On designing hardware accelerator-based systems: interfaces, taxes and benefits

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

ON DESIGNING HARDWARE ACCELERATOR-BASED SYSTEMS: INTERFACES, TAXES AND BENEFITS

by

ZAHRA AZAD

B.S., Shahid Beheshti University, Iran, 2014
M.S., Sharif University of Technology, Iran, 2016

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2023

Approved by

First Reader

Ajay J. Joshi, PhD
Professor of Electrical and Computer Engineering

Second Reader

Ayse K. Coskun, PhD
Professor of Electrical and Computer Engineering

Third Reader

Rabia Yazicigil, PhD
Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Tali Moreshet, PhD
Senior Lecturer & Research Assistant Professor of Electrical and Computer Engineering

Acknowledgments

I would like to begin by expressing my gratitude to my advisor, Prof. Ajay Joshi, for his continuous guidance, support, and encouragement throughout my PhD journey. I am also thankful to my thesis committee members, Prof. Ayse Coskun, Prof. Tali Moreshet, and Prof. Rabia Yazicigil, for their invaluable feedback, generous support, and precious time. I am extremely grateful to all my collaborators and co-authors, especially Prof. Michael Taylor, Prof. Vijay Reddi, Dr. Rathijit Sen, Dr. Kwanghyun Park, Dr. Leila Delshad, Dr. Rashmi Agrawal, Daniel Petrisko, Guowei Yang, and Michael Buch, for their contributions, expertise, and insights, which have significantly enhanced the quality of my research. Additionally, I would like to express my appreciation to all my friends in the ICSG research group, Peac Lab research group, and CAAD research group for their support and encouragement. I also want to thank my friends at Boston University, especially Marcia, Saiful, Furkan, Chathura, Prachi, Zihao, Sahan, and Pouya, for their friendship, which has made my stay at the university more enjoyable. I am deeply grateful to my mother and sisters for their constant love, support, and encouragement throughout my academic journey. Despite being unable to visit them during my PhD, their unwavering emotional support and motivation have been invaluable to me. Finally, I would like to pay tribute to my father who is no longer with us, but whose memory and values serve as a constant source of inspiration in my life.

ON DESIGNING HARDWARE ACCELERATOR-BASED SYSTEMS: INTERFACES, TAXES AND BENEFITS

ZAHRA AZAD

Boston University, College of Engineering, 2023

Major Professor: Ajay J. Joshi, PhD
Professor of Electrical and Computer Engineering

ABSTRACT

Complementary Metal Oxide Semiconductor (CMOS) Technology scaling has slowed down. One promising approach to sustain the historic performance improvement of computing systems is to utilize hardware accelerators. Today, many commercial computing systems integrate one or more accelerators, with each accelerator optimized to efficiently execute specific tasks.

Over the years, there has been a substantial amount of research on designing hardware accelerators for machine learning (ML) training and inference tasks. Hardware accelerators are also widely employed to accelerate data privacy and security algorithms. In particular, there is currently a growing interest in the use of hardware accelerators for accelerating homomorphic encryption (HE) based privacy-preserving computing.

While the use of hardware accelerators is promising, a realistic end-to-end evaluation of an accelerator when integrated into the full system often reveals that the benefits of an accelerator are not always as expected. Simply assessing the performance of the accelerated portion of an application, such as the inference kernel in ML applications, during performance analysis can be misleading. When designing an accelerator-based system, it is critical to evaluate the system as a whole and account for all the accelerator taxes.

In the first part of our research, we highlight the need for a holistic, end-to-end analysis of workloads using ML and HE applications. Our evaluation of an ML application for a database management system (DBMS) shows that the benefits of offloading ML inference to accelerators depend on several factors, including backend hardware, model complexity, data size, and the level of integration between the ML inference pipeline and the DBMS. We also found that the end-to-end performance improvement is bottlenecked by data retrieval and pre-processing, as well as inference. Additionally, our evaluation of an HE video encryption application shows that while HE client-side operations, i.e., message-to-ciphertext and ciphertext-to-message conversion operations, are bottlenecked by number theoretic transform (NTT) operations, accelerating NTT in hardware alone is not sufficient to get enough application throughput (frame rate per second) improvement. We need to address all bottlenecks such as error sampling, encryption, and decryption in message-to-ciphertext and ciphertext-to-message conversion pipeline.

In the second part of our research, we address the lack of a scalable evaluation infrastructure for building and evaluating accelerator-based systems. To solve this problem, we propose a robust and scalable software-hardware framework for accelerator evaluation, which uses an open-source RISC-V based System-on-Chip (SoC) design called BlackParrot. This framework can be utilized by accelerator designers and system architects to perform an end-to-end performance analysis of coherent and non-coherent accelerators while carefully accounting for the interaction between the accelerator and the rest of the system.

In the third part of our research, we present RISE, which is a full RISC-V SoC designed to efficiently perform message-to-ciphertext and ciphertext-to-message conversion operations. RISE comprises of a BlackParrot core and an efficient custom-designed accelerator tailored to accelerate end-to-end message-to-ciphertext and ciphertext-to-message conversion operations. Our RTL-based evaluation demonstrates that RISE improves the throughput of the video encryption application by 10$\times$-27$\times$ for different frame resolutions.

Contents

1 Introduction 1

1.1 Motivation 1

1.1.1 Tax for AI operations 3

1.1.2 Tax for HE Computing 12

1.2 Thesis Contribution 22

1.2.1 BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs 22

1.2.2 RISE: RISC-V SoC for HE Client-Side Operations Acceleration . . 25

1.3 Organization 27

2 BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs 28

2.1 RISC-V Instruction Set Architecture 28

2.2 Related Work 29

2.2.1 Open-Source Software-Level simulator 29

2.2.2 Open-Source Hardware-Level Platforms 30

2.3 BlackParrot: Tile-Based Micro-Architecture 32

2.3.1 Core Tile 33

2.3.2 L2 Extension Tile 35

2.3.3 Accelerator Tiles 35

2.4 BlackParrot: Networks-On-Chip 37

2.4.1 BedRock Network 38

2.4.2 DRAM Network 38

2.4.3 I/O Network 39
2.5 BlackParrot Software 39
2.5.1 Baremetal and Linux Libraries 39
2.5.2 Cosimulation Testing Framework 40
2.5.3 ParrotLine: BlackParrot Microsoft Zipline Accelerator Integration . 41
2.6 Summary 43

3 RISE: RISC-V SoC for HE Client-Side Operations Acceleration 44
3.1 Background 44
3.1.1 Homomorphic Encryption and Ring-Learning with Errors 44
3.1.2 Residue Number System 46
3.1.3 Number Theoretic Transform 46
3.2 Related Work 47
3.3 RISE: System View 49
3.4 RISE: Micro-Architecture 49
3.4.1 Error Sampling Unit 51
3.4.2 Encryption and Decryption Unit 54
3.5 Evaluation 64
3.5.1 Experimental Setup 64
3.5.2 Experimental Results 65
3.6 Summary 76

4 Summary and Future Work 78
4.1 Summary of Major Contributions 78
4.1.1 AI and HE Tax 78
4.1.2 BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs 80
4.1.3 RISE: RISC-V SoC for HE Edge Side Operations Acceleration . . 81

4.2 Future Research Directions 82

4.2.1 Runtime Task Scheduler in Accelerator-based Systems 82

4.2.2 In-Pipeline Hardware Accelerators 84

4.2.3 Coherent Accelerators Using Compute Express Link (CXL) 86

4.3 Final Remarks 87

References 88

Curriculum Vitae 98

List of Tables

3.1 NTT operation Performance comparison with the state-of-the-art designs. . 69

List of Figures

1·1 High-level architecture of ML inference in SQL Server with Python. 6

1·2 Speedups of the best-performing hardware over CPU. 8

1·3 Inference latency for different models on CPU, GPU, and FPGA. 10

1·4 End-to-end T-SQL query latency breakdown 11

1·5 Latency breakdown of HE client-side operations. 18

1·6 Throughput for software execution and NTT hardware acceleration. 20

2·1 BlackParrot multi-core SoC. 32

2·2 BlackParrot core micro-architecture. 34

2·3 Zipline integration. 41

3·1 System-level view of RISE. 50

3·2 Unified encryption and decryption dataflow. 53

3·3 RISE memory reuse technique. 58

3·4 RISE memory reorder technique. 60

3·5 RISE latency results. 66

3·6 RISE power and area results. 68

3·7 RISE area efficiency results. 71

3·8 RISE energy efficiency results. 73

3·9 RISE frame rate per second results. 75

4·1 The best-performing hardware for ML inference. 83

List of Abbreviations

ADP	Area-Delay Product
AI	Artificial Intelligence
ALU	Arithmetic and Logical Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BFU	ButterFly Unit
BFV	Brakerski/Fan-Vercauteren
BGV	Brakerski-Gentry-Vaikuntanathan
CKKS	Cheon-Kim-Kim-Song
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
CXL	Compute Express Link
DBMS	Database Management System
DMA	Direct Memory Access
DSP	Digital Signal Processor
EDP	Energy-Delay Product
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
FPS	Frame Per Second
GPU	Graphics Processing Unit
HE	Homomorphic Encryption
IFFT	Inverse Fast Fourier Transform
INTT	Inverse Number Theoretic Transform
ISA	Instruction Set Architecture
ML	Machine Learning
MMIO	Memory-mapped I/O
NNAPI	Android's Neural Networks API
NPU	Neural Processing Unit
NTT	Number Theoretic Transform
OS	Operating System
PRNG	Pseudo Random Number Generator
QQVGA	Quarter Quarter VGA
RNS	Residue Number System
RTL	Register Transfer Level
SBI	Supervisor Binary Interface

SDK	Software Development Kit
SoC	System-on-a-Chip
SNPE	Snapdragon Neural Processing Engine
TFLite	TensorFlow Lite
TPU	Tensor Processing Unit

Chapter 1

Introduction

1.1 Motivation

One of the main challenges facing modern computing systems is the end of Dennard scaling. For many years, the performance of general-purpose processors was able to keep pace with the increasing demands of applications and workloads. However, as transistors continue to shrink, the power density of these devices has increased to the point where it is no longer possible to maintain the same level of performance without consuming significantly more power. This has resulted in diminishing performance gains of general-purpose processors, necessitating alternative solutions. Hardware accelerators are becoming increasingly important in this context because they offer a way to improve performance without consuming excessive amounts of power. By offloading computationally intensive tasks to specialized hardware, these units can perform specific tasks faster and more efficiently than general-purpose processors. This allows computing systems to continue to scale their performance while staying within reasonable power limits.

Examples of hardware accelerators commonly found in modern computer systems include graphics processing units (GPUs) for accelerating graphics and parallel computing workloads (Nickolls and Dally, 2010), digital signal processors (DSPs) for accelerating signal processing tasks (Tan and Jiang, 2019), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs) for accelerating custom algorithms and computations (Kuon and Rose, 2010), and tensor processing units (TPUs) for accelerating machine learning (ML) workloads (Jouppi et al., 2017).

Many commercial computing systems integrate one or more of these accelerators, each of which is designed to efficiently execute a specific task. Good examples of modern systems with numerous hardware accelerators include the Apple M1 (Apple Inc., 2020), which includes a GPU, a neural engine, and a dedicated video encoding and decoding unit; the Qualcomm Snapdragon 888 (Qualcomm Technologies, Inc., 2020), which includes a GPU, a DSP, and a new Artificial Intelligence (AI) engine designed specifically for ML; and the NVIDIA Drive AGX platform for autonomous vehicles (Nvidia Corp., 2023), which includes a GPU for visual processing, a deep learning accelerator for ML, and a computer vision accelerator for real-time image processing.

One domain where hardware acceleration can provide significant benefits is ML. ML is used in a wide range of applications, including natural language processing (Bouraoui et al., 2022), image and video recognition (Camastra and Vinciarelli, 2015), speech recognition (Deng and Li, 2013), recommendation systems (Nawrocka et al., 2018), and anomaly detection (Omar et al., 2013) to perform human-like tasks. As ML models become more complex, they require more computational power for training and inference. Over the years, a large amount of work has been done on designing hardware accelerators for ML that can greatly improve the performance and efficiency of training and inference tasks. These solutions include massively-threaded GPUs (Choquette et al., 2021), (Chen et al., 2014c), (Huynh et al., 2017), FPGAs and ASICs (Lecun et al., 2011), (Li et al., 2016), (Sankaradas et al., 2009), (Chen et al., 2014a), (Chung et al., 2018), (Fowers et al., 2018), (Askarihemmat et al., 2023), (Chen et al., 2014b), (Li et al., 2022).

Hardware accelerators are also widely used for accelerating data privacy and security algorithms. In particular, lately, there is a growing interest in the use of hardware accelerators for accelerating homomorphic encryption (HE) based privacy-preserving computing. HE allows encrypted data to be processed and analyzed without having to decrypt it, thus providing strong data privacy and security guarantees. Although HE-based privacy-preserving computing seems plausible, it is several orders of magnitude slower than operating on unencrypted data. Prior studies have attempted to address this performance gap by utilizing hardware accelerators (Jung et al., 2021), (Bootland et al., 2020), (Badawi et al., 2021), (Gupta et al., 2020), (Wang et al., 2014), (Cousins et al., 2017), (Reis et al., 2020), (Turan et al., 2020), (Riazi et al., 2020).

While, broadly, the use of a hardware accelerator is promising, unfortunately, the evaluation of a hardware accelerator in the context of the full system reveals that the accelerator's benefits are not always as expected. In fact, in some cases, using a hardware accelerator leads to a performance loss (Azad et al., 2021), (Buch et al., 2021). Hence, an end-to-end performance evaluation of hardware accelerators is critical to understand the overheads associated with using the hardware accelerator. Such an evaluation helps to ensure that the use of the accelerator improves the overall application performance. In section 1.1.1 and section 1.1.2 below, we provide concrete examples to back up this claim.

1.1.1 Tax for AI operations

Hardware accelerators can significantly enhance the performance and efficiency of an ML model execution, i.e., inference. Nevertheless, to enable ML inference, an application typically requires additional steps, such as retrieving data from sensors/databases, preprocessing inputs and models, initializing backend hardware accelerators, transferring data, and post-processing results. While every ML application must go through these necessary stages, such overheads are often overlooked in performance analysis. In this section, we will first explain our proposed methodology for the end-to-end evaluation of ML applications, followed by an example that utilizes it to understand and quantify the effects of individual execution stages of a Database Management System (DBMS) ML application on end-to-end performance.

Evaluation Methodology

To evaluate the performance of ML applications, we recommend the following methodology.

- Understand the processing stages of the ML application, including data capture or retrieval, data pre-processing, inference, and data post-processing. We quantify their individual contributions to the overall end-to-end application time. This understanding will facilitate the understanding of bottlenecks in the application execution pipeline.

- Identify the key performance metrics that are relevant to the application. These may include accuracy, inference time, throughput, latency, power consumption, and memory usage.

- Choose a dataset that accurately represents the problem domain and includes a diverse range of features and samples. This allows observation of the impact of dataset complexity on the application's performance. Additionally, evaluate the application's performance using various input distributions that it is likely to encounter in the real world.

- Select a suitable ML model that aligns with the problem domain and available dataset in the ML application. Additionally, it is important to adjust the model's complexity by tuning its parameters to observe how changes in complexity impact the application's performance.

- Configure and optimize the ML pipeline for each hardware backend. This may involve selecting the optimal batch size, tuning the hyperparameters of the model, and optimizing the code for the target hardware. To ensure the robustness and generalizeability of the evaluation results, use multiple software tools for building and evaluating the ML pipeline, such as Scikit-learn, Tensorflow, or PyTorch.

- Use different hardware backends such as multi-core CPUs, GPUs, and ASIC or FPGA accelerators to run inference. When using accelerators, consider the offloading overhead, i.e., accelerator setup and the data transfer time. Additionally, study the different accelerator integration options, such as tightly coupled, loosely coherent/non-coherent coupled, and decoupled, to determine the best integration method for the application. To carry out this step, we need to utilize a scalable evaluation/simulation infrastructure that can accurately account for the interaction of the accelerators with the rest of the system.

- Analyze the performance data and compare the results across different hardware backends and software tools. Identify any bottlenecks or limitations specific to a particular backend and evaluate the trade-offs between performance and other factors such as cost, energy efficiency, and scalability.

- Draw conclusions and make recommendations based on the evaluation results. Use the insights gained from the evaluation to guide the selection of the optimal hardware backend for your application and to inform future improvements to the ML application pipeline.

Today's DBMS applications integrate machine learning algorithms for the execution of advanced analytical tasks such as predictive modeling, data mining, and natural language processing, directly within the database environment to improve the speed, accuracy, and efficiency of their data analysis. Figure 1·1 shows the high-level architecture of the SQL Server DBMS analytics and ML inference pipeline (Microsoft Corp., 2023). Users submit analytic queries that invoke custom Python scripts that run ML inference, to SQL Server. SQL Server launches an external Python process to execute the script that involves pre-processing, transferring the inputs, running inference, post-processing, and returning results. Below we elaborate on DBMS ML inference pipeline stages.

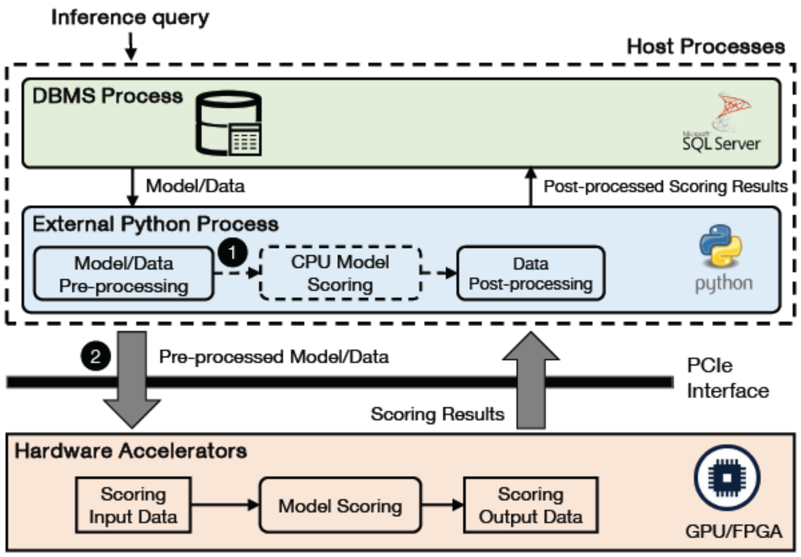


Figure 1·1: High-level architecture of ML inference, using CPUs or hardware accelerators, in SQL Server with Python.

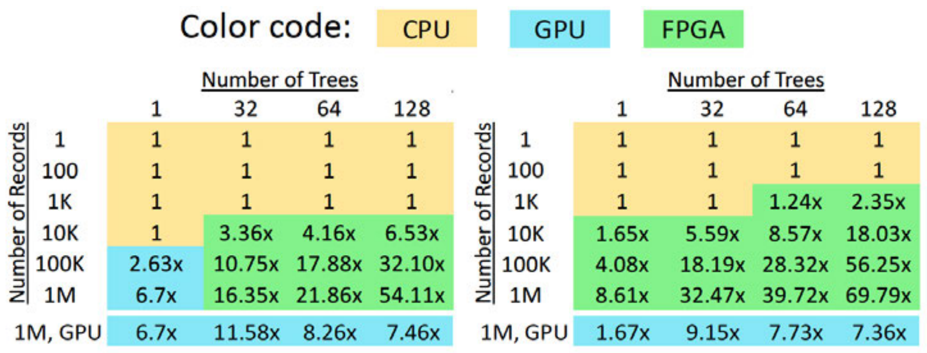
- **Model and Data Pre-Processing:** In SQL Server DBMS, ML models are typically serialized and stored in a binary format within the database. When it is time to use the model for inference or other analytical tasks, the model needs to be deserialized, which involves converting the binary data into an object that can be loaded into memory and executed by the system. The time taken for deserialization can impact the overall performance of the system. There are also some overheads associated with extracting features and preparing the input data for inference. Hence, it is important to consider these factors when optimizing the system for end-user performance experience.

- **Python Invocation Time and Data Transfer:** When using Python code within the SQL Server DBMS environment, there is typically an external Python process that is launched to execute the code. The time taken for launching this external process and initializing the required Python runtime environment can impact the overall performance of the system. Thus, it is important to consider the launch time when optimizing the performance of the system for end-user performance experience. In addition, when using Python within SQL Server DBMS, we need to transfer the data to an external process as input parameters. These input parameters can be of various data types such as integer, float, string, or Pandas DataFrame. However, we need to serialize the data into a format (e.g., binary format) that can be transferred between SQL Server DBMS and Python process. The input parameters are then deserialized and used within the Python script to perform the desired ML tasks. The results of the code execution are returned back to the DBMS environment for further processing. These data serialization/deserialization processes and data transfer add overheads to end-to-end performance.

- **Model Scoring/Inference:** The inference can happen either on the CPU (see ❶ in Figure 1·1), or on a PCIe-attached hardware accelerator (see ❷ in Figure 1·1). If the inference task is offloaded to hardware accelerators, we need to explicitly transfer the input data to the accelerator main memory and transfer the inference results back to the host main memory through a PCI-e link or other similar interfaces. This data transfer process can introduce overhead that depends on several factors, including the size and complexity of the data being transferred and PCI-e bandwidth.

We evaluate the performance of each stage in the SQL Server DBMS ML pipeline by measuring it for tree ensemble models and in particular, on the random forest model, which is one of the top models being used in a wide range of classification and regression applications (Psallidas et al., 2019). However, users can score any ML model with SQL Server by invoking custom Python scripts. We use two datasets with different numbers of data features: 1) **IRIS** (Dua, Dheeru and Graff, Casey, 2017), which is a multi-class classification dataset with 4 features, 3 classes, and 150 data samples; and 2) **HIGGS** (Baldi et al., 2014), which is a binary classification dataset with 28 features and 11M data samples.

