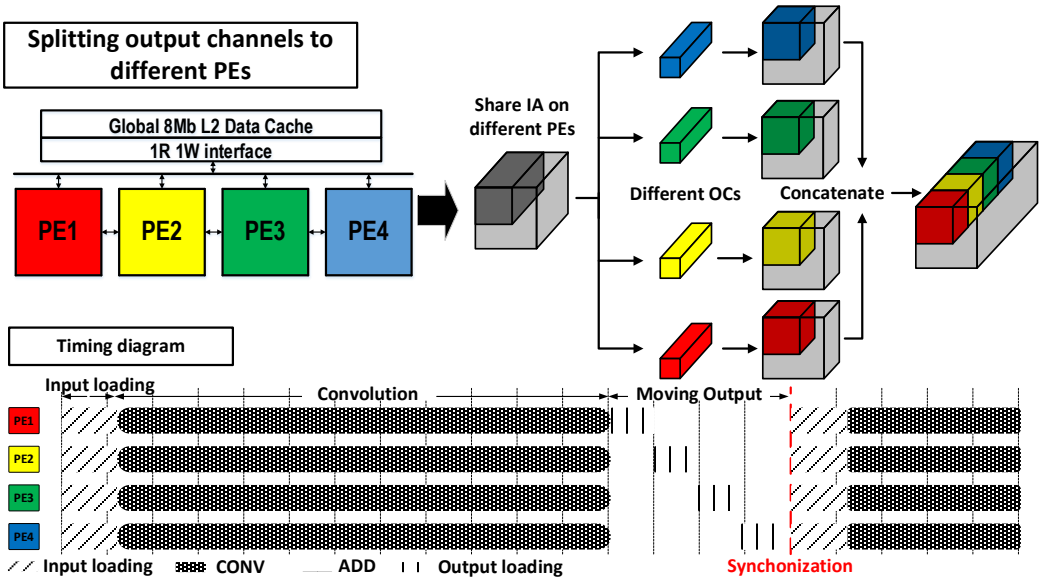

Figure 3.7 Split Convolution onto Multiple PEs by Output Channels.

execution trace for evaluation / verification, and also generates VLIW instructions (Figure 3.3) to control the chip.

3.3.1 Partition Workload onto PEs by Output Channels

One example of mapping a DNN layer to the architecture is shown in Figure 3.7. The colors in Figure 3.7 indicate weight/kernel mapping of a convolutional layer to the architecture, where the weights are split by different output channels mapped on dedicated PEs. In this example, the weights are pre-partitioned on these 4 PEs, and each PE is programmed to compute different output channels through instructions. Meanwhile, the input activations are partitioned into 8 × 8 blocks, with all associated input channels, for processing to match the local memory capacity in each PE. When the processing of an 8 × 8 block for all input channels finishes, the PE re-organizes the output and moves it back to the global memory. The outputs from multiple PEs are concatenated in this process to form the complete layer output. The timing diagram of the process is shown in Figure 3.7 (bottom). Although each PE stores only 1/4 of the total weights and also processes only 1/4 of the convolutions, the same complete input activations from the prior layer must be copied to the local memories of each PE. To minimize this potentially redundant traffic and save data

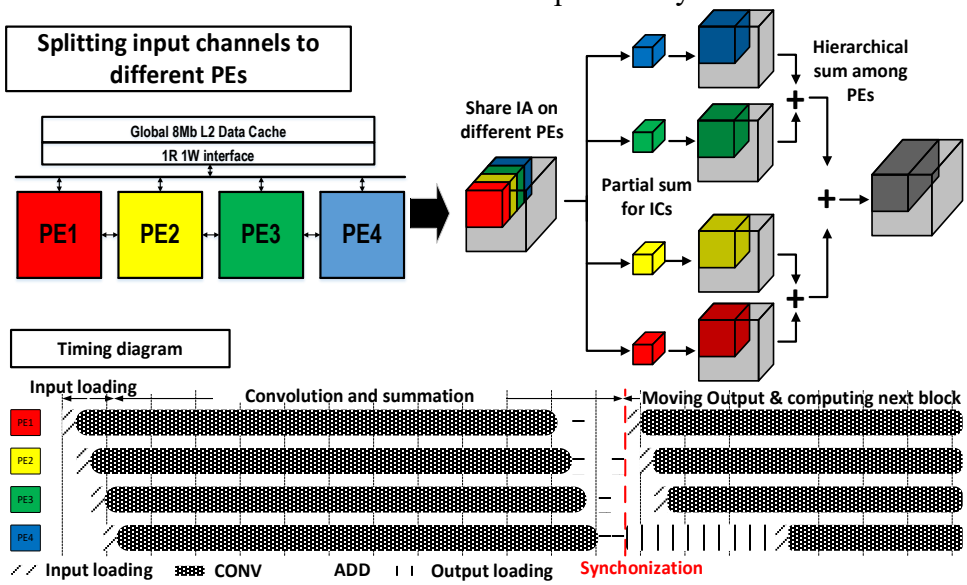

Figure 3.8 Split Convolution onto Multiple PEs by Input Channels.

transfer time, we enable the bus to broadcast input activation to all PEs. In simulation, the combination of input activation broadcasting and global memory access coalescing improves the MAC utilization and reduces the inference latency by 7% (Figure 3.9).

3.3.2 Partition Workload onto PEs by Input Channels

Another possible mapping of a convolutional layer to different PEs is input channel-based partitioning. In that case, each PE processes a partial sum of different input channels as shown in Figure 3.8. A selected PE merges the results from neighboring PEs hierarchically and then writes the merged results to the global memory (Figure 3.8, bottom). Thanks to the local connection between neighboring PEs, no extra data movement is needed as each PE has read access to each neighboring PE's local memory. Similar to the output channel split mapping in Section 3.3.1, weights are pre-partitioned on these 4 PEs and input activations are partitioned into 8 × 8 blocks (and ¼ of all input channels) for processing. Compared with splitting the output channels, this input channel partitioning scheme involves uneven workload distribution among the PEs because a selected PE(s) needs to perform the extra merge and move operations. This can potentially lead to idle cycles and low MAC utilizations for the other PEs. However, the workload can be balanced throughout the entire convolution layer if the merge and move operations are mapped onto all PEs

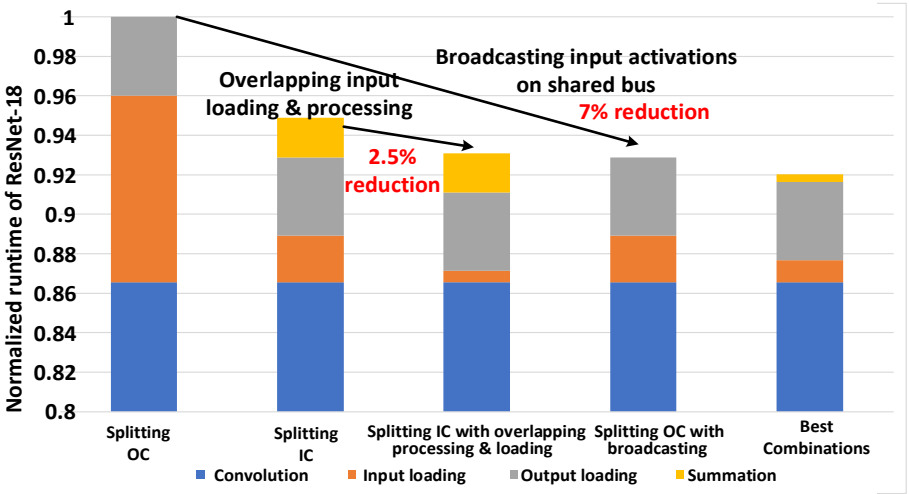

Figure 3.9 Optimizing Processing Latency with Multiple PEs (Simulation).

in a round-robin fashion. Because the input activations are partitioned into 8 × 8 × 4 (input channels) blocks per PEs, and typically there are > 64 blocks in a layer, the workload can be balanced for the overall layer. This workload balancing scheme improves the overall MAC array utilization by 2.5%. Figure 3.9 summaries the different workload partitioning methods and their impacts on MAC utilization with ResNet-18.

Each individual layer in the network can be separately programmed for the best workload partitioning based on the layer characteristic. For the ResNet-18 example shown, 8% overall latency reduction can be achieved with layer-dependent best combinations of aforementioned partitioning schemes compared to a naïve approach (Figure 3.9).

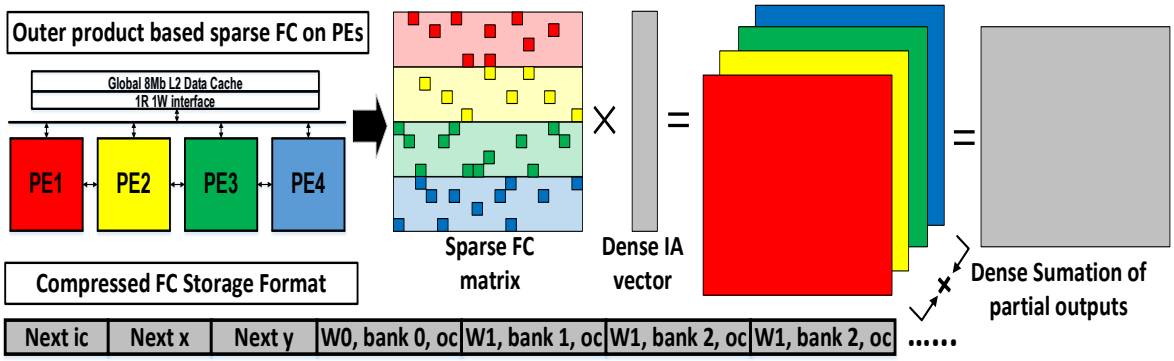

Figure 3.10 Sparse FC Operation of the RRAM-DNN Design.

3.3.3 Data Reuse for Efficient Convolution Processing

Convolution operations on a single PE are optimized with massive parallelism and data-reuse. Similar to the CNN design in [96], each PE consists of 4 clusters of 8 × 4 MAC arrays, processing 8 consecutive pixels and 4 consecutive input channels in parallel. Partial convolution is performed with shifting input activations (IAs) using a row of 8 MAC units with k (kernel size) cycles. This operation is repeated on the second and third row of IAs to complete the 2D convolution. The pooling operations are performed in the same fashion except that MAC is

replaced by Max in the 128 MAC units. Partial products in 2D convolution are accumulated locally in each MAC unit to improve the energy efficiency without unnecessary memory accesses.

3.3.4 Data Reuse for Efficient Processing of Sparse Fully Connected Layers

The compressed weights for the fully connected layers (FCL) are very sparse, with typically less than 20% density [34] (Table 3-1), whereas the input activations for the fully connected layers are densely populated. Inspired by [97], we deploy outer-product-based matrix vector multiplication to efficiently compute the FCL and skip all zero multiplications. Similar to the convolution operation, sparse FCL weights are stored in the compressed format and are pre-partitioned onto each PE during the compile time to enable highly parallelized processing with multiple PEs (Figure 3.10). Different from convolution layers, weights for FCL are only pruned and quantized without Huffman coding to increase the decompression rate to match the throughput of parallelized processing without weight reuse. During processing, each element of input activations is multiplied with sparse non-zero weights from RRAM in each PE. Partial outputs are then accumulated and stored in the accumulators depending on the location of the non-zero

Table 3-1 Example of Applying Compression Scheme on Resnet18

	Total number non-zeros parameters (TNZ)	Weight bitwidth (w)	Index bitwidth (r)	Top-1 Error	Top-5 Error	Compression rate
Baseline	Conv layers: 10.99 million FC layers: 0.513 million	FP32	-	28.22%	9.42%	-
Pruning	Conv layers: 4.565 million FC layers: 0.094 million	FP32	-	28.20%	9.42%	-
Pruning + Quantization	Conv layers: 4.565 million FC layers: 0.094 million	INT8	-	28.17%	9.40%	-
Pruning + Quantization + Runlength coding	Conv layers: 4.565 million FC layers: 0.094 million	INT8	INT5	28.17%	9.40%	6.08x
Pruning + Quantization + Runlength coding + huffman coding	Conv layers: 4.565 million FC layers: 0.094 million	5.5	2.4	28.17%	9.40%	10.01x

$$Bit/weight = \frac{(r+w) * TNZ\text{-}conv + (r+w) * TNZ\text{-}fc}{TNZ\text{-}uncompressed}$$

weights. As multiple PEs finish processing subsets of an FCL, selected PEs merge the FCL output hierarchically and write the results to the global memory.

3.4 Compressed Model for Simulation and Measurements

To enable the single-chip implementation for DNN models, we leverage an idea from a state-of-the-art deep compression scheme [34] for compressing DNN models. However, the compression scheme also has to be co-designed to maximize the performance and efficiency of the architecture. The convolution layers typically require less bandwidth to decompress weights because each weight can be reused over multiple cycles for different input/output activations. Therefore, the weights for convolution layers are pruned, non-linearly quantized with 64 weight centroids and run-length coded using 5-bit codes to achieve maximum compression. On the other hand, weights are only used once for fully connected layers. Thus, it is necessary to simplify the compression scheme to balance the weight decompression throughput with the FCL computation throughput. The weights for fully connected layers in our design are pruned without any entropy coding. On average, the proposed compression scheme achieves ~5.5 bit per weight in convolution layers and 2.4 bit per weight in FC layers. Table 3-1 shows an example of applying the compression on ResNet18 when trained and evaluated under the ImageNet dataset [98]. Pruning the weights reduces the model size of the convolution layers by 68% and fully connected layers by 82%. Run-length coding and Huffman coding further compress the pruned convolution layers by 40% (from 8-bit weight and 5-bit run-length to 5.5-bit weight and 2.4-bit run-length for non-zero weights). With both methods combined, the average bits/weight is 3.2.

After the proposed weight compression, the DNN model exhibits negligible accuracy degradation compared with 8-bit uncompressed weights under ImageNet [98] evaluation.

3.5 Customized Memory

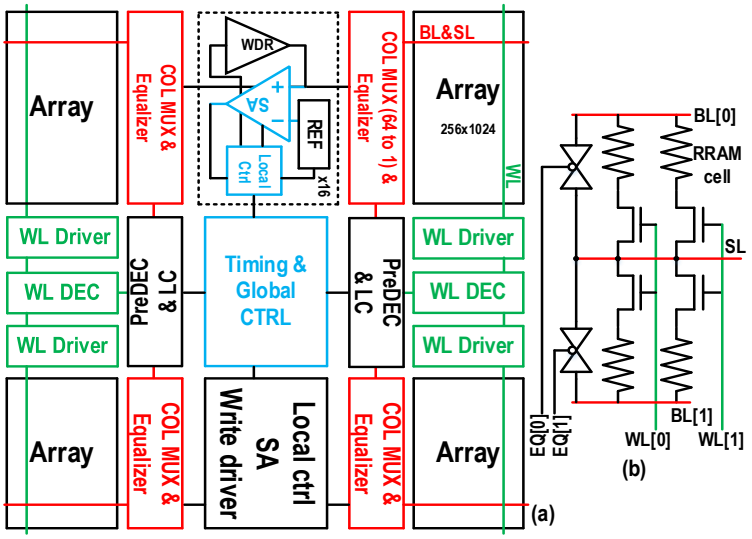

Figure 3.11 RRAM Bank Architecture(a) and Common SL Arrangement(b).

The 1 Mb custom-designed RRAM bank uses a butterfly architecture, as shown in Figure 3.11 (a), which is composed of four 256 × 1024 RRAM arrays with 32b word length. The three colors denote the three main power domains used in the bank for testing flexibility: 1.4V for the word line (WL, green), 1.25V for the column mux (red), and 1V for the sense amplifier (SA) and ctrl (blue). The RRAM array employs a common source line (SL) cell arrangement [99]. Thus, the column-wise peripherals include an equalizer (Figure 3.11 (b)) to virtually short the half-selected column when the other one is being written.

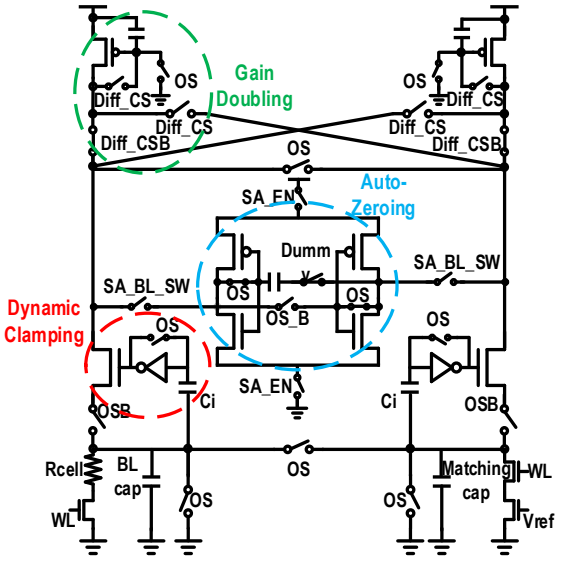

Figure 3.12 Proposed DCOCSA.

3.5.1 Dynamic Clamping Offset-Canceling SA

RRAM typically suffers from high variation in cell resistance which can vary by 2~10x for the low resistance state and 5~100x for the high resistance state [100], leaving a small sensing margin on sensing circuits. To address the high variation nature of the RRAM, the 2-stage offset-cancelling current-mode SA shown in Figure 3.12 is proposed. The first stage is composed of two cross-coupled current sampling branches similar to the scheme in [101], which doubles the input current difference and effectively halves the offset. In addition, the first stage incorporates dynamic clamping, instead of typical static clamping, to bring down the bit line settling time and increase the sensing speed. Unlike conventional clamping amplifiers, which are large and power hungry, a carefully designed self-biased inverter provides the feedback loop. The settling time is reduced by 50% (simulation) compared to a static clamping SA under the same load. The second stage

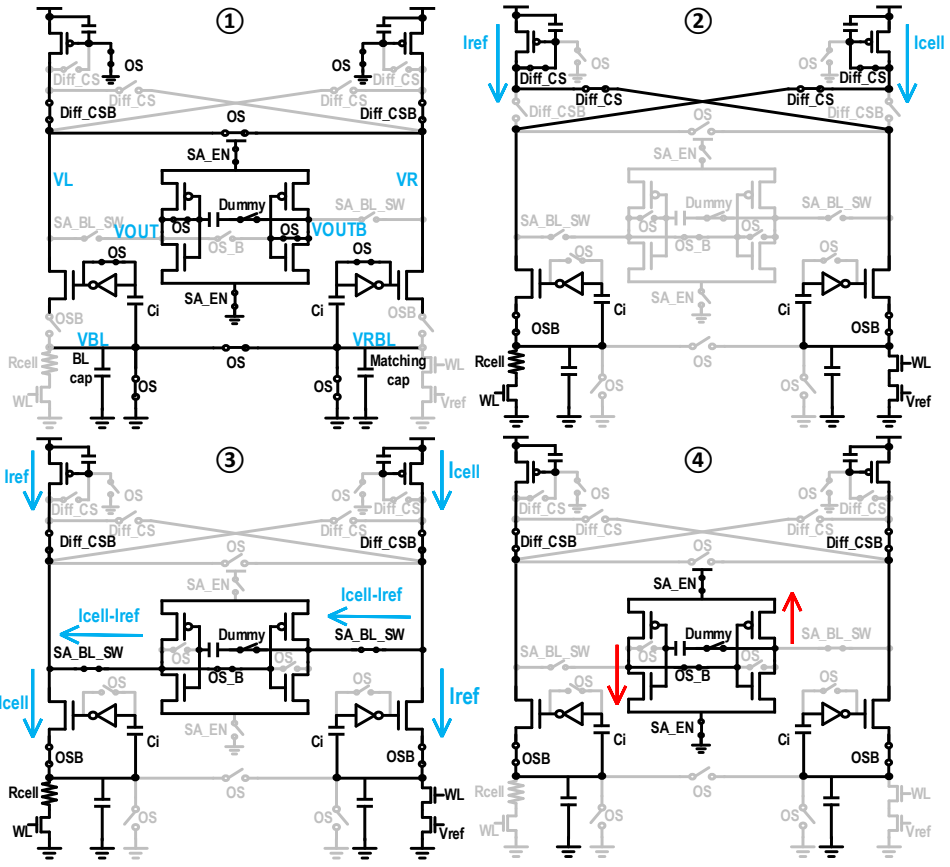

Figure 3.13 Operation of DCOCSA.

provides further amplification and offset-reduction with a single-cap auto-zero regenerative amplifier [91].

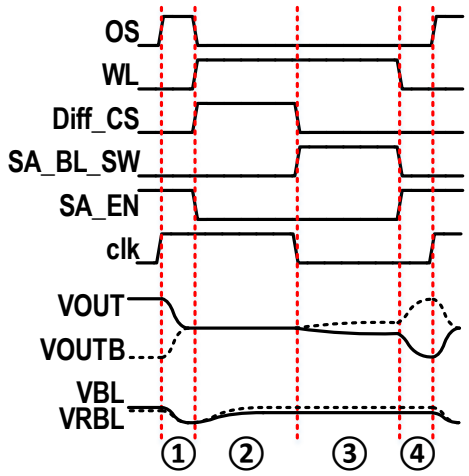

Figure 3.14 Timing waveform of DCOCSA.

Figure 3.13 details the operation of the proposed SA. In step ①, the input and output of the inverter are shorted to self-bias the clamp transistors, with bias voltage sampled on the Ci's. Meanwhile, the regenerative amplifier of the second stage is also shorted to sample the offset and cancel it out in the following steps. This step overlaps with address decoding to avoid a timing penalty. Then, in step ②, the shorted inverter in phase one is disconnected to function as a negative feedback amplifier, and the WL is turned on to allow the two diode-connected PMOS headers to sample the currents I_{ref} and I_{cell} on their respective branches. After the current settles, in step ③, the two headers are switched to the other branch and function as a current source, which generates current difference $I_{cell} - I_{ref}$ on both input nodes of the second stage. However, note that the directions of the two current differences are opposite, which effectively doubles the current difference of the input to the second stage to $2(I_{cell} - I_{ref})$. Finally, in step ④, the second stage is fired and latches the output. The voltage waveforms of the internal nodes are shown in Figure 3.14. A sub-μA current offset is achieved at 21 μA common mode input under 1.2 V VDD from

Monte Carlo (MC) simulation with 500 samples; Figure 3.15 gives the offset distribution at different temperatures.

3.5.2 Write-Verify Process

RRAM also suffers from variation in write time. At a fixed write voltage, the write time of slow cells and fast cells can differ by more than 100× [102]. Thus, applying a write pulse of the same length to both fast and slow cells causes unnecessary power and endurance losses on the fast

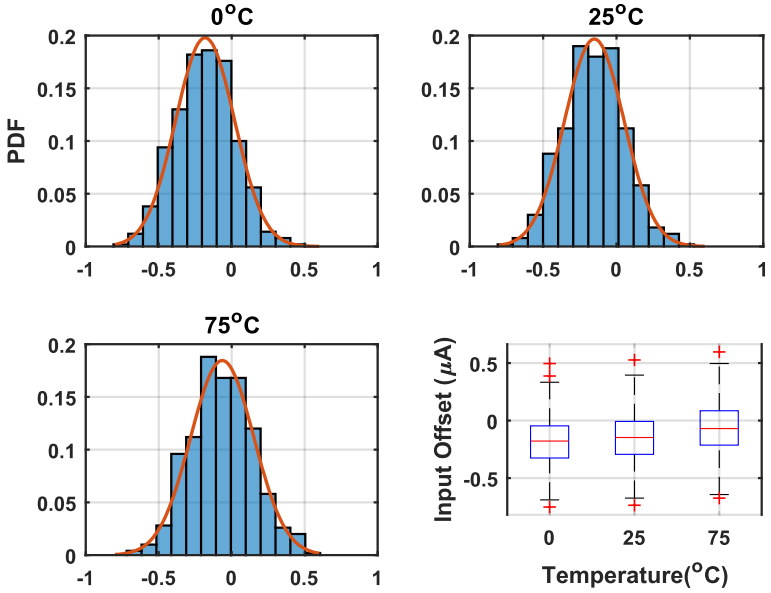

Figure 3.15 DCOCSA Input Current Offset MC Simulation Distribution.

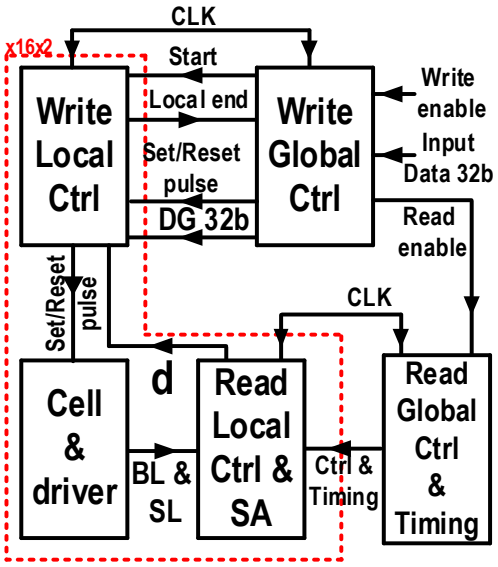

Figure 3.16 Write-Verify Signal Flow.

cell. So, a fine-grained iterative Write-Verify control is adopted. Each bit in a word is separately controlled based on the read result, ruling out correlation between fast and slow cells, which alleviates locality-dependent variation. Furthermore, with Write-Verify, each cell automatically adapts to the corresponding SA offset, further reducing the locality dependency.

Figure 3.16 illustrates the block diagram of the Write-Verify control. Following a write request, each RRAM cell of the target address is read out first to compare with the input data (DG) initiated by the global control. If the read-out value (d) of a cell is the same as the corresponding bit in DG, the write process of that cell concludes for better endurance. On the other hand, the cell is programed to the desired value by the iterative Write-Verify process.

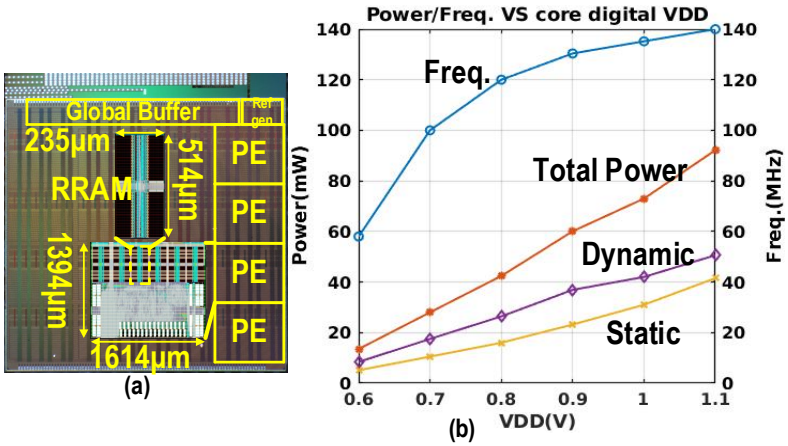

Figure 3.17 Die Photo (a) and Power/Freq vs. VDD for Core Digital Logic (b).

3.6 Measurement

We implement the proposed accelerator in 22nm ULL CMOS technology with each PE of size $1614 \times 1394\ um^2$ and each 1 Mb RRAM bank of size $235 \times 514\ um^2$ as shown in the die photo (Figure 3.17 (a)). The test chip achieves 120 MHz core clock frequency at 0.8 V VDD and consumes 42.4 mW when evaluating a CNN layer of size 4×3×3×16 as depicted in the measured power/frequency vs. VDD plot (Figure 3.17 (b)) for core digital logic, which is everything except the RRAM banks.

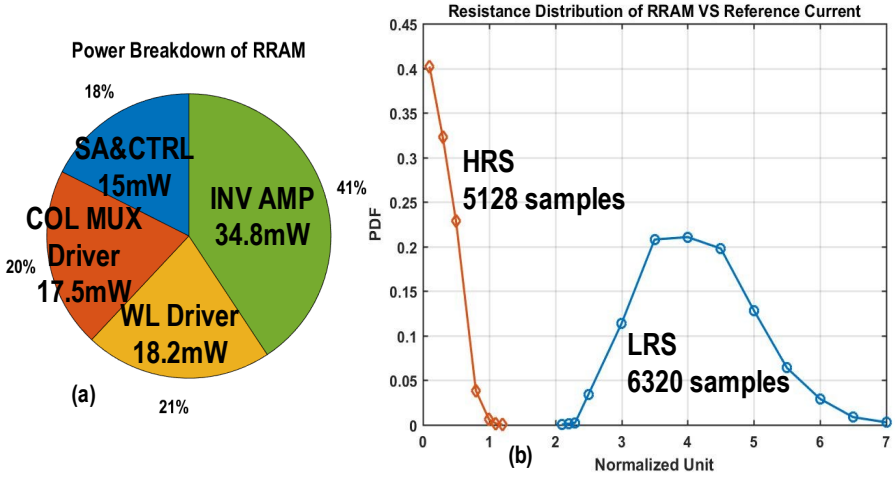

Figure 3.18 RRAM Power Breakdown (a) and Measured RRAM Resistance Distribution (b).

In the implementation, the RRAM clock is hard coded to be half of the core clock; the RRAM operates at 60 MHz for the 120 MHz core frequency. A power breakdown of the four RRAM power domains is shown in Figure 3.18 (a), with 1V for the SA and control, 1.4 V for the WL, 1.25 V for the column mux, and 1.1 V for the inverter amplifier, noting that this breakdown includes the effect of the possible static errors in RRAM. A measured RRAM resistance distribution across ~10k cells randomly sampled from the 24 banks in one test chip at room

Table 3-2 Comparison to Other Works

	This Work	QUEST[12]	SNAP[103]	STICKER[41]	UNPU[37]	Envision[104]
Technology	**ULL 22nm**	40nm	16nm	65nm	65nm	28FD-SOI
On-chip RAM(B)	**3M RRAM 1.3M SRAM**	7.68M+96M 3D SRAM	280.6K	170K	256K	148K
Max On-chip Weight	**16M@8b Non-Volatile**	15.36M@ 4b Volatile	140.3K@16b Volatile	170K@8b Volatile	256K@8b Volatile	148K@8b Volatile
Off-chip Memory	**No**	Yes	Yes	Yes	Yes	Yes
MACs	**4x128 (8x8b)**	24x512 (1x1b log)	252 (16x16b)	256 (8x8b)	4x576 (1x16b)	256 (8x8b)
Voltage (V)	**1.0-1.2 RRAM 0.6-1.1 Core**	1.1	0.55-0.8	0.67-1.0	0.63-1.1	1.05
Freq. (MHz)	**60 RRAM 120 Core**	300	33-480	20-200	200	200
TOPS/W	***0.96@8b**	†0.59@4b	‡3.61@16b	‡1.038@8b	‡5.57@8b	‡1@8b
GOPS	**123@8b**	1960@4b	65.52@16b	102@8b	690@8b	102@8b
Power (mW)	**127.9 @120MHz**	3300 @300MHz	364 @480MHz	284.4 @200MHz	297 @200MHz	44 @200MHz
Chip Area (mm^2)	**10.8**	122	2.4	12	16	1.87

* Including power of loading weights from RRAM to SRAM and MAC
† Including power of loading weights from 3D SRAM to on-chip SRAM & MAC
‡ Excluding power of loading weights from off-chip memory

temperature is shown in Figure 3.18 (b). The proposed accelerator consumes 127.9 mW in total, including weight decompression and transfer from RRAM to SRAM, resulting in a power efficiency of 0.96 TOPS/W. Table 3-2 compares the work to recent NN accelerators. The proposed design achieves the highest number of on-chip stored weights due to the model compression and better density of RRAM and is also the only design employing non-volatile memory as dedicated weight storage, thereby reducing standby power for edge devices.

3.7 Conclusion

In summary, we present the first energy-efficient digital DNN accelerator featuring RRAM for dedicated weight storage to enable efficient single-chip inference of NN models for mobile devices. Using on-the-fly weight decompression, we achieve a total capacity of 16 M 8bit weights on chip. To reliably read from and write to the RRAM, we propose a dynamic clamping offset-canceling sense amplifier (DCOCSA) achieving sub-μA input-sensing offset. Together, these techniques help us eliminate fully off-chip weight access. The proposed processor is prototyped and measured in TSMC 22nm ULL with RRAM technology. This design supports single-chip NN model inference with ~16 million parameters. It achieves 123 GOPs throughput in real-time, consuming 127.9 mW from a 0.8 V supply, with measured 0.96 TOPS/W efficiency. The proposed design achieves the highest number of on-chip-stored weights and is also the only design employing non-volatile memory as dedicated weight storage, reducing standby power for edge devices.

Publications related to the DNN accelerator can be found in [95] and [131].

Chapter 4. Accelerator Design for Third Generation Fully Homomorphic Encryption and Private Set Intersection

4.1 Introduction

The past decade has witnessed the rapid development of Neural Networks (NN) / Machine Learning (ML) with unprecedented accuracy on various emerging computational tasks such as computer vision (CV) and Natural Language Processing (NLP). These data-driven approaches are being adopted in almost every aspect of human life, extending from online advertising, stock trading, and autopiloting vehicles to biomedical research. However, such ubiquitous employment of ML introduces severe concerns regarding the privacy of the users' data.

As modern NNs grow in parameter size and complexity of network structure to achieve higher accuracy and cope with more complex tasks, it is common to outsource ML computation to data centers for their excellent storage capacity and computational performance. For example, it took approximately 16 TPUv3s (equivalent to ~100–200 GPUs) of Google Cloud and a few

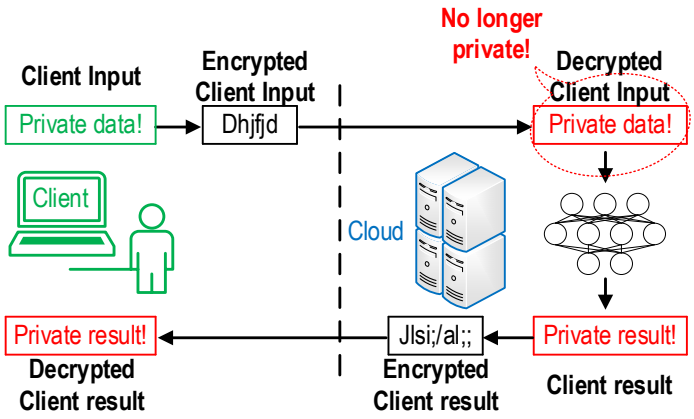

Figure 4.1 Conventional Client/Server Computation Model.

weeks of time to train AlphaFold2, which is a state-of-the-art protein structure predictor NN that achieves groundbreaking accuracy in predicting protein structure [132]. Although most research in NNs takes advantage of public data sets, the scenario is quite different when it comes to commercial applications. For instance, when outsourcing data analysis to a third-party cloud service provider, a medical institute has to reveal patients' medical records to said provider, which not only violates HIPAA regulations but also allows the data to be potentially exploited by security loopholes in the cloud, even if the data is encrypted during transfer. As illustrated in the typical client/server computation model (Figure 4.1), the client input is secure until the computation takes place. However, to work on the input from a client, the encrypted client data must be decrypted first, which reveals the private data to a potentially untrusted cloud server.

In fact, not only ML, but any computation that complies with a server/client or multiparty model can potentially leak a user's private data, even if the data being transferred is securely encrypted. For a social media app to discover which person in a cellphone contact list also owns an account with the same social media, the contact list needs to be uploaded to the cloud for cross comparison, which reveals private information to the cloud. Privacy is not only a personal concern but also has profound implications for other aspects of society. For instance, in an e-voting system,

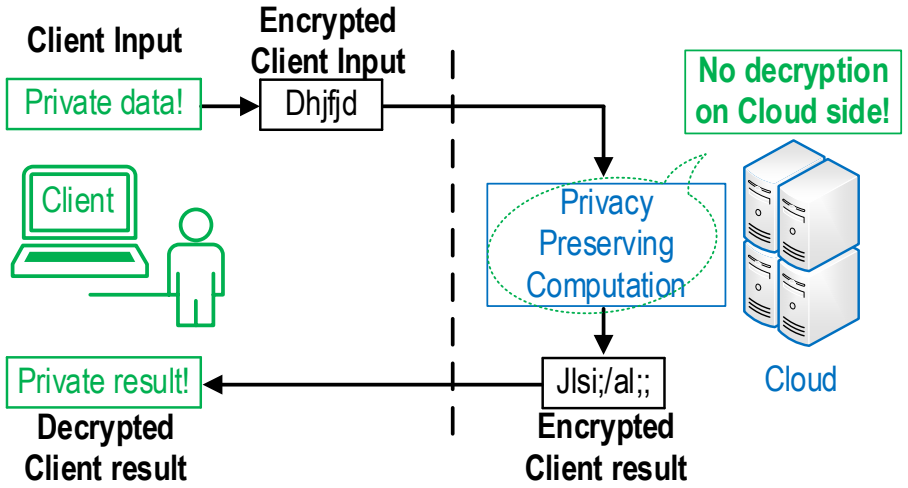

Figure 4.2 Privacy Preserving Computation Model.

although the ballots filled in by voters are encrypted, counting the results requires all individual ballots to be decrypted first, which renders the results susceptible to tampering. Thus, with the expansion of data centers and cloud services, and as a growing number of services are moving online, especially after the COVID-19 pandemic, privacy-preserving computation (Figure 4.2) is expected to play a pivotal part of this industry in the future.

There have been extensive investigations focused on the use of security primitives to preserve privacy in various aspects of computation. Examples include but are not limited to the following: Private Set Intersection (PSI) [105][106] [110][117][133]-[135], which allows two parties to learn about the intersection of their private data base without leaking any excess information beyond the intersection; Oblivious RAM (ORAM) [108][109][111], which conceals the data access pattern of a user when accessing a cloud storage; and Fully Homomorphic Encryption (FHE) [48]-[54][58]-[60] [113], which permits a more general direct computation on encrypted data without decrypting it first.

Since 2009, when the first general FHE framework [48] was proposed by Gentry, several FHE schemes have been published [50]-[53][58]-[60]. Due to the generality of the concept, FHE can also be employed to build other security primitives, like PSI [110][117] and ORAM [111]. So far, although several open-source libraries [55]-[57] have pushed the processing time of the above FHE schemes from weeks to seconds or even milliseconds, the computation on encrypted data is still order of magnitudes slower than computing on plaintext data, due to the prohibitive amount of compute power that challenges the current compute systems. Thus, hardware and software optimization for accelerating the FHE process is critical to the practical employment of it.

4.1.1 Fully Homomorphic Encryption

Homomorphic encryption (HE) is an encryption scheme that allows operations to be performed directly on encrypted data. To be precise, for a given function f(x), a homomorphic encryption scheme satisfies $f(Enc(m)) = Enc(f(m))$. If it is homomorphic to any function, it is characterized as fully homomorphic encryption.

Exploration of HE started following the advent of RSA encryption due to its multiplicative homomorphism. However, no scheme with fully homomorphic capability was devised until 2009, when Gentry proposed a general FHE framework [48][113], which all the subsequent works follow. It has been proven that from a Boolean circuit model perspective of computation, if an encryption scheme is homomorphic to its own decryption function followed by a universal logic gate, then it is homomorphic to any function. The operation that fulfills this property by refreshing the noise level of the ciphertext after each operation, thus transforming a partially HE or Leveled HE (LHE) into an FHE, is called bootstrapping or recryption. Based on this idea, Gentry also presented a concrete construction based on the ideal lattice and sparse subset sum problem; however, the software implementation requires around 30 minutes of computation time per bootstrapping [49].

Since then, various schemes have been proposed offering more efficient implementations. Among them, the most well-known are BGV [50], BFV [51][52], and CKKS [53]. These second-generation schemes differ from Gentry's approach in the underlying hard problem: the Ring Learning with Errors (RLWE) problem is employed for its better-studied hardness analysis and efficiency obtained via SIMD-styled computation [54]. BGV and BFV schemes operate on polynomial functions of integer numbers and have shared much in common during their development since the first publication, whereas CKKS works with complex numbers in the

plaintext space. Several open-source libraries that implement these schemes are available, including PALISADE [55], HElib [56] and SEAL [57]. These implementations can potentially reduce the recryption time to minutes depending on the security parameters.

After the GSW [58] scheme was proposed in 2013, two schemes, FHEW [59] and TFHE [60], were published as third-generation approaches. Compared to the second-generation solutions, the third-generation schemes focus on finding an efficient implementation of single-bit logic and the bootstrap operation. Although performance-wise the third generation may not be superior to earlier schemes (since the amortized cost of the second generation equipped with SIMD-like construction is estimated to be within the same order of magnitude as that of the third generation), the third generation is superior in its simplicity and flexibility in terms of both concept and implementation. Typically, to achieve a 128-bit security level, the polynomial length is less than 2048, which is much shorter than the 10K required in the second generation. Further, the ciphertext modulus is less than 64 bits, compared to ~200 bits in the second generation. The reported recryption time of the third generation is generally around 0.1s to 1s.

Although the FHE processing time is reduced from about half an hour [48] to minutes [55][56] in second-generation schemes or even sub-seconds [59][60] in third-generation systems, it is still many orders of magnitudes slower compared to operations on plaintext due to the prohibitive requirement of compute power that challenges current computing systems. This still renders FHE largely impractical. Thus, various hardware solutions have been proposed in recent years [119]-[129]. [119][121] focused on encryption/decryption of RLWE in a post-quantum scenario, which is less computation demanding due to the smaller size of the polynomial. Reference [120] presented a crypto-engine for the encryption/decryption of RLWE for homomorphic encryption, which is less heavy lifting compared to homomorphic evaluation. The

authors in [128] explored acceleration for large number multiplication, while [126][127] discussed approaches to accelerate long polynomial multiplications in homomorphic encryption. Other works [123]-[125][129] implemented accelerators for LHE schemes based on the BFV scheme, with limited computation depth and security levels. Finally, an architecture for the CKKS scheme was proposed in [122]. To date, there has been no hardware acceleration published for third-generation FHE schemes.

4.1.2 Private Set Intersection

PSI allows two parties (Sender and Receiver) to exchange the intersection of their private sets without leaking any excess information other than the intersection set. In other words, it protects the privacy of two parties that exchange information. Thus, its applications include private human genome testing, contact list discovery of social media apps, and conversion rate measuring of online advertisement. Recently, Microsoft introduced Password Monitor in the latest release of the Edge web browser, which compares a user's private passwords saved to Edge with a known database of leaked passwords to figure out whether there is a leak in the user's passwords. With the underlying PSI protocol, privacy of the comparison is ensured, meaning that the server that facilitates the comparison learns nothing about the user's passwords.

The PSI problem has been explored extensively, seeking efficient protocols [105][106][133]-[135]. However, most of them have a communication complexity linear with the size of both sets. Therefore, in an unbalanced scenario where one set is significantly smaller than the other, these protocols still perform linearly based on the size of the large set. In recent years, unbalanced PSI protocols based on second-generation FHE [110][117] were proposed that provide significant communication overhead reduction compared to previous approaches but maintain comparable performance. However, they still suffer from the large encryption parameters of

second-generation FHE. Therefore, it is worth exploring a possible PSI protocol that exploits the lightweight parameters of third-generation FHE.

4.1.3 Our Contributions

This paper has four major contributions:

1) We present the first accelerator architecture for third-generation FHE, targeting the $RLWE \otimes RGSW$ operation (defined in the following section), which is a fundamental function of both second-generation and third-generation FHE. By exploiting the asymmetric nature of the encryption, the architecture incorporates an asymmetric Inverse Number Theory Transform (INTT) module and Number Theory Transform (NTT) module, which are capable of maintaining high throughput with less resource usage while addressing different parameter sets. An extensive analysis of the architecture is included.
2) We propose a novel unbalanced PSI protocol that is based on third-generation FHE and is optimized for the proposed hardware architecture. The proposed PSI protocol makes the computation cost independent of the Sender's set size. The core block of the PSI that facilitates the cross comparison of the PSI in [110] is replaced with a homomorphic lookup table (LUT) implemented with third-generation FHE. Unlike the multiplication used in [110], which returns a nonzero value when the cross comparison misses and potentially leaks the content of the Sender's set, the LUT only returns one bit indicating whether an element is inside the Sender's set and thus avoids sending any excess information about the Sender's set. Therefore, the noise flooding process adopted in [110] is not necessary. We introduce several additional algorithm-architecture co-optimizations to reduce the computation and

communication costs, rendering a practical application of the proposed PSI protocol.

3) A prototype of the proposed architecture is implemented with AWS cloud FPGA service. We develop all necessary high-level functions in C++ and benchmark the implemented architecture with different parameter sets. We make the SystemVerilog HDL code of the proposed accelerator and supporting software code publicly available at [136].
4) We quantify and analyze the performance of the proposed hardware acceleator and PSI protocol. The measurements show over 21× performance improvement compared to a software implementation for various subroutines of the third-generation FHE and the proposed PSI.

4.2Preliminaries

Before diving into the detailed algorithm and architecture, it is worth to review the preliminary knowledge on which the aforementioned FHE schemes are built.

4.2.1 Notations

In the remaining chapter, bold face lower-case letters $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \ldots$ are used to denote vectors or polynomials depending on the context, and bold face upper-case letters $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \ldots$ are used for matrices. The set of integers is denoted by $\mathbb{Z}$, and the quotient ring of integers modulo q is denoted by $\mathbb{Z}_q$. "×" denotes the scalar multiplication with either another scalar or a vector/polynominal. "·" denotes vector inner product or polynomial product depending on the context, while "⊙" denotes the outer product or element wise product of polynomial. At last, "⊗" is used to represent

a special operation that multiplies and accumulates a polynomial with a set of polynomials, which will be defined later.

Throughout the paper, boldface lower-case letters $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \ldots$ are used to denote vectors or polynomials depending on the context, and boldface upper-case letters $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \ldots$ are used for matrices. The set of integers is denoted by $\mathbb{Z}$, and the quotient ring of integers modulo q is denoted by $\mathbb{Z}_q$. The polynomial ring is denoted by $R = \mathbb{Z}[X]/(X^N + 1)$, where N is a power of two. And $R_Q = R/QR$ represents the residue ring of R modulo an integer Q. "$\times$" denotes the scalar multiplication with either another scalar or a vector/polynomial. "$\cdot$" denotes the vector inner product or polynomial product depending on the context, while "$\odot$" denotes the outer product or element wise product of a polynomial. Lastly, "$\otimes$" represents a special operation, which will be defined later in this section.

4.2.2 Learning with Errors

Just like RSA encryption relies on the hardness of factoring large numbers, almost all FHE schemes published so far are built upon Learning with Errors (LWE) or Ring Learning with Errors (RLWE) problem, which can be reduced to a lattice problem that is proved to be quantum safe within polynomial time [47][115].

The LWE problem states that given a polynomial number of vectors of form $[\boldsymbol{a}, b = \boldsymbol{a} \cdot \boldsymbol{s} + e]$, where $\boldsymbol{a}$, a vector of dimension n, is sampled uniformly from the integer vector space $\mathbb{Z}_q^n$ and error e is sampled from an error distribution χ, it is hard to extract the secret vector $\boldsymbol{s}$ [47].

In practice, given a plaintext modulo t and a ciphertext modulo q, an LWE encryption of a plaintext $m \bmod t$ is defined as

$$LWE_{\boldsymbol{s}}^{\frac{q}{t}}(m) = \left[\boldsymbol{a}, b = \boldsymbol{a} \cdot \boldsymbol{s} + e + m \times \frac{q}{t}\right], \tag{4.1}$$

with the vector $\boldsymbol{a}$ and noise e as described above. As long as $|e| < \frac{q}{2t}$, the plaintext can be successfully recovered by $m = \left\lfloor (b - \boldsymbol{a} \cdot \boldsymbol{s}) \times \frac{t}{q} \right\rceil$, which rounds off the noise.

A special operation called key switch can convert a LWE ciphertext encrypted under a secret key $\boldsymbol{s_1}$ into another LWE ciphertext encrypted by a different secret key $\boldsymbol{s_2}$. The operation is fulfilled by a LWE key switch key that encrypts the first key $\boldsymbol{s_1}$ into a lookup table (LUT) using the second key $\boldsymbol{s_2}$. For each element $\boldsymbol{s_1}[i]$ in $\boldsymbol{s_1}$, a vector of LWE ciphertexts with secret key $\boldsymbol{s_2}$ is built, as in Equation 4.2. As a result, a 2-dimensional LUT is generated, which contains $n \times q$ LWE ciphertexts encrypted by key $\boldsymbol{s_2}$.

$$[LWE_{\boldsymbol{s_2}}(0 \times \boldsymbol{s_1}[i]), LWE_{\boldsymbol{s_2}}(1 \times \boldsymbol{s_1}[i]), \dots, LWE_{\boldsymbol{s_2}}((q-1) \times \boldsymbol{s_1}[i])] \quad 4.2$$

For a LWE ciphertext $LWE_{\boldsymbol{s_1}}(m) = [\boldsymbol{a}, b]$ with key $\boldsymbol{s_1}$, to switch to key $\boldsymbol{s_2}$, the new ciphertext is generated by Equation 4.3, with each LWE ciphertext from the LUT index by i and $a[i]$. It is easy to verify the correctness of the process with a small amount of noise increase. A formal definition can be found in [116].

$$LWE_{\boldsymbol{s_2}}(m) = [\boldsymbol{0}, b] - \sum_{i=0}^{n-1} LWE_{\boldsymbol{s_2}}(\boldsymbol{a}[i] \times \boldsymbol{s_1}[i]) \; mod \; q \quad 4.3$$

There exists a time-space trade-off for the key switch process. Given a decomposition base B_{KS}, the product of each $\boldsymbol{a}[i] \times \boldsymbol{s_1}[i]$ in Equation 4.3 can be rewritten with a decomposed $\boldsymbol{a}[i]$ as Equation 4.4, where $\sum_{j=0}^{\log_{B_{KS}}(q)-1} \boldsymbol{a}[i][j] \times B_{KS}^{j} = \boldsymbol{a}[i]$. Therefore, when building the LUT, instead of a vector of LWE ciphertexts for each $\boldsymbol{s_1}[i]$, a 2-D array of LWE ciphertexts is built, with each element of the form $LWE_{\boldsymbol{s_2}}(v \times B_{KS}^{j} \times \boldsymbol{s_1}[i])$ for $v \in [0, B_{KS} - 1]$ and $j \in [0, \log_{B_{KS}}(q) - 1]$. In total, the new LUT is of 3 dimensions. When switching the key from $\boldsymbol{s_2}$ to $\boldsymbol{s_1}$, each $\boldsymbol{a}[i]$ is first decomposed by B_{KS}, and then the new ciphertext is generated by Equation

4.5. With the decomposition, the size of the 3-D LUT is $B_{KS} \times log_{B_{KS}}(q) \times n$, and with a proper B_{KS}, it can be much smaller than the size of a 2-D LUT, $n \times q$, at the cost of more LWE additions/subtractions, facilitating a time-space trade off.

$$\sum_{j=0}^{\log_{B_{KS}}(q)-1} \boldsymbol{a}[i][j] \times B_{KS}^{j} \times \boldsymbol{s_1}[i] \tag{4.4}$$

$$LWE_{\boldsymbol{s_2}}(m) = [\boldsymbol{0}, b] - \sum_{i=0}^{n-1} \sum_{j=0}^{log_{B_{KS}}(q)-1} LWE_{\boldsymbol{s_2}}\left(\boldsymbol{a}[i][j] \times B_{KS}^{j} \times \boldsymbol{s_1}[i]\right) \ mod\ q \tag{4.5}$$

Another operation that can be applied is modulus switch, which converts a LWE ciphertext that is defined on a modulus q_1 to another LWE ciphertext with a different modulus q_2. This operation is simple in practice by scaling and rounding the LWE ciphertext to the nearest integer, as in Equation 4.6. A formal definition can be found in [116].

$$LWE_{\boldsymbol{s}}^{\frac{q_2}{t}}(m) = \left\lfloor \frac{q_2}{q_1} \times LWE_{\boldsymbol{s}}^{\frac{q_1}{t}}(m) \right\rceil = \left[\left\lfloor \frac{q_2}{q_1} \times \boldsymbol{a} \right\rceil, \left\lfloor \frac{q_2}{q_1} \times b \right\rceil \right] \tag{4.6}$$

4.2.3 Ring Learning with Errors

LWE suffers from low ciphertext efficiency, as the ciphertext utilization is $\frac{1}{n+1}$. So, most of the FHE schemes exploits a polynomial ring construction. A ring is defined as a nonempty set R, equipped with two operations, denoted as $+$ and $\times$, that satisfies the following conditions:

1) if $a, b \in R$, then $a + b \in R$ and $a \times b \in R$.