We consider FPGA-based and GPU-based acceleration of the random forest inference. For CPU experiments, we use up to 52 threads on a dual-socket Intel Xeon Platinum 8171M



Color code: 🟨 CPU 🟦 GPU 🟩 FPGA

Number of Records	Number of Trees: 1	32	64	128
1	1	1	1	1
100	1	1	1	1
1K	1	1	1	1
10K	1	3.36x	4.16x	6.53x
100K	2.63x	10.75x	17.88x	32.10x
1M	6.7x	16.35x	21.86x	54.11x
1M, GPU	6.7x	11.58x	8.26x	7.46x

(a) IRIS

Number of Records	Number of Trees: 1	32	64	128
1	1	1	1	1
100	1	1	1	1
1K	1	1	1.24x	2.35x
10K	1.65x	5.59x	8.57x	18.03x
100K	4.08x	18.19x	28.32x	56.25x
1M	8.61x	32.47x	39.72x	69.79x
1M, GPU	1.67x	9.15x	7.73x	7.36x

(b) HIGGS

Figure 1·2: Speedups of the best-performing hardware over CPU.

processor with 26 cores (52 threads) per socket running at 2.6 GHz. For FPGA experiments, we use Intel Stratix 10 GX 2800 on this machine. For GPU experiments, we use NVIDIA Tesla P100 available in an Azure NC6s_v2-series Virtual Machine. We use both Scikit-learn (Pedregosa et al., 2011) and ONNX (Bai et al., 2019) models for CPU experiments. For GPU experiments, we use NVIDIA RAPIDS (NVIDIA, 2018) and Microsoft Hummingbird (Nakandalam et al., 2020) libraries. For FPGA experiments, we implement a random forest inference engine and map it to the FPGA implementation, extract the ONNX model information and transfer it to the tree memories on the FPGA through PCIe 3.0 x16. The FPGA design is clocked at 250 MHz and is programmed only once for all the experiments with different tree ensemble structures.

Figure 1·2 shows the ‘shmoo’ plot for which backend among CPU, GPU, and FPGA gives us the best ML inference performance for a given combination of the number of trees in the random forest model (X-axis) and the number of records (Y-axis) for both IRIS and HIGGS dataset. The plot indicates that offloading ML inference to an accelerator is not always the best choice. For a simple model (small number of trees) or a small number of records (i.e., the first 3 rows in the tables) we don’t gain any performance benefit from offloading the inference to an accelerator because the offloading overhead is the dominant time component in the overall inference time. A wrong decision to offload to an accelerator in this case can increase the latency by $10\times$. However, as the model complexity and/or the number of records increases, we can gain $>69\times$ speedup in the inference time by offloading to an accelerator. Another interesting point to note in Figure 1·2 is that given that HIGGS has more data features than IRIS dataset and it generates larger models (which are more compute-intensive to score), even with a smaller number of records (1K in HIGGS compared to 10K in IRIS), offloading to an accelerator is still beneficial.

Figure 1·3 shows the overall inference latency (in ms) on CPU, GPU, and FPGA for different numbers of records and model complexities for the IRIS and the HIGGS datasets. As we can see, for different models, with an increase in the number of records, the inference latency increases, and it affects the choice of the backend hardware we should use for the inference.

Figures 1·3 (a) and 1·3 (b) indicate that with a simple model (only 1 tree with a different number of levels) if we have less than 10K records, CPU has the lowest latency and is the most suitable hardware for inference. However, as the number of records increases over 10K, performing inference on FPGA or GPU is faster compared to CPU. The reason is that, for a large number of records, the inference time is the dominant element in the overall inference time, which can be greatly accelerated by FPGA and GPU due to their massively parallel computing capabilities and deep computation pipelines.

In Figures 1·3 (c) and 1·3 (d), the model complexity is increased by increasing the number of trees in the forest (from 1 tree to 128 trees). As we can see, even for a complex model, for less than 1K records, the CPU is still the best hardware backend to run inference for the small number of records.

We see the same trend in the latency graphs for both IRIS (Figure 1·3 (a)-(d)) and HIGGS (Figure 1·3 (e)-(h)) datasets. However, in the case of the IRIS dataset, the crossover point is higher for both models with 1 and 128 trees (10K, 1K) compared to HIGGS (5K, 500). The reason is that a dataset with a low number of features such as IRIS generates sim-

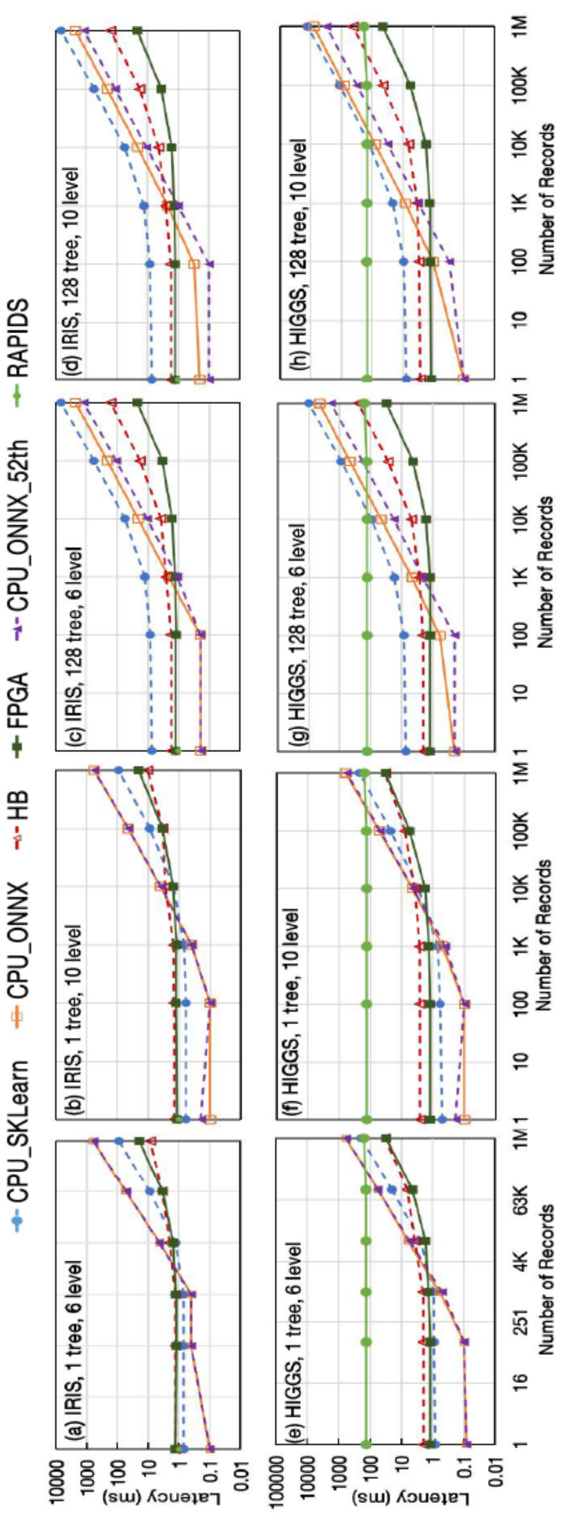


Figure 1·3: Inference latency for models with different tree counts and tree levels for the IRIS and HIGGS datasets running on CPU, GPU, and FPGA.

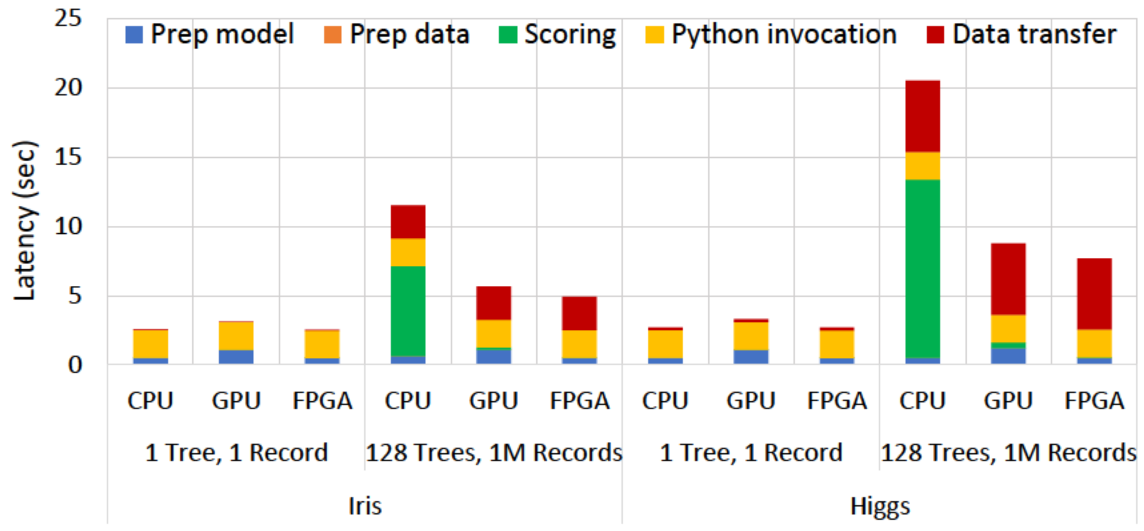


Figure 1·4: End-to-end T-SQL query latency breakdown.

pler/smaller models than a dataset with more features such as HIGGS. Hence, for a simple model (such as for IRIS), the speedup we get from accelerators over the CPU is smaller and the offloading overhead is not amortized for the small number of records. However, even with a smaller number of records, for a complex model (such as for HIGGS), the overhead can be amortized because the accelerator speedup is more significant.

Another key observation is that by increasing the number of dataset features, the amount of GPU/FPGA speedup grows. For example, for 128 trees and 1M records, FPGA and GPU speedups are about 69.7× and 16.5× for HIGGS dataset compared to 54× and 7.5× in IRIS dataset. This is because the generated model for HIGGS dataset is more complex than the one generated for IRIS. So we can accelerate inference more significantly on the accelerators compared to the CPU with its more limited computation capability.

Overall, we can see that offloading ML inference to an accelerator is not always the best choice. The choice of the best hardware backend for inference depends on various factors such as the model complexity (e.g., number of features and tree depth), the input data size, and the overheads associated with data movement. When evaluating end-to-end performance, it is important to consider all these factors.

Figure 1·4 illustrates the end-to-end time breakdown for the T-SQL query that involves ML model inference. It can be observed that for a small model, i.e., a small tree count and a small number of records, the scoring/inference time is insignificant, and the dominant elements are Python invocation and model pre-processing times. However, as the number of records and model complexities increases, Python invocation, model pre-processing, and data transfer (between SQL Server and Python process) contribute only up to 25% of the execution time, with inference time becoming the dominant component. Offloading the inference step to an accelerator can significantly reduce the inference time, making the data transfer time the dominant time component in the overall query time.

When running inference with 128 trees and 1 million input data from the HIGGS dataset, hardware accelerators yield a 69.7$\times$ performance improvement (see Figure 1·2). However, due to these application overheads, this only translates into a 2.6$\times$ end-to-end query time improvement (see Figure 1·4). Alternatively, tighter integration of the ML inference functionality within the DBMS would reduce many of these application overheads, but an external Python invocation could allow for more customizations, including in the choice of the ML inference engine and accelerator. Therefore, application developers and system designers should prioritize a balanced approach to the entire system and end-to-end application pipeline, rather than focusing solely on optimizing ML inference in isolation when optimizing the system for the end-use performance experience.

1.1.2 Tax for HE Computing

HE allows data to be processed while it is still in an encrypted state without compromising its privacy. Over time, numerous HE schemes have been developed, including Brakerski-Gentry-Vaikuntanathan (BGV) (Gentry et al., 2012), Brakerski/Fan-Vercauteren (BFV) (Brakerski, 2012), and Cheon-Kim-Kim-Song (CKKS) (Cheon et al., 2017). Among these, the CKKS scheme stands out as it supports operations on real numbers, which are essential for various applications such as ML, scientific research, and graph analysis. Therefore, we use the CKKS scheme to explore and understand any tax associated with HE computing

Typically, in HE the client-side device encrypts the data and sends it to the cloud for processing. The cloud servers operate on the encrypted data, generate a result that is in encrypted form, and then send the result back to the client-side device. The client-side device then decrypts the data. To encrypt the data, the client-side device must perform encoding, error sampling, and encryption, which together form the "message-to-ciphertext" conversion operation. Similarly, to decrypt the data received from the cloud, the client-side device must perform decryption and decoding, which together form the "ciphertext-to-message" conversion operation. In this section, we elaborate on the message-to-ciphertext and ciphertext-to-message conversion operations.

- **Encoding and Decoding:** The CKKS scheme works with a native plaintext data type that is a vector of length $N/2$, where each vector element is chosen from the field of complex numbers $\mathbb{C}$. The encoding operation takes this $N/2$-dimensional vector as input and returns polynomial $\mathsf{m}(X)$ with integer coefficients. Encoding involves several operations, including scaling, rounding, and polynomial interpolation. First, the input message is scaled by a factor determined by the encryption parameters. This is done to ensure that the message can be represented as an integer polynomial with coefficients that fall within the range that can be encrypted. Next, the scaled message is rounded to the nearest integer. This step is necessary because the CKKS scheme can only encrypt integer polynomials. After that, the rounded message is used to generate a polynomial. Finally, the polynomial is interpolated using Inverse Fast Fourier Transform (IFFT) to obtain a set of complex numbers that can be encrypted. Once the message has been encoded into a polynomial, it can be encrypted using the CKKS encryption algorithm.

  The decoding step is used to recover the plaintext message from the decryption output, which is a polynomial with integer coefficients. First, the integer polynomial is scaled back to the original range of the plaintext message using the scaling factor that was determined during the encoding process. During the encoding process, the message was rounded to the nearest integer, which introduces a rounding error. This error is removed by subtracting a small multiple of the ciphertext modulus from each coefficient of the scaled polynomial. The scaled polynomial is then interpolated using the Fast Fourier Transform (FFT) to obtain a set of complex numbers. Finally, the real part of the complex numbers is extracted to obtain the original plaintext message.

- **Pseudo Random Number Generator (PRNG) and Error Sampling:** In HE, random polynomials are used to add noise to the encrypted data and prevent attackers from learning information about the plaintext Generating high-quality error samples is a crucial factor in maintaining the necessary security level during HE operations. However, creating these error samples is a bottleneck in client-side operations. The error sampling process involves two main steps: first, generating pseudo-random numbers using a true random seed, and second, using these generated numbers to produce uniform and binomially distributed error samples.

- **Encryption and Decryption:** The polynomial $\mathsf{m}(X)$, obtained from the encoding step, can be encrypted under the public key pk, generating a ciphertext ct by computing:

$$\mathsf{c}_0 = \mu \cdot \mathsf{pk}_0 + \mathsf{m} + \mathsf{e}_0, \tag{1.1}$$

$$\mathsf{c}_1 = \mu \cdot \mathsf{pk}_1 + \mathsf{e}_1 \tag{1.2}$$

Here, the μ polynomial is sampled from a uniform distribution, and the error polynomials e_0 and e_1 are sampled using a binomial distribution. The coefficients in the ciphertext polynomials $(\mathsf{c}_0, \mathsf{c}_1)$ are elements of $\mathbb{Z}_Q$, where $\mathbb{Z}$ is a set of integers and Q

defines the order of finite field. Here modulus Q is typically on the order of thousands of bits to account for the noise growth in HE computation. The CKKS scheme supports the use of the Residue Number System (RNS) (also known as the Chinese Remainder Theorem (CRT) representation) to compute such large operands efficiently. Using the RNS approach, each coefficient is represented modulo $Q = \prod_{i=1}^{\ell} q_i$, where each q_i is a prime number. We can represent $x \in \mathbb{Z}_Q$ as a length-ℓ vector of scalars $[x]_{\mathcal{B}} = (x_1, x_2, \ldots, x_\ell)$, where $x_i \equiv x \pmod{q_i}$. We refer to each x_i as a *limb* of x.
The ciphertext is decrypted to obtain the original message back using the following equations:

$$\mathsf{m} = \mathsf{c}_0 + \mathsf{c}_1 \cdot \mathsf{s} \mod q_\ell \tag{1.3}$$

Here s is the secret key. Using RNS, both encryption and decryption can be performed w.r.t. a smaller modulus q_i instead of a large modulus Q.
During encryption, all inputs are transformed into the NTT domain, where they can be manipulated more efficiently. Polynomial multiplication is a critical step in the encryption operation, and a naïve approach to perform it has a computational complexity of $O(N^2)$ multiplications for a polynomial of degree N, which can be computationally expensive. However, NTT can be used to reduce this complexity. NTT is a variant of the FFT that operates on elements of finite fields, rather than complex numbers. During NTT, coefficients of the input polynomial are multiplied with the power of an N-th primitive root of unity and combined with each other in a butterfly fashion. This process is repeated recursively, dividing the polynomial into smaller and smaller subproblems until the entire polynomial is transformed. In the NTT domain, polynomial multiplication can be performed using point-wise multiplication of the transformed data, which has a computational complexity of $O(N \log N)$, which is much more efficient than the naïve approach.

Similarly, we need to perform an inverse NTT (iNTT) operation in the decryption operation to transform the ciphertext from the NTT domain back into the plaintext domain. Both NTT and iNTT operations add high computational complexity to the encryption and decryption operations, respectively.

We use an example of video encryption to further illustrate message-to-ciphertext and ciphertext-to-message conversion steps described above. A video is made up of multiple frames, where a frame size is defined by $f_w \times f_h \times b_{pp}$. Here, b_{pp} defines the bits per pixel and assumes a value of 8 for a grayscale pixel. For a given N, $\log q$, and $limbs$ value, we can encode $N/2 \times \log q$ bits in a single ciphertext, which implies that a single frame will be encoded and encrypted within multiple ciphertexts (cts) and will have a total size of $N \times \log q \times limbs \times \#cts$ bits. For a Quarter Quarter VGA (QQVGA), the frame resolution is 120×160 pixels. If this frame is in grayscale, the frame size will be $120 \times 160 \times 8 = 153,600$ bits = 18.75 KB. With $N = 4096$ and $\log q = 30$ bits, we can encode $N/2 \times \log q = 2048 \times 30 = 61,440$ bits in a single ciphertext, which implies that a single frame will be encoded and encrypted within 3 ciphertexts and will have a total size of 270 KB.

Evaluation Methodology

We evaluate the performance of message-to-ciphertext and ciphertext-to-message conversions operations for HE video application using a methodology based on the one described in section 1.1.1, with the following modifications:

- We study the processing stages of the HE client-side operations, including error sampling, encoding/decoding, and encryption/decryption to quantify their individual contributions to the overall application time. This understanding will help in better designing and optimizing the overall pipeline.
- We evaluate the application performance metric for input with different features such as different video frame resolutions (e.g., QVGA and QQVGA). This allows us to

observe the impact of input data complexity on the application's performance.

- We select a suitable HE scheme (such as CKKS) that aligns with our problem domain and available dataset.

- We use the SEAL-Embedded software library (Natarajan and Dai, 2021) instead of the other available options such as SEAL (Chen et al., 2017), PALISADE (Kurt Rohloff, 2018), and HElib (Halevi and Shoup, 2020) because it is optimized for resource-constrained client devices. We modify SEAL-Embedded to call hardware accelerators whenever needed.

- We use our accelerator integration framework (Chapter 2) to integrate an NTT accelerator (Azad et al., 2022) into the BlackParrot SoC and accurately account for the interaction of the accelerator with the rest of the system and measure the offloading overhead. We implement both baseline and accelerator (NTT_Accel) systems in SystemVerilog and simulate them using VCS, a cycle-accurate simulator, to collect the performance results.

- We draw conclusions and make recommendations based on the evaluation results. We use the insights gained from the evaluation to identify the bottlenecks in the HE client-side operations pipeline, guide the selection of the optimal hardware backend, and inform future improvements for the application.

We use BlackParrot RISC-V core (with 32 KB each of Icache and Dcache and running at 1 GHz) and also an NTT hardware accelerator (Azad et al., 2022) (NTT_Accel) as our hardware backends. The NTT operation is parallelizable, as it involves multiple butterfly operations that can be computed simultaneously. Hence, the NTT accelerator uses parallel butterfly units (BFUs) to further improve its performance. We study the impact of faster NTT operation on the application performance metrics.