2) $a + (b + c) = (a + b) + c$, and $a \times (b \times c) = (a \times b) \times c$.

3) $a + b = b + a$.

4) There exists an element 0_R such that $a + 0_R = 0_R + a = a$ for every $a \in R$.

5) For each $a \in R$ the equation $a + x = 0_R$ has a solution in R.

6) $a \times (b + c) = a \times b + a \times c$ and $(a + b) \times c = a \times c + b \times c$.

The polynomial ring that is adopted in FHE is the set defined by $R_Q = Z_Q[x]/(x^N + 1)$ with N being a power of 2, which is the set of all polynomials, with coefficients in integer ring $\mathbb{Z}_Q$, modulo the cyclotomic polynomial $x^N + 1$. And similar to LWE, the RLWE problem states that it is hard to extract the secret polynomial $\boldsymbol{z}$ from samples of the form $[\boldsymbol{a}, \boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{z} + \boldsymbol{e}]$, where $\boldsymbol{a}$ is sampled uniformly from the set R_Q, and $\mathbf{e}$ is a noise polynomial sampled from an error distribution χ [115]. The "·" and "+" here represent polynomial multiplication and addition.

Similarly, in practice, given a plaintext modulo T and a ciphertext modulo Q, an RLWE encryption of a plaintext $\boldsymbol{m}\ mod\ T$ is defined as in Equation 4.7, with the vector $\boldsymbol{a}$ and noise vector $\boldsymbol{e}$ as described above. As long as $|\boldsymbol{e}|_\infty < \frac{Q}{2T}$, the plaintext can be successfully recovered by $\boldsymbol{m} = \left\lfloor (\boldsymbol{b} - \boldsymbol{a} \cdot \boldsymbol{z}) \times \frac{T}{Q} \right\rceil$, which rounds off the noise. In some contexts, the scale Q/T is omitted for clarity.

$$RLWE_{\boldsymbol{z}}^{\frac{Q}{T}}(\boldsymbol{m}) = \left[\boldsymbol{a}, \boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{z} + \boldsymbol{e} + \boldsymbol{m} \times \frac{Q}{T}\right], \tag{4.7}$$

A similar key switch can also be applied to RLWE to convert an RLWE ciphertext encrypted under a secret key $\boldsymbol{z_1}$ into another RLWE ciphertext encrypted by a different secret key $\boldsymbol{z_2}$. Given a decomposition base B_{KS}, a RLWE key switch key is created by encrypting the secret key $\boldsymbol{z_1}$ into a vector of RLWE ciphertexts, as in Equation 4.8. And for a RLWE ciphertext $RLWE_{\boldsymbol{z_1}}(\boldsymbol{m}) = [\boldsymbol{a}, \boldsymbol{b}]$ encrypted with key $\boldsymbol{z_1}$, to switch to key $\boldsymbol{z_2}$, the new ciphertext is generated by Equation 4.9, with $\boldsymbol{a} = \sum_{j=0}^{\log_{B_{KS}}(Q)-1} \boldsymbol{a}[j]$. And the product of a polynomial $\boldsymbol{p}$ and RLWE ciphertext $RLWE_{\boldsymbol{z}}(\boldsymbol{m}) = [\boldsymbol{a}, \boldsymbol{b}]$ is defined as $\boldsymbol{p} \cdot RLWE_{\boldsymbol{z}}(\boldsymbol{m}) = [\boldsymbol{p} \cdot \boldsymbol{a}, \boldsymbol{p} \cdot \boldsymbol{b}]$. A formal definition of the process can be found in [50]-[52].

$$\left[RLWE_{\boldsymbol{z_2}}(1 \times \boldsymbol{z_1}), RLWE_{\boldsymbol{z_2}}(B_{KS} \times \boldsymbol{z_1}), \dots, RLWE_{\boldsymbol{z_2}}\left(B_{KS}^{\log_{B_{KS}}(Q)-1} \times \boldsymbol{z_1}\right)\right] \quad 4.8$$

$$RLWE_{\boldsymbol{z_2}}(\boldsymbol{m}) = [\boldsymbol{0}, \boldsymbol{b}] - \sum_{j=0}^{log_{B_{KS}}(Q)-1} \boldsymbol{a}[j] \cdot RLWE_{\boldsymbol{z_2}}\left(B_{KS}^{j} \times \boldsymbol{z_1}\right) mod\ Q, \quad 4.9$$

Finally, since RLWE is a special form of LWE, each coefficient of the polynomial $\boldsymbol{b}$ of an RLWE ciphertext $RLWE_{\boldsymbol{z}}(\boldsymbol{m}) = [\boldsymbol{a}, \boldsymbol{b}]$ can be converted into multiple separate LWE ciphertexts under the same secret key, with some transformation of polynomial $\boldsymbol{a}$. For example, in an RLWE ciphertext $RLWE_z(\boldsymbol{m}) = [\boldsymbol{a} = \sum \boldsymbol{a}[i]X^i, \boldsymbol{b} = \sum \boldsymbol{b}[i]X^i]$, the coefficient at index 0,

$$\boldsymbol{b}[0] = \boldsymbol{z}[0] \times \boldsymbol{a}[0] - \sum \boldsymbol{z}[i] \times \boldsymbol{a}[N-i] + \boldsymbol{e}[0] + \boldsymbol{m}[0], \quad 4.10$$

can be viewed as an LWE ciphertext $LWE_{\boldsymbol{z}}(\boldsymbol{m}[0]) = [\boldsymbol{a_{LWE}}, \boldsymbol{b}[0]]$ encrypted by secret key $\boldsymbol{z}$, where $\boldsymbol{a_{LWE}} = [\boldsymbol{a}[0], Q - \boldsymbol{a}[N-1], Q - \boldsymbol{a}[N-2], \dots Q - \boldsymbol{a}[1]]$.

4.2.4 Number Theory Transform

The polynomial multiplication of RLWE can be efficiently computed with Number Theory Transform (NTT), which is also advantageous compared to the matrix-vector multiplication of LWE. NTT is a generalization of the famous FFT algorithm, which reduces the complexity of polynomial multiplication from $O(N^2)$ to $O(Nlog(N))$.

For a polynomial of the above polynomial ring R_Q with coefficients $a(x) = [a_0, a_1, \dots a_{N-1}]$, the NTT representation of it, $A(x) = [A_0, A_1, \dots A_{N-1}]$, is defined as

$$A_i = \sum_{j}^{N-1} a_j \omega_N^{ij}\ mod\ Q. \quad 4.11$$

And the inverse NTT (INTT) operation is defined as

$$a_i = \frac{1}{n} \sum_{j}^{N-1} A_j \omega_N^{-ij}\ mod\ Q. \quad 4.12$$

Notice that the major difference from FFT is that the N-th primitive root of unity ω_N of the ring $\mathbb{Z}_Q$, which satisfies $\omega_N^N = 1 \; mod \; Q$ and $\omega_N^j \neq 1 \; mod \; Q \; \forall \; j \in (0, N)$, takes place of the complex root of unity $e^{-\frac{2\pi i}{N}}$ to form the twiddle factors.

However, direct application of this NTT process to a polynomial multiplication, for example,

$$\boldsymbol{c} = INTT\big(NTT(\boldsymbol{a}) \odot NTT(\boldsymbol{b})\big), \tag{4.13}$$

gives out polynomial modulo reduction with respect to $(x^N - 1)$, rather than $(x^N + 1)$, which means the polynomial product is not in the desired polynomial ring. In order to fix this, negacyclic/anticyclic convolution is adopted [112]. With $\psi_N = \sqrt{\omega_N}$ as the $2N$-th primitive root of unity of the ring $\mathbb{Z}_Q$, the polynomial multiplication is defined as

$$\boldsymbol{c} = \left[1, \psi_N^{-1}, \dots \psi_N^{-(N-1)}\right] \odot INTT\left(NTT(\widetilde{\boldsymbol{a}}) \odot NTT\left(\widetilde{\boldsymbol{b}}\right)\right), \tag{4.14}$$

where $\widetilde{\boldsymbol{a}} = \left[1, \psi_N^1, \dots \psi_N^{(N-1)}\right] \odot \boldsymbol{a}$ and $\widetilde{\boldsymbol{b}} = \left[1, \psi_N^1, \dots \psi_N^{(N-1)}\right] \odot \boldsymbol{b}$.

Further optimization can be achieved by merging the powers of ψ_N into the twiddle factors. A detailed description is beyond the scope of this work, the reader is referred to [112] for more information. In the remaining text, NTT/INTT is referred to the negacyclic version, unless stated otherwise.

4.2.5 Framework of Fully Homomorphic Encryption

The last piece of the puzzle for getting a taste of the FHE schemes is the general blueprint proposed in Craig Gentry's dissertation [113] in 2009. Besides Gentry's initial construction, all following FHE schemes comply with this blueprint. Thus, a good understanding of it is necessary.

Almost all present FHE schemes starts with a partially homomorphic encryption, such as LWE and RLWE. It is easy to find that LWE, for example, is homomorphic to addition. Let

$\left[\boldsymbol{a_1}, b_1 = \boldsymbol{s}\cdot\boldsymbol{a_1} + e_1 + m_1 \times \frac{q}{t}\right]$ and $\left[\boldsymbol{a_2}, b_2 = \boldsymbol{s}\cdot\boldsymbol{a_2} + e_2 + m_2 \times \frac{q}{t}\right]$ be two ciphertexts that encrypt two plaintext integers m_1 and m_2 with secret key vector $\boldsymbol{s}$. By adding the two ciphertexts, a new ciphertext that encrypts $m_1 + m_2$ can be achieved, as in Equation 4.15.

$$\begin{aligned}[\boldsymbol{a_3}, b_3] &= [\boldsymbol{a_1}, b_1] + [\boldsymbol{a_2}, b_2] \\ &= [\boldsymbol{a_1} + \boldsymbol{a_2}, b_1 + b_2] \\ &= [\boldsymbol{a_1} + \boldsymbol{a_2}, \boldsymbol{s}\cdot(\boldsymbol{a_1} + \boldsymbol{a_2}) + e_1 + e_2 + (m_1 + m_2) \times \frac{q}{t}]\ mod\ q.\end{aligned} \qquad 4.15$$

And as long as the new noise $e_1 + e_2$ is small enough, the sum can be successfully decrypted by $m_1 + m_2 = \left\lfloor (b_3 - \boldsymbol{a_3}\cdot\boldsymbol{s}) \times \frac{t}{q} \right\rfloor$. This homomorphism also applies to a plaintext or a clean ciphertext, by setting the second ciphertext as $[\mathbf{0}, m]$, where $\mathbf{0}$ stands for zero vector. But if the addition continues, eventually, the excessively growing noise will contaminate the plaintext, rendering a decryption failure. So, the operation should be stopped before the noise exceeds the limit. And from this perspective, LWE is a somewhat homomorphic encryption (SHE).

However, if one can "secretly" reset the noise level after each addition, then the operation can resume. In fact, it is shown by Gentry that, from a circuit point of view, an encryption scheme can achieve full homomorphism if it is homomorphic to its own decryption function plus a universal gate [113][114]. Although most SHE schemes do not naturally come with this nice property, a process called "bootstrap", which reset the noise in the ciphertext homomorphically, is introduced. An excellent physical analogy about the "bootstrap" process can be found in [114] that is highly recommend, the reader can refer to it for better understanding.

Thus, all present FHE schemes can be decomposed into two parts. One is the basic SHE that facilitates the encryption of plaintexts. And another is a bootstrap process that convert the SHE into FHE by homomorphically reset the noise after the computation.

4.3 Adapted FHEW Scheme

Ever since the advent of the FHE framework proposed by Gentry in 2009, there have been several generations of schemes. FHEW [59][116] is one of the third-generation schemes. Compared to second-generation schemes, like BGV [50] and BFV [51][52], FHEW focuses on efficient implementation of homomorphic Boolean logic operation and the corresponding bootstrap process. The performance of it is on the same order of magnitude compared to second-generation schemes due to the widely adopted SIMD-like construction [54] which amortizes the computation cost of second-generation schemes. But it is well accepted for the simplicity and flexibility of both its concept and implementation. And in practice, these two types are used in different scenarios.

Figure 4.3 gives an overview of the FHEW scheme, which is divided into two parts, local and remote, since FHE is intended to be used in such scenario. The local part covers the key generations, encryption of the input and decryption of the output. And the remote part is

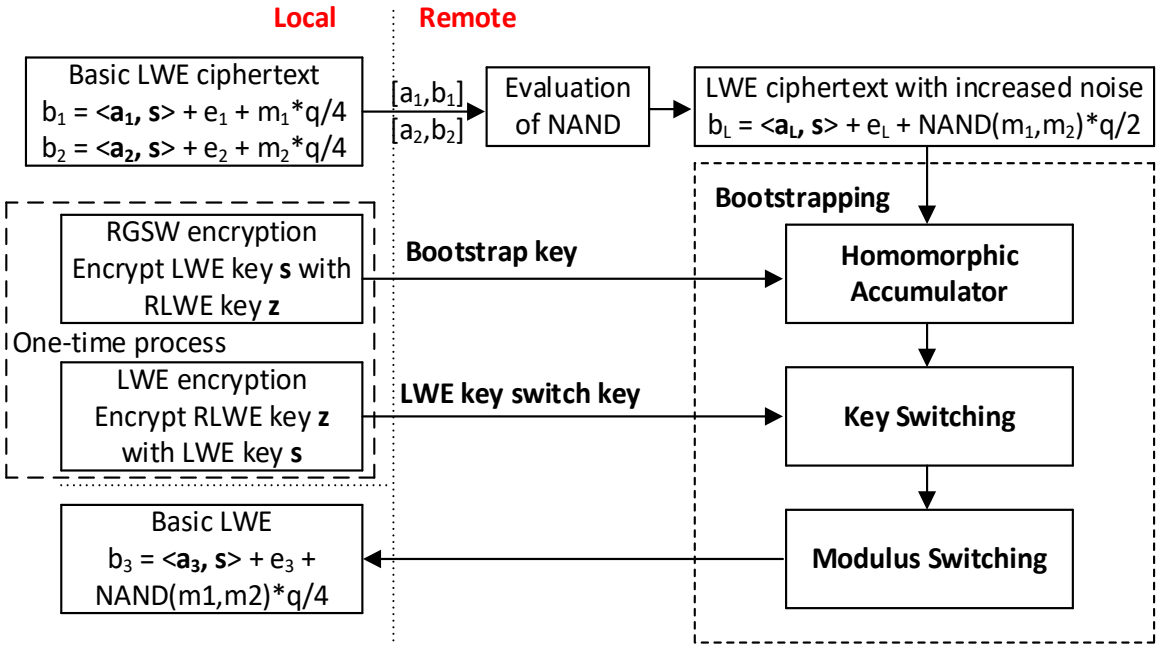

Figure 4.3 An Overview of the FHEW Scheme.

responsible for the evaluation of the Boolean logic and the bootstrap process. The task on the remote side is much more heavy lifting compared to what is on the local side, which will be clear in the following sections, so the remaining chapter focuses mainly on the acceleration of the bootstrap process on the server.

4.3.1 The Basic LWE/SHE of FHEW and the Evaluation of Boolean Logic

Since FHEW [59] deals with Boolean logic, it is natural to set the plaintext of the LWE encryption to $t = 2$. However, since LWE is only homomorphic to addition, Boolean logic cannot be directly applied to it. Instead, the sum of two bits is relied on to extract the Boolean logic result. Take NAND(m_1, m_2) for example, with m_1, m_2 being either 0 or 1, it can be extracted from the sum of $m_1 + m_2 + 2\ mod\ 4$. If the sum is 2 or 3, then NAND(m_1, m_2) = 0. Otherwise, if the sum is 0, then NAND(m_1, m_2) = 1. Therefore, the LWE of FHEW uses $t = 4$ as the plaintext modulo and limits the plaintext to be either $2'b00$ or $2'b01$.

Furthermore, the noise bound is set to $|e| < \frac{q}{4t} = \frac{q}{16}$, rather than $|e| < \frac{q}{2t} = \frac{q}{8}$, to leave some room for the addition operation. With that in mind, the above sum can be evaluated homomorphically to get an encrypted sum of the input, as in Equation 4.16. And this process is visualized in Figure 4.4. Each colored sector represents the possible range of a encode plaintext $(m_1, m_2\ or\ sum)$. Apparently, after the summation, the noise in the output ciphertext is doubled

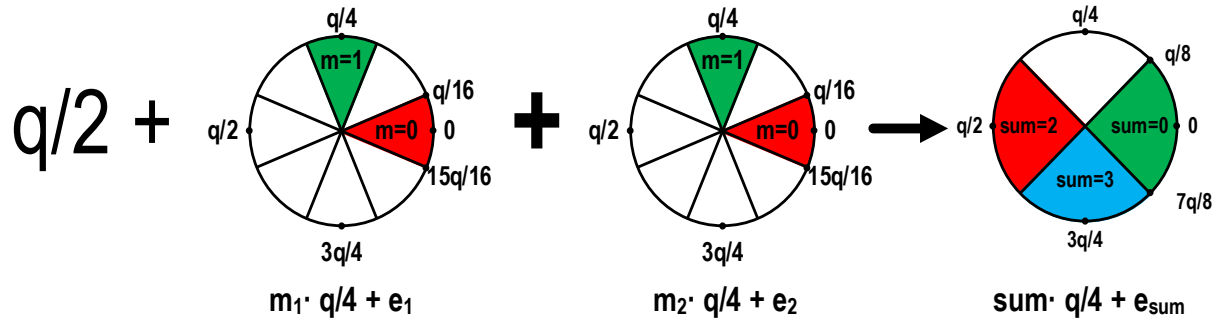

Figure 4.4 Visualization of the Evaluation of the NAND Operation with LWE.

due to the sum of the input noise, as indicated by the increase of the sector area in Figure 4.4. Thus, the size of the noise has exceeded the designated bound in a clean ciphertext. And the plaintext encoding is also different from the original.

$$\begin{aligned} LWE_{\boldsymbol{s}}^{\frac{q}{4}}(sum) &= LWE_{\boldsymbol{s}}^{\frac{q}{4}}(m_1) + LWE_{\boldsymbol{s}}^{\frac{q}{4}}(m_1) + \left[\mathbf{0}, \frac{q}{2}\right] \\ &= \left[\boldsymbol{a_1} + \boldsymbol{a_2}, b_1 + b_2 + \frac{q}{2}\right] \\ &= [\boldsymbol{a_1} + \boldsymbol{a_2}, (\boldsymbol{a_1} + \boldsymbol{a_2}) \cdot \boldsymbol{s} + e_1 + e_2 + (m_1 + m_2 + 2) \times \frac{q}{t}] \end{aligned} \quad 4.16$$

So, further addition cannot be applied to the result LWE ciphertext, unless there is a way to "secretly" apply the decryption equation $b - \boldsymbol{a} \cdot \boldsymbol{s}$, and a function $f(x)$ that maps the encoded sum into the original plaintext space, as illustrated in Figure 4.5. This is applied homomorphically with the bootstrap process which will be detailed in the following sections. Other Boolean logics can be done in the same fashion with a different mapping function.

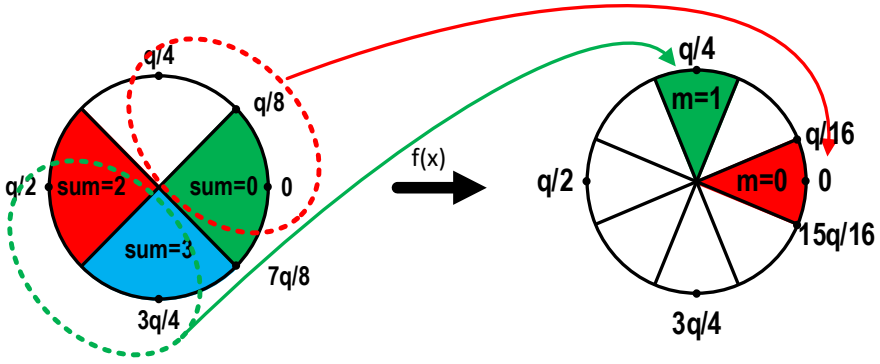

Figure 4.5 "Secret" Function That Maps a Dirty Ciphertext to a Clean Ciphertext.

4.3.2 Ring GSW Encryption

To evaluate the decryption equation $b - \boldsymbol{a} \cdot \boldsymbol{s}$ without leaking the plaintext, another encryption $E(x)$ that is homomorphic to addition and multiplication should be utilized, since the LWE itself reaches its noise limit. To be specific, encrypt $\boldsymbol{s}$ as a vector $E(\boldsymbol{s}) = [E(\boldsymbol{s}[0]), E(\boldsymbol{s}[1]), \dots E(\boldsymbol{s}[n-1])]$. By the definition of homomorphism, it is obvious that the

inner product is absorbed into the encryption, as shown in Equation 4.17. Thus, the decryption equation is secretly evaluated without leaking the encoded plaintext. Note that it is not necessary to encrypt $\boldsymbol{a}$ and b, since they are both public as a ciphertext. Only the secret key of the LWE needs to be kept secret.

$$\begin{aligned} b - \sum_{i}^{n-1} \boldsymbol{a}[i] \times E(\boldsymbol{s}[i]) &= b - \sum_{i}^{n-1} E(\boldsymbol{a}[i] \times \boldsymbol{s}[i]) \\ &= E\left(b - \sum_{i}^{n-1} \boldsymbol{a}[i] \times \boldsymbol{s}[i]\right) \\ &= E(b - \boldsymbol{a} \cdot \boldsymbol{s}). \end{aligned} \tag{4.17}$$

And by applying a transform that converts the ciphertext $E(b - \boldsymbol{a} \cdot \boldsymbol{s})$ to a LWE ciphertext that encrypts the same encoded plaintext, a bootstrapped LWE ciphertext is achieved and further Boolean operation on it can continue. Note that two encryptions are involved, one is the basic LWE, and the other is the encryption $E(x)$ that is used for bootstrapping.

Obviously, the RLWE encryption is a candidate for the encryption $E(x)$ due to the additive and multiplicative homomorphism, for example a polynomial $\boldsymbol{d}$ can be multiplied into a RLWE ciphertext homomorphically,

$$RLWE_{\boldsymbol{z}}^{\frac{Q}{T}}(\boldsymbol{m}) \cdot \boldsymbol{d} = [\boldsymbol{a} \cdot \boldsymbol{d}, \boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{d} \cdot \boldsymbol{z} + \boldsymbol{e} \cdot \boldsymbol{d} + \boldsymbol{m} \cdot \boldsymbol{d}] = RLWE_{\boldsymbol{z}}^{\frac{Q}{T}}(\boldsymbol{m} \cdot \boldsymbol{d}). \tag{4.18}$$

However, multiplication expands the noise in an RLWE ciphertext rapidly, as the noise polynomial $\boldsymbol{e}$ is directly multiplied by $\boldsymbol{d}$ whose coefficients modulo Q. To reduce the noise growth, instead of encrypting the polynomial $\boldsymbol{m}$, $B_G^j \times \boldsymbol{m}, \forall j \in [0, \log_{B_G}(Q) - 1]$ is encrypted as a vector of RLWE ciphertext with a predefined decomposition base B_G,

$$RLWE'_{\boldsymbol{z}}(\boldsymbol{m}) = \left[RLWE_{\boldsymbol{z}}^{\frac{Q}{T}}(\boldsymbol{m}), \dots, RLWE_{\boldsymbol{z}}^{\frac{Q}{T}}\left(B_G^{log_{B_G}(Q)-1} \times \boldsymbol{m}\right)\right]. \tag{4.19}$$

Therefore, the above RLWE multiplication turns into an inner product of two vectors, denoted by $\boxdot$,

$$\begin{aligned} \boldsymbol{d} \boxdot RLWE'_{z}(\boldsymbol{m}) &= \sum_{j=0}^{\log_{B_G}(Q)-1} RLWE_{\boldsymbol{z}}^{\frac{Q}{T}}\left(B_G^j \times \boldsymbol{m}\right) \cdot \boldsymbol{d}'[i] \\ &= RLWE_{\boldsymbol{z}}^{\frac{Q}{T}}\left(\sum_{j=0}^{\log_{B_G}(Q)-1} B_G^j \times \boldsymbol{d}'[i] \cdot \boldsymbol{m}\right) \\ &= RLWE_{\boldsymbol{z}}^{\frac{Q}{T}}(\boldsymbol{m} \cdot \boldsymbol{d}), \end{aligned} \tag{4.20}$$

where the polynomials $\boldsymbol{d}'[i]$ are decomposition of polynomial $\boldsymbol{d}$ satisfying $\boldsymbol{d} = \sum_{i=0}^{\log_{B_G}(Q)} \boldsymbol{d}'[i]$. This breaks the noise multiplication into an accumulation of smaller products, which reduces the growth rate of the noise.

However, this noise growth control is far from enough, since there is a long accumulation in Equation 4.17. A further reduction can be achieved with Ring-GSW (RGSW) encryption. RGSW [116] is adapted from GSW encryption [58], which is named after the initials of the authors. An RGSW ciphertext is composed of two RLWE' ciphertexts as in Equation 4.21 (in some literatures, the two vectors are concatenated as a one-dimensional vector).

$$RGSW_{\boldsymbol{z}}(\boldsymbol{m}) = [RLWE'_{\boldsymbol{z}}(-\boldsymbol{z} \cdot \boldsymbol{m}), RLWE'_{\boldsymbol{z}}(\boldsymbol{m})]. \tag{4.21}$$

Given an RLWE ciphertext $RLWE_{\boldsymbol{z}}(\boldsymbol{m_1}) = [\boldsymbol{a}, \boldsymbol{b}]$ and an RGSW ciphertext $RGSW_{\boldsymbol{z}}(\boldsymbol{m_2})$, an operation that output an RLWE encryption of $\boldsymbol{m_1} \cdot \boldsymbol{m_2}$ is defined as

$$\begin{aligned} RLWE_{\boldsymbol{z}}(\boldsymbol{m_1}) \otimes RGSW_{\boldsymbol{z}}(\boldsymbol{m_2}) &= \boldsymbol{a} \boxdot RLWE'_{\boldsymbol{z}}(-\boldsymbol{z} \cdot \boldsymbol{m_2}) + \boldsymbol{b} \boxdot RLWE'_{\boldsymbol{z}}(\boldsymbol{m_2}) \\ &= RLWE_{\boldsymbol{z}}\left((\boldsymbol{b} - \boldsymbol{a} \cdot \boldsymbol{z}) \cdot \boldsymbol{m_2}\right) \\ &= RLWE_{\boldsymbol{z}}\left((\boldsymbol{m_1} + \boldsymbol{e_1}) \cdot \boldsymbol{m_2}\right) \\ &= RLWE_{\boldsymbol{z}}(\boldsymbol{m_1} \cdot \boldsymbol{m_2} + \boldsymbol{e_1} \cdot \boldsymbol{m_2}). \end{aligned} \tag{4.22}$$

Therefore, if $\boldsymbol{m_2}$ only has one nonzero coefficient, a linear noise growth can be achieved. The RGSW encryption is that target encryption scheme that is used to encrypt the secret key of LWE in Equation 4.17. In practice, a RGSW ciphertext is calculated as

$$RGSW_{\boldsymbol{z}}(\boldsymbol{m}) = [RLWE_{\boldsymbol{z}}(\boldsymbol{0}), \dots, RLWE_{\boldsymbol{z}}(\boldsymbol{0})] + \boldsymbol{m} \cdot \boldsymbol{G}, \tag{4.23}$$

where $\boldsymbol{G} = \boldsymbol{I} \odot [1, B_G, B_G^2, \dots, B_G^{\log_{B_G}(Q)-1}]$ is called the "gadget matrix".

4.3.3 Bootstrap in FHEW

The bootstrap process is composed of three steps, a homomorphic accumulator that calculates the inner product of Equation 4.17 and generates a RLWE ciphertext that is encrypted by the RLWE secret key $\boldsymbol{z}$, an LWE key switch step that switches LWE ciphertext with secret key $\boldsymbol{z}$ into an LWE ciphertext that is encrypted by the LWE secret key $\boldsymbol{s}$, and a modulus switch step that switches the ciphertext modulo from RLWE modulo Q to LWE modulo q [116],

With $\otimes$ operation of a RLWE and RGSW ciphertext, the homomorphic accumulator is defined on an RLWE ciphertext and a bootstrap key which is a LUT of RGSW that encrypts the secret key $\boldsymbol{s}$ of the basic LWE. First, for each element $\boldsymbol{s}[i]$ of the secret key vector $\boldsymbol{s}$ of LWE, a vector of polynomial that contains powers of X is used to encode the $\boldsymbol{s}[i]$ into the exponent of X, as

$$\left[1, X^{\boldsymbol{s}[i]}, X^{2\times\boldsymbol{s}[i]}, \dots, X^{(q-1)\times\boldsymbol{s}[i]}\right]. \tag{4.24}$$

Then this vector is encrypted with RGSW encryption into a vector of RGSW ciphertexts,

$$\left[RGSW_{\boldsymbol{z}}(1), RGSW_{\boldsymbol{z}}(X^{\boldsymbol{s}[i]}), RGSW_{\boldsymbol{z}}(X^{2\times\boldsymbol{s}[i]}), \dots, RGSW_{\boldsymbol{z}}(X^{(q-1)\times\boldsymbol{s}[i]})\right]. \tag{4.25}$$

Thus, a two-dimensional LUT is built, containing $n \times q$ RGSW ciphertexts.

To homomorphically evaluate $b - \boldsymbol{a} \cdot \boldsymbol{s}$ of the LWE decryption, b is encoded into a clean RLWE ciphertext $RLWE_{\boldsymbol{z}}(X^b)$ which is called a homomorphic accumulator. And for each i, the

RGSW ciphertexts from the LUT indexed by $-\boldsymbol{a}[i] = q - \boldsymbol{a}[i]$ are multiplied to the accumulator by the $\otimes$ operation defined in Equation 4.22, which accumulates the inner product and encodes the result onto the exponent of a polynomial as in Equation 4.26.

$$RLWE_{\boldsymbol{z}}(X^{b}) \otimes RGSW_{\boldsymbol{z}}\left(X^{(q-\boldsymbol{a}[0])\times \boldsymbol{s}[0]}\right) \dots \otimes RGSW_{\boldsymbol{z}}\left(X^{(q-\boldsymbol{a}[n-1])\times \boldsymbol{s}[n-1]}\right) = RLWE_{\boldsymbol{z}}(X^{b-\sum \boldsymbol{a}[i]\times \boldsymbol{s}[i]}). \quad 4.26$$

A similar time-space tradeoff also applies to the accumulator, as mentioned in section 4.2.2. Given a decomposition base B_r to decompose each element of vector $\boldsymbol{a}$, the LUT is increased by one dimension. For each element $\boldsymbol{s}[i]$ of the secret key vector $\boldsymbol{s}$ of LWE, a 2-D array of RGSW ciphertexts of the form $RGSW_{\boldsymbol{z}}(X^{v\times B_r^j \times s[i]})$ for $v \in [0, B_r - 1]$ and $j \in [0, \log_{B_r}(q) - 1]$. To accumulate, each element $\boldsymbol{a}[i]$ of $\boldsymbol{a}$ is decomposed by B_r with $\boldsymbol{a}[i] = \sum_{j=0}^{\log_{B_r}(q)-1} \boldsymbol{a}[i][j] \times B_r^j$, and the decomposition is used to index the LUT to facilitate the homomorphic accumulation, as in Equation 4.27. As a result, the size of the 3-D LUT is $B_r \times log_{B_r}(q) \times n$, and with a proper B_r, it can be much smaller than the size of a 2-D LUT, $n \times q$, at the cost of more accumulation steps, facilitating a time-space trade off.

$$RLWE_{\boldsymbol{z}}(X^{b}) \otimes RGSW_{\boldsymbol{z}}\left(X^{(q-\boldsymbol{a}[i][j])\times B_r^j \times \boldsymbol{s}[i]}\right) \dots = RLWE_{\boldsymbol{z}}(X^{b-\sum \boldsymbol{a}[i][j]\times B_r^j \times \boldsymbol{s}[i]}). \quad 4.27$$

So far, the inner product of the LWE decryption is secretly evaluated and encoded into the exponent of a polynomial. However, to facilitate the mapping function $f(x)$ in Figure 4.5, i.e., transformation from RLWE to LWE, the accumulator is initialized with a precomputation of the mapping based on which Boolean operation is performed. The accumulator of NAND, for example, is initialized as a clean RLWE ciphertext of a polynomial $\sum_{i<\frac{q}{2}} g(b-i)X^{i\times p}$, where $p = 2N/q$, and the function $g(x)$ is defined in Equation 4.28. A more general explanation of the initialization can be found in [116].

$$g(x) = \begin{cases} \frac{Q}{8}, x \in (-\frac{q}{8}, \frac{3q}{8}] \\ -\frac{Q}{8}, x \in (\frac{3q}{8}, \frac{7q}{8}] \end{cases} \quad 4.28$$

After the accumulation, the inner product in Equation 4.17 is secretly evaluated and encrypted as the coefficient of index 0 in a RLWE ciphertext with secret key $\boldsymbol{z}$, which needs to be converted back to a LWE ciphertext with secret key $\boldsymbol{s}$. Before that, the RLWE is converted into an LWE ciphertext under key $\boldsymbol{z}$ to extract the coefficient at index 0 as described in Section 4.2.3.

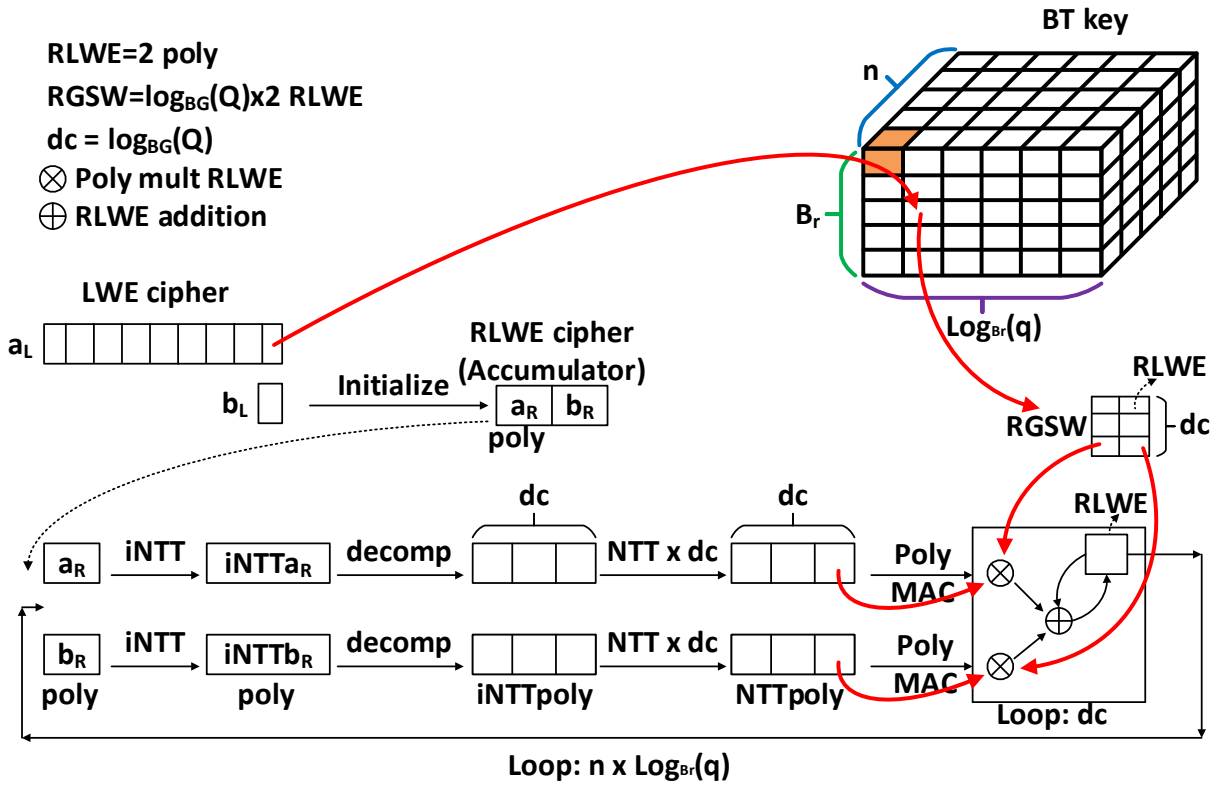

Figure 4.6 Data Flow of Bootstrap in FHEW (Last Two Steps not Included).

Finally, the LWE key switch process followed by the LWE modulus switch operation is applied to get the bootstrap output, an LWE ciphertext, complying the original basic LWE format, that encrypts the Boolean logic output of the two inputs. One caveat to be noted is that when

creating the LWE key switch key, the original LWE key $\boldsymbol{s}$, which is with modulus q, should be mapped into modulus Q first.

Figure 4.6 illustrates the bootstrap process, without the last two steps. The main computation takes place in the center loop that is composed of a INTT followed by dc NTT and dc polynomial multiplication and accumulation (MAC), where $dc = \log_{B_G}(Q)$. The loop in Figure 4.6 comprise 98% of the processing time in one bootstrap [116], so the proposed accelerator architecture in Section 4.6 focuses on this part. Other functions are offloaded to software.

4.4 Enhanced Features for FHEW

Proposed in another third-generation scheme, TFHE [60], some additional features are adopted into the FHEW scheme for flexible application.

4.4.1 Homomorphic MUX Function

The $\otimes$ operation between RLWE and RGSW ciphertext defined in Section 4.3.2 can be used to construct a homomorphic MUX function [60]. Let $\boldsymbol{m} = [sel, 0,0, \dots, 0]$ be the selection signal of the MUX gate with sel equals to either 0 or 1, and $RLWE_{\boldsymbol{z}}(\boldsymbol{p_0})$ and $RLWE_{\boldsymbol{z}}(\boldsymbol{p_1})$ be two input RLWE ciphertexts for the MUX.

The MUX function $CMUX\big(RGSW_{\boldsymbol{z}}(\boldsymbol{m}), RLWE_{\boldsymbol{z}}(\boldsymbol{p_0}), RLWE_{\boldsymbol{z}}(\boldsymbol{p_1})\big)$ is defined as in Equation 4.29, and illustrated in Figure 4.7. Obviously, the output of the function is a RLWE ciphertext that follows the encrypted selection signal.

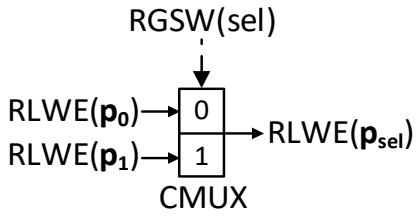

Figure 4.7 CMUX Function.

$$RGSW_z(\boldsymbol{m})\otimes\big(RLWE_z(\boldsymbol{p_1})-RLWE_z(\boldsymbol{p_0})\big)+RLWE_z(\boldsymbol{p_0})=RLWE_z(\boldsymbol{m}\cdot(\boldsymbol{p_1}-\boldsymbol{p_0})+\boldsymbol{p_0})$$
$$=RLWE_z(\boldsymbol{p}_{sel}) \quad (4.29)$$

4.4.2 Blind Rotate

Following the definition of the CMUX function, another function that can homomorphically rotate the plaintext polynomial is formulated, which multiplies polynomial with a power of X. A simplified version is show in Equation 4.30 and illustrated in Figure 4.8. Let $\boldsymbol{m}=[sel,0,0,\dots,0]$ be the selection signal of the MUX gate with sel equals to either 0 or 1, and $RLWE_z(\boldsymbol{p})$ be the RLWE ciphertexts for the $BlindRotate$. j denotes the number of steps for the rotation.

$$\begin{aligned} BlindRotate &= CMUX\left(RGSW_z(\boldsymbol{m}),RLWE_z(\boldsymbol{p}),X^{-j}\cdot RLWE_z(\boldsymbol{p})\right)\\ &= CMUX\left(RGSW_z(\boldsymbol{m}),RLWE_z(\boldsymbol{p}),RLWE_z\left(X^{-j}\cdot\boldsymbol{p}\right)\right)\\ &= RLWE_z\left(\boldsymbol{m}\cdot\left(X^{-j}\cdot\boldsymbol{p}-\boldsymbol{p}\right)+\boldsymbol{p}\right). \end{aligned} \quad (4.30)$$

Thus, the output RLWE contains a plaintext that is either rotated or not based on the selection. A comprehensive definition can be found in [60].

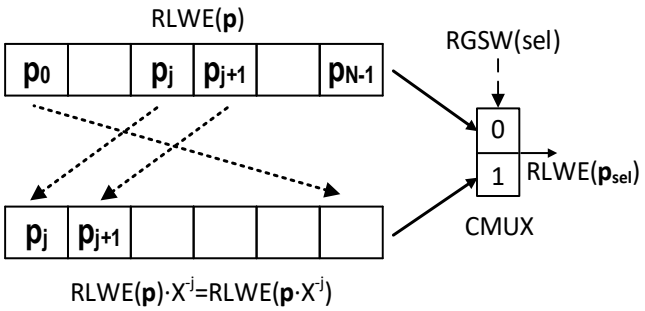

Figure 4.8 Blind Rotate Function

4.4.3 Homomorphic LUT and Plaintext Packing

Intuitively, the CMUX gate can be concatenated into a CMUX tree to evaluate an arbitrary binary function homomorphically as shown in Figure 4.10. The function is precomputed and

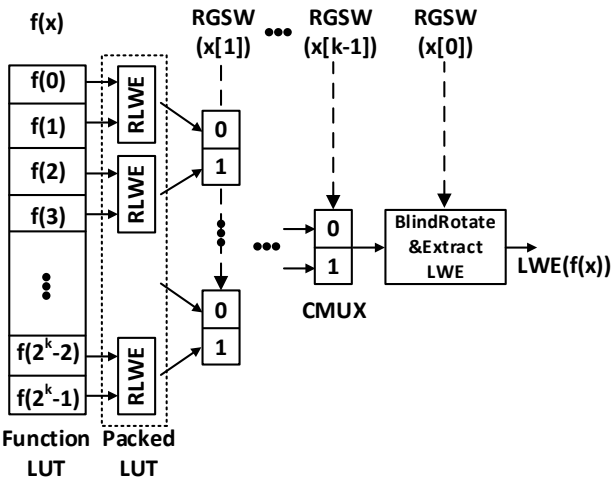

Figure 4.9 Vertical Packing Scheme of the LUT and Modified CMUX Tree.

encrypted into an LUT of RLWE ciphertexts, and after traversing the CMUX tree indexed by RGSW encryptions of each bit of an input x, an RLWE ciphertext that encrypts the corresponding $f(x)$ is output.

However, the size of the LUT is large if only one function value is encrypted in each RLWE ciphertext, resulting in 2^k RLWE ciphertexts. Also, the amount of CMUX is $2^k - 1$. The size can be reduced by a factor of the length of the polynomial in a RLWE ciphertext if several function values are packed into a RLWE ciphertext. For example, if each coefficient of a plaintext polynomial $\boldsymbol{m}$ is taken as a plaintext slots, then a contiguous block of function values can be packed into one polynomial, such as $\boldsymbol{m} = [f(0), f(1), \dots, f(N-1)]$, where N is the length of the polynomial. Thus, an RLWE ciphertext can encrypt at most N function values, which reduces the size of the LUT and the amount of CMUX by a factor of N. Figure 4.9 details this packing scheme, where each RLWE encrypts two function values. In the example, the MSBs of the input x are first used to find the desired RLWE ciphertext, and then the target slot is rotated to the position 0 by the LSB of x with $BlindRotate$. At last, the desired slot is extracted from the RLWE into a LWE ciphertext with a process the same as described in Section 4.2.3.

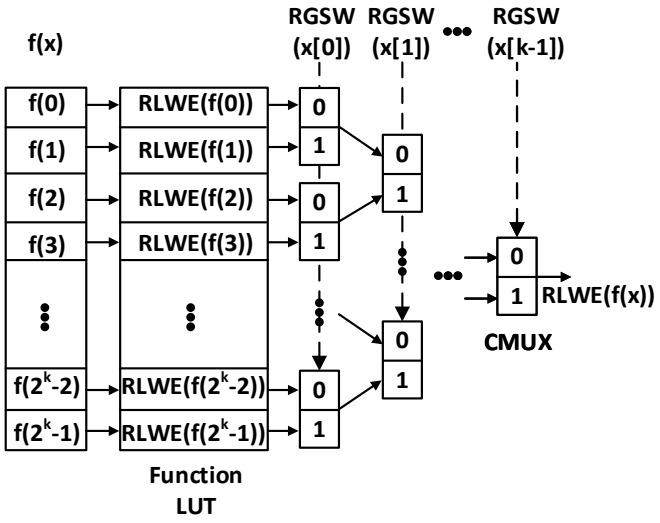

Figure 4.10 LUT and CMUX Tree for an Arbitrary Binary Function $f(x)$.

There are other types of packing schemes, the reader is referred to [60] for further information.

4.5 Private Set Intersection Protocol Based on Adapted FHEW

PSI allows two parties to exchange the intersection of their private sets without leaking any excessive information other than the intersection set. The problem has been explored extensively before [105]-[107], and in recent years, PSI protocols based on second generation FHE [110][117] have been proposed. In this section, a PSI protocol based on the adapted FHEW scheme is proposed, with a framework adopted from [110].

4.5.1 High Level Construction

For two parties, Receiver and Sender, to find the intersection of their private sets $\{x_i\}$ and $\{y_j\}$ containing some 32-bit integers, as show in Figure 4.11, each element of the Receiver set is compared with the elements of the Sender set, looking for a match.

However, in an unencrypted scenario, one of the parties needs to reveal all its content to the other party, which is undesirable. So, in [110], the comparison is fulfilled by a homomorphic

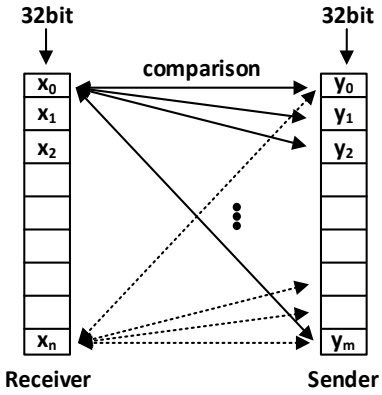

Figure 4.11 General Concept to Find the Intersection of Two Parties.

product of the difference between elements in the two sets. For example, Receiver encrypts all elements in $\{x_i\}$ with RLWE encryption and send to Sender. While Sender evaluates the product of difference homomorphically for each element in $\{y_j\}$, as in Equation 4.31, and send back the result to Receiver.

$$\prod_j \left([\mathbf{0}, y_j] - RLWE_{\mathbf{z}}(x_i) \right) = RLWE_{\mathbf{z}}\left(\prod_j (y_j - x_i) \right) \tag{4.31}$$

Obviously, after Receiver decrypts the result, the product evaluates to 0 if x_i finds a match in Sender's set $\{y_j\}$. Extra caution needs to be taken to prevent the product from leaking y_j if the product evaluates to a non-zero number. But it is beyond the scope of this work, [110] is referred to for a more comprehensive description.

In this work, the comparison is facilitated with the homomorphic LUT described in Section 4.4.3. As shown in Figure 4.12, on the Sender side, an LUT is precomputed based on the content of the Sender set $\{y_j\}$, with $LUT[y_j] = 1$ otherwise it is set to 0. While on the receiver side, each element x_i is decomposed into its bit representation and encrypted with a vector of RGSW ciphertext, $[RGSW_{\mathbf{z}}(x_i[0]), RGSW_{\mathbf{z}}(x_i[1]), \dots, RGSW_{\mathbf{z}}(x_i[31])]$, and send to the Sender. Then, the RGSW encrypted x_i are pass into the CMUX tree to index the LUT on the Sender side, and

the result is sent back to Receiver. Obviously, after decryption, 1 indicates that the x_i is in the intersection, otherwise, it is not.