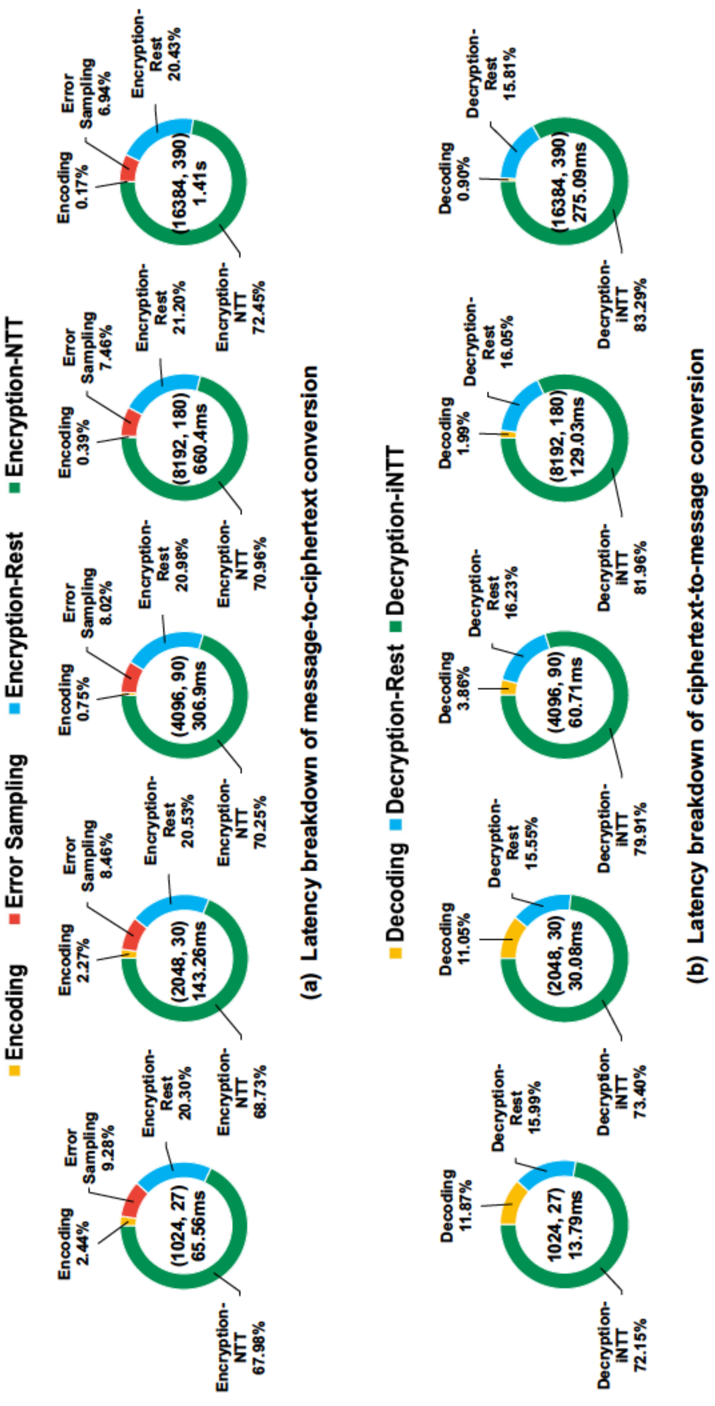


Figure 1·5: Latency breakdown of (a) message-to-ciphertext and (b) ciphertext-to-message conversion operations running on BlackParrot using SEAL-Embedded library. Corresponding scheme parameters (N, $\log Q$) and latencies are specified inside the doughnut charts.

Figure 1·5 (a) and (b) show the latency breakdown of the message-to-ciphertext and ciphertext-to-message conversion for different scheme parameters (N, $\log Q$) using the baseline system (software execution). For all the parameter sets that we evaluated, the encryption and decryption operations incur the highest latency because they perform multiple polynomial multiplications. The latency of the encryption and decryption operations is dominated by NTT (67.98%-72.45%) and iNTT (72.15%-83.29%) operations. Error sampling is also a bottleneck operation, accounting for up to 10% of the total message-to-ciphertext conversion latency.

Due to the computationally intensive nature of NTT, hardware acceleration is often used to speed up the computation. Several research efforts have focused on designing efficient hardware accelerators for NTT operation, including FPGA-based implementations, ASIC designs, and custom-built processors (Nannipieri et al., 2021), (Banerjee et al., 2019), (Li and Liu, 2021), (Chen et al., 2022), (Li and Liu, 2021), (Ye et al., 2022), (Paludo and Sousa, 2022), (Ye et al., 2022), (Su et al., 2022), (Ye et al., 2022), (Su et al., 2022), (Duong-Ngoc et al., 2023).

Figure 1·6 (a), (b), and (c) show the NTT per second, message-to-ciphertext (M-t-C) conversion operation per second, and the video encryption throughput, i.e., frame per second (FPS) metrics for the baseline system and NTT_Accel with different numbers of BFUs for a range of scheme parameters. As we can see, using hardware acceleration for NTT operation (NTT_Accel) can lead to 381$\times$ improvement in NTT per second for different scheme parameters. However, when we move from the baseline system to NTT_Accel, there is only up to 4$\times$ improvement in the message-to-ciphertext conversions per second. In the case of video encryption, with the baseline system (software execution), we cannot encrypt even one frame per second. This is due to the slow and inefficient nature of the software-based implementation of modular arithmetic in HE message-to-ciphertext and ciphertext-to-message conversion process, which makes it unsuitable for real-time applica-

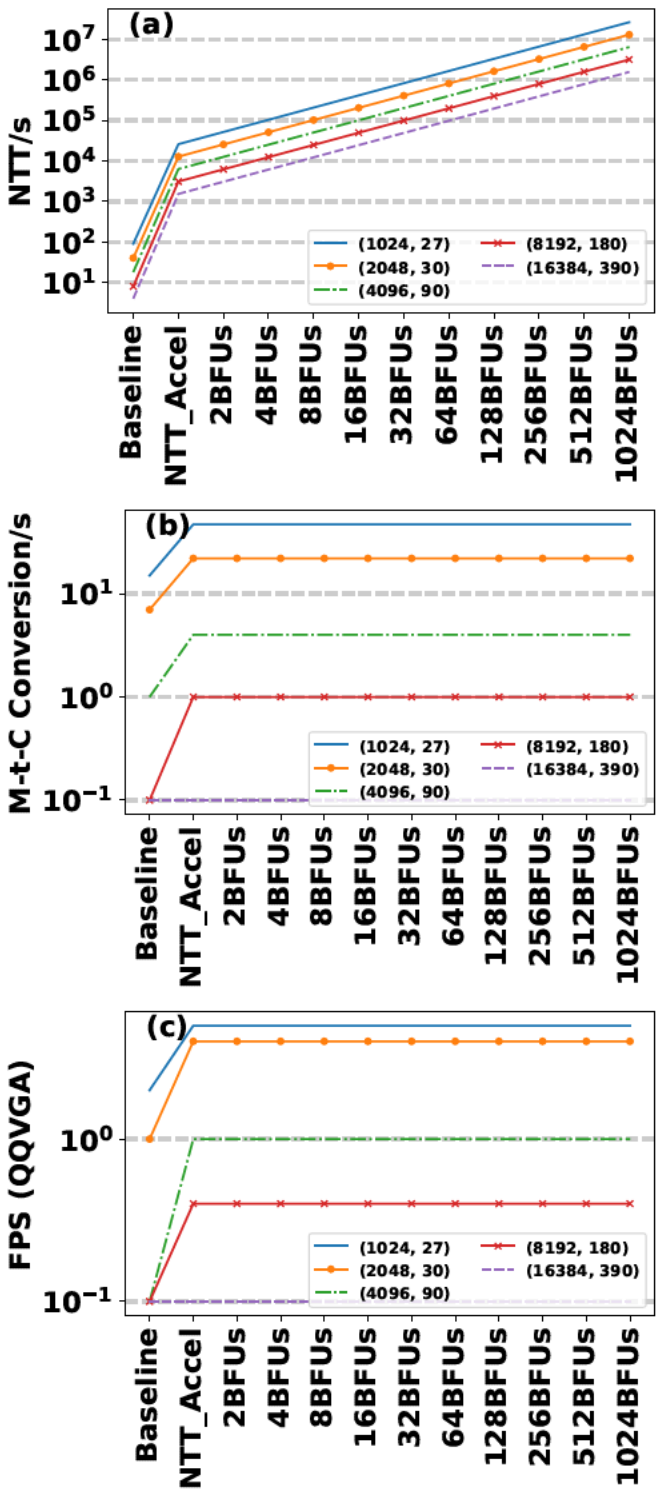


Figure 1·6: (a) NTT/s, (b) message-to-ciphertext/s, and (c) FPS (c) for software execution and NTT hardware acceleration with different numbers of BFUs.

tions. Using NTT_Accel with 1BFU configuration will only improve FPS by 3 frames per second for smaller scheme parameters and will provide no improvement for larger scheme parameters ($N >= 2^{12}$).

In addition, as we increase the parallelism level (number of BFUs) in NTT_Accel, we observe a linear increase in NTT throughput. However, there is no corresponding increase in the message-to-ciphertext conversion throughput and FPS metric. This is because the error sampling and non-NTT operations in the encryption process account for up to 10% and 21.2% of the end-to-end latency, respectively, making them the new bottlenecks. As a result, further NTT acceleration does not enhance the end-to-end performance, resulting in low message-to-ciphertext conversion per second and FPS metrics. One could use a more powerful processor to address all bottlenecks, but then the power consumption would be higher which would not be sustainable in a typical edge device.

Accelerating error sampling and non-(i)NTT operations in the encryption and decryption pipeline is needed to improve the end-to-end message-to-ciphertext and ciphertext-to-message conversion latency, which results in improved application performance metrics. Therefore, to measure the end-user performance experience in a realistic way, it is crucial to evaluate end-to-end application performance metrics instead of evaluating the performance of just one kernel, such as NTT.

It is worth noting that we conducted our experiments in bare metal mode and so the direct memory access (DMA) and data transfer overhead is minimal. Bare metal refers to running code directly on the hardware without the intervention of an operating system or other software layers. In this scenario, direct access to hardware resources, including the DMA capabilities, typically results in lower overhead and higher performance compared to running the same code on an operating system, which adds additional layers of abstraction and control. If we use an operating system, the data transfer overhead for NTT_Accel system will be significant as we need a round-trip through the kernel device driver to transfer the inputs (μ, pk_0, m, e_0, pk_1, e_1) and return their results (inputs in NTT domain). Therefore, this overhead degrades the end-to-end performance as we discussed in Section 1.1.1, and needs to be taken into account when optimizing the performance for the end-user experience.

1.2 Thesis Contribution

End-to-end evaluation of hardware accelerator-based systems is essential for ensuring that these systems are correctly optimized for their intended applications and that the system as a whole is delivering the desired level of performance (end-user performance experience). End-to-end evaluation typically involves several stages, including benchmarking, profiling, and tuning. Benchmarking is used to evaluate the performance of the system under different workloads and scenarios, while profiling is used to identify potential bottlenecks or areas for optimization. Tuning involves adjusting the hardware and software configurations to optimize for performance and energy efficiency. This evaluation should be an ongoing process, as new workloads and hardware configurations emerge, and as the requirements of applications and users continue to evolve. To this end, we present the thesis statement as follows:

It is essential to evaluate the system as a whole and account for all the accelerator taxes when designing an accelerator-based system.

The main contributions of this Ph.D. research are discussed in the following sections.

1.2.1 BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs

One of the challenges of building and evaluating accelerator-based systems is the lack of a scalable evaluation/simulation infrastructure that can accurately account for the interaction of the accelerators with the rest of the system. This can be attributed to the lack of open-source software and hardware platforms that can be used to build these systems. While there are some open-source tools available for building and evaluating these systems, they are often limited in their capabilities.

Software simulators such as gem5-Aladdin (Shao et al., 2016), gem5-gpu (Power et al., 2014), PARADE (Cong et al., 2015), Scale-Sim (Samajdar et al., 2018), and SMAUG (Xi et al., 2020) are useful tools for evaluating hardware designs and software systems, but they have limitations when it comes to performance and accuracy. Moreover, they are much slower than native execution and both their performance and accuracy degrade considerably as the system size scales up. Similarly, there are open-source hardware-level platforms like Rocket Chip Generator (Asanović, Krste., 2016), HERO (Kurth et al., 2017), OpenPiton (Balkind et al., 2020), ESP (Giri et al., 2018), and Chipyard (Amid et al., 2020). However, these platforms either lack scalability, lack support for different accelerator types, lack a generalized accelerator interface, lack silicon validation, or a lack of user-friendly development language.

We develop BlackParrot[1], an agile open-source Linux-capable RISC-V multi-core processor for accelerator SoCs, which is optimal in terms of power, performance, area, and complexity. BlackParrot implements the RISC-V "RV64G" architecture and supports three privilege levels-machine, supervisor, and user-as well as SV39 virtual memory; these extensions are sufficient to efficiently run full-featured operating systems such as Linux. BlackParrot is designed as a scalable, heterogeneously tiled multi-core micro-architecture.

As part of the BlackParrot project, we develop a robust and scalable software-hardware framework for integrating accelerators in heterogeneous systems. Each accelerator tile features specialized hardware for executing coarse-grained tasks in a loosely-coupled manner. BlackParrot supports both streaming and coherent loosely-coupled accelerators. To achieve modularity, we design a socket for each accelerator tile that decouples the accelerator design from the rest of the SoC. This socket provides components that handle various functions, such as memory-mapped registers, interrupt requests, DMA, and hardware-coherent

[1]This work was in done in collaboration with the University of Washington.

transactions. We design a new tile socket for third-party accelerator integration in a modular way such that at design time the SoC architect may select a different adapter for each specific accelerator tile. The concept of socket plays a key role in supporting the flexibility of the BlackParrot methodology because it accommodates accelerators designed using a variety of design flows as a third-party IP block.

Accelerators invocation utilizes a software stack and an application programming interface (API) to allocate shared data and configure accelerators in bare metal or on top of Linux. The lightweight API can be easily targeted by a user program or by a compiler, and it invokes the accelerators through Linux device drivers that are automatically generated. The BlackParrot accelerator framework also includes the necessary synthesis and simulation scripts for testing the accelerator implementation.

Accelerator developers can use this ecosystem to evaluate different accelerator integration mechanisms as well as accelerator designs for the targeted application. This helps to measure and minimize the end-to-end latency for applications, with different invocation and completion semantics, parallelism granularity, synchronization types, and memory access patterns.

In the ParrotLine case study, we integrate the Microsoft Zipline accelerator with the BlackParrot Core to validate our framework. Zipline is a compression engine that supports lz77 compression formats such as xp10, gzip, and zlib. Zipline contains 2 main engines, Compression and Encryption IP, and Decompression and Decryption IP. We integrate Zipline as a streaming accelerator tile as it requires a large amount of input/output data transfers. We observed up to 22$\times$ speedup for different input file sizes. In addition, our end-to-end performance analysis shows that speedup decreases for larger file sizes. This decrease in speedup is attributed to the offloading overhead becoming a significant component in the overall latency.

1.2.2 RISE: RISC-V SoC for HE Client-Side Operations Acceleration

There is plenty of research being done on accelerating server-side HE operations. Unfortunately, little attention has been paid to client-side operations, even though they can be non-trivial. On the client side, there have been some research works to accelerate the NTT kernel (Nannipieri et al., 2021), (Banerjee et al., 2019), (Li and Liu, 2021), (Chen et al., 2022), (Li and Liu, 2021), (Ye et al., 2022), (Paludo and Sousa, 2022), (Ye et al., 2022), (Su et al., 2022), (Ye et al., 2022), (Su et al., 2022), (Duong-Ngoc et al., 2023). However, these efforts often fail to optimize the end-to-end message-to-ciphertext and ciphertext-to-message conversions, resulting in suboptimal application performance metrics (refer to Section 1.1.2).

To address this gap, we profiled client-side HE operations for a range of scheme parameters (N and log Q). We observed that during message-to-ciphertext and ciphertext-to-message conversion operations, error sampling, encryption, and decryption operations are the performance bottlenecks. Additionally, we found that the NTT operation presents a significant bottleneck in the encryption and decryption process, requiring a large number of modular multiplications and additions. Based on the profiling results, we architect RISE, an SoC that includes a RISC-V BlackParrot core and an area and energy-efficient hardware accelerator (integrated as a streaming accelerator) to efficiently perform end-to-end HE client-side operations.

RISE speeds up the error sampling process by combining a lightweight pseudo-random number generator with fast sampling techniques. To accelerate encryption and decryption operations, RISE uses scalable, data-level parallelism for the NTT operation. This parallelism-based approach enables RISE to meet the diverse needs of different applications and platforms. Furthermore, the non-(i)NTT portion of the encryption and decryption process is accelerated in the hardware, resulting in improved overall performance. We use several optimizations in RISE such as a shared data path for the en/decryption operations, on-the-fly twiddle factor computation, memory reuse, and data reorder techniques to meet the performance requirements of resource-constrained client devices.

We conduct a thorough evaluation of RISE using a complete RTL design containing a BlackParrot RISC-V processor interfaced with our accelerator. We analyze RISE's performance, area, and energy efficiency by executing end-to-end message-to-ciphertext and ciphertext-to-message conversion operations. Our analysis reveals that across a range of parameters, RISE reduces the message-to-ciphertext and ciphertext-to-message conversion latency by 28.79×-104.39× and 7.95×-66.08×, respectively, as compared to the software execution. RISE also achieves much lower energy-delay product (EDP) and area-delay product (ADP) than software execution. Specifically, RISE achieves 471.24×-6191.19× lower EDP when performing message-to-ciphertext conversion and 36×-2481.44× lower EDP when performing ciphertext-to-message conversion, as compared to the software execution. Similarly, RISE has 24.06×-55.36× lower ADP when performing message-to-ciphertext conversion and 6.65×-35.05× lower ADP when performing ciphertext-to-message conversion, as compared to the software execution.

We also conduct a case study using a video application to evaluate RISE and determine how it improves the video application performance (end user's performance experience). When using software execution for message-to-ciphertext conversion, we are unable to encrypt even one frame per second. Accelerating NTT in hardware while running the remaining operations on software did not improve the frame rate for large scheme parameters. However, by optimizing and running the entire pipeline in hardware, RISE is able to encrypt up to 17 QVGA frames and 64 QQVGA frames per second for the smallest ($N = 2^{10}$) scheme parameters, and up to 10 QVGA frames and 27 QQVGA frames for the largest ($N = 2^{14}$) scheme parameters. This frame rate is typically sufficient for most surveillance cameras and mobile platforms.

1.3 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we review the background on RISC-V, state-of-the-art on heterogeneous system simulators and hardware platforms, and present our work on the design and implementation of BlackParrot, an agile open-source RISC-V multicore for accelerator SoCs. In Chapter 3, we review the state-of-the-art HE accelerators and introduce RISE, our RISC-V SoC for HE client-side operations acceleration. In Chapter 4, we discuss the future directions and conclude this thesis.

Chapter 2

BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs

In recent years, there has been a growing trend toward the use of accelerator-based systems with multiple accelerators. These systems are designed to perform complex computations and accelerate a wide range of applications. The emergence of these systems has led to new challenges in system design, such as the need for a scalable evaluation/simulation infrastructure that can accurately account for the interaction between multiple accelerators and the rest of the system including the CPU, memory, and I/O subsystems. To address the lack of a widely adopted and scalable infrastructure for building and evaluating accelerator-based systems, this chapter discusses our design of BlackParrot, an open-source hardware platform for designing such systems (Azad et al., 2019) (Petrisko et al., 2020).[1]

2.1 RISC-V Instruction Set Architecture

RISC-V (Waterman et al., 2014) is an open-source instruction set architecture (ISA) that has been widely used in recent years by both academia and industry for building processors and the surrounding software environment The RISC-V ISA was originally developed at the University of California, Berkeley in 2010, with the goal of creating a simple and modular ISA that could be easily extended to meet the needs of different applications and markets. Unlike proprietary ISAs, which are typically licensed and controlled by a single company, RISC-V is an open standard that anyone can use, modify, and distribute without paying.

[1]This work was done in collaboration with the University of Washington.

One of the key advantages of RISC-V as an open-source ISA is the flexibility and adaptability it provides. With a modular design, RISC-V allows for the creation of custom instruction sets that can be tailored to specific workloads and applications, resulting in improved performance and energy efficiency. Additionally, the open-source nature of RISC-V enables collaboration and innovation across the industry, as developers and companies can freely share and build upon each other's work. As a result, RISC-V has gained significant traction in a variety of markets, including edge computing, IoT, and high-performance computing, among others.

RISC-V supports three different address spaces, namely RV32, RV64, and RV128, corresponding to 32-bit, 64-bit, and 128-bit architectures, respectively. In this chapter, our focus is on the commonly used RV64 processors. The RISC-V ISA defines a base integer instruction set, such as RV64I, as well as several standard extensions. For 64-bit address space, the standard general-purpose RISC-V ISA is RV64G, where "G" represents the base integer prefix, and the names of the other general extensions, i.e., RV64IMAFD. The "M" extension is for integer multiplication and division, "A" is for atomic instructions, and "F" and "D" are for single-precision and double-precision floating-point, respectively. Additionally, the RISC-V ISA provides a standard extension for compressed instructions (“C”).

2.2 Related Work

In this section, we provide an overview of the state-of-the-art on heterogeneous system simulators and hardware platforms, along with their limitations.

2.2.1 Open-Source Software-Level simulator

Software simulators such as gem5-Aladdin (Shao et al., 2016), gem5-gpu (Power et al., 2014), PARADE (Cong et al., 2015), Scale-Sim (Samajdar et al., 2018), and SMAUG (Xi et al., 2020) are widely used in the field of computer architecture to evaluate and compare different hardware designs and software systems. These simulators provide a way to test a design's performance and functionality without requiring the actual hardware to be available. However, despite their usefulness, software simulators have limitations when it comes to accuracy and performance.

One major limitation of software simulators is their slower execution speed compared to native execution. Software simulators have to simulate the behavior of hardware, which requires significant computation and memory resources. As a result, software simulations can be orders of magnitude slower than running the same code natively on actual hardware. This slower execution speed can be a significant drawback when running simulations that require a lot of computational resources or that need to run for long periods.

Another limitation of software simulators is their accuracy. While software simulators try to replicate the behavior of hardware as closely as possible, they may not always accurately model all aspects of the hardware. As a result, the results obtained from software simulations may not be entirely accurate, especially for complex designs. This lack of accuracy can be a significant issue when evaluating new designs or when trying to optimize existing ones.

Both the accuracy and performance of software simulators degrade considerably as the system size scales up. As the size of the system being simulated increases, the amount of computation required also increases. This increased computational demand can cause significant performance degradation and inaccuracies, making software simulators less useful for large-scale simulations.