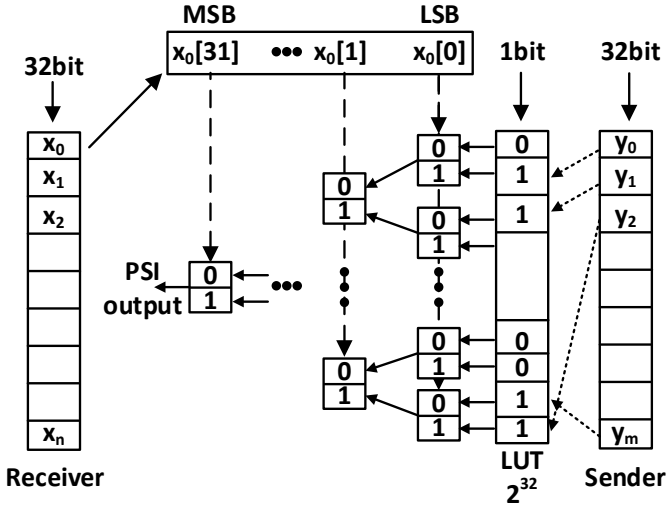

Figure 4.12 Homomorphic LUT Based PSI

However, this naïve construction is very inefficient in both computation and communication traffic. First, $2^{32} - 1$ CMUXs are evaluated for each element in the Receiver set. Second, 32 RGSW ciphertexts have to be transferred for each element in the Receiver set, resulting in a low ciphertext utilization. Several optimizations can be adopted to mitigate these problems and render practical application of the protocol.

4.5.2 RLWE Substitution and RLWE Expansion

Before tackling the problems, some additional preliminaries are discussed. First is RLWE substitution[111], which transforms a RLWE ciphertext $RLWE_{\boldsymbol{z}}\left(\sum \boldsymbol{m}[i]X^i\right)$ into $RLWE_{\boldsymbol{z}}\left(\sum \boldsymbol{m}[i]\left(X^i\right)^k\right)$, for an odd integer k. A substituted secret key $\boldsymbol{z}(X^k) = \boldsymbol{z}[i]\left(X^i\right)^k$ and an RLWE key switch key from $\boldsymbol{z}(X^k)$ to $\boldsymbol{z}$ are precomputed. An RLWE ciphertext is first substituted to get $RLWE_{\boldsymbol{z}(X^k)}\left(\boldsymbol{m}(X^k)\right) = [\boldsymbol{a}(X^k), \boldsymbol{b}(X^k)]$, and then key-switched to $RLWE_{\boldsymbol{z}}(\boldsymbol{m}(X^k))$. A formal definition can be found in [111].

The RLWE substitution is used extensively in the RLWE expansion [111] operation which expands a RLWE ciphertext from $RLWE_{\boldsymbol{z}}(\sum \boldsymbol{m}[i]X^i)$ into a vector $[RLWE_{\boldsymbol{z}}(\boldsymbol{m}[0]), RLWE_{\boldsymbol{z}}(\boldsymbol{m}[1]), \dots, RLWE_{\boldsymbol{z}}(\boldsymbol{m}[N-1])]$. To find how RLWE substitution fulfills the expansion, $k = N + 1$ is shown as an example. For $k = N + 1$, $\left(X^i\right)^k = (-1)^i X^i$. Thus, the addition of the substituted ciphertext to the original one extracts the even index coefficients of the message $\boldsymbol{m}$, as in Equation 4.32. And the subtraction extracts the odd index coefficients, as in Equation 4.33.

$$\begin{aligned} RLWE_{\boldsymbol{z}}(\sum \boldsymbol{m}[i]X^i) + RLWE_{\boldsymbol{z}}\left(\sum \boldsymbol{m}[i]\left(X^i\right)^k\right) &= RLWE_{\boldsymbol{z}}(\sum \boldsymbol{m}[i]X^i + \sum \boldsymbol{m}[i](-1)^i X^i) \\ &= RLWE_{\boldsymbol{z}}(\sum 2 \times \boldsymbol{m}[2i]X^{2i}), \end{aligned} \tag{4.32}$$

$$\begin{aligned} RLWE_{\boldsymbol{z}}(\sum \boldsymbol{m}[i]X^i) - RLWE_{\boldsymbol{z}}\left(\sum \boldsymbol{m}[i]\left(X^i\right)^k\right) &= RLWE_{\boldsymbol{z}}(\sum \boldsymbol{m}[i]X^i - \sum \boldsymbol{m}[i](-1)^i X^i) \\ &= RLWE_{\boldsymbol{z}}(\sum 2 \times \boldsymbol{m}[2i+1]X^{2i+1}). \end{aligned} \tag{4.33}$$

Therefore, by recursively substituting with $k = \frac{n}{2^s} + 1$, for $s \in [0, \log_2 N - 1]$, each coefficient $\boldsymbol{m}[i]$ of the message is extracted into a separate RLWE ciphertext $RLWE_{\boldsymbol{z}}(N \times \boldsymbol{m}[i])$. The scale can be offset by pre-scaling the message with the multiplicative inverse of the N in $\mathbb{Z}_Q$.

The data flow of RLWE substitution is shown in Figure 4.13, with the key switch highlighted in the dotted red box. An RLWE ciphertext in the NTT domain is first transformed

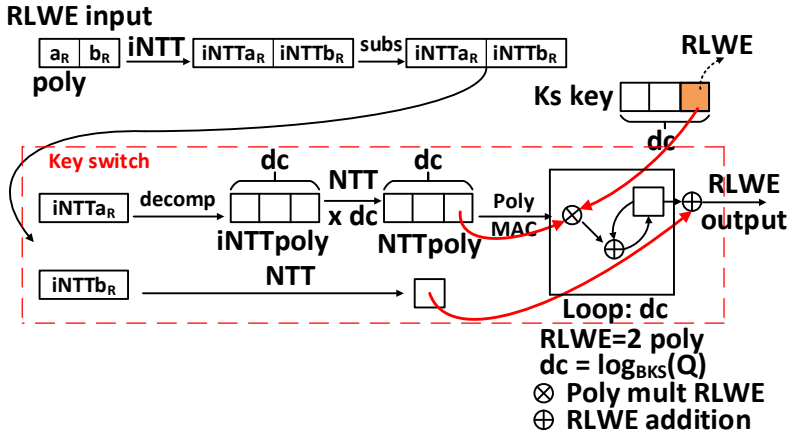

Figure 4.13 Data Flow of the RLWE Substitution Subroutine (RLWE key Switch Included).

into INTT form and substituted. Then it is decomposed with base B_{KS} and key-switched to the original secret key to get a substituted RLWE ciphertext. After that, the output ciphertext is postprocessed for RLWE expansion. Based on our experiment, 97% of the processing time of RLWE expansion is dedicated to substitution and key switch functions, so in our implementation, these two functions are offloaded to a FPGA. Note that the key switch data flow is very similar to the bootstrap data flow. Therefore, the proposed architecture merges both data flows, which will be detailed in Section 4.6.

4.5.3 Optimizations for the Proposed PSI

It is obvious that, in the proposed PSI in Section 4.5.1, both the computation and communication cost depend directly on the bit width of the elements in the set. So, the first optimization is to reduce the size of the element with permutation-based hashing [118]. In permutation-based hashing, to insert a 32-bit element x_i from the Receiver set into 2^k bins, the number is divided into $x_{iH}||x_{iL}$, with x_{iL} consisting of k bits. The position of the element is calculated as in Equation 4.34, where $H(x)$ is the hash function. In other words, the position of an element also stores some information about the element. And instead of inserting x_i into the hash table, only x_{iH} is inserted, which reduces the bit width to $32 - k$.

$$pos(x_i) = H(x_{iH})\ XOR\ x_{iL}, \tag{4.34}$$

The comparison in the homomorphic LUT still holds with permutation-based hashing. Assuming that x_{iH} from Receiver and y_{jH} from Sender are in the same bin after hashing and $x_{iH} = y_{jH}$, from Equation 4.34, it is apparent that

$$pos(x_i) = H(x_{iH})\ XOR\ x_{iL} = H(y_{jH})\ XOR\ x_{iL} = pos(y_j) = H(y_{jH})\ XOR\ y_{jL}. \tag{4.35}$$

Therefore, $x_{iL} = y_{jL}$, resulting in $x = y$. Thus, the correctness of the LUT based PSI holds with permutation-based hashing, reducing the transferred RGSW by k and amount of CMUX by a factor of 2^k.

The second optimization achieves a further reduction of computation by exploiting the vertical packing described in Section 4.4.3. With vertical packing, at most N LUT elements can be packed into one RLWE ciphertext, which shrinks the amount of CMUXs by roughly a factor of N. So, with $k = 14$, $N = 2048$, for example, the amount of CMUXs to compare each element in the Receiver set is reduced from $2^{32} - 1$ to $2^7 - 1 + 11$, by a factor of 2^{25}.

The last optimization aims at decreasing the communication payload. Instead of transferring an RGSW ciphertext, containing $\log_{B_G}(Q) \times 2$ RLWE ciphertexts, for each bit of an element in the Receiver set, it is observed that the first column of an RGSW ciphertext, as in Equation 4.21, can be calculated from the second column, which is detailed in Equation 4.36.

$$RLWE_{\boldsymbol{z}}\left(B_G^j \times (-\boldsymbol{z} \cdot \boldsymbol{m})\right) = RLWE_{\boldsymbol{z}}\left(B_G^j \times \boldsymbol{m}\right) \otimes RGSW_{\boldsymbol{z}}(-\boldsymbol{z}) \quad 4.36$$

Thus, only the second column of it needs to be transferred, together with a RGSW encryption of the secret key $-\boldsymbol{z}$, which is shared for all the transfers [111].

However, the ciphertext utilization is still very low, because each RLWE only works for one element in the Receiver set. So, N elements, for example, $[x_0, x_1, \dots, x_{N-1}]$, from the Receiver set are packed into a 2-D array of RLWE ciphertexts. Each element of the array is formed as $RLWE_{\boldsymbol{z}}(\sum_i x_i[k] \times B_G^j \times X^i)$, and is index by $j \in \left[0, \log_{B_G}(Q) - 1\right]$, $k \in [0,17]$ (assuming applying permutation-based hashing, the bit width is 18). Upon receiving the array, the Sender unpacks it, with the RLWE expansion described in Section 4.5.2, into arrays of RLWEs for each element x_i, of the form $RLWE_{\boldsymbol{z}}(x_i[k] \times B_G^j)$. At last, the RLWEs are converted into RGSWs with Equation 4.36, and passed into the LUT to complete the PSI.

Together, compared to transferring complete RGSWs, the communication overhead is reduced from 18 RGSWs (216 RLWEs) per element to $18 \times \log_{B_G}(Q)$ RLWEs per N elements, amounting to 4096x reduction if $\log_{B_G}(Q) = 6$, $N = 2048$, at the cost of increased computation on the Sender side for unpacking and reconstructing the RGSW.

Figure 4.14 gives an example of the data flow for the proposed homomorphic LUT based PSI. It assumes that after the permutation-based hashing, the data bit width is 18 bits, and the polynomial length $N = 2048$. The hashing process is not shown in the figure.

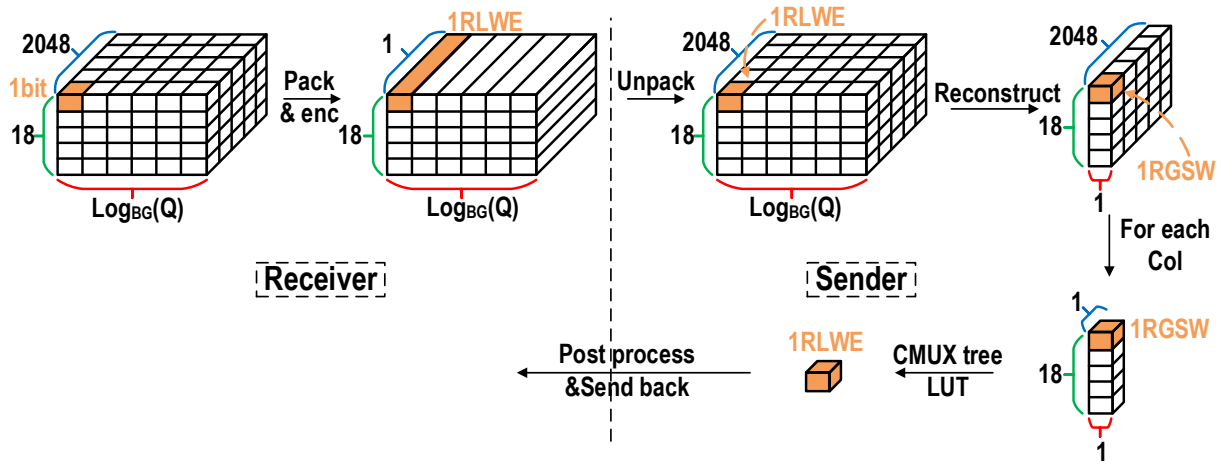

Figure 4.14 Data Flow of the Homomorphic LUT Based PSI.

4.6 Accelerator Architecture for the Adapted FHEW

In this section, an accelerator architecture for the core functions of the adapted FHEW scheme that is detailed in the above chapters is proposed. It features an asymmetric architecture for INTT/NTT module, which exploits the characteristic of the fundamental algorithms.

4.6.1 Overall Architecture

Figure 4.15 shows the overall architecture of the proposed accelerator with a zoom-in view of the compute pipeline in Figure 4.16. Implemented with AWS F1 instance, the accelerator is controlled and monitored by the host software running on an x86 processor through various AXI

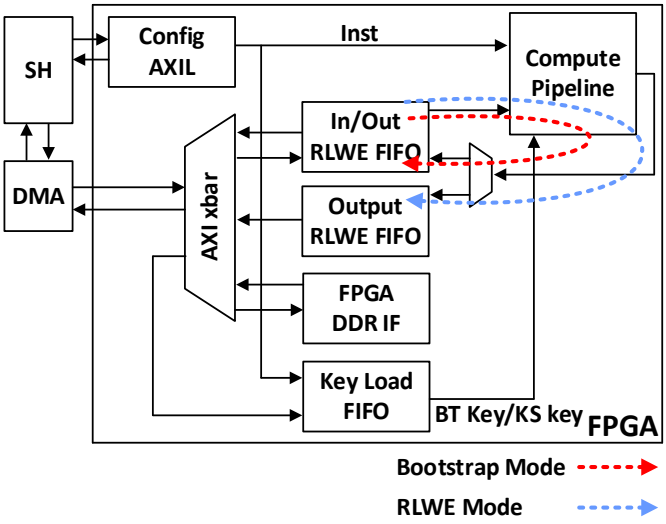

Figure 4.15 Overall Architecture of the Proposed Accelerator.

interfaces. The configure parameters and instructions are programed with the AXI-Lite interface, and the FIFO states are also read from it. The DMA module communicates with the FPGA through the AXI bus to program the FPGA DDR and read/write the RLWE FIFOs. The RLWEs streamed in and out of the FPGA are in the NTT domain. Further, the modulo multiplication in the accelerator is facilitated by the standard Barrett Reduction [137].

The architecture works in a pipelined fashion, with necessary inter-stage double buffering. Upon an input instruction, the key load module reads the corresponding key from the preprogramed FPGA DRAM into its own key load FIFO. In parallel, the INTT/NTT modules inside the compute pipeline manipulate the input RLWEs, hiding the DDR access delay of the key load module since the keys are only needed at the poly MAC stage, which facilitates the

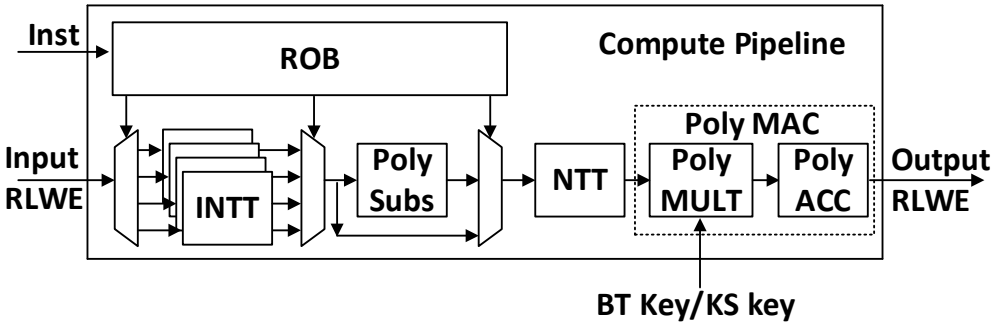

Figure 4.16 Overview of the Main Compute Chain.

polynomial and RLWE vector inner product introduced in Section 4.3.3 and Section 4.5.2. Once the computation finishes, the output RLWEs are written back to the RLWE FIFO dictated by the mode of the accelerator, which will be detailed later, and then streamed out to the host.

As mentioned in Section 4.5.2, the accelerator merges the two data flows, the RLWE substitution and the bootstrap process. Note that the data flow of evaluating the homomorphic LUT introduced in Section 4.4.3 is mostly the same as the bootstrap flow since they both incorporate the $\otimes$ operation, so they will not be differentiated in the remaining text. There are three primary differences between the two data flows. The first is the RLWE key switch versus the $RLWE \otimes RGSW$ operation as highlighted in Figure 4.13 and Figure 4.6, respectively. Second, in RLWE substitution, after INTT, the subroutine that transforms the $RLWE_z(\boldsymbol{m})$ into $RLWE_{z(X^k)}(\boldsymbol{m}(X^k))$, as stated in Section 4.5.2, is needed; this subroutine is unnecessary in the bootstrap process. Lastly, an RLWE ciphertext, streamed into the in/out FIFO, only passes through the compute pipeline once for RLWE substitution and is then streamed out from the output FIFO after the computation. In contrast, in the bootstrap process, after initialization, the same RLWE (homomorphic accumulator) must be looped $n \times \log_{B_r} q$ times through the compute pipeline before being streamed out, meaning that the output RLWE from the compute pipeline should go to the same FIFO as the input RLWE.

The first two differences regarding the computation are automatically taken care of by the different instructions passed into the compute pipeline. For the third one, a mode configuration is added to the FIFOs to differentiate the situations, as shown by the dotted lines in Figure 4.15. In *RLWE* mode, the in/out FIFO acts only as an input FIFO that receives the input RLWEs, whereas the output FIFO holds the processed RLWEs. While in the *bootstrap* mode, the output FIFO is turned off and the in/out FIFO holds the intermediate RLWEs. The compute pipeline continuously

reads and writes the in/out FIFO until the loop finishes. Then the RLWEs in the FIFO are streamed out to the host.

4.6.2 INTT module

The pseudo-code of the INTT algorithm adopted from [138] with the Gentleman-Sande (GS) butterfly is shown in Algorithm 1, which is mapped onto the INTT module. Figure 4.17 (a) details the structure of the INTT module. The dataflow follows Algorithm 1 except that in the first outer loop, the input RLWE is read from the global in/out RLWE FIFO, and the intermediate result is written to its own two polynomial buffers since each RLWE contains two polynomials. Starting from the second outer loop, the input is read from the polynomial buffers and written back after being processed by the butterflies. Each INTT module stores its own copy of the twiddle factors (TFs) in its local memory.

Algorithm 1 Inverse NTT | $INTT(\boldsymbol{a_{NTT}})$

Input: $\boldsymbol{a_{NTT}} \in \mathbb{Z}_Q^N$ in bit reverse order, $Q \equiv 1 \ mod \ 2N$, a vector of twiddle factors $TF \in \mathbb{Z}_Q^N$ storing the powers of ψ_N^{-1} in bit reverse order.

Output: $\boldsymbol{a} \leftarrow INTT(\boldsymbol{a_{NTT}})$ in INTT domain with normal order

1: $t = 1;$
2: $for(m = N; m > 1; m = m/2)$
3: $\quad j_1 = 0; h = m/2;$
4: $\quad for(i = 0; i < h; i++)$
5: $\quad\quad j_2 = j_1 + t - 1; S = TF[h + i];$
6: $\quad\quad for(j = j_1; j \leq j_2; j++)$
7: $\quad\quad\quad U = \boldsymbol{a_{NTT}}[j];$ GS Butterfly
8: $\quad\quad\quad V = \boldsymbol{a_{NTT}}[j + t];$
9: $\quad\quad\quad \boldsymbol{a_{NTT}}[j] = U + V \ mod \ Q;$
10: $\quad\quad\quad \boldsymbol{a_{NTT}}[j + t] = (U - V) \times S \ mod \ Q;$
11: $\quad\quad j_1 = j_1 + 2t;$
12: $\quad t = 2t;$
13: $for(j = 0; j < n; j++)$
14: $\quad \boldsymbol{a_{NTT}}[j] = \boldsymbol{a_{NTT}}[j] \times N^{-1} \ mod \ Q;$
15: Return $\boldsymbol{a} = \boldsymbol{a_{NTT}}$

Two parallel butterfly units are included in each INTT module to achieve better performance while maintaining a reasonable FPGA resource usage. Thus, to feed enough data, each address of the polynomial buffer contains two consecutive coefficients of a polynomial. The

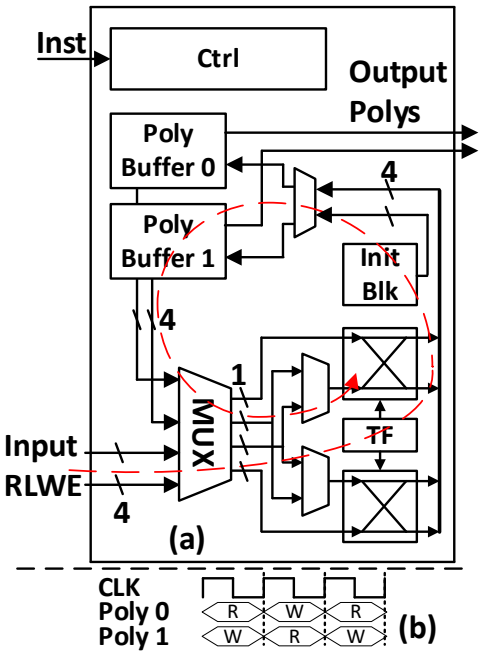

Figure 4.17 Architecture and Dataflow of the INTT Module (a), and the Time-Interleaving of the Polynomial Buffers.

BRAMs of the FPGA that are used to build the polynomial buffers are inherently composed of two read/write ports, fitting the butterfly data access pattern and allowing it to read/write two different addresses at the same time. However, the read and write can only be done in separate clock cycles, resulting in 50% butterfly utilization and halving the throughput. Therefore, we time interleave the two polynomial buffers, as shown in Figure 4.17 (b), to achieve full utilization of the butterflies.

Due to the variation of the data access pattern of the butterfly units in each outer loop of the INTT algorithm, there is a mismatch between the data access pattern and data storage pattern, resulting in two different data flows from the buffers to the butterflies. As shown in Figure 4.18, in pattern 1, the data passes into a butterfly are from different addresses, while in pattern 2, they are from the same address. Therefore, necessary data MUXs are appended to the butterfly units to reorder the input/output data as needed. All the necessary loop counters and step counters are implemented inside the control block, together with the control of the MUXs.

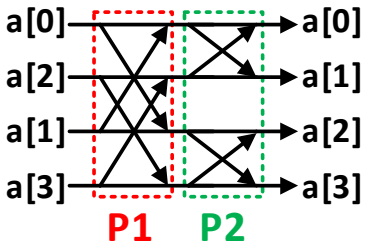

Figure 4.18 Types of Data Access Pattern of The INTT Module.

Besides the INTT functionality, the INTT module also incorporates an init block for the homomorphic accumulator initialization function introduced in Section 4.3.3.

4.6.3 Pipelined NTT module

The NTT algorithm with a Cooley-Tukey (CT) butterfly, shown in Algorithm 2, is very similar to the INTT algorithm, except for the last scaling loop [138]. But a different construction from the INTT module is proposed for the NTT module. The structure of the module is shown in Figure 4.19, and a discussion of this construction is included in a later section.

Algorithm 2 NTT | $NTT(\boldsymbol{a})$

Input: $\boldsymbol{a} \in \mathbb{Z}_Q^N$, $Q \equiv 1\ mod\ 2N$, a vector of twiddle factors $TF \in \mathbb{Z}_Q^N$ storing the powers of ψ_N in bit reverse order.
Output: $\boldsymbol{a_{NTT}} \leftarrow NTT(\boldsymbol{a})$ in NTT domain with bit reverse order

1: $t = n;$
2: $for(m = 1; m < N; m = 2m)$
3: $\quad t = t/2;$
4: $\quad for(i = 0; i < m; i++)$
5: $\quad\quad j_1 = 2 \cdot i \cdot t; j_2 = j_1 + t - 1; S = TF[m + i];$
6: $\quad\quad for(j = j_1; j \le j_2; j++)$
7: $\quad\quad\quad U = \boldsymbol{a}[j];$ CT Butterfly
8: $\quad\quad\quad V = \boldsymbol{a}[j + t] \times S;$
9: $\quad\quad\quad \boldsymbol{a}[j] = U + V\ mod\ Q;$
10: $\quad\quad\quad \boldsymbol{a}[j + t] = U - V\ mod\ Q;$
11: Return $\boldsymbol{a_{NTT}} = \boldsymbol{a}$

To achieve higher throughput for the NTT module, the outer loop of Algorithm 2 is unrolled into $log(N)$ pipeline stages, with each stage only processing one fixed data access pattern, which greatly reduces the control complexity of each stage. Compared to the structure of the INTT module, this implementation offers the same processing latency for an input polynomial but $log(N)$ times higher throughput.