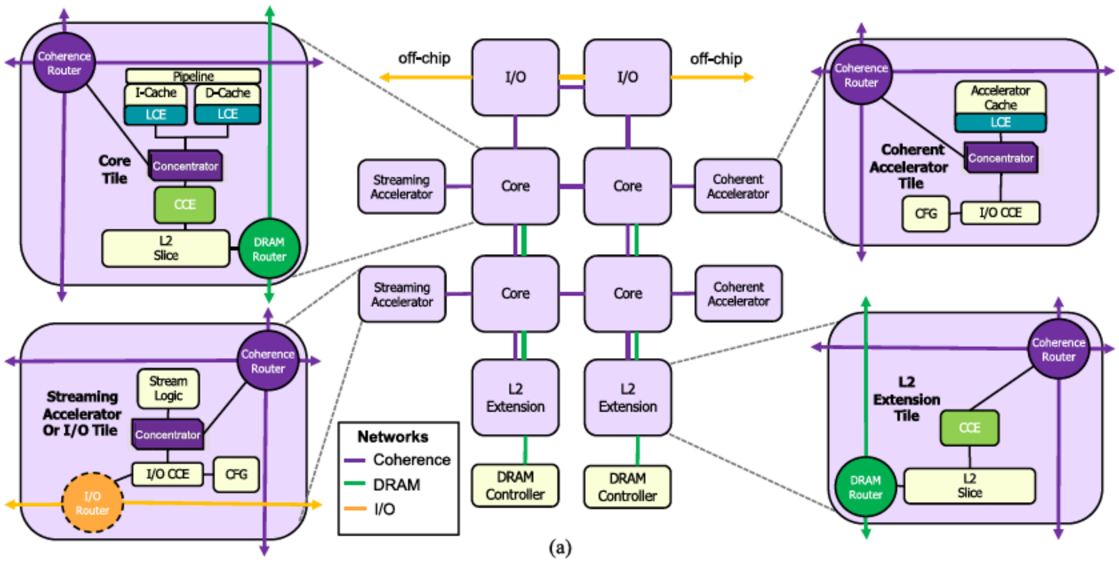
2.2.2 Open-Source Hardware-Level Platforms

There are open-source hardware-level platforms like Rocket Chip Generator (Asanović, Krste., 2016), HERO (Kurth et al., 2017), OpenPiton (Balkind et al., 2020), ESP (Giri et al., 2018), and Chipyard (Amid et al., 2020). However, these platforms lack scalability, lack support for different accelerator types, lack a generalized accelerator interface, lack silicon validation, or a lack of user-friendly development language.

Rocket Chip Generator (Asanović, Krste., 2016) is an open-source SoC that is developed using Chisel RTL language. In Rocket Chip, multiple cores are connected to a coherent TileLink bus. Each core in Rocket Chip has a RoCC (Rocket Chip Coprocessor) interface, which can be used to tightly integrate an accelerator with the core. Loosely-coupled accelerators can be connected to the TileLink bus. However, this bus-based architecture limits the SoC scalability. Moreover, the Rocket Chip Generator uses Chisel, which isn't an industry-standard language. This has limited the adoption of the Rocket Chip Generator by both academia and industry.

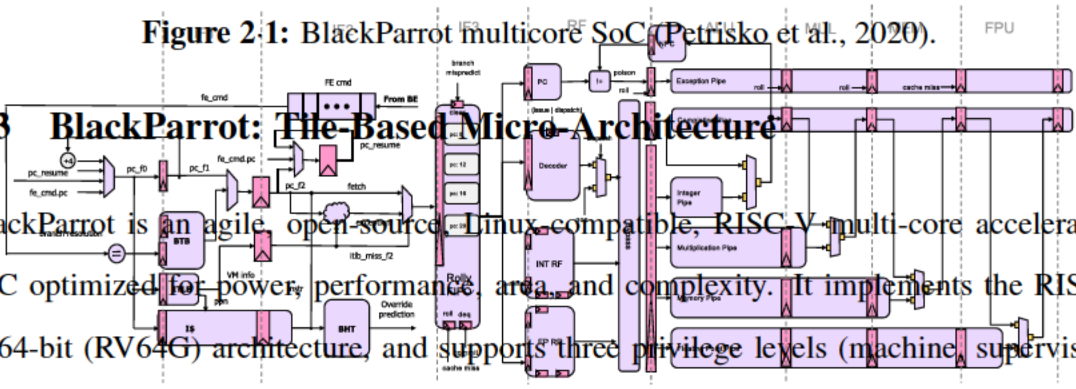
HERO (Kurth et al., 2017) is an FPGA-based research platform that combines a PULP-based open-source parallel manycore accelerator with an ARM Cortex-A multicore processor. PULP platform only supports streaming accelerator integration, and its focus is mainly on ultra-low power systems rather than general-purpose compute systems. OpenPiton (Balkind et al., 2020) is the first open-source Symmetric Multiprocessing (SMP) Linux-booting RISC-V multicore processor. It supports the research of heterogeneous ISAs and provides a coherence protocol that extends across multiple chips. However, OpenPiton does not have a generalized accelerator interface, and the connection they expose is specific to a processor model.

All three platforms mentioned above are built with a processor-centric perspective. BlackParrot, on the other hand, is designed with a system-centric perspective. It has a scalable tile-based architecture with two different tile types (fully-coherent and non-coherent) for accelerator integration. ESP (Giri et al., 2018) is also an open-source RISC-V based research platform for heterogeneous SoC design. It has a tile-based architecture with a dedicated tile type for accelerator integration. Unlike BlackParrot, ESP only has an FPGA prototype and is not silicon validated.



(a)

Figure 2.1: BlackParrot multicore SoC (Petrisko et al., 2020).



2.3 BlackParrot: Tile-Based Micro-Architecture

BlackParrot is an agile, open-source Linux-compatible, RISC-V multi-core accelerator SoC optimized for power, performance, area, and complexity. It implements the RISC-V 64-bit (RV64G) architecture, and supports three privilege levels (machine, supervisor, and user), along with the SV39 virtual memory extension that enables it to efficiently run full-featured operating systems like Linux. This makes it an extremely versatile platform suitable for a wide range of applications.

BlackParrot is designed using a scalable, heterogeneously tiled multicore micro-architecture. The main advantage of this tile-based approach is its scalability and flexibility, which allows the chip to be easily adjusted in size and performance by adding or removing tiles. Additionally, it enables high parallelism, as each tile can perform independent processing and communicate efficiently with neighboring tiles. Furthermore, this modular architecture allows for easy integration with other hardware accelerators and interfaces.

BlackParrot implements a distributed directory-based cache coherence protocol that supports MSI and MESI. The underlying implementation, BedRock, comprises a collection of local cache engines (LCEs), each of which controls an L1 cache, connecting over an interconnection network to the programmable cache coherence engines (CCEs), which collectively maintain the address-sharded directory state.

The micro-architectural tiles of BlackParrot are classified into four categories as shown in Figure 2·1, which we will elaborate on below.

2.3.1 Core Tile

Each BlackParrot Core Tile includes a complete BlackParrot processor. It consists of one or multiple coherent caches, along with a directory shard and an L2 slice. A standard system incorporates several Core Tiles. Figure 2·2 shows the BlackParrot core micro-architecture.

The front-end of the processor presents a stream of instructions to the back end, which follows an in-order but speculative approach. To avoid any delays caused by long-latency back-end operations like servicing cache misses, the issue queue decouples the front-end fetch from the back-end execution, enabling speculative fetching. During the instruction fetch, exceptions can occur, but these are purely speculative and sent to the back-end to be serviced along with other instructions. Apart from the PC, instruction, and exception, the front-end also transmits metadata linked to the branch prediction that led to the specific PC fetch.

The back-end of the processor is responsible for executing instructions, managing exceptions, and ensuring the architectural state of the processor is maintained. Messages are transmitted from the back-end to the front-end to rectify any mispredictions and update the shadow state in the front-end. These messages can include branch resolution, interrupt redirection, iTLB manipulation, and privilege mode changes. Once a branch is resolved, the associated branch metadata is transmitted back to the front-end. The back-end does not inspect this metadata, and the specific branch prediction scheme employed by the front end is entirely unknown to the back end.

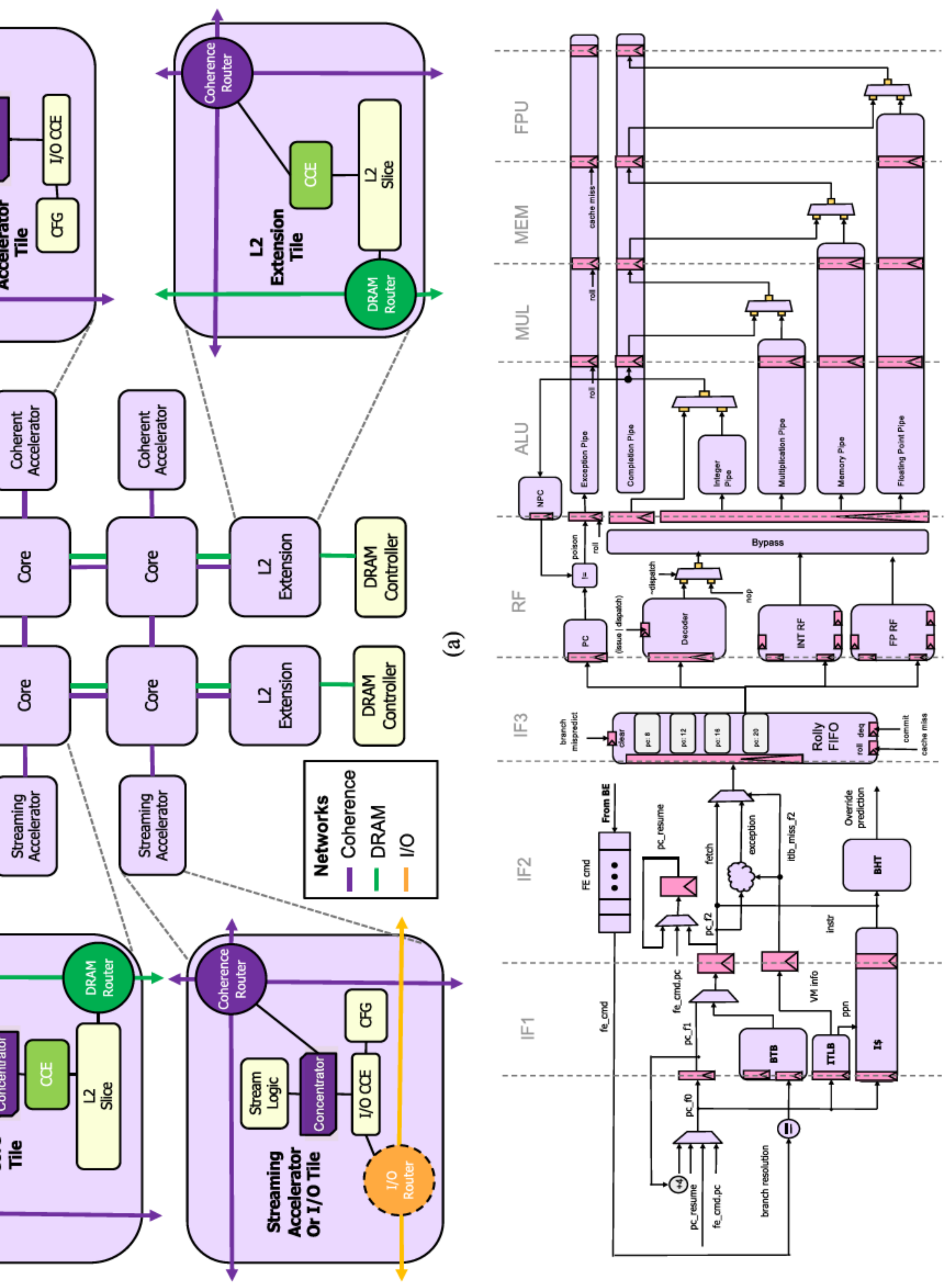


Figure 2·2: BlackParrot core micro-architecture (Petrisko et al., 2020).

2.3.2 L2 Extension Tile

To expand the on-chip L2 memory in a BlackParrot system, an L2 extension tile can be employed as a straightforward scale-out solution. Each L2 extension is equipped with a directory and a non-inclusive, non-exclusive L2 slice. The distribution of L2 slices enables a system designer to adjust the compute-to-cache ratio of a BlackParrot system effortlessly without disturbing the critical paths in the cores or the NoCs.

2.3.3 Accelerator Tiles

Integrating accelerators into existing system architectures is a labor-intensive process, and it introduces additional challenges related to data management. Tightly-coupled accelerators are specialized hardware functional units, one or more of which are tightly coupled into a processor core. They are invasive to the core architecture and typically rely on the processor pipeline to perform memory accesses. The accelerator shares the register file, MMU, and L1 data cache with the processor core.

Loosely-coupled (coherent and non-coherent) accelerators, on the other hand, are separated from the processor pipeline and can be attached at different levels of the memory system hierarchy. In decoupled integration, the processor and accelerator have their own memory subsystem with independent address spaces. A dedicated I/O interconnect, such as PCIe, which is a low-bandwidth and high-latency protocol, provides communication between host memory and accelerator memory.

BlackParrot provides a scalable and robust end-to-end framework for integrating accelerators. This framework simplifies the integration of both loosely-coupled coherent accelerators (with data caching and hardware coherency) and streaming accelerators, as well as the offloading of jobs from the user application. This framework offers a hardware implementation in SystemVerilog for simulation and FPGA prototypes. A socket is included in each accelerator tile, which decouples the accelerator design from the rest of the SoC.

This socket provides components that handle various functions, such as memory-mapped registers, interrupt requests, DMA, and hardware-coherent transactions. The design of a new tile socket for third-party accelerator integration is modular, allowing the SoC architect to select a different adapter for each specific accelerator tile at design time. The socket concept is crucial in supporting the flexibility of the BlackParrot methodology because it accommodates accelerators designed using a variety of design flows as a third-party IP block.

The BlackParrot accelerator framework allows programs to run on both bare-metal and Linux environments. This capability is enabled through software libraries, kernel drivers, and firmware extensions. To invoke accelerators, a software stack and an application programming interface (API) are utilized to allocate shared data and configure accelerators in bare metal or on top of Linux. The API is lightweight and can be easily targeted by a user program or compiler. Accelerators are invoked through Linux device drivers that are automatically generated. Additionally, the framework provides some M-mode extensions to enable low-level access to accelerators' memory-mapped registers for configuration purposes.

These features enable accelerator designers and system architects to evaluate their accelerator-related ideas using hardware implementation instead of simulation and to find the integration strategy with low offload and synchronization overheads for their applications to improve the end-to-end application time.

Coherent Accelerator Tile

A coherent accelerator tile includes an LCE with a coherent cache backing it. A shared memory space helps to reduce accelerator offloading overhead and improves programmability. Additionally, hardware coherence simplifies the software stack by eliminating the need for explicit software synchronization, such as cache flushes and invalidation. Accelerator developers can select one of the following options for the integration:

1. attach the accelerator directly to the provided BlackParrot data cache;
2. reuse the provided BlackParrot LCE and provide a specialized cache; or
3. provide a specialized LCE implementation that interfaces with directly with the network.

Streaming Accelerator Tile

Streaming (non-coherent) accelerator tiles lack a cache memory behind their LCE link and do not manage any physical memory. Consequently, utilizing streaming accelerators entails costly data movement to the accelerator's internal memories through specialized drivers (DMA operation) and synchronization mechanisms. Streaming accelerators can be useful in applications where data coherence is not a critical requirement, but high-speed data processing or I/O operations are necessary Typical applications for streaming tiles include basic I/O devices, network interface links, and GPUs.

2.4 BlackParrot: Networks-On-Chip

BlackParrot tiles are organized in a regular 2D mesh, with communication between adjacent tiles facilitated by a network-on-chip (NoC) interconnect. Unlike bus-based architectures, BlackParrot's NoC interconnect provides a scalable and high-bandwidth connection between the various processing elements in the system. The NoC in BlackParrot is divided into three classes: coherence (BedRock), DRAM, and I/O. Each class is optimized for its specific protocol, which includes factors such as flit width, packet length, and coordinate width. By tailoring the NoC classes to the particular requirements of each protocol, BlackParrot achieves an efficient and optimized physical design.

2.4.1 BedRock Network

The BedRock network forms the cache-coherent fabric that connects all tiles in a BlackParrot system, acting as the central hub for all LCEs and CCEs. The protocol underlying this network employs three logical channels: request, command, and response. To initiate a transaction, an LCE uses the request channel, specifying details such as read or write, cached or uncached, and additional metadata for the return address. Meanwhile, the command channel is utilized by a CCE to modify the LCE state of the system. This includes setting tags, filling data, and completing synchronization sequences, as well as allowing for transfers of cache lines between LCEs. All such transfers are carried out via the command network. Finally, the response channel is responsible for coherence acknowledgments, ensuring that requests are serialized and handled appropriately. Although the BedRock protocol does not mandate it, the current implementation of the network employs a wormhole-routed 2-D mesh, with one physical channel per logical channel.

2.4.2 DRAM Network

The DRAM network in BlackParrot is responsible for connecting the CCEs to devices that are capable of handling memory requests, such as DRAM, Flash, or on-chip ROMS. As all requests are initiated by a tile and serviced by memory devices at the bottom of the chip, the memory network is designed to be a lightweight 0.5-D network. To optimize performance, the DRAM network is particularly well-suited to wormhole routing, as DRAM controllers tend to return the least significant word first. In wormhole routing, packets (data messages) are divided into smaller segments, and each segment is sent separately from the source to the destination through a network of interconnected nodes. At each node, a small part of the packet, called the "header," is examined to determine the next node to which the packet should be forwarded. This routing approach is advantageous for DRAM network because it enables the data to begin transmitting before the entire packet has arrived, reducing latency and increasing throughput. By utilizing the DRAM network for memory requests, BlackParrot is able to efficiently handle large amounts of data while maintaining high levels of performance.

2.4.3 I/O Network

The I/O network in BlackParrot is designed to establish a connection between the processor and external peripherals like serial ports, PCIe controllers, I/O devices, and debugging interfaces. This network operates as a 1-D wormhole network as messages can be initiated both on and off the chip. he network's 1-D wormhole design allows for efficient routing of messages, reducing the chances of congestion and ensuring optimal performance. It is important to note that the I/O network is only present in the I/O complex and functions as a physical transport layer and transducer between BlackParrot protocols and standard protocols such as AXI, WishBone, and simple bit-banging. Since the I/O network serves as a transducer between BlackParrot protocols and standard protocols, it enables seamless integration with external devices that use different communication protocols. This means that BlackParrot can interface with a wide range of peripherals without requiring significant modifications to its hardware or software architecture.

2.5 BlackParrot Software

In this section, we elaborate on BlackParrot's software development kit (SDK), which is a collection of tools and example benchmark suites to demonstrate how to develop the BlackParrot SoC.

2.5.1 Baremetal and Linux Libraries

Libperch is BlackParrot's baremetal library, designed to provide M-mode firmware code for managing and controlling BlackParrot tiles. The library includes sample linker scripts for supported SoC platforms, start code for running bare-metal tests, emulation codes for missing instructions, ecalls, and firmware routines for printing, serial input and output, and program termination. Additionally, Libperch includes all the MMIO routines and functions necessary for user programs to communicate with and control the accelerators. When compiling a new bare-metal program for BlackParrot, users should link this library.

To facilitate program execution with POSIX I/O, BlackParrot employs PanicRoom. PanicRoom is a library that was derived from Newlib (Johnston, Jeff and Fitzsimmons, T, 2011), a C library designed for use in embedded systems. It includes a DRAM-based filesystem, LittleFS (ARM, 2017), as well as a minimal C library. The primary advantage of PanicRoom is that it eliminates the need for a complex host interface by providing an operational filesystem through the implementation of only a few platform-level operations. This makes it a valuable tool for architectures that need to run programs with POSIX I/O, as it simplifies the process of interfacing with the filesystem. The PanicRoom library is based on open-source Newlib and open-source ARM LittleFS, providing flexibility for customization and usage.

In the Linux environment, BlackParrot uses Buildroot (Petazzoni and Electrons, 2012) to generate a BusyBox root filesystem and also uses OpenSBI (OpenSBI Community, 2021) as the machine-mode firmware. BlackParrot also provides a Linux kernel driver that enables communication with the accelerator device file for configuring, reading, and writing. Additionally, the firmware extends the supervisor binary interface (SBI) calls, providing control over the accelerators.

2.5.2 Cosimulation Testing Framework

BlackParrot heavily relies on a system-level testbench, driven by program-level execution. To ensure the design's quality, a variety of testing strategies are employed, including a few directed white-box tests, and functional regression tests like the RISCV-tests suite, BEEBS suite, Spec, and CoreMark.

To provide a thorough and efficient means of testing, BlackParrot uses a hybrid ap-

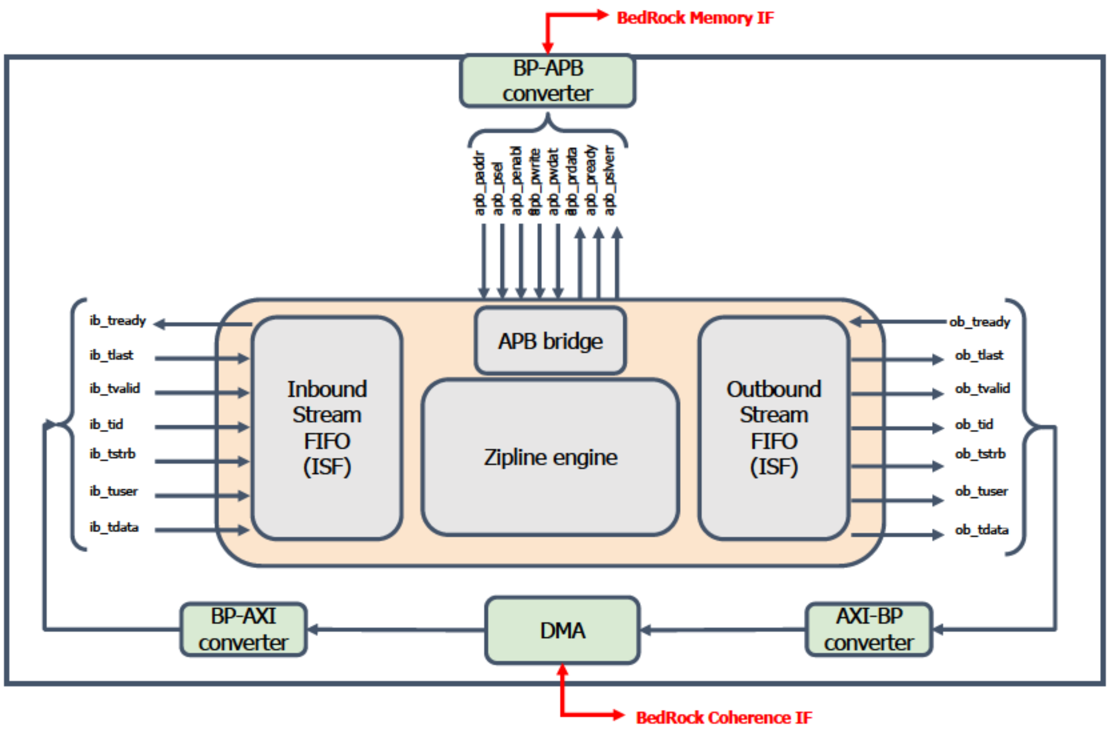


Figure 2·3: Zipline integration

proach of parallel cosimulation, utilizing Dromajo (Esperanto Technologies, 2019), an open-source RISC-V ISA simulator. Initially, a long-running program is simulated using Dromajo, with the system's architectural state collected every N cycles. This process generates both a memory dump and a checkpoint ROM, which comprises RISC-V instructions designed to initialize a freshly rebooted processor to a well-defined architectural state. Next, the BlackParrot RTL model is restored using each checkpoint ROM and cosimulated alongside the Dromajo model in parallel. To maintain alignment between the two versions, imprecise events such as interrupts and device I/O are relayed from RTL to Dromajo.

2.5.3 ParrotLine: BlackParrot Microsoft Zipline Accelerator Integration

Microsoft Zipline (Microsoft Corp., 2021) is an open-source, custom-designed compression accelerator that implements high-throughput, low-latency compression and decompression techniques. Zipline combines dictionary-based LZ77 and Huffman encoding-based lossless compression formats to achieve high compression ratios and low latency. The exact throughput and latency will depend on the chosen technology and the integration of Zipline into the system.

The Project Zipline has a Compression Encryption IP (CCEIP) that compresses and encrypts frames of data while also providing validation services (by decompressing and decrypting the frame and comparing it against CRCs). It also has a Decompression Decryption IP (CDDIP) that decrypts and decompresses frames of data.

A Command and its associated frame data will be transmitted to the CCEIP via an AXI Streaming interface. The incoming data is first stored in a front-end decoupling FIFO with a size of 12KB. To differentiate control information from data information, headers called the type/length/value (TLV) Headers have been constructed. These headers describe the type of information, the length of the data, and the actual data contents. This format allows downstream blocks to manipulate and examine the TLVs while skipping over information not intended for them. The frame data includes a series of TLV headers, and the downstream blocks will process them accordingly. In the end, the compressed/encrypted data is sent to the outbound interface formatter. This formatter contains an 8KB FIFO to manage the bursty nature of the traffic. Finally, the outbound interface formatter readies the data for transmission to the receiver engine. The same computation flow applies to CDDIP.

Additional RTL is used to integrate Zipline into BlackParrot, as shown in Figure 2·3. Zipline has an APB interface for configuring the accelerator and an AXI bus for memory communication. We integrate Zipline with BlackParrot as a streaming accelerator, allowing it to initiate data reads and writes through DMA to and from the system. To facilitate this, we develop AXI/APB converters to connect the BlackParrot memory network with the accelerator memory and add a DMA engine to fetch the input data and TLV headers into the accelerator memory.

We evaluated the performance speedup of Zipline by compressing files of various sizes.

As we increased the file size from 1KB to 8MB, the speedup decreased from 22× to 4.8×. This was due to the offloading overhead or DMA overhead becoming the dominant time component. The results illustrate how the accelerator tax can limit the end-to-end speedup and emphasize the need for realistic performance analysis.

2.6 Summary

In this chapter, we have presented BlackParrot, an agile open-source RISC-V multicore for accelerator SoCs. We have demonstrated the flexibility and ease of adoption of BlackParrot due to its tiled-based architecture. We also discussed how BlackParrot supports both streaming and coherent accelerator integration. This helps accelerator designers and system architects to evaluate their accelerator-related ideas using hardware implementation rather than software simulation and find the integration strategy that has low offload and synchronization overheads for their application to improve the overall application time.

Chapter 3

RISE: RISC-V SoC for HE Client-Side Operations Acceleration

Homomorphic encryption provides a means to process confidential data while preserving its privacy. However, HE operations are memory and compute-intensive on both the cloud and client sides, making efficient hardware acceleration necessary. Several existing works take advantage of software and hardware optimizations to accelerate cloud-side HE operations. However, unfortunately, there is no hardware accelerator to address the end-to-end performance bottlenecks in the client-side operations, i.e., the message-to-ciphertext and ciphertext-to-message conversion operations, to enable practical HE applications. To bridge this gap, this chapter presents RISE: a RISC-V SoC for accelerating HE client-side operations. RISE speeds up error sampling, encryption, and decryption operations, which are the performance bottleneck in client-side operations, based on profiling results discussed in Section 1.1.2.

3.1 Background

3.1.1 Homomorphic Encryption and Ring-Learning with Errors

Ring-Learning with Errors (RLWE) is a mathematical problem that has been widely used in HE. The RLWE problem involves finding a polynomial with random errors in a ring structure, and its hardness is based on the assumption that solving it is difficult. Given a key *sk* chosen from a key distribution over *R*, an RLWE sample $(b,a) \in r_q^2$ is constructed by sampling a from $\mathcal{U}(R_q)$ and noise e randomly from an error distribution over R and computing $b = as + e(modq)$, where q is a prime number that determines the modulus of the ring R. The RLWE assumption is that the distribution of (b,a) and $\mathcal{U}(R_q^2)$ are computationally indistinguishable. The hardness of RLWE has been extensively studied and has been shown to be closely related to the hardness of various mathematical problems, such as lattice-based problems (Peikert et al., 2016).

The security of the most efficient HE schemes, such as CKKs, BGV, and BFV, relies on RLWE. HE based on RLWE is a powerful technique that allows computations to be performed on encrypted data without requiring decryption, thus preserving data privacy. In RLWE-based HE, the encryption process maps plaintext messages to encrypted messages in a ring structure, enabling various computations on the encrypted data, such as addition and multiplication. However, the decryption process is computationally intensive and requires solving the RLWE problem, which is a challenging task.

In HE computation, N and $\log Q$ values are important parameters that determine the security and efficiency of the scheme. N represents the dimension of the ring or vector space over which the homomorphic operations are performed. In other words, it defines the size of the plaintext space, and larger N values can support more complex computations but require more computational resources. $\log Q$ is the bit-length of the modulus used in the scheme. The modulus is a large prime number that is used to reduce the ciphertext values to a fixed range, preventing them from growing too large during homomorphic operations. $\log Q$ determines the size of the ciphertext space and influences the security and efficiency of the scheme. Larger $\log Q$ values increase the security of the scheme but require more computational resources. Therefore, choosing appropriate N and logQ values is crucial to balance the trade-off between security and efficiency in homomorphic computation.

The CKKS variant of the HE scheme allows computations on encrypted data containing real numbers. This is particularly important for applications such as machine learning, scientific computation, and graph analysis, which require processing and manipulating data in its native real number format. Therefore, we use the CKKS scheme to design RISE, and Section 1.1.2 provides a summary of the CKKS message-to-ciphertext and ciphertext-to-message conversion process.

3.1.2 Residue Number System

The Residue Number System (RNS) is a mathematical tool used in homomorphic encryption to speed up modular arithmetic operations by breaking down the computations on the large integer values. RNS represents an integer by its value modulo a set of k integers (called moduli) $q_1, q_2, q_3, ..., q_k$, which generally should be pairwise coprime. An integer, x, can be represented in RNS by a set of its remainders $x_1, x_2, x_3, ..., x_k$ under Euclidean division by the respective moduli. That is, $xi = x(modq)_i$ and $0 \leq x_i < q_i$ for every i. When RNS is used, HE operations can be performed with respect to each small moduli instead of a large modulus. Implementations of HE schemes take advantage of RNS to reduce computational and memory costs of HE operations on both the client and server sides.

3.1.3 Number Theoretic Transform

Polynomial multiplication is a critical step in CKKs encryption and decryption operations. A naïve approach to perform a polynomial multiplication has a complexity of $O(N^2)$ multiplications for a polynomial of degree N. Therefore, to reduce this computational complexity, an NTT operation is applied to the polynomials so as to perform a point-wise multiplication. Using NTT we can reduce the polynomial multiplication complexity to $O(N \log N)$. NTT can be viewed as the finite field version of fast Fourier Transform (FFT). During NTT, coefficients of the input polynomial are multiplied with the power of an N-th primitive root of unity and combined with each other in a butterfly fashion. Before each polynomial multiplication takes place in an encryption operation (see Equation (1.1) and (1.2)), the polynomials are converted into an NTT domain. Similarly, we need to perform an inverse

NTT (iNTT) operation in the decryption operation. Both NTT and iNTT operations add high computational complexity to the encryption and decryption operations, respectively.

3.2 Related Work

HE schemes rely on lattice-based and post-quantum techniques. Therefore, they exhibit numerous algorithmic similarities with post-quantum encryption schemes such as CRYSTALS-KYBER (Avanzi et al., 2021), NewHope (Alkim et al., 2016), SABER (Ribeiro et al., 2021), and NTRU (Guillen et al., 2017). However, HE schemes operate on vectors and in rings that are considerably larger and its message-to-ciphertext and ciphertext-to-message conversion algorithm is compute intensive and has a very high memory usage. For the client devices that are constrained by power, performance, and area, we need to develop efficient software and hardware solutions.

Software-Based Solutions

Microsoft SEAL (Microsoft Research, 2018) (Secure Encrypted Arithmetic Library) is a powerful HE library that enables arithmetic operations on encrypted integers and real numbers. However, the computations required for HE can be resource-intensive, particularly on client devices with limited processing power and memory. To address this issue, a new version of SEAL called SEAL-Embedded (Natarajan and Dai, 2021) has been developed. SEAL-Embedded uses techniques like RNS partitioning, data type compression, memory pooling, and reuse to reduce memory consumption and make HE more feasible on resource-constrained devices. However, since SEAL-Embedded is a software-based implementation of encryption operations, it is still relatively slow and may not be suitable for real-time applications.

As mentioned earlier, for a video application with a low resolution of QQVGA, SEAL-Embedded fails to encrypt even one frame per second running at 1 GHz on a RISC-V core like BlackParrot (Petrisko et al., 2020) for a practical set of scheme parameters (polynomial degree of $N = 4096$ and three 30-bit primes). This highlights the ongoing challenge of achieving efficient and practical HE on resource-constrained client devices.

Hardware-Based Solutions

Few studies have focused on accelerating client-side operations for HE (Su et al., 2020), (Yoon et al., 2019). Su et al. (Su et al., 2020) present an FPGA-based accelerator for the BGV HE scheme, which differs from the CKKS scheme supported by our work. Their BGV accelerator only supports small scheme parameters ($N = 128$, $\log Q = 27$), which are impractical for HE computation. Although the authors claim that their accelerator can be extended to larger polynomial degrees to support higher security levels, support for larger parameters is left as future work. Moreover, the accelerator is mainly optimized to achieve high performance and throughput, while ignoring area/energy efficiency. Meanwhile, Yoon et al. (Yoon et al., 2019) propose an ASIC-based en/decryption accelerator for HE operations. However, their evaluation only covers small parameters ($N = 16$). Even to support these small polynomials, it needs large buffers to store the in/outputs and the pre-computed twiddle factors, increasing the memory area.

In our work, we architect an accelerator that can perform message-to-ciphertext and ciphertext-to-message conversions for practical scheme parameters. We use CKKS HE scheme as it is a natural choice for client devices because it can efficiently perform secure computation on the type of floating-point data often sampled by sensors. Our accelerator uses data-level parallelism, shares the datapath between encryption and decryption operations, adopts memory reuse and memory reordering strategies, and eliminates the need for additional on-chip memory to store twiddle factors by computing them on-the-fly.

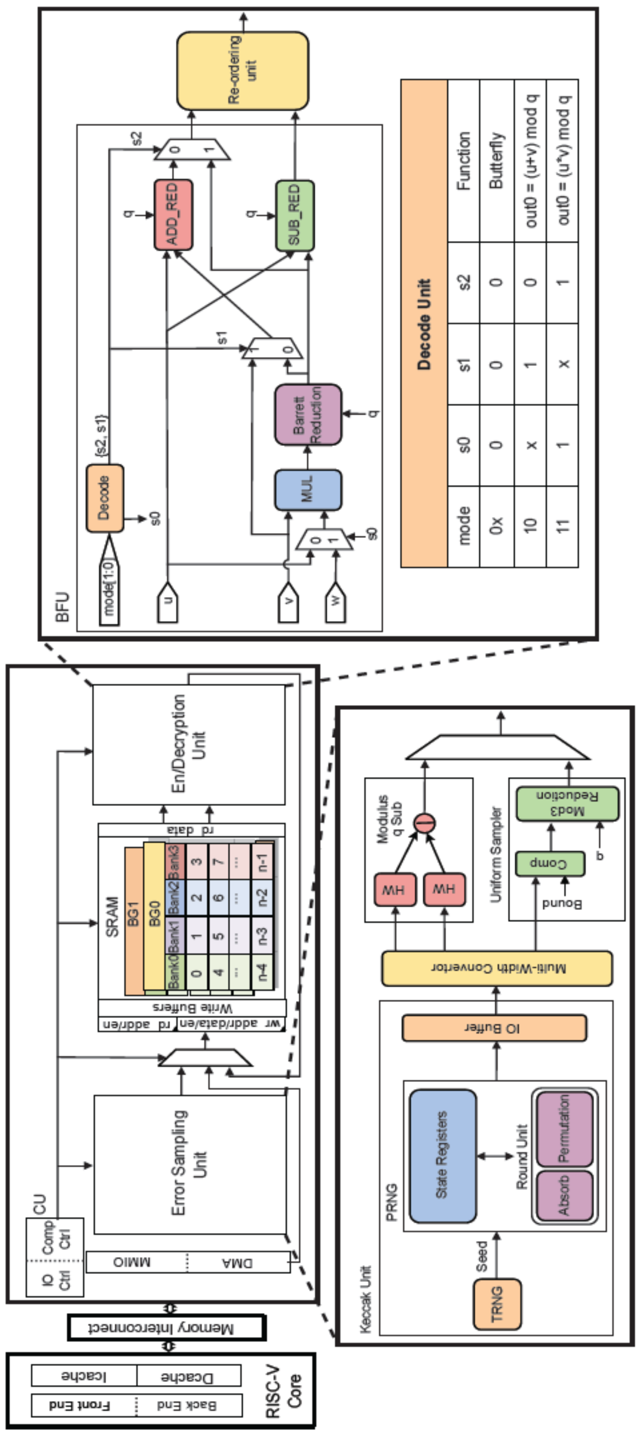
3.3 RISE: System View

This section presents the overall design of RISE, which is an end-to-end SoC (see Figure 3·1). RISE consists of a single BlackParrot RISC-V core and an accelerator that performs error sampling, encryption, and decryption. The accelerator is interfaced with the BlackParrot core in a streaming fashion because a large amount of data needs to be frequently transferred between the two. To transfer all the input data from the main memory of the BlackParrot core to the accelerator, we utilize a hardware DMA logic. The user provides public keys (pk_0, pk_1) and input message m to the BlackParrot core, which is responsible for performing en/decoding operations and the random seed generation using SEAL-Embedded library.

The accelerator contains a PRNG unit that receives a random seed from the BlackParrot core and generates a bit stream of pseudo-random numbers These pseudo-random numbers are passed to a fast error sampler to generate the required error polynomials, i.e., e_0, e_1, and μ. These error polynomials along with the encoded message and public keys are then used to perform encryption. The encryption operation performs the operations described in the Equations (1.1) and (1.2). Similarly, the decryption operation performs the operations listed in Equation (1.3). For the decryption operation, we need the ciphertext, i.e., c_0 and c_1, that is sent by the cloud and the secret key that is generated by the BlackParrot core as inputs. Once the en/decryption operation is completed, the BlackParrot core receives an interrupt from the accelerator. Then, the DMA logic transfers the output of the accelerator back to the main memory of the BlackParrot core.

3.4 RISE: Micro-Architecture

In this section, we provide a detailed description of the micro-architecture of our accelerator (see Figure 3·1).



Decode Unit

mode	s0	s1	s2	Function
0x	0	0	0	Butterfly
10	x	1	0	out0 = (u+v) mod q
11	1	x	1	out0 = (u*v) mod q

Figure 3·1: System-level view of RISE, a RISC-V SoC for accelerating message-to-ciphertext and ciphertext-to-message conversion operations on the edge for supporting homomorphic operations in the cloud.

3.4.1 Error Sampling Unit

Error samples are critical to maintaining the required security level while performing HE operations. However, generating these high-quality error samples is one of the bottlenecks in client-side operations. As shown in Figure 3·1, error sampling basically consists of two steps: generation of pseudo-random numbers using a true random seed, and generation of uniform and binomially distributed error samples using the generated pseudo-random numbers. Below we present the micro-architecture of a lightweight PRNG, a binomial sampler, and a uniform sampler.

Pseudo-Random Number Generator

We have a customized PRNG unit as part of the accelerator to speed up the pseudo-random number generation process (Keccak Team, 2018). One of the prior works (Banerjee et al., 2019) evaluated various PRNGs and concluded that the SHA-3 hash family in the SHAKE mode (Morris Dworkin, 2015), is $2\times$ and $3\times$ more energy efficient than ChaCha20 (Bernstein, 2008) and AES (Heron, 2009), respectively. This is due to the fact that SHA-3 in SHAKE mode generates the highest number of pseudo-random numbers per round. Therefore, in our PRNG unit design, we use a SHAKE function, which is more commonly referred to as Keccak. For our use case of en/decryption operation that requires a large number of error samples (as N is $>2^{12}$), Keccak makes a perfect PRNG because its output length is not predetermined. Hence, we can generate as many error samples as needed for the en/decryption operation with just one invocation of the Keccak unit.

A Keccak unit typically consists of a round unit with two sub-units: Absorb and Permutation. A true random seed (generated by TRNG), and the desired length of the pseudo-random number, and the rate at which the pseudo-random numbers are generated (provided by the BlackParrot core via control and status registers (CSRs)) are input to the Absorb sub-unit. In our design, the true random seed consists of 1600 bits. A Keccak round operates on the data organized as an array of 5×5 computation lanes, each of length 64. Hence, the absorption phase changes the random seed from a 1D 1600-bit representation into a 2D 25×64-bit representation, and we store this 2D representation in a state register (see Figure 3·1). The value in the state register is permuted by performing a series of shift, XOR, AND, and NOT operations in the Permutation unit (Keccak Team, 2018). We store the output of the Permutation unit in the state register. We set the length of the pseudo-random number to 1088 bits, which is the maximum length supported by Keccak.

Error Sampler

The output of PRNG is passed to a uniform sampler and a binomial sampler to generate error polynomials. For RLWE cryptosystems, the original worst-case to average-case security reductions hold for both continuous (rounded) Gaussian distributions and discrete Gaussian distributions. However, the implementation of efficient and constant-time Gaussian sampling is a challenging problem (Agrawal et al., 2020). Prior works (Natarajan and Dai, 2021), (Xin et al., 2020), (Duong-Ngoc et al., 2021) address this by calculating the difference of the hamming weights of two random bit streams of length k, and can rapidly obtain samples from a zero-centered binomial distribution[1] in a constant time. We adopt the same approach in our design.

Additionally, we implement a uniform sampling unit that uniformly samples the coefficients of the polynomial from $\mathcal{R}_3$ (i.e., N coefficients sampled uniformly from $\{\text{-}1, 0, 1\}$). We implement this functionality using a rejection sampling algorithm (Alkim et al., 2016). The implementation is a constant time implementation of modulo3 reduction (see Figure 3·1).

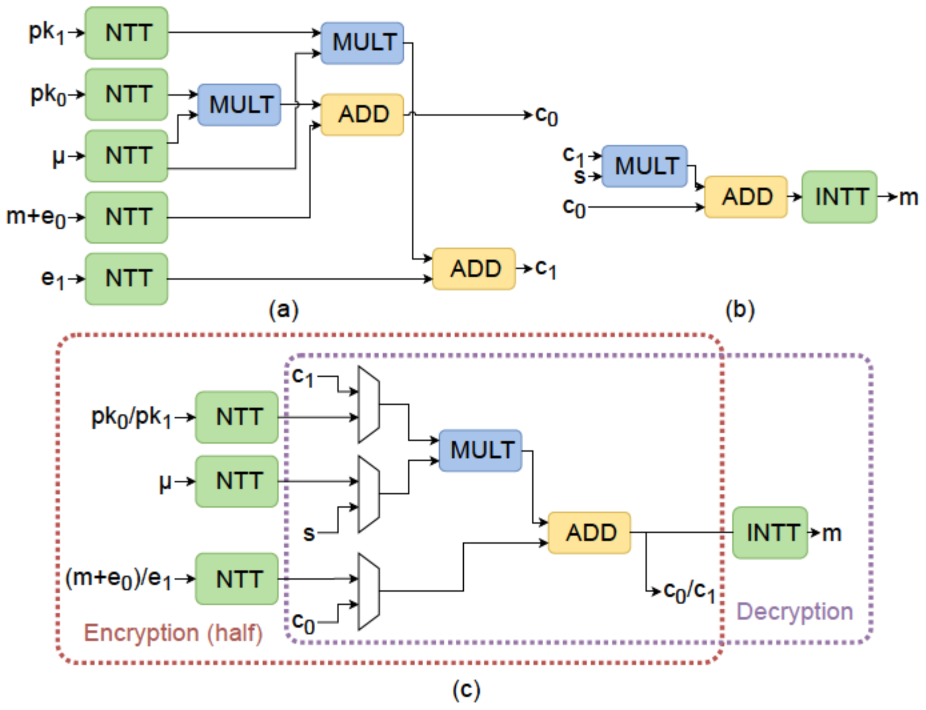


Figure 3.2: (a) Encryption dataflow. (b) Decryption dataflow. (c) Unified en/decryption dataflow. In the unified dataflow, encryption, and decryption operations share the datapath and the control logic.