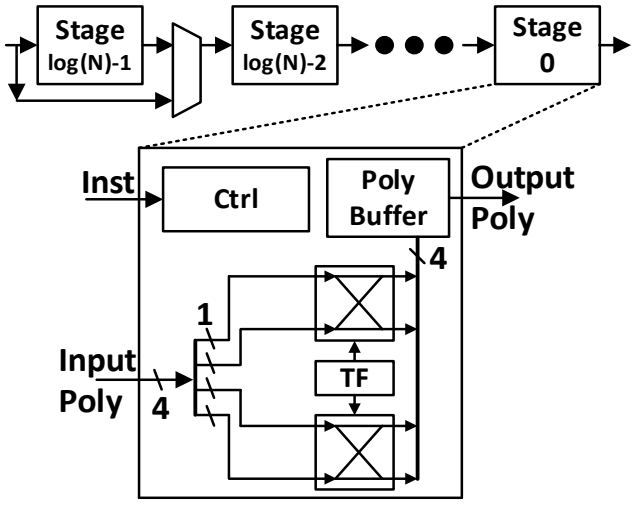

Figure 4.19 Architecture and Dataflow of the Pipelined NTT Module.

Each stage reads the input from the polynomial buffer of the previous stage and processes it with a predetermined data access pattern that is specific to that stage at design time. So, there is no on-the-fly control/MUXs for the data flow, which not only reduces resource usage but also allows a better timing requirement. Note that there is no read/write from/to the same buffer memory; therefore, it is not necessary to employ the time-interleave trick as in the INTT module.

Although the internal structures of the stages are mostly the same, except for the loop counter and step counter inside the control block, extra care should be taken in actual implementation. First, to adapt to different polynomial lengths, MUXs are needed to skip the leading stages for short polynomials (Figure 4.19). Second, the leading stages also incorporate the decomposition functionality as stated in Sections 4.3.3 and 4.5.2, which is just a bitwise AND with a binary decomposition basis and is not detailed in the figure.

4.6.4 Compute Pipeline Analysis: Asymmetric INTT and NTT

Before further discussion of the proposed compute pipeline architecture, some notations are defined. In the following section, we refer to the overall latency of the INTT/NTT algorithms as one NTT latency (ONL) and the latency of one outer loop of the algorithm as one stage latency

(OSL). Therefore, it is obvious that $ONL = log\,(N) \times OSL$. And our compute pipeline architecture utilizing the pipelined NTT module, detailed in the above section, with non-pipelined INTT modules is referred to as an asymmetric structure due to the throughput difference of the two types of the modules. The conventional implementation of similar structures and latencies for the NTT and INTT modules as introduced in Section 4.6.2 is defined as a symmetric structure.

The design of our compute pipeline concentrates on balancing high throughput with optimized resource usage and parameter flexibility. With this in mind, the main compute pipeline is built around an asymmetric structure, as shown in Figure 4.16. A comparison of the symmetric and asymmetric structures is given in Fig. 18, with the poly subs block omitted as it is not a throughput bottleneck. The dataflows of the $RLWE \otimes RGSW$ and RLWE substitution introduced in Figure 4.6 and Figure 4.13, respectively, can be mapped to both architectures with the same throughput. However, the asymmetric structure consumes less resources than its symmetric counterpart.

In the symmetric pipeline (Figure 4.20 (a)), to have balanced throughput, one INTT module is accompanied with dc-many NTT modules since each input polynomial is decomposed into dc polynomials after the INTT operation. The throughput of both modules is one polynomial per ONL due to the non-pipelined construction. The NTTs are also followed by dc-many polynomial/RLWE multiplication blocks to facilitate the inner product of the two dataflows. Although the trailing stages can operate with higher throughput, the overall throughput of the whole pipeline is capped by the first two stages, resulting in a throughput of one polynomial per ONL. Higher throughput can be achieved by operating multiple pipeline instances in parallel.

Most of the prior arts implemented an architecture that is similar to the symmetric structure with the INTT and NTT modules separated without considering the data flow connecting the

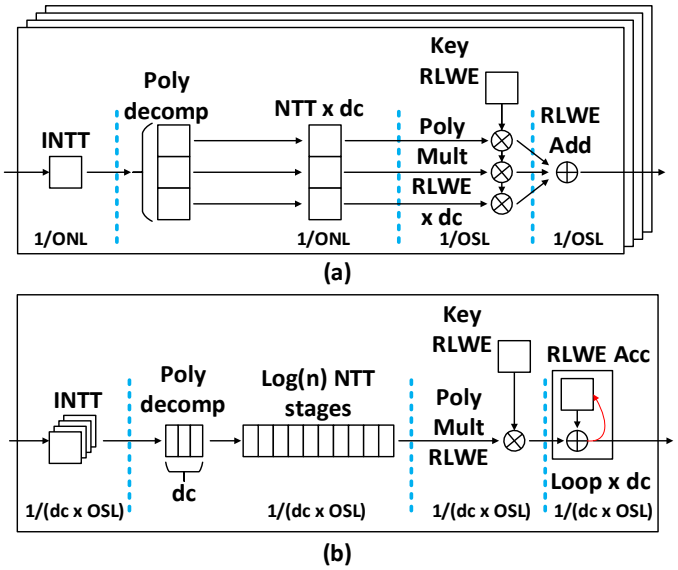

Figure 4.20 Comparison of Symmetric (a) and Asymmetric (b) Compute Pipelines.

modules. We take it one level up and make use of the asymmetric structure to cope with the different throughput requirements of the INTT and NTT modules, as shown in Figure 4.20 (b). Since the throughput of the pipelined NTT is OSL, the overall throughput of the whole pipeline is one input polynomial per $dc \times OSL$ because of the polynomial decomposition, with one caveat that to balance the throughput between the INTT and NTT, $\log(N)/dc$ INTT modules should operate in parallel. Note that in the asymmetric scenario, the trailing stages are also changed to the pipelined form (1 vs. dc poly mult RLWE and an accumulation vs. a wide addition). In practice, $\log(N)$ is always greater than dc; therefore, the asymmetric pipeline enables higher throughput than a single instance of the symmetric pipeline.

Though the symmetric structure can achieve the same throughput as the asymmetric one, with $log\ (N)/dc$-many instances operating in parallel, as seen in Figure 4.20 (a), the asymmetric pipeline uses less FPGA resources. The reduced resources stem from three sources. First, it is clear that in both cases, the number of INTT modules is the same, amounting to $log\ (N)/dc$. The number of NTT modules seems to be the same as well since there are $log\ (N)/dc \times dc = log\ (N)$ NTT modules for the symmetric pipeline and the asymmetric one also incorporates

$log\ (N)$ NTT stages. However, in the symmetric case, the NTT module has a similar structure as the INTT module shown in Figure 4.17, which is much more complex than the NTT stage used in the pipelined NTT. Synthesis shows that with pipelined NTT, 23% less LUT usage is achieved.

Furthermore, the pipelined NTT module has not only smaller control logic but also lower memory requirements. Part of the savings comes from less TF memory in pipelined NTT. Each of the NTT modules used in the symmetric pipeline stores a complete copy of the polynomial of the TF in its own local memory, similar to what is shown in Figure 4.17, so that they can operate independently. Therefore, in total, log(N) copies of the TF are stored. In contrast, in the asymmetric version, there is only one complete copy of the TF. Because each stage of the pipelined NTT is only responsible for one outer loop of the NTT algorithm, it only needs to store the portion of the TF that is used in that outer loop. For example, in the first stage of the pipelined NTT, instead of a complete polynomial of TF with N coefficients, only one TF needs to be stored. Thus, overall, a $log\ (N)$ times reduction of the TF memory usage is achieved with pipelined NTT, equivalent to over 10x reduction in practice. It is possible to reduce the memory usage in the symmetric version by sharing one TF memory within one pipeline and force all the NTT modules to act at the same pace, but that implies stricter timing requirements since the capacitive load of the memory output is dc times higher, exacerbating performance. Also forcing all NTTs to synchronize degrades the flexibility of the architecture.

The memory size of the pipelined NTT module is also reduced due to fewer polynomial buffers. In the non-pipelined NTT module, similar to the INTT module in Section IV.B, two polynomial buffers are instantiated for time-interleaved buffer access to maintain 100% butterfly utilization. In contrast, each stage of the pipelined NTT module reads and writes different buffers;

therefore, time-interleaving is unnecessary. So, the pipelined NTT poses a 50% saving on the polynomial buffer compared to the non-pipelined version.

Lastly, the trailing stages of the asymmetric pipeline are also less complex than that of its symmetric counterpart. As shown in Figure 4.20, since the pipelined NTT outputs one polynomial at a time, only one poly mult RLWE module is needed in the asymmetric structure, compared to $log\ (N)$ parallel mult modules in the symmetric one. In practice, it reduces the amount of mult modules by 11 times, with $N = 2048$. Although the amount of poly mult RLWE modules can be reduced in the symmetric pipeline by reusing one mult module across different NTTs in a time-interleaved manner due to the higher throughput compared to the INTT/NTT, a very wide MUX, log(N) to one, must be inserted between the stages, which would greatly impact timing and performance and introduce more control complexity. In the asymmetric structure, there is a similar MUX between the INTT and NTT modules; however, it is only $log\ (N)/dc$ to one, which is much smaller. In addition, the dc-wide RLWE addition in the symmetric pipeline is also replaced with an RLWE accumulation with ordinary word-size modulo addition.

Besides the resource savings, the asymmetric construction also automatically adjusts to different parameter settings. In the symmetric pipeline, the number of NTTs should be set as the largest possible number of dc of the application at design time. If at design time, the parallelism is 3 for NTT, when $dc = 2$ at run time, the utilization of the NTTs is only 66.7%. Extra effort can be applied to remap the connection between INTT and NTT to reach 100% utilization, but that comes with more control overhead, negatively impacting performance. However, with the pipelined NTT module, as long as the INTT continuously feeds input to it, 100% utilization is always maintained with no extra control overhead involved since the design space of the pipelined NTT itself is independent of the parameter dc. In fact, even when the dc of run time is higher than

the designated dc of design time, the pipelined NTT requires no extra control to handle it. However, it should be noted that in the above case, the INTT of the asymmetric pipeline can be underutilized. But since the number of INTT stages is less than the number of NTT stages in general, it is not optimized in this work.

4.7 Measurement

4.7.1 Experiment Setup

The proposed architecture is implemented at 125MHz system frequency on an AWS F1 instance with the system construction specified in Figure 4.15. The implementation supports up to 54-bit input data word size. But to reduce the complexity of the modulo multiplication block in the butterfly, only a subset of the bit widths is implemented, as detailed in the following sections. Two polynomial lengths, 1024 and 2048, which are typical for third-generation FHE and fit our experiment for the PSI protocol, are implemented. Therefore, the INTT and NTT modules can be configured to process both lengths. The polynomial buffers in the FIFOs, implemented with BRAM, are configured to the size of the longer length, 2048. In addition, since 2 butterfly units operate at the same time in the INTT and NTT, each buffer line contains two consecutive polynomial coefficients. Thus, the size of each polynomial buffer is predefined as 1024×108 bit. However, in this prototype, no optimization on the BRAM utilization is devised, so when the input polynomial length is 1024, only the first half of the buffer is used. Following the analysis of Section 4.6.4, the number of INTT is set to $log\,(N)/dc$ to keep a balanced throughput. In our implementation, it is set to the largest possible value derived from the parameter sets, which is 4.

4.7.2 Measurement of Bootstrap of The Third Generation FHE

The parameter sets used to benchmark our implementation of the third-generation FHE are listed in Table 4-1 and adopted from [116]. The MEDIUM parameter set corresponds to ~100 bits of security level. Since our work only implements the homomorphic accumulation of the bootstrap process (including evaluation, accumulation, and key switch) on the hardware, we only report the measurement of this operation to emphasize our advancement. It is composed of two parts, the processing time and the time required to stream out the result for post processing. Table 4-2 summarizes the measurement of the homomorphic accumulation function. Due to the pipelined nature of the proposed accelerator, the maximum parallelism that can be achieved is 12 accumulations at the same time. So. the reported time is the amortized time of 12 inputs. Because the homomorphic accumulation function is independent of the input binary gate, we do not differentiate it during the measurement. The reported time is averaged over all measured input gates.

The software implementation [116] of the FHEW scheme from the PALISADE library [55] operates on the same host machine and is used for comparison. Table 4-3 gives the comparison result of the proposed accelerator over software implementation. As stated above, only the homomorphic accumulation part is compared. On average, a 21× speed-up for the homomorphic accumulation function is achieved compared with the software implementation.

Table 4-1 Parameter Sets of The Third Generation FHE

Parameter set	n	q	N	$Log_2(Q)$	B_{ks}	B_G	B_r
MEDIUM	256	512	1024	27	25	2^9	23
STD128_AP	512	512	1024	27	25	2^9	23
STD192	512	512	2048	37	25	2^{13}	23
STD256	1024	1024	2048	29	25	2^{10}	32
STD192Q	1024	1024	2048	35	25	2^{12}	32
STD256Q	1024	1024	2048	27	25	2^7	32

Table 4-2 Processing Time of Homomorphic Accumulation

Parameter set	*Amortized Processing Time (us)*	*Amortized Stream Out Time (us)*
MEDIUM	6615	49
STD128_AP	13238	48
STD192	26253	54
STD256	52523	54
STD192Q	52524	59
STD256Q	70031	58

4.7.3 Measurement of The Proposed PSI

In our implementation of the proposed PSI protocol, we set the encryption-related parameters to be $N = 2048$, $\log_2(Q) = 54$, with $\sigma = 3.19$, which achieves around 128-bit security level according to the LWE estimator [139]. The B_G of the RGSW and B_{KS} of the RLWE key-switch key are both set to 2^9. Since the proposed PSI is not directly available in open-source libraries, we developed the necessary components of the scheme ourselves for baseline comparison.

The average processing time of the two basic operations of the proposed PSI, RLWE substitution and $RLWE \otimes RGSW$, as deployed on the hardware are shown in Table 4-5. A comparison to our own software implementation is also included in the table. The raw measurement shows a speed-up factor of over 140, which is much higher compared to the improvement of the bootstrap process. A discussion of this discrepancy is incorporated in a later

Table 4-3 Comparison of Processing Time of The Proposed Accelerator Over Software

Parameter set	*Proposed Accelerator (ms)*	*Software [116] (ms)*	*Improvement*
MEDIUM	6.7	141.1	21.1×
STD128_AP	13.3	283.8	21.3×
STD192	26.3	578.4	22.0×
STD256	52.6	1180.8	22.4×
STD192Q	52.6	1270.5	24.2×
STD256Q	70.1	1571.5	22.4×

Table 4-4 Comparison of The Processing Time of The Two Operations For The Proposed PSI

Operation	*Proposed Accelerator (μs)*	*Software (μs)*	*Improvement*	*Scaled Improvement*
RLWE Substitution	105	17616	167.8×	27.9×
$RLWE \otimes RGSW$	105	14739	140.4×	23.4×

section. The last column, 'Scaled Improvement,' is added for this purpose and is discussed later, as well.

Based on the time consumption of the basic operations from Table 4-4, the processing times on the Sender's side with the proposed accelerator and the communication size of the proposed PSI are listed in Table 4-5. Since the complexity of our scheme is only directly dependent on the bit width b and the hash table size 2^k, assuming only one element in each bin on the Receiver's side, we only list these two factors as design parameters in the table, with the security parameters set as above. In the Receiver-to-Sender communication size, the key-switch keys and the RGSW-encrypted $-\boldsymbol{z}$ are not included, which are of size 2.1 MB and 384 KB, respectively. Note that a modulus switch process can be applied to the returning LWE ciphertexts from the Sender to the Receiver, which can further reduce the message size by 15~20% [110]. Figure 4.21 shows a time breakdown of the proposed PSI operating with the proposed accelerator. Four parts are included: (a) RLWE substitution; (b) $RLWE \otimes RGSW$; (c) RGSW transfer, which transfers the reconstructed RGSWs to the FPGA DDR; and (d) software post process. The first three are

Table 4-5 Sender's Processing Time and Communication Size Of The Proposed PSI

Parameters		*Sender's Processing time (s)*		*Communication Size (MB)*	
b	k			$R \rightarrow S$	$S \rightarrow R$
32	14	1642		27.0	256
	12	814		7.5	64
	10	585		2.1	16
30	14	1148		24.0	256
	12	410		6.8	64
	10	203		1.9	16
28	14	935		21.0	256
	12	287		6.0	64
	10	102		1.7	16

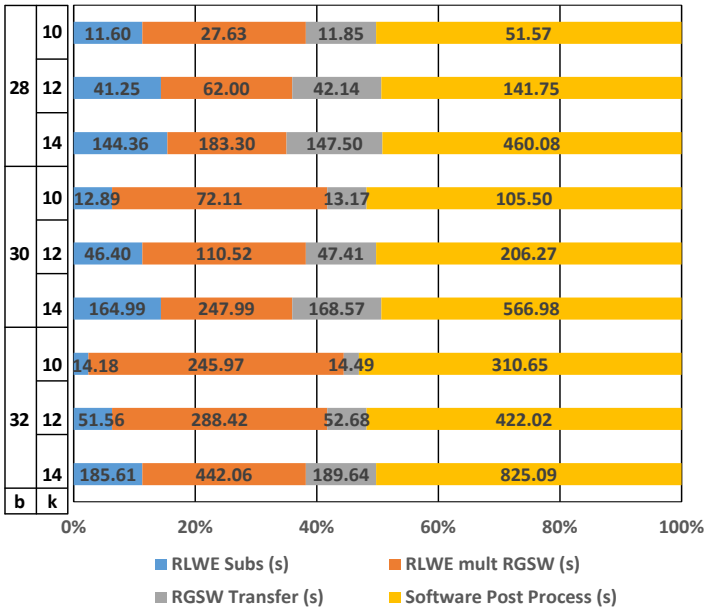

Figure 4.21 Time Breakdown of the Proposed PSI.

attributed to hardware. The measured time consumption of each part is also included in the diagram. The software post processing times are raw measurement data and not scaled, which will be discussed in next section.

4.7.4 Analysis Of The Measurement Results

1) Software Inefficiency Encountered in the Measurement

As mentioned in the above section, compared to the improvement of the bootstrap process listed in Table 4-3, we see a higher speed-up in the basic PSI operations, as shown in Table 4-4. The discrepancy mainly results from the different software implementations that are being used in the comparison. Since the proposed PSI and its basic operations are not directly available in open-source libraries, we developed the software implementation ourselves from scratch for both verifying the hardware design and baseline comparison. We also built our own software for the bootstrap process for the purpose of hardware verification and comparison.

However, due to our relatively limited effort, our own software code may not perform as efficiently as the highly optimized open-source libraries. In order to estimate the potential software

performance discrepancy, a comparison between our own software and an open-source library [55] is conducted with the same host machine using commonly available operations such as NTT/INTT, polynomial operations, and bootstrap process. Based on the comparison, our own software code is around 6× slower compared to open-source library. Hence, the measured improvement in the third column of Table 4-4 is scaled by 6 to factor in potential software optimization for a more realistic speed-up number for the basic operations of the proposed PSI. This scaled number is shown in the last column of Table 4-4.

The inefficiency in our software code includes unoptimized post processing, which takes about 50% of the total processing time of the proposed PSI operating on the accelerator (Figure 4.21). Thus, by factoring out this inefficiency, the total time consumption of the proposed PSI could be reduced by around 42% (which is not accounted for in the reported performance in Table V).

2) I/O Bandwidth Bottleneck of the Implemented Accelerator

During the measurement, we find that the latency of processing just one input on the proposed acceleration hardware is ~350 μs for RLWE substitution and 309 μs for $RLWE \otimes RGSW$, which includes 120 μs streaming in and out. Due to the pipelined nature of the proposed accelerator, a maximum parallelism of 13 can be achieved in the RLWE mode. Therefore, ideally, the average time consumption of processing one input on the hardware should be ~17 μs, which is 6× faster compared to the numbers listed in the first column of Table 4-4. This shows that, in the RLWE mode, the accelerator is bottlenecked by the I/O bandwidth. In the case that an optimized I/O is achieved, 6× better performance can be extracted from the proposed accelerator.

Table 4-6 summarizes the (estimated) attainable bound of processing time of the proposed PSI, which both factors out software inefficiency and operates on an optimized I/O.

Table 4-6 Attainable Bound of Sender's Processing Time of The Proposed PSI

Parameters		*Sender's Processing time (s)*	
b	***k***	***Measured***	***Attainable Bound***
32	14	1642	273
	12	814	135
	10	585	97
30	14	1148	191
	12	410	68
	10	203	33
28	14	935	155
	12	287	47
	10	102	17

4.8 Discussion

4.8.1 Future Improvements to The Proposed FHE Accelerator

Since the depths of the buffers in the accelerator are designed to hold the longest possible polynomial, which is 2048, when running with shorter polynomials, such as 1024, the BRAM utilization of the accelerator is limited. The same problem also applies to the word width of the buffers. Static or dynamic remapping of the BRAM can potentially improve the utilization.

In the current implementation, interrupt functionality is not included. Therefore, the host is busy waiting by continuously reading the status register during the operation of the accelerator, which not only occupies the software resource but also introduces unnecessary communication traffic to the operation. As analyzed above, the I/O operation also bottlenecks the accelerator. Thus, an optimized interface between the host and the accelerator should be included in future implementations.

In *bootstrap* mode, only the in/out RLWE FIFO is in full utilization, while the output RLWE FIFO is always idle. Therefore, after each bootstrap process, while streaming out from the in/out FIFO, the compute pipeline has to wait until the FIFO is empty. However, if the output FIFO

acts as a second in/out FIFO, the compute pipeline will have better utilization due to the overlap of the data streaming out and data processing.

In *RLWE* mode, although both RLWE FIFOs can work independently of each other, our current software stack does not include multithreading capability that can stream into and out of the FPGA at the same time in a producer/consumer fashion. Also, since both input and output transactions pass through the AXI crossbar, in some extreme cases, it can bottleneck the whole system. Therefore, the input and output FIFO should be designated to separate interfaces between the host machine and the FPGA to avoid transaction congestion while operating at the same time.

Finally, as mentioned in the above sections, we built the software stack that controls the proposed accelerator ourselves from scratch. Therefore, APIs that are compatible with the open-source libraries are not available. We would like to support that for better performance.

4.8.2 Future Improvements to The Proposed PSI Protocol

It is clear that the security proof of our protocol is derived directly from [110], which is secure in a semi-honest setting. The authors of [110] improved the security proof to malicious adversary in [117]. Therefore, it is worth exploring an adaptation of our proposed protocol to a malicious setting.

In addition, even with all the optimizations proposed in Section 4.5.3, the proposed protocol still imposes a communication complexity linear to the bit-width of the elements of the sets and a computation complexity exponential to the bit-width. It is manageable when the data sets are 32-bit or less, but with wider numbers, the computation explodes. One possible solution is to first divide the wide number into several small segments (one 64 bits to two 32 bits for instance) and perform the proposed PSI on each segment. Then, the outputs of the PSI for all the segments of the same wide number are ANDed to get the final result. Note that to prevent

information leaks of the Sender's set, this AND is also performed by the Sender together with the homomorphic LUT.

4.9 Related Work

4.9.1 Hardware Acceleration of Fully Homomorphic Encryption

Ever since the discovery of Gentry's fully homomorphic blueprint [48][113] in 2009, which featured the bootstrap concept and followed an exploration that spanned over 30 years since the proposal of RSA encryption, much advancement has been made to improve the initial construction, propelling practical applications of this idea. Among them, BGV [50], BFV [51][52], CKKS [53], FHEW [59], and THFE [60] are the most well accepted and are built upon RLWE [115] and LWE [47] problems, different from Gentry's initial formula.

These schemes are categorized based on their application emphasis. The second-generation schemes, BGV, BFV and CKKS, focus on faster polynomial evaluation on encrypted data using arithmetic multiplication and addition, with the difference being that BGV and BFV schemes operate on integer numbers, whereas CKKS works with approximate complex numbers as plaintext space. SIMD-styled operation [54] is widely adopted in these schemes to pack multiple input data into one ciphertext, thus increasing parallelism. Several open-source libraries that implement these schemes are publicly available, including PALISADE [55], HElib [56] and SEAL [57]. With the maturity of the algorithm and implementation, applications of these schemes to homomorphic evaluation of Neural Networks have also been published in recent years [140]-[144]. Other applications like PSI based on second-generations schemes are also proposed [110][117].

Compared to the second-generation counterparts, FHEW and THFE excel in efficient and flexible homomorphic binary logic gate evaluation. Although, the performance of the third-

generation schemes is estimated to be on the same order as that of the second-generation approaches equipped with SIMD-like construction, they well accepted for their simplicity and flexibility in terms of both concept and implementation. The binary logic nature of the third-generation FHEs makes them suitable for applications that are logic intensive, such as ORAM as proposed in [111].