3.4.2 Encryption and Decryption Unit

Figure 3·2 (a) shows the encryption datapath, which follows Equation (1.1) and (1.2). Each encryption operation calls the accelerator twice, once to compute c_0 with $(\mathsf{pk}_0, \mu, \mathsf{m}, \mathsf{e}_0)$ input set and then to compute c_1 with $(\mathsf{pk}_1, \mu, \mathsf{e}_1)$ input set. The datapath consists of polynomial addition and multiplication operations. The polynomial addition involves simple element-wise modular addition of the polynomial coefficients and has a complexity of $O(N)$. In contrast, polynomial multiplication has a complexity of $O(N^2)$, and like prior efforts, we accelerate it using NTT (more details about NTT are in Section 3.1.3). Acceleration using NTT reduces the complexity of polynomial multiplication to $O(N)$. Similarly, Figure 3·2 (b) shows the datapath for the decryption operation that follows Equation (1.3). Decryption datapath again performs polynomial addition and multiplication operation. It receives input polynomials that are already in the NTT domain. However, the decrypted polynomial is required to be in coefficient form for performing the decoding operation (we perform this operation on the BlackParrot core using the SEAL-Embedded library). Therefore, the decryption datapath has an iNTT operation.

Unified En/Decryption Datapath

In order to reduce the area overhead of the accelerator, we share the datapath and control logic of the accelerator between encryption and decryption operations (see Figure 3·2 (c)). This is possible because the sequence of operations performed in the encryption and decryption operations are the same. Moreover, the encryption operation uses the exact same sequence of operations to compute both c_0 and c_1. Thus, we use the same datapath twice to perform the complete encryption operation.

[1]Presuming this error distribution's standard deviation is sufficiently large, no known attack exploits the shape of this distribution. For our binomial distribution, we use a standard deviation of $\sqrt{21/2} \approxeq 3.24$ to comply with the HE security standard (Albrecht et al., 2021).

NTT Acceleration

The main performance bottleneck in the en/decryption unit is the NTT operation. Consequently, we propose several optimization techniques to efficiently perform NTT while incurring a low memory and area overhead. We discuss these optimizations in detail in the rest of this section.

Butterfly Unit (BFU): A Butterfly operation is the basic building block of NTT/iNTT operation. An NTT/iNTT operation consists of $\log_2 N$ stages (for a polynomial of degree N), and each stage requires $N/2$ Butterfly operations. Each BFU takes two coefficients (say a and b) out of the N polynomial coefficients as input and computes $(a,b) = (a+\omega \cdot bq,$ $a-\omega \cdot bq)$ (refer Algorithm 1 line 13 and 14). Here, ω is the twiddle factor. A degree N polynomial requires $N/2$ twiddle factors, where each twiddle factor needs $\log q$ bits. Our accelerator computes twiddle factors on-the-fly within BFU to reduce the memory overhead for storing them as pre-computed values.

BFU is fully pipelined with the throughput of 1 Butterfly operation per cycle. It is designed to perform NTT, iNTT, polynomial addition, and multiplication operations that are required by both encryption and decryption operations (see Figure 3·2). BFU has an integer adder and subtractor unit that performs modular reduction using a conditional operator. BFU contains a modular multiplier where modular reduction operation is performed using a Barrett reduction (Barrett, 1986) unit.

Barrett reduction computes modular reduction operation without performing any division and only involves two multiplications and one subtraction, shift, and conditional subtraction operation (Barrett, 1986). In addition, it does not exploit any property of the modulus q, which makes it ideal for supporting configurable moduli. The modular multiplier lies on the critical path in the accelerator. Hence, we pipeline the multiplier to reduce the critical path and improve the operating frequency of the accelerator. As power and area are the primary design goals for edge devices, all the above computations are performed by sequentially leveraging the pipelined BFU.

Memory Reuse Technique: All the necessary polynomials (m, e_0, e_1, μ, pk_0, pk_1, c_0, c_1) should be kept in the accelerator's on-chip memory for efficient en/decryption computation. A single polynomial typically needs memory of $\sim$60 KB with $N = 2^{14}$ and $\log q = 30$. Therefore, we require a total of 480 KB to hold all the in/output polynomials. In our memory reuse strategy, we manage the encryption and decryption operations such that at any given time, we need to store a maximum of only two polynomials, which takes 120 KB of space.

For memory reuse, we divide the entire on-chip SRAM memory into multiple banks that are organized into two bank groups, i.e., BG0 and BG1. Each bank group corresponds to a single polynomial and each polynomial is stored across multiple banks within a bank group. During the encryption and decryption operation, we use these bank groups to store the input, output, and intermediate polynomials. Hence, we share each of the two bank groups among several polynomials as shown in Figure 3·3 (a) and (b).

As an illustration (see Figure 3·3), we carry out an in-place NTT in an encryption operation that gets the data for polynomial μ from BG0, processes it, and then writes the results back to BG0. While we are still performing NTT on the polynomial μ, we load the next input polynomial pk_1 into BG1 in parallel. The modular addition and multiplication operations involve memory reuse as well. Both of these operations read the input from BG0 and BG1 while writing the output to bank group BG1. Therefore, after the modular addition or multiplication operations are complete, we can reuse BG0 for the subsequent operation. Therefore, by utilizing a memory reuse strategy, we can efficiently perform en/decryption operations while incurring a minimal memory footprint.

Memory Reorder Technique: The next memory level optimization that we perform is mem-

Algorithm 1: NTT_swap4

```
**Input:** Polynomial $a(x) \in Z_q[x]$ in bit-reversed order
**Output:** $NTT(a(x))$ in normal order
$m = 2$;
**for** $(stage = 0; stage < (\log N - 1); stage+ = 1)$ **do**
    $\omega = 1$; $\omega_m = \omega_n^{2^{\log N - 1 - stage}}$; $upd\_cnt = 1$;
    **for** $(j = 0; j < m * 2; j+ = 4)$ **do**
        **for** $(k = 0; k < N; k+ = m * 4)$ **do**
            i0=[]; i1=[];
            **for** $(l = 0; l < 4; l+ = 1)$ **do**
                **switch** $l$ **do**
                    **case** 0 **do** $idx = j + k$ ;
                    **case** 1 **do** $idx = j + k + 2$ ;
                    **case** 2 **do** $idx = j + k + m * 2$ ;
                    **case** 3 **do** $idx = j + k + m * 2 + 2$ ;
                $a[idx] = a[idx] + a[idx + 1] * \omega q$;
                $a[idx + 1] = a[idx] - a[idx + 1] * \omega q$;
                i0.append($idx$); i1.append($idx + 1$);
                **if** $upd\_cnt == N/(2^{stage+1})$ **then**
                    $\omega = \omega * \omega_m q$; $upd\_cnt = 1$;
                **else** $upd\_cnt+ = 1$ ;
            $(a[i0[0]], a[i1[0]], a[i0[1]], a[i1[1]],$
             $a[i0[2]], a[i1[2]], a[i0[3]], a[i1[3]]) =$
            $(a[i0[0]], a[i0[1]], a[i0[2]], a[i0[3]],$
             $a[i1[0]], a[i1[1]], a[i1[2]], a[i1[3]])$
    $m = (m == N/4)$ ? 2 : $(m * 2)$;
**for** $(i = 0; i < N; i+ = 1)$ **do**
    Bit manipulation $phy\_addr = \{i[\log N - 3 : 2], i[\log N - 1 : \log N - 2], i[1 : 0]\}$ ;
    $a\_out[i] = a[phy\_addr]$;
return $a\_out$;
```

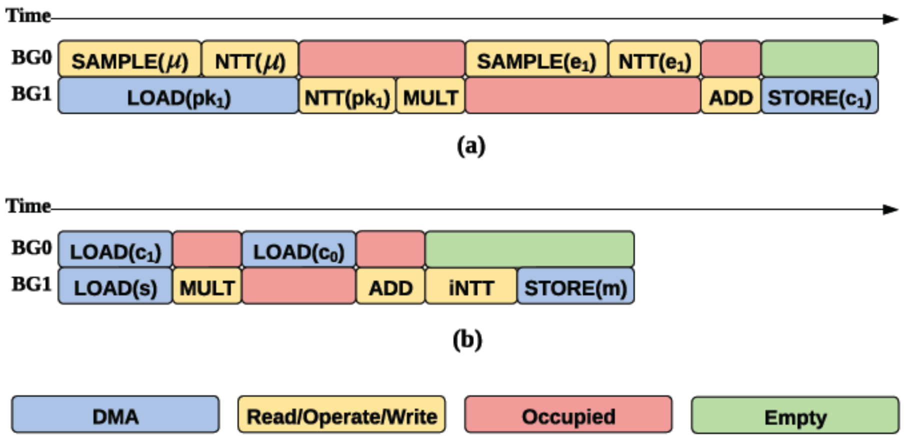


Figure 3·3: Memory reuse during (a) encryption and (b) decryption operations. Each BG can store only one polynomial. "Read/Operate/Write" means the bank group is being accessed during the operations. "Occupied" means the bank group stores intermediate results.

ory reorder, which helps reduce the number of memory ports required, resulting in low memory area overhead. Every Butterfly operation takes as input two coefficients of the polynomials, operates on them, and stores the resultant values back to the same memory banks. As a result, a naïve implementation of NTT will require 2 read and 2 write ports (2R2W) for each memory bank that is of size N. Typically, a 2R2W memory bank is roughly twice as large as 1 read and 1 write port (1R1W) memory bank of the same size. Consequently, we can save half of the memory area simply by switching from a single 2R2W bank of size N to two 1R1W banks of size $N/2$.

However, managing memory access patterns for NTT, with 1R1W banks, becomes challenging as the memory accesses can lead to bank conflicts. Throughout all the NTT stages, the distance $((j-i))$ between the two inputs of a Butterfly operation changes. This leads to bank conflicts in several stages of NTT as each stage in NTT iterates through all values from 1 to $N/2$. Thus, replacing 2R2W bank with 1R1W banks is not trivial. Some of the prior works (Nannipieri et al., 2021), (Li and Liu, 2021), (Chen et al., 2022), (Duong-Ngoc et al., 2021), (Xin et al., 2020), (Banerjee et al., 2019), (Roy et al., 2014) address this issue by modifying the NTT algorithm itself. For example, to use 1R1W memory banks for an NTT, Roy et al. (Roy et al., 2014) proposed a memory-efficient NTT algorithm, which we refer to as the NTT_swap2 algorithm. Their technique rearranges the output of the two subsequent Butterfly operations to prevent bank conflicts (two 1R1W banks). As a result, it guarantees that the input pair required by the Butterfly operation in the following stage is in distinct memory banks.

Although using a 1R1W memory bank reduces memory space by half, there is still scope for improvement. To further reduce memory area overhead, we suggest replacing two 1R1W banks of size $N/2$ with four 1 read/write port (1RW) banks of size $N/4$. This causes newer bank conflicts, which cannot be addressed by using the NTT_swap2 method. For example, if a bank receives both read and write requests at the same time, we will need an additional write buffer to store the write requests. Now the write requests must wait in the write buffer until there are no incoming reads before opportunistically writing back the results. Although using a write buffer is a good way to solve bank conflicts, the size of the write buffer quickly grows. Our observation is that if there are $N/4$ continuous read and write accesses to the same bank in a given stage, the write buffer must be the same size as the banks ($N/4$) in order to hold all write requests that overlap with read requests to the same bank. If we were to use the same size buffers as the memory banks, we incur the same memory overhead as the $1R1W$ memory bank, making this solution impractical.

We propose a method called NTT_swap4 (refer Algorithm 1) to avoid using these large write buffers. NTT_swap4 reorders the output of four successive Butterfly operations, while NTT_swap2 reorders the result of only two Butterfly operations. (see Figure 3·4). This is to ensure that not only the two inputs of each Butterfly operation are stored in different banks (like NTT_swap2), but also the inputs of consecutive Butterfly operations are stored in different banks (NTT_swap4). As the same bank is not repeatedly used in this scenario, the write buffer can immediately write back the outcomes in the subsequent

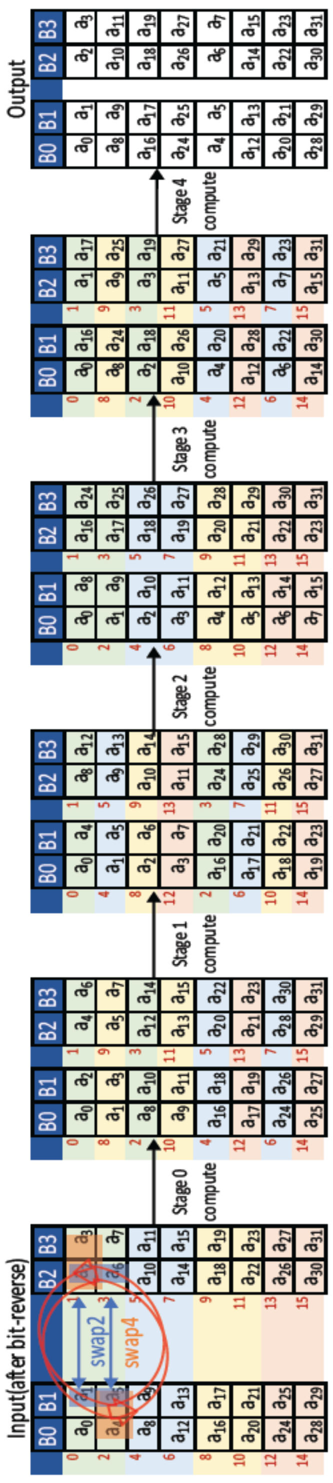


Figure 3·4: NTT_swap4 with $N = 32$. The red-colored numbers before each pair of cells denote the order of Butterfly operations. The four consecutive Butterfly operations (2 rows) being reordered are denoted with the same color.

cycle. Thus, the write buffer can be as small as one element wide ($\log q$) for a memory bank.

We demonstrate NTT_swap4 technique example (for $N = 32$) in Figure 3·4. The order of the Butterfly operations is indicated by the numbers (in red) before each pair of cells. For example, in stage 0, the first four Butterfly operations access the following pairs: (a_0, a_1), (a_2, a_3), (a_4, a_5), (a_6, a_7). However, stage 1 expects elements in the order of (a_0, a_2), (a_4, a_6), (a_1, a_3), (a_5, a_7). To prevent successive Butterfly operations in stage 1 from accessing the same banks for reads and writes, we reorganize stage 0's outputs into the order anticipated by stage 1 (refer Algorithm 1 line 19). To carry out this reordering, we use a Reordering Unit (RU).

Re-ordering Unit (RU): The RU reorders the output generated by the BFU and writes it back into the memory banks. A small register array that can store up to 8 pairs of Butterfly outputs and a reordering logic make up the RU. Reordering logic begins by sequentially writing the two results of a Butterfly operation and their addresses to the register array in each cycle. Once there are eight elements in the register array or four pairs of BFU outputs, the reordering logic will send out the elements stored in the registers to the corresponding memory bank. Both NTT and iNTT operations can be reordered effectively using RU. The RU will be active only while doing NTT/iNTT computations based on the *mode* signal (see Figure 3·1).

Control Unit (CU): The CU consists of two components – the computation controller and the I/O controller. Based on the current operation (error sampling, NTT/iNTT, modular addition, and multiplication), the computation controller, which is an FSM, chooses the BFU and RU mode signals. In addition, it generates the enable signal and read/write addresses for memory bank accesses. During NTT/iNTT operation, the computation controller is also in charge of setting up the NTT unit to compute the twiddle factors on-the-fly. Depending on the type of CPU request received by the accelerator (encryption or decryption), the I/O controller chooses the necessary set of BFU operations. Besides, based on the current en/decryption stage, it also configures the DMA unit for the input/output data transfer to/from memory banks.

Parallel NTT Computation

In this section, we present a technique to parallelize NTT to further improve its performance. This is due to the fact that the area-efficient NTT design that we discussed above cannot meet the performance requirements of high-end edge devices and several high-speed applications. Therefore, by parallelizing the NTT computation, we can improve the performance at the cost of area and power overhead. We evaluate this performance vs. area/power trade-off to identify the optimal architectures for different design objectives in Section 3.5.

To improve the performance of NTT computation, we can perform multiple Butterfly operations in parallel. Therefore, we propose a scalable parallel implementation of NTT with multiple BFUs. To support a parallel NTT architecture using multiple BFUs, we need to address two main requirements: 1. Multi-port memory banks to read/write multiple BFUs' inputs and outputs simultaneously and 2. On-the-fly computation of multiple twiddle factors to enable multiple Butterfly operations in parallel.

Memory Bank Organization for Parallel NTT: Moving to a multi-port memory bank design is not an efficient solution as increasing the number of ports will quadratically increase the memory area overhead (Azad et al., 2022). To reduce the memory area overhead, we still use the $1RW$ memory banks but we linearly increase the number of memory banks as we increase the number of BFUs. However, we keep the total memory size the same by proportionally decreasing the size of each memory bank. With an increase in the number of memory banks, the data access pattern within each stage of the NTT becomes complicated resulting in data dependencies that need to be carefully managed. Consequently, our proposed memory reorder technique (see Section 3.1.3) will not work as it is and requires modifications.

We extend our memory reorder scheme to get rid of the memory bank conflicts by reordering the output of $4 \times \#BFUs$ Butterfly operations instead of reordering the output of only four successive Butterfly operations (as proposed in NTT_swap4 technique). For example, for $N = 32$ with 2 parallel BFUs, in stage 0, the first eight Butterfly operations access the following pairs: (a_0, a_1), (a_2, a_3), (a_4, a_5), (a_6, a_7), (a_8, a_9), (a_{10}, a_{11}), (a_{12}, a_{13}), (a_{14}, a_{15}). However, stage 1 expects elements in the order of (a_0, a_2), (a_4, a_6), (a_8, a_{10}), (a_{12}, a_{14}), (a_1, a_3), (a_5, a_7), (a_9, a_{11}), (a_{13}, a_{15}). To prevent successive Butterfly operations in stage 1 from accessing the same banks for reads and writes, we reorganize stage 0's outputs into the order anticipated by stage 1.

Twiddle Factor Computation for Parallel NTT: As discussed earlier, to minimize the memory area overhead RISE computes the required twiddle factors on-the-fly instead of storing the precomputed values. However, now as we increase the number of BFUs for the parallel NTT approach, we need to compute many twiddle factors in parallel, thus introducing stalls in the NTT computation pipeline that offsets the performance gains. The stalls are introduced because RISE's area-efficient design shares the BFU to compute the twiddle factor and to perform the Butterfly operation. To eliminate pipeline stalls, we introduce a separate modular multiplier to compute twiddle factors in parallel with the Butterfly operations. We note that we also need to increase the number of modular multipliers that are used to compute twiddle factors, as we increase the number of BFUs.

3.5 Evaluation

In this section, we evaluate RISE performance, area and energy efficiency for different security parameters and with different level of NTT parallelism.

3.5.1 Experimental Setup

For our analysis, we run all edge-side operations on the following systems in bare-metal mode:

- Baseline: BP processor executes all the operations from SEAL-Embedded library.
- RACE (Azad et al., 2022): In the RACE SoC, the hardware accelerator executes the en/decryption operation, while the remaining operations (error sampling and en/decoding) are performed on the BP processor. We modified SEAL-Embedded library to invoke calls to en/decryption operations on the accelerator.
- RISE: In the RISE SoC, the hardware accelerator performs error sampling and executes the en/decryption operation while the remaining operations (en/decoding) are performed on the BP processor. RISE supports a range of parallel BFUs (1 to 32) within a single NTT operation. In our evaluation, RISE-1BFU and RISE-MaxBFU correspond to configurations with 1 BFU and 32 BFUs, respectively.

All three systems, i.e., baseline, RACE, and RISE, make use of a single core BP configuration (32 KB each of Icache and Dcache) running at 1 GHz. We implement all three systems in SystemVerilog and simulate them using VCS. The hardware implementation is cycle-accurate and captures the nuances of data movement between all parts of the systems. For power, performance, and area evaluation, we use GlobalFoundries 12nm technology. We synthesize the logic components in baseline, RACE, and RISE using Synopsys Design Compiler, and use memory compiler for designing the SRAM arrays.

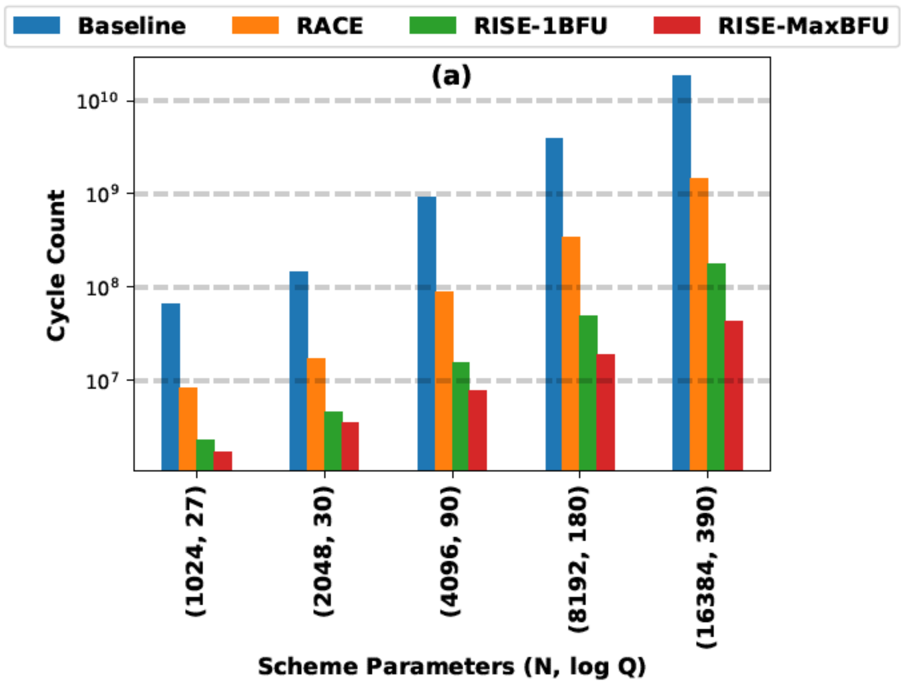
3.5.2 Experimental Results

Performance Results

We evaluate RISE performance with different numbers of BFUs (1 to 32) for both message-to-ciphertext and ciphertext-to-message conversion operations for a range of scheme parameters.

As shown in Figure 3·5 (a), across different scheme parameter (N, $\log Q$) values, RACE configuration achieves 7.8×-12.58× and 7.9×-66.08× better performance for message-to-ciphertext and ciphertext-to-message conversion operations, respectively, compared to the baseline. The performance improvement in RACE is because we offload the encryption and decryption operations to the hardware accelerator, which speeds up encryption by 89.82×-123.76× and decryption by 204.66×-244.44×. In RISE-1BFU configuration, on top of encryption and decryption operations, we also offload the error sampling operation to the hardware accelerator, which results in 1726.39×-1734.08× speed up in the error sampling. However, RISE-1BFU configuration achieves just 3.68×-8.29× better performance for message-to-ciphertext conversion operation compared to the RACE as the performance improvement is limited by Amdahl's law.

RISE-1BFU configuration achieves similar performance as RACE for ciphertext-to-message conversion operation (refer Figure 3·5 (b)), as this conversion does not include the error sampling step. As we increase the number of BFUs within the NTT/iNTT operation to perform multiple Butterfly operations in parallel, we observe a speed-up in encryption and decryption operation. As shown in Figure 3·5 (a) and (b) for RISE-MaxBFU configuration, with maximum number of BFUs (32), compared to RISE-1BFU the message-to-ciphertext and ciphertext-to-message conversion performance improves by 37.7%-414.92% and 3.4%-35%, respectively. Overall, compared to the baseline system, our RACE-MaxBFU improves the message-to-ciphertext conversion performance by 38.27×-433.14×, and the ciphertext-to-message conversion performance by 8.2×-89.25×.



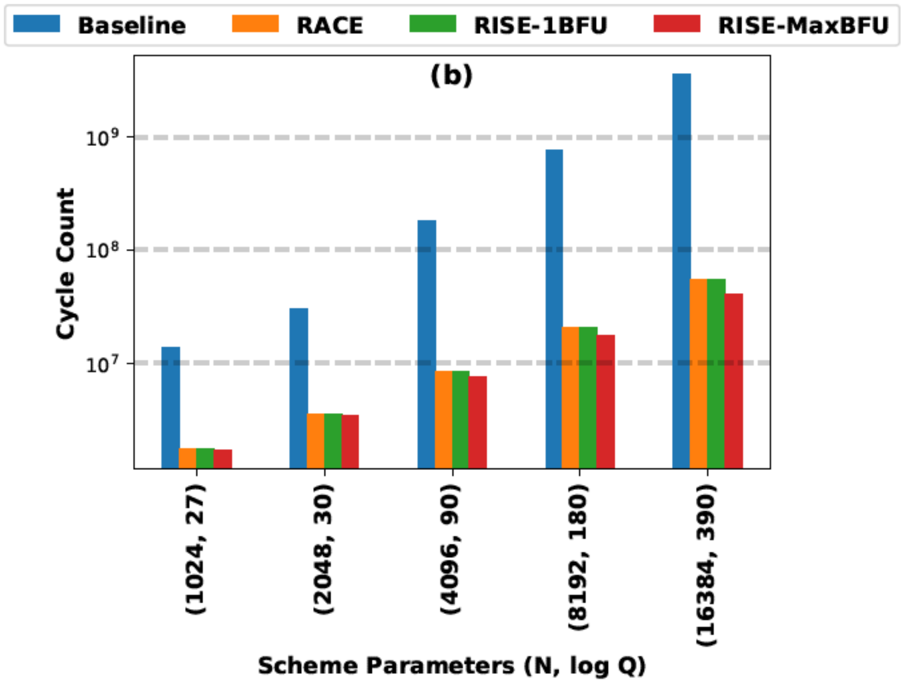


Figure 3·5: Latency (in clock cycle count) of (a) message-to-ciphertext and (b) ciphertext-to-message conversion operations for baseline, RACE, RISE-1BFU, and RISE-MaxBFU with 1 GHz frequency.

Comparison with Related Works:

Table 3.1 presents the performance comparison between RISE and other relevant state-of-the-art works. The performance comparison includes the latency of NTT operation for a single limb (the dominant operation in message-to-ciphertext and ciphertext-to-message conversion), the SRAM size, the number of memory ports, and the evaluation platform (ASIC or FPGA). As other existing works use different parameters (N and $\log q$), we compute the performance numbers for RISE for all of these parameter sets. It is evident from the table that RISE performs faster NTT computation when compared to other designs for all values of N except for (Li and Liu, 2021). Moreover, for every value of N, RISE utilizes only single-port memory with the smallest SRAM size.

Thanks to our highly parallel and pipelined NTT computation design that leads to low NTT computation latency. Li et al. (Li and Liu, 2021) can perform a single NTT in 38 cycles as they store precomputed twiddle factors in SRAM, which leads to $10\times$ higher memory requirement than RISE. In addition, they need dual-port memories to feed the input to their vectorized NTT unit. RISE manages to perform NTT computations while using only a single-port memory by leveraging NTT_swap4 method. Compared to the related work, RISE has the smallest memory footprint because of our on-the-fly twiddle factor generation unit and in-place NTT computation. We do not provide a head-to-head comparison of the performance of RISE with the works by (Su et al., 2020), (Yoon et al., 2019) as those prior works accelerate en/decryption operations to support HE operations for the BGV scheme while we enable the support for CKKS scheme.

Power Results

Figure 3·6 (a) shows the power consumption in the message-to-ciphertext and ciphertext-to-message conversion operations for different scheme parameter (N, $\log Q$) values when using baseline, RACE, RISE-1BFU, and RISE-MaxBFU systems. message-to-ciphertext

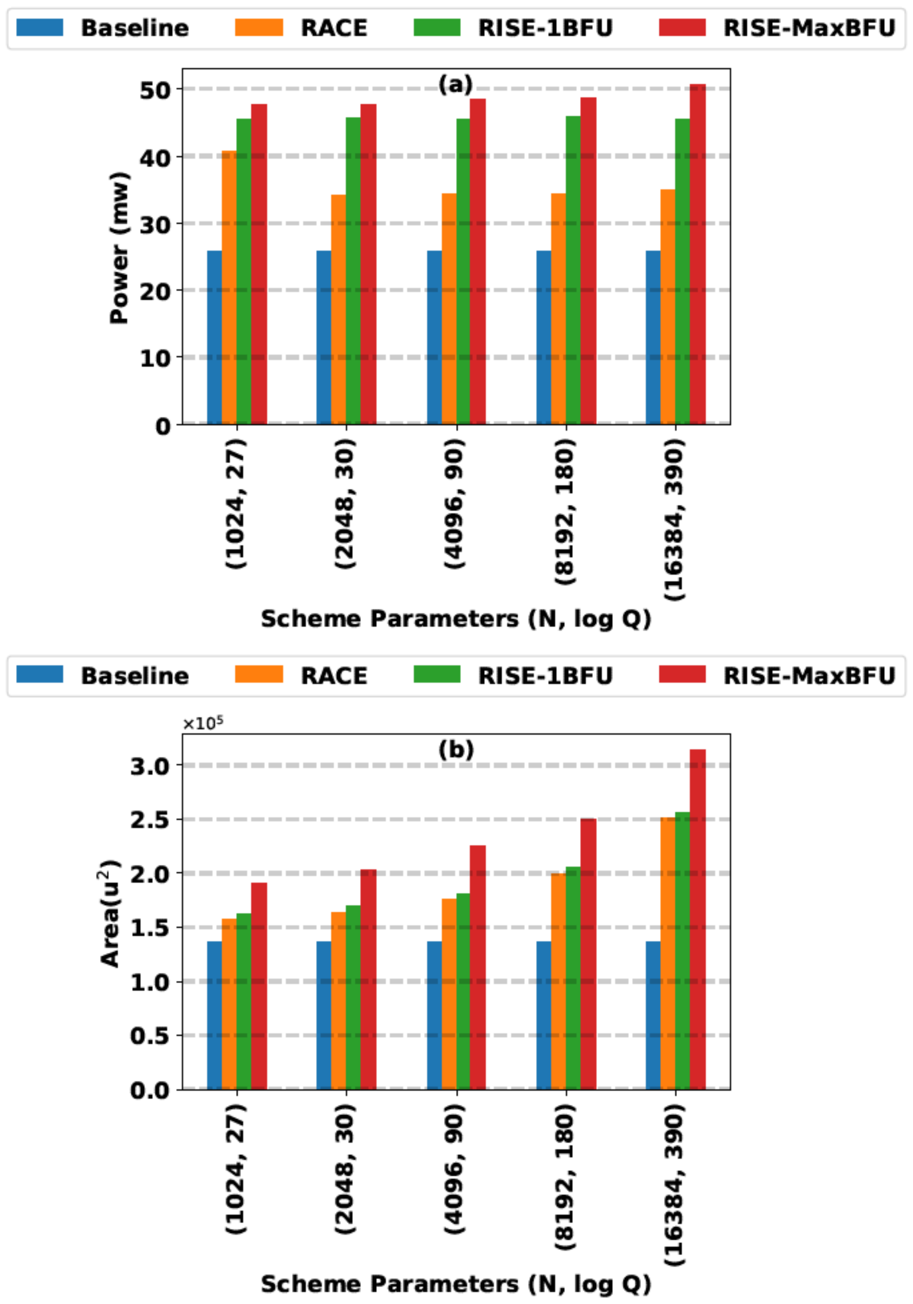


Figure 3·6: (a) Power consumption and (b) area utilization for baseline, RACE, RISE-1BFU, and RISE-MaxBFU.

Table 3.1: NTT operation Performance (cycle count) comparison with the state-of-the-art designs in related works. A head-to-head comparison in terms of frequency, power and area numbers cannot be done because of differences in platforms (ASIC vs FPGA) and technology nodes.

Design	N	$\log q$	Latency (Clock Cycles)	SRAM Size (KB)	SRAM R/W Ports	Platform
(Nannipieri et al., 2021)	256	16	18554	2.25 KB	Dual	FPGA
(Banerjee et al., 2019)		24	1289	45 KB	Single	ASIC
(Li and Liu, 2021)		16	556	13.5 KB	Dual	FPGA
(Chen et al., 2022)		14	327	22.5 KB	Dual	FPGA
RISE		30	103	0.93 KB	Single	ASIC
(Li and Liu, 2021)		16	38	10 KB	Dual	FPGA
(Ye et al., 2022)	512	16	1074	18 KB	Dual	FPGA
RISE		30	215	1.87KB	Single	ASIC
(Paludo and Sousa, 2022)	1024	28	2568	108 KB	Dual	FPGA
(Ye et al., 2022)		28	2114	27 KB	Dual	FPGA
(Su et al., 2022)		32	650	355.5 KB	Dual	FPGA
RISE		30	447	3.75KB	Single	ASIC
(Ye et al., 2022)	4096	60	8284	110.25 KB	Dual	FPGA
(Su et al., 2022)		32	3075	355.5 KB	Dual	FPGA
RISE		30	1918	15KB	Single	ASIC
(Duong-Ngoc et al., 2023)	16384	60	536832	616.5 KB	Dual	FPGA
RISE		60	34814	480 KB	Single	ASIC

and ciphertext-to-message conversion operations have similar power consumption, within 0.01%, and we report the power consumption for the message-to-ciphertext[2]. The total power consumption for a message-to-ciphertext and ciphertext-to-message conversion in the baseline system is 27.19 mW, out of which the SRAM power consumption is 41.49% = 11.4 mW and the digital logic consumes the remaining power. Overall, the power consumption of RACE is about 25%-28% (for a range of scheme parameters) higher than the baseline system for both message-to-ciphertext and ciphertext-to-message conversion operations. The increase in the power consumption is due to 41.92%-43.55% and 3.36%-7.81% power increase in the digital logic and SRAM, respectively. The power consumption in RISE-1BFU configuration increases by 11.62%-30.15% compared to RACE

[2]The ciphertext-to-message conversion does not perform the error sampling operation and so should have lower power consumption than the message-to-ciphertext conversion. However, we did not power gate or clock gate the error sampling unit during the ciphertext-to-message conversion and so it consumes some power even during the ciphertext-to-message conversion. The error sampling operation takes less than 10% of the total time required to perform message-to-ciphertext conversion, and so is not the dominant component. Hence, the power consumed during message-to-ciphertext and ciphertext-to-message conversions are comparable

due to the additional digital logic required for the error sampling unit. As we increase the number of BFUs from 1 to 32, the power consumption increases by 4.49%-10.98% due to the more complex memory banking logic (14.61%-30.66%) and multiple parallel BFUs (1.01%-4.11%).

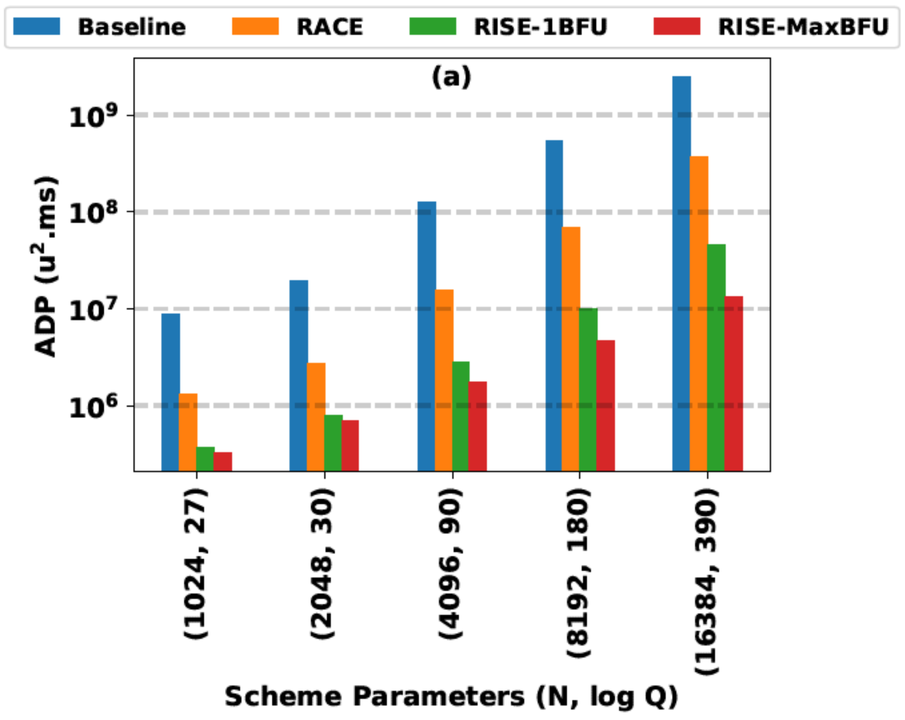
Area Results

The area of RACE is 15% (smallest N) to 84% (largest N) larger than the area of the baseline system. (see Figure 3·6 (b)). This increase in the area is due to the area required by the accelerator where SRAMs primarily contribute to the increase in area. The area overhead in RISE-1BFU is (3.65%-2.14%) compared to RACE, as error sampling contributes very little to the overall area of RISE.-1BFU With an increase in the number of parallel BFUs, the complexity of control logic and memory banking increases. Hence, as shown in Figure 3·6 (b), by increasing the number of BFUs from 1 to 32, the area overhead of RISE-MaxBFU increases by 17.59% to 22.38% as compared to RISE-1BFU for different scheme parameters.

Area and Energy Efficiency

RISE aims to improve the performance of message-to-ciphertext and ciphertext-to-message conversion at the cost of increase in area and power. Thus, Area-Delay Product (ADP) and Energy-Delay Product (EDP) metrics need to be considered for evaluating our RISE design. Figure 3·7 (a) and (b) compares the ADP value for baseline, RACE, RISE-1BFU, and RISE-MaxBFU systems. As we can see, RACE decreases the message-to-ciphertext and ciphertext-to-message conversion ADP by 6.76×-7.78× and 6.89×-35.80× compared to the baseline, respectively. The improvement is the result of 7.8×-12.58× and 7.95×-66.08× improvement in performance while incurring only a 15%-84% increase in the area.

The Figure 3·7 (a) also shows that in RISE-1BFU, the message-to-ciphertext conversion ADP outperforms RACE (3.55×-8.12× lower). This is due to 3.68×-8.29× perfor-



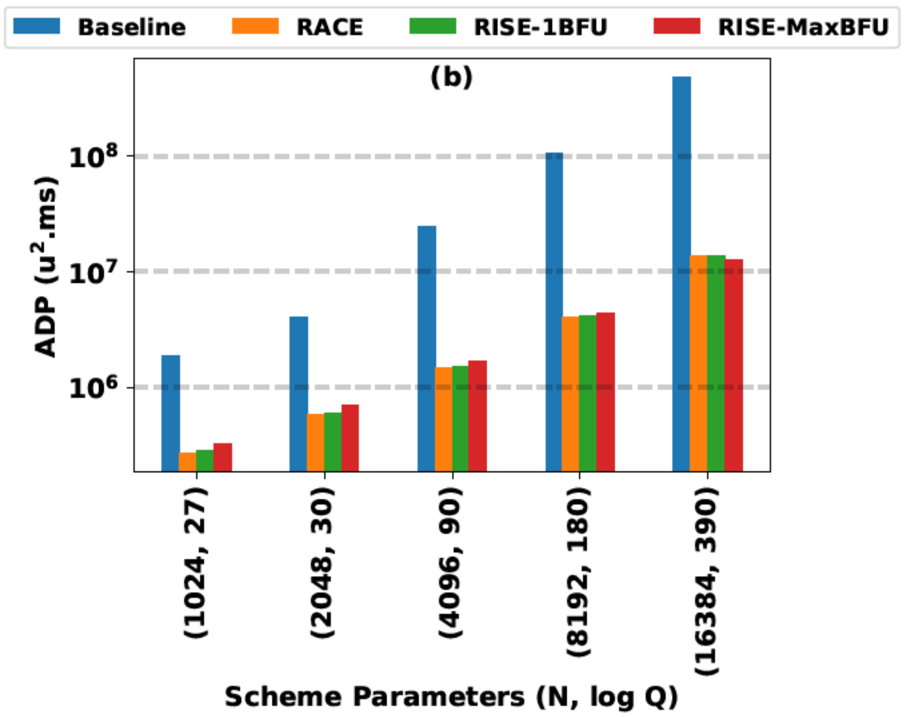
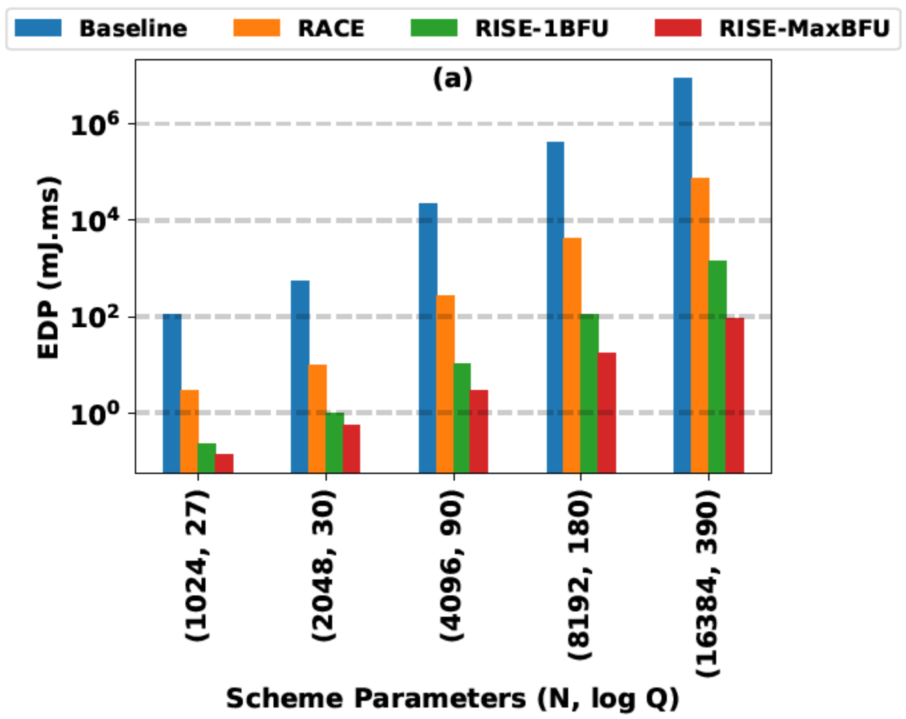


Figure 3·7: ADP of (a) message-to-ciphertext/ and (b) ciphertext-to-message conversion operations for baseline, RACE, RISE-1BFU, and RISE-MaxBFU.

mance improvement while incurring only 3.65%-2.14% increase in area. Increasing the number of BFUs from 1 to 32 improves the message-to-ciphertext conversion ADP by 1.13×-3.39× for different scheme parameters (due to 1.37×-4.14× performance improvement and 17.59%-22.38% area overhead). For the ciphertext-to-message conversion the ADP (refer Figure 3·7 (b)) of the RISE-1BFU system underperforms RACE by 3.65% for the smallest N and 2.14% for the largest N as there is an increase in area overhead due to the additional error sampling unit, which is not used by ciphertext-to-message conversion. Increasing the number of BFUs to 32 worsens the ciphertext-to-message conversion ADP of RISE-1BFU by up to 12.3% for small N values. This is because in RISE-1BFU the decryption operation only accounts for 3.43%-10.40% of the total latency, which when improved by adding parallel BFUs within iNTT, does not improve the performance by the same proportion as the area overhead (17.59%-22.38%). For large N values, the ciphertext-to-message conversion ADP increases by up to 10.35% as now decryption operation contributes significantly to the total latency, which can be accelerated by instantiating parallel BFUs.