Various hardware solutions have been proposed in recent years [119]-[129] that aim at mitigating the performance gap between direct evaluation of plaintext and homomorphic evaluation. [119][121] proposed acceleration for encryption/decryption of RLWE in post-quantum scenarios; however, due to the smaller size of the applicable polynomial, it is not suitable for homomorphic operation. A crypto-engine for the encryption/decryption of RLWE for homomorphic encryption is presented in [120], which involves less heavy lifting compared to the homomorphic evaluation (bootstrap). Accelerating large number multiplication is also a direction pursued in [128]; however, it is not necessary for third-generation schemes. [126][127] explored different approaches to accelerate long polynomials in homomorphic encryption. [123]-[125][129] proposed accelerators for BFV-based LHE schemes, but this approach suffered from limited computation depth and security level. An architecture for a leveled CKKS scheme is proposed in [122].

4.9.2 Private Set Intersection

Early PSI protocols [105][133] were built upon the multiplicative homomorphism of the Diffie-Hellman public-key encryption, featuring good communication cost but prohibitive computation toll as the set size grows.

By far, the most efficient schemes are Oblivious Transfer (OT)-based [106][134][135], in which the Receiver obliviously encodes each element in its set by initiating multiple OTs with the

Sender and then cross compares it with the encoded Sender's set that it receives to find the intersection. Because of the randomized encoding of the protocol, neither of the two parties learns about the other's data set except that the intersection is revealed to the Receiver. It is clear that such an approach involves sending both parties' sets through the network, resulting in communication cost that is linear in both sets' sizes, an undesirable consequence in an unbalanced scenario where one party's set is significantly smaller.

In recent years, unbalanced PSI protocols based on second-generation FHE [110][117] have been proposed. These provide communication overhead linear in the size of the smaller set, a substantial reduction compared to previous approaches in such scenarios, while maintaining a comparable performance.

4.10 Conclusion

In conclusion, the first hardware acceleration architecture for third-generation FHE is proposed in this paper. Featuring an asymmetric INTT/NTT configuration, the proposed compute pipeline achieves less resource usage while maintaining a high throughput. An extensive analysis of the architecture is presented. An unbalanced PSI protocol based on third-generation FHE is also proposed to better demonstrate the architecture. Supplemented by several optimizations for reducing the communication and computation costs, the proposed PSI achieves a computation cost independent of the Sender's set size. Implemented with AWS cloud FPGA, the proposed accelerator achieves over 21× performance improvement compared with a software implementation on various subroutines of the FHE and the proposed PSI at 125 MHz.

Chapter 5. Conclusion

5.1 Contributions of This Thesis

This work focuses on domain-specific accelerators. Three emerging applications, namely, DNA sequencing, ML, and post quantum/homomorphic encryption are explored.

In Chapter 2, an accelerator for seed extension, a critical and computationally intensive step in genome sequencing, is proposed. The accelerator, implementing a string-independent automata, consists of a triangular array of 25×25 custom-designed processing elements. It performs 2.46M reads/s, achieving a 1581x improvement in power efficiency and 165.5x smaller silicon footprint compared to a system with dual-socket Xeon E5-2697 v3 server processors.

Chapter 3 presents an energy-efficient deep neural network (DNN) accelerator with non-volatile embedded resistive random-access memory (RRAM) for mobile ML applications. This DNN accelerator implements weight pruning, non-linear quantization, and Huffman encoding to store all weights on RRAM, enabling single-chip processing for large neural network models without external memory. A 4-core parallel and programmable architecture adapts to various neural network configurations with high utilization. We introduce a customized RRAM macro with a dynamic clamping offset-canceling sense amplifier (DCOCSA) that achieves sub-μA input offset. The on-chip decompression and memory error-resilient scheme enables 16 million (M) 8-bit (decompressed) weights on a single chip using 24 Mb RRAM. The proposed RRAM-DNN is the first digital DNN accelerator featuring 24 Mb RRAM as all-on-chip weight storage to eliminate energy-consuming off-chip memory accesses. The fabricated design performs the complete

inference process of the ResNet-18 model while consuming 127.9 mW power in TSMC-22nm ULL CMOS. The RRAM-DNN accelerator achieves peak performance of 123 GOPs with 8-bit precision, exhibiting measured energy efficiency of 0.96 TOPs/W.

Chapter 4 presents the first accelerator architecture for third-generation FHE, targeting at the $RLWE \otimes RGSW$ operation, which is a fundamental function of both second-generation and third-generation FHE. By exploiting the asymmetric nature of the encryption, the architecture incorporates an asymmetric Inverse Number Theory Transform (INTT) module and Number Theory Transform (NTT) module, which are capable of maintaining high throughput with less resource usage while addressing different parameter sets. An extensive analysis of the architecture is included. A novel unbalanced PSI protocol that is based on third-generation FHE and is optimized for the proposed hardware architecture is proposed. The proposed PSI protocol makes the computation cost independent of the Sender's set size. We introduce several additional algorithm-architecture co-optimizations to reduce the computation and communication costs, rendering a practical application of the proposed PSI protocol. A prototype of the proposed architecture is implemented with AWS cloud FPGA service. We develop all necessary high-level functions in C++ and benchmark the implemented architecture with different parameter sets. We make the SystemVerilog HDL code of the proposed accelerator and supporting software code publicly available at [136]. At last, we quantify and analyze the performance of the proposed hardware accelerator and PSI protocol. The measurements show over 21× performance improvement compared to a software implementation for various subroutines of the third-generation FHE and the proposed PSI.

5.2 Future Directions

The DNA sequencing accelerator demonstrated in Chapter 2 targets only at the seed-extension step of the post processing pipeline. It would be interesting to see how much overall improvements can be extracted for the complete pipeline if all the sub-steps are optimized on hardware. In that case, the communication bandwidth between the subroutines and the total memory bandwidth are a potential bottleneck of the system. So, a system level perspective must be honored when integrating the whole pipeline, rather than just optimizing the subroutines separately. In addition, during the backend implementation, we found that some global signal limits the dimension of the accelerator. The large loading of the global signal limits the frequency of the design when dimension is large. Therefore, it would be good to deal with the problem from algorithm and architecture level.

The RRAM blocks designed for the DNN accelerator introduced in Chapter 3 have SAR ADC based test structure for up to 3 bit per cell multilevel capability. Due to the large peripheral of the RRAM, the density of RRAM does not pose significant advantage compared to SRAM. So multilevel bit cells are expected to provide higher memory density. However, in the measurement we were not able to verify that capability due to the malfunctioned level converters inside the RRAM bank. It would be helpful to get the data point of how a high density on-chip nonvolatile memory system would impact the power/area/performance of the system compared to SRAMs. Furthermore, to perform more comprehensive measurement, DFT structure should be included to provide better observability of the chip during measurement.

In Chapter 4, a concept proving accelerator for third-generation FHE is demonstrated with FPGA. We focused on the architecture and hardware implementation, but limited effort was applied to the corresponding software running on the host CPU that talks to the FPGA. As a result,

the software pre/post processing and APIs impose a bottleneck to the overall performance. In addition, we found that the interface bandwidth to the FPGA also limited the overall throughput of the accelerator especially in *RLWE* mode. We performed simple hand calculation to extract attainable upper bound of the performance, it would be ideal to improve the efficiency of the software and interface to better study the accelerator.

Currently, the third-generation FHE is a very emerging encryption scheme that has not been fully explored regarding application of it. In Chapter 4, we introduced a PSI protocol based on third-generation FHE, which sheds light on how the homomorphic binary operation can be utilized in the future. Therefore, expanding the use case of the encryption is an open question that is full of possibilities.

The accelerators demonstrated in this work shows great benefits of domain specific solutions in terms of performance and energy efficiency. And we have seen various directions of domain-specific design. In Chapter 2, a novel algorithm that is suitable for hardware implementation is shown. Chapter 3 exemplifies the impact of technology advancements on the domain specific architecture. At last, Chapter 4 manifests how hardware-software co-design potentially benefits the application. Together, we have shown that domain specific architecture brings technology, circuit, and software even closer to achieve an efficient system. And we envision that this close interaction of technology, circuit, architecture, and software can potentially counteract the declining of technology scaling. Therefore, the key to future application-specific hardware is to build an application centric thought-process that unifies all abstraction levels, including technology, circuit, architecture and software.

Bibliography

[1] Kar Rupp, 48 Years of Microprocessor Trend Data, https://github.com/karlrupp/microprocessor-trend-data, accessed on 01/11/2020.

[2] L. T. Su, S. Naffziger and M. Papermaster, "Multi-chip technologies to unleash computing performance gains over the next decade," 2017 IEEE International Electron Devices Meeting (IEDM), San Francisco, CA, 2017, pp. 1.1.1-1.1.8, doi: 10.1109/IEDM.2017.8268306.

[3] A. P. Chandrakasan, S. Sheng and R. W. Brodersen, "Low-power CMOS digital design," in IEEE Journal of Solid-State Circuits, vol. 27, no. 4, pp. 473-484, April 1992, doi: 10.1109/4.126534.

[4] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), San Francisco, CA, 2014, pp. 10-14, doi: 10.1109/ISSCC.2014.6757323.

[5] "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, 2018, pp. 27-29, doi: 10.1109/ISCA.2018.00011.

[6] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17). Association for Computing Machinery, New York, NY, USA, 1–12. DOI:https://doi.org/10.1145/3079856.3080246.

[7] Y. Chen, T. Krishna, J. Emer and V. Sze, "14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," 2016 IEEE International Solid-State

Circuits Conference (ISSCC), San Francisco, CA, 2016, pp. 262-263, doi: 10.1109/ISSCC.2016.7418007.

[8] Sanghoon Kang, Donghyeon Han, Juhyoung Lee, Dongseok Im, Sangyeob Kim, Soyeon Kim, Hoi-Jun Yoo, "7.4 GANPU: A 135TFLOPS/W Multi-DNN Training Processor for GANs with Speculative Dual-Sparsity Exploitation," 2020 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2020, pp. 140-142, doi: 10.1109/ISSCC19947.2020.9062989.

[9] Yang Jiao, Liang Han, Rong Jin, Yi-Jung Su, Chiente Ho, Li Yin, Yun Li, Long Chen, Zhen Chen, Lu Liu, Zhuyu He, Yu Yan, Jun He, Jun Mao, Xiaotao Zai, Xuejun Wu, Yongquan Zhou, Mingqiu Gu, Guocai Zhu, Rong Zhong, Wenyuan Lee, Ping Chen, Yiping Chen, Weiliang Li, Deyu Xiao, Qing Yan, Mingyuan Zhuang, Jiejun Chen, Yun Tian, Yingzi Lin, Wei Wu, Hao Li, Zesheng Dou, "7.2 A 12nm Programmable Convolution-Efficient Neural-Processing-Unit Chip Achieving 825TOPS," 2020 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2020, pp. 136-140, doi: 10.1109/ISSCC19947.2020.9062984.

[10] Jinshan Yue, Ruoyang Liu, Wenyu Sun, Zhe Yuan, Zhibo Wang, Yung-Ning Tu, Yi-Ju Chen, Ao Ren, Yanzhi Wang, Meng-Fan Chang, Xueqing Li, Huazhong Yang, Yongpan Liu, "7.5 A 65nm 0.39-to-140.3TOPS/W 1-to-12b Unified Neural Network Processor Using Block-Circulant-Enabled Transpose-Domain Acceleration with 8.1 × Higher TOPS/mm2and 6T HBST-TRAM-Based 2D Data-Reuse Architecture," 2019 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2019, pp. 138-140, doi: 10.1109/ISSCC.2019.8662360.

[11] AMD 6000 series graphics cards, https://www.amd.com/en/graphics/amd-radeon-rx-6000-series, accessed on 01/11/2021.

[12] Kodai Ueyoshi, Kota Ando, Kazutoshi Hirose, Shinya Takamaeda-Yamazaki, Junichiro Kadomoto, Tomoki Miyata, Mototsugu Hamada, Tadahiro Kuroda, Masato Motomura, "QUEST: A 7.49TOPS multi-purpose log-quantized DNN inference engine stacked on 96MB 3D SRAM using inductive-coupling technology in 40nm CMOS," 2018 IEEE International Solid - State Circuits Conference - (ISSCC), San Francisco, CA, 2018, pp. 216-218, doi: 10.1109/ISSCC.2018.8310261.

[13] Yu-Der Chih, Yi-Chun Shih, Chia-Fu Lee, Yen-An Chang, Po-Hao Lee, Hon-Jarn Lin, Yu-Lin Chen, Chieh-Pu Lo, Meng-Chun Shih, Kuei-Hung Shen, Harry Chuang, Tsung-Yung Jonathan Chang, "13.3 A 22nm 32Mb Embedded STT-MRAM with 10ns Read Speed, 1M Cycle Write Endurance, 10 Years Retention at 150°C and High Immunity to Magnetic Field Interference," 2020 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2020, pp. 222-224, doi: 10.1109/ISSCC19947.2020.9062955.

[14] Qing Dong, Zhehong Wang, Jongyup Lim, Yiqun Zhang, Yi-Chun Shih, Yu-Der Chih, Jonathan Lee Chang, David Blaauw, Dennis Sylvester, "A 1Mb 28nm STT-MRAM with 2.8ns read access time at 1.2V VDD using single-cap offset-cancelled sense amplifier and in-situ self-write-termination," 2018 IEEE International Solid - State Circuits Conference - (ISSCC), San Francisco, CA, 2018, pp. 480-482, doi: 10.1109/ISSCC.2018.8310393.

[15] P. Jain, U. Arslan, M. Sekhar, B. Lin, Liqiong Wei, Tanaya Sahu, Juan Alzate-vinasco, Ajay Vangapaty, M. Meterelliyoz, N. Strutt, Albert B. Chen, P. Hentges, P. Quintero, C. Connor,

O. Golonzka, K. Fischer, F. Hamzaoglu, "13.2 A 3.6Mb 10.1Mb/mm2 Embedded Non-Volatile ReRAM Macro in 22nm FinFET Technology with Adaptive Forming/Set/Reset Schemes Yielding Down to 0.5V with Sensing Time of 5ns at 0.7V," 2019 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2019, pp. 212-214, doi: 10.1109/ISSCC.2019.8662393.

[16] Intel Optane Memory, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html, accessed on 01/12/2021.

[17] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16). IEEE Press, 27–39. DOI:https://doi.org/10.1109/ISCA.2016.13.

[18] L. Song, X. Qian, H. Li and Y. Chen, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, 2017, pp. 541-552, doi: 10.1109/HPCA.2017.55.

[19] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. Nature 409, 860–921 (2001). https://doi.org/10.1038/35057062.

[20] K. A. Wetterstrand, “DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)”, [Online]. Available: https//www.genome.gov/sequencingcostsdata, Accessed on: May 20, 2020.

[21] A. Grada, and K. Weinbrecht, “Next-generation sequencing: methodology and application.” The Journal of investigative dermatology vol. 133,8 (2013): e11. doi:10.1038/jid.2013.248.

[22] M. A. Hamburg and F. S. Collins, “The path to personalized medicine,” N Engl J Med, vol. 2010, no. 363, pp. 301–304, 2010.

[23] Erin D. Pleasance, R. Keira Cheetham, Philip J. Stephens, David J. McBride, Sean J. Humphray, Chris D. Greenman, Ignacio Varela, Meng-Lay Lin, Gonzalo R. Ordóñez, Graham R. Bignell, Kai Ye, Julie Alipaz, Markus J. Bauer, David Beare, Adam Butler, Richard J. Carter, Lina Chen, Anthony J. Cox, Sarah Edkins, Paula I. Kokko-Gonzales, Niall A. Gormley, Russell J. Grocock, Christian D. Haudenschild, Matthew M. Hims, Terena James, Mingming Jia, Zoya Kingsbury, Catherine Leroy, John Marshall, Andrew Menzies, Laura J. Mudie, Zemin Ning, Tom Royce, Ole B. Schulz-Trieglaff, Anastassia Spiridou, Lucy A. Stebbings, Lukasz Szajkowski, Jon Teague, David Williamson, Lynda Chin, Mark T. Ross, Peter J. Campbell, David R. Bentley, P. Andrew Futreal, Michael R. Stratton, A comprehensive catalogue of somatic mutations from a human cancer genome. Nature 463, 191–196 (2010). https://doi.org/10.1038/nature08658.

[24] Lacour A, Espinosa A, Louwersheimer E, Heilmann S, Hernández I, Wolfsgruber S, Fernández V, Wagner H, Rosende-Roca M, Mauleón A, Moreno-Grau S, Vargas L, Pijnenburg YA, Koene T, Rodríguez-Gómez O, Ortega G, Ruiz S, Holstege H, Sotolongo-Grau O, Kornhuber J, Peters O, Frölich L, Hüll M, Rüther E, Wiltfang J, Scherer M, Riedel-Heller S, Alegret M, Nöthen MM, Scheltens P, Wagner M, Tárraga L, Jessen F, Boada M, Maier W, van der Flier WM, Becker T, Ramirez A, Ruiz A. Genome-wide significant risk factors for Alzheimer's disease: role in progression to dementia due to Alzheimer's disease

among subjects with mild cognitive impairment. Mol Psychiatry. 2017 Jan;22(1):153-160. doi: 10.1038/mp.2016.18. Epub 2016 Mar 15. PMID: 26976043; PMCID: PMC5414086.

[25] An Introduction to Next-Generation Sequencing Technology, Illumina, Inc. [Online]. Available: https://www.illumina.com/content/ dam/illumina-marketing/documents/products/illumina_sequencing_in troduction.pdf. Accessed on: July 15, 2020.

[26] Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, Gene E. Robinson, (2015) Big Data: Astronomical or Genomical?. PLOS Biology 13(7): e1002195. https://doi.org/10.1371/journal.pbio.1002195.

[27] F. Rosenblatt, "The Perceptron—a perceiving and recognizing automaton," Report, 85-460-1. Cornell Aeronautical Laboratory, 1957.

[28] J. Deng, W. Dong, R. Socher, L. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12). Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[30] K. Simonyan, A. Zisserman, "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[31] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[32] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size." arXiv preprint arXiv:1602.07360 (2016).

[33] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, Liang-Chieh Chen; Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 4510-4520.

[34] Song Han, Huizi Mao, William J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." arXiv preprint arXiv:1510.00149 (2015).

[35] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'14). MIT Press, Cambridge, MA, USA, 1269–1277.

[36] Y. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127-138, Jan. 2017, doi: 10.1109/JSSC.2016.2616357.

[37] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim and H. Yoo, "UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," 2018 IEEE

International Solid - State Circuits Conference - (ISSCC), 2018, pp. 218-220, doi: 10.1109/ISSCC.2018.8310262.

[38] Suyoung Bang, Jingcheng Wang, Ziyun Li, Cao Gao, Yejoong Kim, Qing Dong, Yen-Po Chen, Laura Fick, Xun Sun, Ron Dreslinski, Trevor Mudge, Hun Seok Kim, David Blaauw, Dennis Sylvester, "14.7 A 288μW programmable deep-learning processor with 270KB on-chip weight storage using non-uniform memory hierarchy for mobile intelligence," 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, 2017, pp. 250-251, doi: 10.1109/ISSCC.2017.7870355.

[39] B. Moons and M. Verhelst, "A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets," 2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits), Honolulu, HI, 2016, pp. 1-2, doi: 10.1109/VLSIC.2016.7573525.

[40] Paul Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, G.-Y. Wei, "14.3 A 28nm SoC with a 1.2GHz 568nJ/prediction sparse deep-neural-network engine with >0.1 timing error rate tolerance for IoT applications," 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, 2017, pp. 242-243, doi: 10.1109/ISSCC.2017.7870351.

[41] Zhe Yuan, Jinshan Yue, Huanrui Yang, Zhibo Wang, Jinyang Li, Yixiong Yang, Qingwei Guo, Xueqing Li, M. Chang, H. Yang, Yongpan Liu, "Sticker: A 0.41-62.1 TOPS/W 8Bit Neural Network Processor with Multi-Sparsity Compatible Convolution Arrays and Online Tuning Acceleration for Fully Connected Layers," 2018 IEEE Symposium on VLSI Circuits, Honolulu, HI, 2018, pp. 33-34, doi: 10.1109/VLSIC.2018.8502404.

[42] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 9, no. 2, pp. 292-308, June 2019, doi: 10.1109/JETCAS.2019.2910232..

[43] Ronald L. Rivest, Adi Shamir and Leonard M. Adleman, “Cryptographic communications system and method,” U.S. Patent 4 405 829A, Dec. 14, 1977.

[44] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM J. Comput. 26, 5 (Oct. 1997), 1484–1509. DOI:https://doi.org/10.1137/S0097539795293172

[45] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher,

Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, John M. Martinis, Quantum supremacy using a programmable superconducting processor. Nature 574, 505–510 (2019). https://doi.org/10.1038/s41586-019-1666-5.

[46] NIST Post Quantum Cryptography Round 3 Finalist, https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions, accessed on 01/13/2021.

[47] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. J. ACM 56, 6, Article 34 (September 2009), 40 pages. DOI:https://doi.org/10.1145/1568318.1568324

[48] C. Gentry. 2009. "Fully homomorphic encryption using ideal lattices. In Proceedings of the forty-first annual ACM symposium on Theory of computing (STOC '09). Association for Computing Machinery, New York, NY, USA, 169–178. DOI:https://doi.org/10.1145/1536414.1536440

[49] Gentry C., Halevi S. (2011) Implementing Gentry's Fully-Homomorphic Encryption Scheme. In: Paterson K.G. (eds) Advances in Cryptology – EUROCRYPT 2011. EUROCRYPT 2011. Lecture Notes in Computer Science, vol 6632. Springer, Berlin, Heidelberg

[50] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12). Association for Computing Machinery, New York, NY, USA, 309–325. DOI:https://doi.org/10.1145/2090236.2090262

[51] Brakerski Z. (2012) Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In: Safavi-Naini R., Canetti R. (eds) Advances in Cryptology – CRYPTO 2012. CRYPTO 2012. Lecture Notes in Computer Science, vol 7417. Springer, Berlin, Heidelberg

[52] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144

[53] Cheon, Jung Hee; Kim, Andrey; Kim, Miran; Song, Yongsoo (2017). "Homomorphic encryption for arithmetic of approximate numbers". Takagi T., Peyrin T. (eds) Advances in Cryptology – ASIACRYPT 2017. ASIACRYPT 2017. Springer, Cham. pp. 409–437. doi:10.1007/978-3-319-70694-8_15.

[54] N. P. Smart, F. Vercauteren, Fully homomorphic SIMD operations. Des. Codes Cryptogr. 71, 57–81 (2014). https://doi.org/10.1007/s10623-012-9720-4.

[55] PALISADE, https://palisade-crypto.org/, accessed on 01/13/2020

[56] Halevi S., Shoup V. (2014) Algorithms in HElib. In: Garay J.A., Gennaro R. (eds) Advances in Cryptology – CRYPTO 2014. CRYPTO 2014. Lecture Notes in Computer Science, vol 8616. Springer, Berlin, Heidelberg

[57] Microsoft SEAL (release 3.5), https://github.com/Microsoft/SEAL.

[58] Gentry C., Sahai A., Waters B. (2013) Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In: Canetti R., Garay J.A. (eds) Advances in Cryptology – CRYPTO 2013. CRYPTO 2013. Lecture Notes in Computer Science, vol 8042. Springer, Berlin, Heidelberg.

[59] Ducas L., Micciancio D. (2015) FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In: Oswald E., Fischlin M. (eds) Advances in Cryptology -- EUROCRYPT

2015. EUROCRYPT 2015. Lecture Notes in Computer Science, vol 9056. Springer, Berlin, Heidelberg.

[60] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène, TFHE: Fast Fully Homomorphic Encryption Over the Torus. J Cryptol 33, 34–91 (2020). https://doi.org/10.1007/s00145-019-09319-x.

[61] Van der Auwera GA, Carneiro MO, Hartl C, Poplin R, Del Angel G, Levy-Moonshine A, Jordan T, Shakir K, Roazen D, Thibault J, Banks E, Garimella KV, Altshuler D, Gabriel S, DePristo MA. From FastQ data to high confidence variant calls: the Genome Analysis Toolkit best practices pipeline. Curr Protoc Bioinformatics. 2013;43(1110):11.10.1-11.10.33. doi: 10.1002/0471250953.bi1110s43. PMID: 25431634; PMCID: PMC4243306.

[62] H. Li, and Richard Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform." Bioinformatics (Oxford, England) vol. 26, no. 5, pp. 589-95, 2010. doi:10.1093/bioinformatics/btp698.

[63] B. Langmead and S. Salzberg, "Fast gapped-read alignment with Bowtie 2". Nature Methods 9, pp. 357–359, 2012. https://doi.org/ 10.1038/nmeth.1923.

[64] Li R, Yu C, Li Y, Lam TW, Yiu SM, Kristiansen K, Wang J. SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics. 2009 Aug 1;25(15):1966-7. doi: 10.1093/bioinformatics/btp336. Epub 2009 Jun 3. PMID: 19497933.

[65] McKenna A, Hanna M, Banks E, Sivachenko A, Cibulskis K, Kernytsky A, Garimella K, Altshuler D, Gabriel S, Daly M, DePristo MA. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. Genome Res. 2010 Sep;20(9):1297-303. doi: 10.1101/gr.107524.110. Epub 2010 Jul 19. PMID: 20644199; PMCID: PMC2928508.

[66] Reinert K, Dadi TH, Ehrhardt M, Hauswedell H, Mehringer S, Rahn R, Kim J, Pockrandt C, Winkler J, Siragusa E, Urgese G, Weese D. The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. J Biotechnol. 2017 Nov 10;261:157-168. doi: 10.1016/j.jbiotec.2017.07.017. Epub 2017 Sep 6. PMID: 28888961.

[67] Zhao M, Lee WP, Garrison EP, Marth GT (2013) SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications. PLOS ONE 8(12): e82138. https://doi.org/10.1371/journal.pone.0082138

[68] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," Proceedings 41st Annual Symposium on Foundations of Computer Science, Redondo Beach, CA, USA, 2000, pp. 390-398, doi: 10.1109/SFCS.2000.892127.

[69] M. Burrows, D.J. Wheeler, "A block sorting lossless data compression algorithm," Technical Report 124, Digital Equipment Corporation, 1994.

[70] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," Journal of Molecular Biology, vol. 147, no.1, pp. 195–197, 1981.

[71] Y. Liu and B. Schmidt, "CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing," IEEE Design & Test, vol. 31, no. 1, pp. 31-39, Feb. 2014, doi: 10.1109/MDAT.2013.2284198.

[72] Wilton R, Budavari T, Langmead B, Wheelan SJ, Salzberg SL, Szalay AS. Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search

space. PeerJ. 2015 Mar 3;3:e808. doi: 10.7717/peerj.808. PMID: 25780763; PMCID: PMC4358639.

[73] H. M. Waidyasooriya and M. Hariyama, “Hardware-Acceleration of Short-Read Alignment Based on the Burrows-Wheeler Transform,” IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 5, pp. 1358-1372, 1 May 2016, doi: 10.1109/TPDS.2015.2444376.

[74] J. Arram, K. H. Tsoi, W. Luk and P. Jiang, "Reconfigurable Acceleration of Short Read Mapping," 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines, 2013, pp. 210-217, doi: 10.1109/FCCM.2013.57.

[75] E. J. Houtgast, V. Sima, K. Bertels and Z. Al-Ars, "An FPGA-based systolic array to accelerate the BWA-MEM genomic mapping algorithm," 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015, pp. 221-227, doi: 10.1109/SAMOS.2015.7363679.

[76] Y. Wu, J. Hung and C. Yang, “14.8 A 135mW fully integrated data processor for next-generation sequencing,” 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, 2017, pp. 252-253, doi: 10.1109/ISSCC.2017.7870356.

[77] Daichi Fujiki, Aran Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. Genax: a genome sequencing accelerator. In Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18). IEEE Press, 69–82. DOI:https://doi.org/10.1109/ISCA.2018.00017.

[78] Zhehong Wang, T. Zhang, Daichi Fujiki, Arun Subramaniyan, Xiao Wu, M. Yasuda, S. Miyoshi, Masaru Kawaminami, R. Das, S. Narayanasamy, D. Blaauw, “A 2.46M reads/s Genome Sequencing Accelerator using a 625 Processing-Element Array,” IEEE Custom Integrated Circuits Conference (CICC), Boston, MA, USA, 2020, pp. 1-4, doi: 10.1109/CICC48029.2020.9075900.

[79] Okada, D., Ino, F., and Hagihara, K. Accelerating the Smith-Waterman algorithm with interpair pruning and band optimization for the all-pairs comparison of base sequences. BMC Bioinformatics 16, 321 (2015). https://doi.org/10.1186/s12859-015-0744-4.

[80] K. Schulz and S. Mihov, “Fast string correction with Levenshtein automata”. IJDAR vol. 5, pp. 67–85, Nov. 2002. https://doi.org/ 10.1007/s10032-002-0082-8.

[81] O. Gotoh, “Optimal Sequence Alignment Allowing For Long Gaps,” Bulletin of Mathematical Biology, vol. 52, no. 3, pp. 359–373, 1990.

[82] Eberle MA, Fritzilas E, Krusche P, Källberg M, Moore BL, Bekritsky MA, Iqbal Z, Chuang HY, Humphray SJ, Halpern AL, Kruglyak S, Margulies EH, McVean G, Bentley DR. A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree. Genome Res. 2017 Jan;27(1):157-164. doi: 10.1101/gr.210500.116. Epub 2016 Nov 30. PMID: 27903644; PMCID: PMC5204340.

[83] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, Dongjun Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications." arXiv preprint arXiv:1511.06530 (2015).

[84] Z. Li et. Al., “Energy-Efficient, Mobile Computer Vision and Machine Learning Processors.” deepblue preprint 2027.42/151423 (2019).

[85] M. Chang, Yu-Fan Lin, Yen-Chen Liu, J. Wu, S. Shen, W. Tsai, Y. Chih, "An Asymmetric-Voltage-Biased Current-Mode Sensing Scheme for Fast-Read Embedded Flash Macros," in

IEEE Journal of Solid-State Circuits, vol. 50, no. 9, pp. 2188-2198, Sept. 2015, doi: 10.1109/JSSC.2015.2424972.

[86] S. Yu and P. Chen, "Emerging Memory Technologies: Recent Trends and Prospects," in IEEE Solid-State Circuits Magazine, vol. 8, no. 2, pp. 43-56, Spring 2016, doi: 10.1109/MSSC.2016.2546199.

[87] Chieh-Pu Lo, Wen-Zhang Lin, Wei-Yu Lin, Huan-Ting Lin, Tzu-Hsien Yang, Yen-Ning Chiang, Y. King, C. Lin, Y. Chih, T. Chang, M. Chang, "A ReRAM Macro Using Dynamic Trip-Point-Mismatch Sampling Current-Mode Sense Amplifier and Low-DC Voltage-Mode Write-Termination Scheme Against Resistance and Write-Delay Variation," in IEEE Journal of Solid-State Circuits, vol. 54, no. 2, pp. 584-595, Feb. 2019, doi: 10.1109/JSSC.2018.2873588.

[88] Wu, Tony F; Le, Binh Q; Radway, Robert; Bartolo, Andrew; Hwang, William; Jeong, Seungbin; Li, Haitong; Tandon, Pulkit; Vianello, Elisa; Vivet, Pascal; Nowak, Etienne; Wootters, Mary K.; Wong, Philip H.-S.; Mohamed M. Sabry Aly; Beigne, Edith; Mitra, Subhasish, "14.3 A 43pJ/Cycle Non-Volatile Microcontroller with 4.7µs Shutdown/Wake-up Integrating 2.3-bit/Cell Resistive RAM and Resilience Techniques," 2019 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2019, pp. 226-228, doi: 10.1109/ISSCC.2019.8662402.

[89] Lee, Seung Ryul, Young-Bae Kim, Man Chang, Kyung Min Kim, Chang Bum Lee, Ji-Hyun Hur, Gyeong Su Park, Dongsoo Lee, Myoung-Jae Lee, Chang Jung Kim, U-In Chung, In-kyeong Yoo and Kinam Kim, "Multi-level switching of triple-layered TaOx RRAM with excellent reliability for storage class memory," 2012 Symposium on VLSI Technology (VLSIT), Honolulu, HI, 2012, pp. 71-72, doi: 10.1109/VLSIT.2012.6242466.

[90] Wei, Liqiong, Juan G. Alzate, Umut Arslan, Justin Brockman, Nilanjan Das, Kevin Fischer, Tahir Ghani, Oleg Golonzka, Patrick Hentges, Rawshan Jahan, Pulkit Jain, Blake C. Lin, Mesut Meterelliyoz, James A. O'Donnell, Conor Puls, Pedro A. Quintero, Tanaya Sahu, Meenakshi Sekhar, Ajay Vangapaty, Chris Wiegand and Fatih Hamzaoglu, "13.3 A 7Mb STT-MRAM in 22FFL FinFET Technology with 4ns Read Sensing Time at 0.9V Using Write-Verify-Write Scheme and Offset-Cancellation Sensing Technique," 2019 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2019, pp. 214-216, doi: 10.1109/ISSCC.2019.8662444.

[91] Qing Dong, Zhehong Wang, Jongyup Lim, Yiqun Zhang, M. Sinangil, Y. Shih, Y. Chih, Jonathan Chang, D. Blaauw, D. Sylvester, "A 1-Mb 28-nm 1T1MTJ STT-MRAM With Single-Cap Offset-Cancelled Sense Amplifier and In Situ Self-Write-Termination," in IEEE Journal of Solid-State Circuits, vol. 54, no. 1, pp. 231-239, Jan. 2019, doi: 10.1109/JSSC.2018.2872584.

[92] G. Sandre, Luca Bettini, A. Pirola, Lionel Marmonier, M. Pasotti, M. Borghi, P. Mattavelli, P. Zuliani, L. Scotti, G. Mastracchio, F. Bedeschi, R. Gastaldi, R. Bez, "A 90nm 4Mb embedded phase-change memory with 1.2V 12ns read access time and 1MB/s write throughput," 2010 IEEE International Solid-State Circuits Conference - (ISSCC), San Francisco, CA, 2010, pp. 268-269, doi: 10.1109/ISSCC.2010.5433911.

[93] Xue, Cheng-Xin; Chang, Ting-Wei; Chang, Tung-Cheng; Kao, Hui-Yao; Chiu, Yen-Cheng; Lee, Chun-Ying; King, Ya-Chin; Lin, Chrong-Jung; Liu, Ren-Shuo; Hsieh, Chih-Cheng;

Tang, Kea-Tiong; Chen, Wei-Hao; Chang, Meng-Fan; Liu, Je-Syu; Li, Jia-Fang; Lin, Wei-Yu; Lin, Wei-En; Wang, Jing-Hong; Wei, Wei-Chen; Huang, Tsung-Yuan, "Embedded 1-Mb ReRAM-Based Computing-in- Memory Macro With Multibit Input and Weight for CNN-Based AI Edge Processors," in IEEE Journal of Solid-State Circuits, vol. 55, no. 1, pp. 203-215, Jan. 2020, doi: 10.1109/JSSC.2019.2951363.

[94] Chen, Wei-Hao; Li, Kai-Xiang; Lin, Wei-Yu; Hsu, Kuo-Hsiang; Li, Pin-Yi; Yang, Cheng-Han; Xue, Cheng-Xin; Yang, En-Yu; Chen, Yen-Kai; Chang, Yun-Sheng; Hsu, Tzu-Hsiang; King, Ya-Chin; Lin, Chorng-Jung; Liu, Ren-Shuo; Hsieh, Chih-Cheng; Tang, Kea-Tiong; Chang, Meng-Fan, "A 65nm 1Mb nonvolatile computing-in-memory ReRAM macro with sub-16ns multiply-and-accumulate for binary DNN AI edge processors," 2018 IEEE International Solid - State Circuits Conference - (ISSCC), San Francisco, CA, 2018, pp. 494-496, doi: 10.1109/ISSCC.2018.8310400.

[95] Zhehong Wang, Ziyun Li, Li Xu, Qing Dong, Chin-I Su, W. Chu, George Tsou, Y. Chih, T. Chang, D. Sylvester, Hun-Seok Kim, D. Blaauw, "An All-Weights-on-Chip DNN Accelerator in 22nm ULL Featuring 24×1 Mb eRRAM," 2020 IEEE Symposium on VLSI Circuits, Honolulu, HI, USA, 2020, pp. 1-2, doi: 10.1109/VLSICircuits18222.2020.9162811.

[96] Ziyun Li, Y. Chen, Luyao Gong, Lu Liu, D. Sylvester, D. Blaauw, Hun-Seok Kim, "An 879GOPS 243mW 80fps VGA Fully Visual CNN-SLAM Processor for Wide-Range Autonomous Exploration," 2019 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2019, pp. 134-136, doi: 10.1109/ISSCC.2019.8662397.

[97] P. Subhankar, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. "Outerspace: An outer product based sparse matrix multiplication accelerator." In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 724-736. IEEE, 2018.

[98] D, Jia, W. Dong, R. Socher, L. Li, K. Li, and F. Li. "Imagenet: A large-scale hierarchical image database." 2009 IEEE conference on computer vision and pattern recognition, pp. 248-255. 2009.

[99] Chung-Cheng Chou, Zheng-Jun Lin, Pei-Ling Tseng, Chih-Feng Li, Chih-Yang Chang, Wei-Chi Chen, Yu-Der Chih, and Tsung-Yung Jonathan Chang, "An N40 256K×44 embedded RRAM macro with SL-precharge SA and low-voltage current limiter to improve read and write performance," 2018 IEEE International Solid - State Circuits Conference - (ISSCC), San Francisco, CA, 2018, pp. 478-480, doi: 10.1109/ISSCC.2018.8310392.

[100] A. Chen and M. Lin, "Variability of resistive switching memories and its impact on crossbar array performance," 2011 International Reliability Physics Symposium, Monterey, CA, 2011, pp. MY.7.1-MY.7.4, doi: 10.1109/IRPS.2011.5784590.

[101] P. Jain, U. Arslan, M. Sekhar, B. Lin, Liqiong Wei, Tanaya Sahu, Juan Alzate-vinasco, Ajay Vangapaty, M. Meterelliyoz, N. Strutt, Albert B. Chen, P. Hentges, P. Quintero, C. Connor, O. Golonzka, K. Fischer, F. Hamzaoglu, "13.2 A 3.6Mb 10.1Mb/mm2 Embedded Non-Volatile ReRAM Macro in 22nm FinFET Technology with Adaptive Forming/Set/Reset Schemes Yielding Down to 0.5V with Sensing Time of 5ns at 0.7V," 2019 IEEE International Solid- State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2019, pp. 212-214, doi: 10.1109/ISSCC.2019.8662393.

[102] M. Chang, J. Wu, Tun-Fei Chien, Yen-Chen Liu, T. Yang, W. Shen, Y. King, C. Lin, Ku-Feng Lin, Y. Chih, T. Chang, "Low VDDmin Swing-Sample-and-Couple Sense Amplifier and Energy-Efficient Self-Boost-Write-Termination Scheme for Embedded ReRAM Macros Against Resistance and Switch-Time Variations," in IEEE Journal of Solid-State Circuits, vol. 50, no. 11, pp. 2786-2795, Nov. 2015, doi: 10.1109/JSSC.2015.2472601.

[103] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Y. Shao, S. Keckler, Zhengya Zhang, "SNAP: A 1.67 - 21.55TOPS/W Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference in 16nm CMOS," 2019 Symposium on VLSI Circuits, Kyoto, Japan, 2019, pp. C306-C307, doi: 10.23919/VLSIC.2019.8778193.

[104] Moons, Bert; Uytterhoeven, Roel; Dehaene, Wim; Verhelst, Marian, "14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI," 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, 2017, pp. 246-247, doi: 10.1109/ISSCC.2017.7870353.

[105] C. Meadows, "A More Efficient Cryptographic Matchmaking Protocol for Use in the Absence of a Continuously Available Third Party," 1986 IEEE Symposium on Security and Privacy, 1986, pp. 134-134, doi: 10.1109/SP.1986.10022.

[106] B. Pinkas, T. Schneider, and M. Zohner, "Scalable Private Set Intersection Based on OT Extension," ACM Trans. Priv. Secur. 21, 2, Article 7 (February 2018), DOI:https://doi.org/10.1145/3154794.

[107] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. "Efficient Batched Oblivious PRF with Applications to Private Set Intersection," Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 818–829. DOI:https://doi.org/10.1145/2976749.2978381.

[108] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," Proceedings of the nineteenth annual ACM symposium on Theory of computing (STOC '87). Association for Computing Machinery, New York, NY, USA, 182–194. DOI:https://doi.org/10.1145/28395.28416.

[109] S. Devadas, M. van Dijk, C. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: A constant bandwidth blowup oblivious RAM," Theory of Cryptography Conference. Springer, 145–174, Springer, Berlin, Heidelberg, 2016.

[110] H. Chen, K. Laine, and P. Rindal, "Fast Private Set Intersection from Homomorphic Encryption," Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17). Association for Computing Machinery, New York, NY, USA, 1243–1255. DOI:https://doi.org/10.1145/3133956.3134061.

[111] H. Chen, I. Chillotti, and L. Ren, "Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE," 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354226.

[112] P. Longa , M. Naehrig, "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography," Cryptology and Network Security, CANS 2016, Lecture

Notes in Computer Science, vol 10052, Springer, Cham. https://doi.org/10.1007/978-3-319-48965-0_8.
[113] C. Gentry. 2009. "A fully homomorphic encryption scheme. " Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Dan Boneh.
[114] C. Gentry, "Computing Arbitrary Functions of Encrypted Data, " Commun. ACM. 53. 97-105. DOI:https://doi.org/10.1145/1666420.1666444.
[115] V. Lyubashevsky, C. Peikert, O. Regev, "On Ideal Lattices and Learning with Errors over Rings, " Advances in Cryptology – EUROCRYPT 2010. EUROCRYPT 2010. Lecture Notes in Computer Science, vol 6110. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-13190-5_1.
[116] D, Micciancio and Y, Polyakov, "Bootstrapping in FHEW-like Cryptosystems, " Cryptology ePrint Archive: Report 2020/086, URL: https://eprint.iacr.org/2020/086/20200224:014328, accessed on 05/07/2021.
[117] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled PSI from Fully Homomorphic Encryption with Malicious Security," Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). Association for Computing Machinery, New York, NY, USA, 1223–1237. DOI:https://doi.org/10.1145/3243734.3243836.
[118] Y. Arbitman, M. Naor and G. Segev, "Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation," 2010 IEEE 51st Annual Symposium on Foundations of Computer Science, 2010, pp. 787-796, doi: 10.1109/FOCS.2010.80.
[119] S. Song, W. Tang, T. Chen and Z. Zhang, "LEIA: A 2.05mm2 140mW lattice encryption instruction accelerator in 40nm CMOS," 2018 IEEE Custom Integrated Circuits Conference (CICC), 2018, pp. 1-4, doi: 10.1109/CICC.2018.8357070.
[120] I. Yoon, N. Cao, A. Amaravati and A. Raychowdhury, "A 55nm 50nJ/encode 13nJ/decode Homomorphic Encryption Crypto-Engine for IoT Nodes to Enable Secure Computation on Encrypted Data," 2019 IEEE Custom Integrated Circuits Conference (CICC), 2019, pp. 1-4, doi: 10.1109/CICC.2019.8780277.
[121] U. Banerjee, A. Pathak and A. P. Chandrakasan, "2.3 An Energy-Efficient Configurable Lattice Cryptography Processor for the Quantum-Secure Internet of Things," 2019 IEEE International Solid- State Circuits Conference - (ISSCC), 2019, pp. 46-48, doi: 10.1109/ISSCC.2019.8662528.
[122] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, HEAX: An Architecture for Computing on Encrypted Data. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 1295–1309. DOI:https://doi.org/10.1145/3373376.3378523.
[123] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren and I. Verbauwhede, "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data," 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019, pp. 387-398, doi: 10.1109/HPCA.2019.00052.
[124] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren and I. Verbauwhede, "HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation,"

in IEEE Transactions on Computers, vol. 67, no. 11, pp. 1637-1650, 1 Nov. 2018, doi: 10.1109/TC.2018.2816640.

[125] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine and G. Gogniat, "Hardware/Software Co-Design of an Accelerator for FV Homomorphic Encryption Scheme Using Karatsuba Algorithm," in IEEE Transactions on Computers, vol. 67, no. 3, pp. 335-347, 1 March 2018, doi: 10.1109/TC.2016.2645204.

[126] Vincent Migliore, Cédric Seguin, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, Guy Gogniat, and Russell Tessier. 2017. A High-Speed Accelerator for Homomorphic Encryption using the Karatsuba Algorithm. ACM Trans. Embed. Comput. Syst. 16, 5s, Article 138 (October 2017), 17 pages. DOI:https://doi.org/10.1145/3126558.

[127] C. Jayet-Griffon, M. -. Cornelie, P. Maistri, P. Elbaz-Vincent and R. Leveugle, "Polynomial multipliers for fully homomorphic encryption on FPGA," 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2015, pp. 1-6, doi: 10.1109/ReConFig.2015.7393335.

[128] Y. Doröz, E. Öztürk and B. Sunar, "Accelerating Fully Homomorphic Encryption in Hardware," in IEEE Transactions on Computers, vol. 64, no. 6, pp. 1509-1521, 1 June 2015, doi: 10.1109/TC.2014.2345388.

[129] F. Turan, S. S. Roy and I. Verbauwhede, "HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA," in IEEE Transactions on Computers, vol. 69, no. 8, pp. 1185-1196, 1 Aug. 2020, doi: 10.1109/TC.2020.2988765.

[130] Zhehong Wang, T. Zhang, Daichi Fujiki, Arun Subramaniyan, Xiao Wu, M. Yasuda, S. Miyoshi, Masaru Kawaminami, R. Das, S. Narayanasamy, D. Blaauw, "A 2.46M Reads/s Seed-Extension Accelerator for Next-Generation Sequencing Using a String-Independent PE Array," in IEEE Journal of Solid-State Circuits, vol. 56, no. 3, pp. 824-833, March 2021, doi: 10.1109/JSSC.2020.3023822.

[131] Ziyun Li, Zhehong Wang, Li Xu, Qing Dong, Bowen Liu, Chin-I Su, Wen-Ting Chu, George Tsou, Yu-Der Chih, Tsung-Yung Jonathan Chang, Dennis Sylvester, Hun-Seok Kim, David Blaauw, "RRAM-DNN: An RRAM and Model-Compression Empowered All-Weights-On-Chip DNN Accelerator," in IEEE Journal of Solid-State Circuits, vol. 56, no. 4, pp. 1105-1115, April 2021, doi: 10.1109/JSSC.2020.3045369.

[132] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli & Demis Hassabis, Highly accurate protein structure prediction with AlphaFold. Nature 596, 583–589 (2021). https://doi.org/10.1038/s41586-021-03819-2.

[133] B. A. Huberman, M. Franklin, and T. Hogg, “Enhancing privacy and trust in electronic communities,” In Proceedings of the 1st ACM conference on Electronic commerce (EC '99). Association for Computing Machinery, New York, NY, USA, 78–86. DOI:https://doi.org/10.1145/336992.337012.

[134] B. Pinkas, T. Schneider, and M. Zohner, “Faster private set intersection based on OT extension,” In Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14). USENIX Association, USA, 797–812.
[135] M. Orrù., E. Orsini, P. Scholl, “Actively Secure 1-out-of-N OT Extension with Application to Private Set Intersection,” In: Handschuh H. (eds) Topics in Cryptology – CT-RSA 2017. CT-RSA 2017. Lecture Notes in Computer Science, vol 10159. Springer, Cham. https://doi.org/10.1007/978-3-319-52153-4_22.
[136] Z. Wang, “Hardware Acceleration for PSI with Third Generation Fully Homomorphic Encryption,” https://github.com/zhehongw/3rd_Gen_FHE_ACC, 2021.
[137] P. Barrett, “Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor,” In: Odlyzko A.M. (eds) Advances in Cryptology — CRYPTO’ 86. CRYPTO 1986. Lecture Notes in Computer Science, vol 263. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-47721-7_24.
[138] P. Longa , M. Naehrig, "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography," Cryptology and Network Security, CANS 2016, Lecture Notes in Computer Science, vol 10052, Springer, Cham. https://doi.org/10.1007/978-3-319-48965-0_8.
[139] M. R. Albrecht, R. Player and S. Scott, “On the concrete hardness of Learning with Errors,” Journal of Mathematical Cryptology. Volume 9, Issue 3, Pages 169–203, ISSN (Online) 1862-2984, ISSN (Print) 1862-2976 DOI: 10.1515/jmc-2015-0016, October 2015.
[140] Hao Chen, Ran Gilad-Bachrach, Kyoohyung Han, Zhicong Huang, Amir Jalali, Kim Laine, Kristin Lauter, Logistic regression over encrypted data from fully homomorphic encryption. BMC Med Genomics 11, 81 (2018). https://doi.org/10.1186/s12920-018-0397-z.
[141] Yichen Jiang, Jenny Hamer, Chenghong Wang, Xiaoqian Jiang, Miran Kim, Yongsoo Song, Yuhou Xia, N. Mohammed, Md. Nazmus Sadat, Shuang Wang, "SecureLR: Secure Logistic Regression Model via a Hybrid Cryptographic Protocol," in IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 16, no. 1, pp. 113-123, 1 Jan.-Feb. 2019, doi: 10.1109/TCBB.2018.2833463.
[142] F. Bourse, M. Minelli, M. Minihold , P. Paillier, “Fast Homomorphic Evaluation of Deep Discretized Neural Networks,” In: Shacham H., Boldyreva A. (eds) Advances in Cryptology – CRYPTO 2018. CRYPTO 2018. Lecture Notes in Computer Science, vol 10993. Springer, Cham. https://doi.org/10.1007/978-3-319-96878-0_17.
[143] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, „CryptoNets: applying neural networks to encrypted data with high throughput and accuracy,” In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16). JMLR.org, 201–210.
[144] J. W. Bos, W. Castryck, I. Iliashenko, F. Vercauteren, “Privacy-Friendly Forecasting for the Smart Grid Using Homomorphic Encryption and the Group Method of Data Handling,” In: Joye M., Nitaj A. (eds) Progress in Cryptology - AFRICACRYPT 2017. AFRICACRYPT 2017. Lecture Notes in Computer Science, vol 10239. Springer, Cham. https://doi.org/10.1007/978-3-319-57339-7_11.