Figure 3·8 (a) and (b) compare the energy efficiency of baseline, RACE, RISE-1BFU, and RISE-MaxBFU systems. As evident from the figures, EDP follows a similar trend as ADP. There is 38.6×-117.09× and 40.19×-3229.81× improvement in the EDP for message-to-ciphertext and ciphertext-to-message conversion, respectively when using RACE as compared to the baseline. The EDP of RISE-1BFU for message-to-ciphertext conversion is 12.19×-52.87× better compared to RACE and by increasing the number of parallel BFUs, EDP further improves by 1.69×-15.51×. Unfortunately, EDP of RISE-1BFU for ciphertext-to-message conversion worsens by 11.62%-30.15% compared to RACE for the same reason as ADP. The EDP of RISE-1BFU for ciphertext-to-message conversion can be improved by clock gating the error sampling unit. Moreover, by using 32 BFUs the ciphertext-to-message conversion EDP improves by 1.71%-64.35% compared to



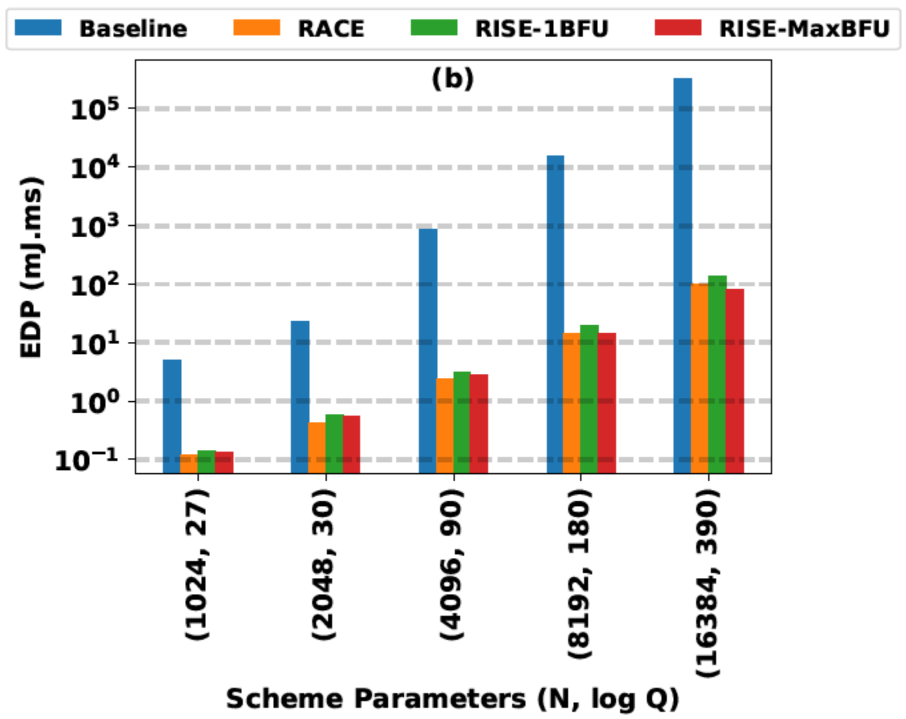


Figure 3·8: EDP of (a) message-to-ciphertext/ and (b) ciphertext-to-message conversion operations for baseline, RACE, RISE-1BFU, and RISE-MaxBFU.

RISE-1BFU. This improvement is due to the fact that increasing the number of BFUs improves decryption operation performance, which leads to up to 35% performance improvement for ciphertext-to-message conversion. We also get up to 21.69% energy consumption reduction as the BlackParrot core consumes less idle energy.

Video Application Evaluation

We evaluate our RISE design using QQVGA and QVGA video frame encryption examples. For calculating the number of ciphertexts required to encode and encrypt each of these frames refer to Section 1.1.2. Figure 3·9 (a) and (b) shows the maximum frames per second (FPS) that the baseline, RACE, RISE-1BFU, and RISE-MaxBFU systems can sustain for different scheme parameter (N, $\log Q$) values when performing message-to-ciphertext conversion operation for QQVGA and QVGA, respectively. The frames are sent to the cloud using a mid-band 5G network, which offers a balance between speed, capacity, and coverage (Thummaluru et al., 2019). As shown in Figure 3·9, in the regions with maximum bandwidth, mid-band 5G network can transfer up to 70 (QQVGA) and 23 (QVGA) frames per second for the largest N value and in the regions with minimum bandwidth, it can only transfer 7 (QQVGA) and 2 (QVGA) frames per second for the largest N value. So the throughput of our designs for message-to-ciphertext conversion and ciphertext-to-message conversion should match with these frame rates.

The baseline system is capable of encrypting up to 2 QQVGA FPS for N values smaller than 2048 (refer Figure 3·9 (b)). However, as we increase N to 4096 or larger values, it cannot encrypt even a single frame per second. On the other hand, for QQVGA, RACE encrypts $\sim$16 FPS for small values of N and 5 FPS for the largest N value (16384). So at large values of N we cannot saturate the 5G network at both the maximum bandwidth and minimum bandwidth.

For QVGA resolution, the baseline system cannot encrypt even one FPS even for the smallest N value (1024). However, RACE can encrypt 5 and 2 FPS for the smallest and

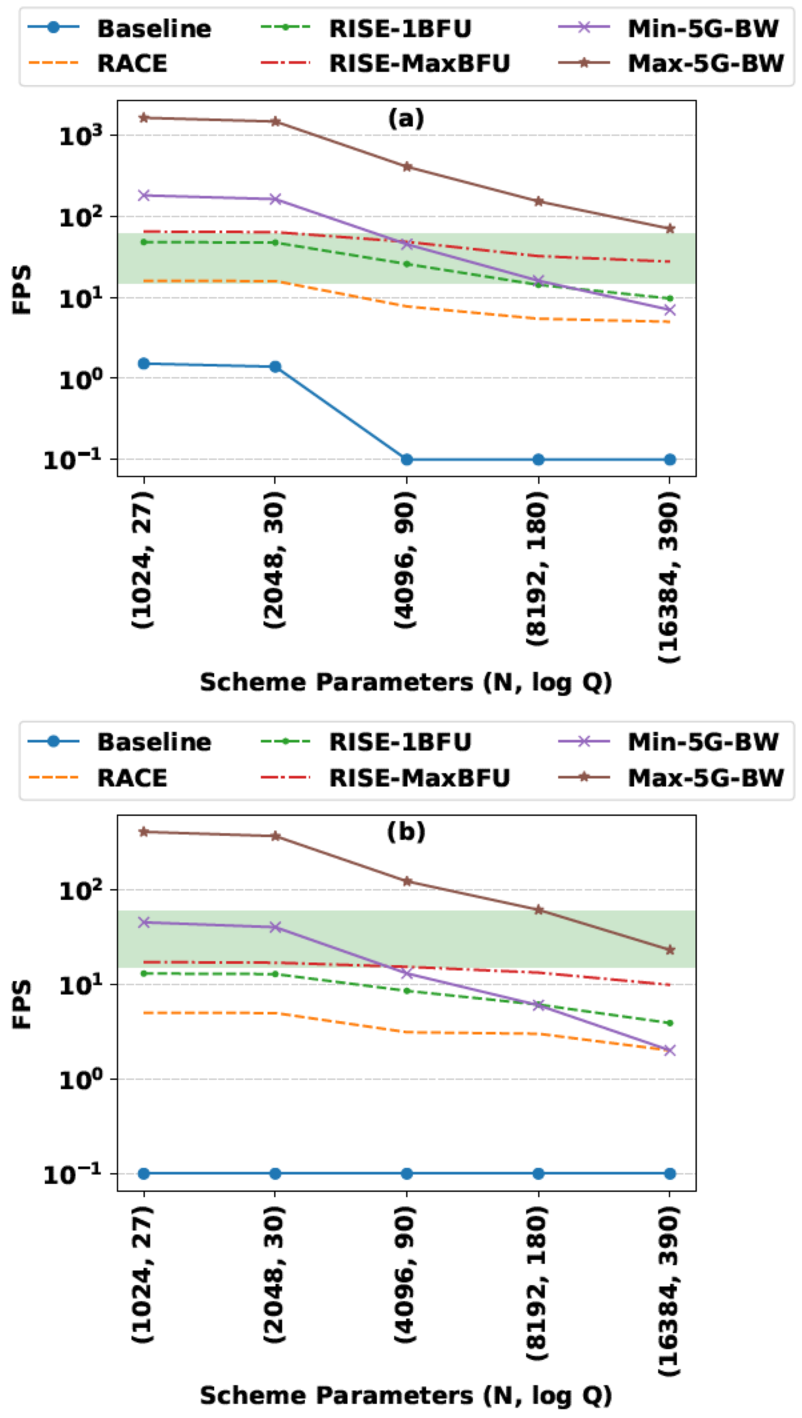


Figure 3·9: Maximum supported (a) QQVGA and (b) QVGA frame rate per second for mid-band 5G, baseline, RACE, RISE-1BFU, and RISE-MaxBFU for different N and $\log Q$ values. The green region indicates the typical frame per second required for surveillance cameras and mobile platforms.

largest N values, respectively. While RACE can support higher FPS than the baseline, it cannot saturate the 5G network at both the maximum bandwidth and minimum bandwidth for QVGA.

The RISE-1BFU system is capable of encrypting up to 48 QQVGA FPS for N values smaller than 2048 (refer Figure 3·9(b)). For the largest N value, RISE-1BFU system is capable of encrypting up to 10 QQVGA FPS. As we increase the number of BFUs from 1 to 32, the FPS numbers change to 64 and 27 QQVGA FPS for the smallest and largest N values, respectively. Thus, we can saturate the 5G network at minimum bandwidth but not at the maximum bandwidth.

For QVGA resolution (refer Figure 3·9 (a)), RISE-1BFU system is capable of encrypting up to 13 FPS for the smallest N value (1024) and 4 FPS for the largest N value. The RISE-MaxBFU configuration can encrypt up to 17 and 10 QVGA FPS for the smallest and largest N values, respectively. Thus, we can saturate the 5G network at minimum bandwidth but not at the maximum bandwidth.

Typically surveillance cameras and mobile platforms have an average frame rate of 15 to 30 FPS (Usman et al., 2018) (shown by the green highlighted area in Figure 3·9 (a) and (b)). For QQVGA resolution, RISE-MaxBFU meets this FPS requirement for all N and $\log Q$ combinations. For QVGA, RISE-MaxBFU can barely meet the FPS requirement for smaller values of N and $\log Q$.

3.6 Summary

In this chapter, we have presented RISE, a RISC-V SoC designed for accelerating HE client-side operations. We demonstrate that, in order to measure the end-user performance experience in a realistic way, it is crucial to evaluate end-to-end application performance metrics, rather than just one kernel (such as NTT) as has been done in most prior works. To achieve an end-to-end optimization, we profile message-to-ciphertext and ciphertext-to-message conversion operations, and our analysis reveals that error sampling, encryption, and decryption operations are the bottlenecks during conversion. RISE utilizes an efficient and lightweight pseudo-random number generator core and combines it with fast sampling techniques to accelerate the error sampling operations. To speed up the encryption and decryption operations, RISE employs scalable, data-level parallelism to implement the number theoretic transform operation, the main bottleneck within the encryption and decryption operations. We thoroughly evaluate RISE and our analysis reveal that RISE significantly reduces message-to-ciphertext and ciphertext-to-message conversion latency, EDP, and ADP compared to the baseline, across a range of parameters. We also demonstrate how RISE's end-to-end performance optimization improves application performance metrics.

Chapter 4

Summary and Future Work

In this thesis, we emphasize the need for a holistic analysis of accelerator-based systems and the consideration of all accelerator taxes to make informed decisions about when and where to use accelerators. We first highlight this through AI and HE applications. Then, we present and evaluate BlackParrot, an agile open-source RISC-V multicore for accelerator SoCs, and RISE, a RISC-V SoC that leverages BlackParrot to accelerate end-to-end HE client-side operations. In this chapter, we summarize the contributions of this thesis and outline the directions for future work.

4.1 Summary of Major Contributions

4.1.1 AI and HE Tax

In this work, we use ML and HE applications as case studies to demonstrate that end-to-end evaluation of hardware accelerator-based systems is essential to ensure that the systems are optimized for their intended applications. This involves analyzing the system from input to output to ensure that all components are working optimally and the system as a whole is delivering the desired performance, resulting in an improved end-user experience.

Our evaluation of a DBMS ML application shows that the benefits of offloading ML inference to accelerators depend on the backend hardware, model complexity, and data size, as well as the level of integration between the ML inference pipeline and the DBMS. Generally speaking, for models with lower complexity and smaller data sizes, it's preferable to use the CPU rather than a dedicated accelerator for inference. This is because the overhead of offloading (initializing the hardware accelerator and transferring data) is the dominant time component in the overall inference time. In contrast, for models with higher complexity and larger data sizes, using an accelerator for ML inference is more efficient because the inference time itself is the dominant factor in the overall execution time.

Additionally, the data retrieval and pre-processing time account for the majority of the execution time for small models and datasets, and up to 25% for large models and datasets. For instance, when running inference on a random forest model with 128 trees and 1 million input data from the HIGGS dataset using a hardware accelerator, we achieve a $69.7\times$ performance improvement. However, due to data retrieval and pre-processing bottlenecks, this only translates into a $2.6\times$ end-to-end performance improvement. Therefore, application developers and system designers should prioritize a balanced approach to the entire system and end-to-end application pipeline rather than focusing solely on optimizing ML inference in isolation. For instance, in the context of DBMS ML applications, tighter integration of the ML inference functionality within the DBMS could reduce data retrieval overheads and improve end-user performance.

For HE applications, the client-side only needs to perform message-to-ciphertext conversion operation, which involves encoding, error sampling, and encryption, and ciphertext-to-message conversion operation, which involves decryption and decoding. Our evaluation results indicate that the encryption and decryption latency is dominated by NTT (67.98%-72.45%) and iNTT (72.15%-83.29%) operations across a range of scheme parameters. As a result, significant efforts have been made to design hardware accelerators for NTT/iNTT operations.

When evaluating a video application, we found that accelerating NTT in the hardware greatly improves the NTT throughput (up to $381\times$) for different scheme parameters. However, there is only a slight improvement in message-to-ciphertext (up to $4\times$) and FPS (up to 3 frames) throughput metrics for smaller scheme parameters ($N <= 2^{11}$), and no improvement for larger parameters ($N >= 2^{12}$). As we increase the parallelism level in NTT computation, we observe a linear increase in NTT throughput but no improvement in the message-to-ciphertext and FPS throughput. This is because the rest of the operations in message-to-ciphertext become the new bottleneck, and further NTT acceleration does not enhance the end-to-end performance. In the case of HE client-side operations, error sampling accounts for up to 10% of the total message-to-ciphertext conversion latency. Additionally, non-NTT operations in the encryption process account for 20.30% to 21.2% of the total latency, while non-iNTT operations in the decryption process account for 15.99% to 16.05% of the total latency. Therefore, accelerating error sampling and non-(i)NTT operations in the encryption and decryption pipeline would greatly improve the message-to-ciphertext and ciphertext-to-message conversion latency, resulting in improved application throughput metrics (such as FPS). Therefore, to realistically optimize end-user performance, it is crucial to evaluate end-to-end application performance metrics instead of the performance of just one kernel, such as NTT.

4.1.2 BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs

We are addressing the lack of a scalable evaluation/simulation infrastructure for building and evaluating accelerator-based systems by introducing BlackParrot. BlackParrot is designed to be a scalable, heterogeneously tiled multicore micro-architecture for accelerator SoCs, enabling a highly flexible and scalable design. BlackParrot's tiles can be categorized into four groups: core, L1 extension, streaming accelerator, and coherent accelerator tiles. These tiles are arranged in a regular 2D mesh, and communication between neighboring tiles is facilitated by different NoCs. BlackParrot has three different NoCs: BedRock, DRAM, and I/O networks.

BlackParrot offers a robust and scalable end-to-end framework for integrating accelerators. This framework simplifies the integration of both loosely-coupled coherent (with data caching and hardware coherency) and streaming accelerators, enabling the offloading of jobs from the user application. The framework provides hardware implementation in SystemVerilog and the capability of running programs in both bare-metal and Linux environments (software libraries, kernel drivers, and firmware extensions). It assists accelerator designers and system architects in evaluating their accelerator-related ideas using hardware implementation, rather than simulation, to identify the integration strategy that has low offload and synchronization overheads for their application. This optimization can improve the end-to-end application time.

To validate this framework, we integrate the Microsoft Zipline compression accelerator with the BlackParrot Core in the ParrotLine case study. We observed a maximum of 22× end-to-end speedup for various input file sizes compared to the software execution. However, our end-to-end performance analysis indicates that the speedup decreases for larger file sizes. This decrease in speedup is attributed to the offloading overhead becoming a significant component in the overall latency. To address this, overlapping data transfer with computations can effectively reduce the overhead.

4.1.3 RISE: RISC-V SoC for HE Edge Side Operations Acceleration

To address the performance gap caused by solely focusing on NTT acceleration, we present and evaluate RISE, an area- and energy-efficient RISC-V SoC that leverages BlackParrot to accelerate message-to-ciphertext and ciphertext-to-message conversion operations. RISE speeds up the error sampling process by combining a lightweight pseudo-random number generator with fast sampling techniques. It also improves end-to-end performance by overlapping error sampling with the encryption process. To accelerate encryption and decryption operations, RISE employs scalable data-level parallelism for the NTT operation and implements the remaining operations in hardware. Furthermore, RISE optimizes area usage by implementing a unified encryption/decryption datapath and utilizing techniques such as memory reuse and data reordering to minimize on-chip memory usage.

We conduct a thorough evaluation of RISE using a complete RTL design and analyze its performance, area, and energy efficiency metrics by executing end-to-end message-to-ciphertext and ciphertext-to-message conversion operations for different scheme parameters. Our analysis reveals that RISE significantly reduces message-to-ciphertext and ciphertext-to-message conversion latency, EDP, and ADP as compared to software execution, across a range of parameters. Specifically, RISE reduces message-to-ciphertext conversion latency by 28.79×-104.39× and ciphertext-to-message conversion latency by 7.95×-66.08×. It reduces EDP by 471.24×-6191.19× for message-to-ciphertext conversion and by 36×-2481.44× for ciphertext-to-message conversion. In addition, it reduces ADP by 24.06×-55.36× for message-to-ciphertext conversion, and by 6.65×-35.05× for ciphertext-to-message conversion.

For a video application case study, software execution for message-to-ciphertext conversion was insufficient to encrypt even one frame per second. Hardware acceleration of NTT, combined with software execution of the remaining operations, did not improve the frame rate for large scheme parameters. However, by optimizing and running the entire pipeline in hardware, RISE was able to encrypt up to 17 QVGA frames and 64 QQVGA frames per second for the smallest ($N = 2^{10}$) scheme parameters, and up to 10 QVGA frames and 27 QQVGA frames for the largest ($N = 2^{14}$) scheme parameters, which is generally sufficient for most client devices.

4.2 Future Research Directions

In this subsection, we outline the potential directions for future work.

4.2.1 Runtime Task Scheduler in Accelerator-based Systems

As we discussed in 1.1.1, the optimal hardware backend for running an inference task depends on several factors including the model complexity, the data size for inference, and the overheads associated with data movement and pipeline stage invocation. Figure 4·1

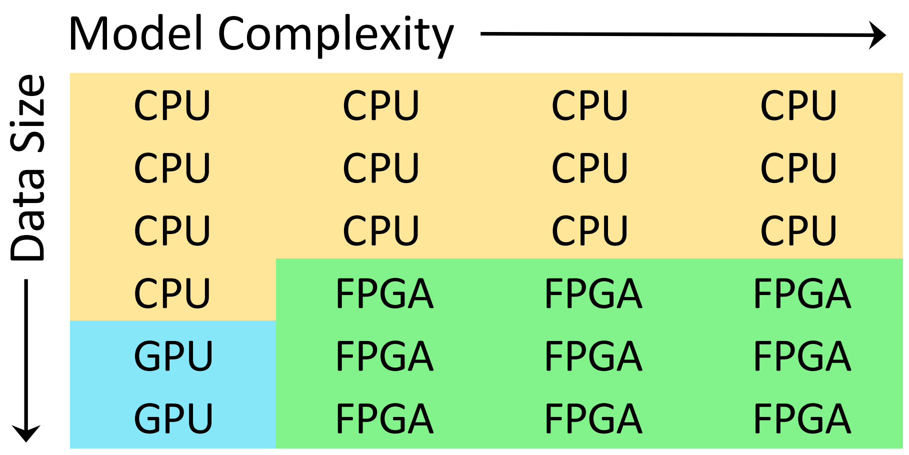


Figure 4·1: The best-performing hardware for ML random forest inference depends on the model complexity and data size.

illustrates the optimal hardware backend for ML inference based on the model complexity and the number of inputs for a random forest model. For example, in cases where the number of inputs is small and the model is less compute-intensive, it is best to keep the ML inference on the CPU to avoid high accelerator offloading costs. A wrong decision to offload to an accelerator in this case can increase the latency by 10×. However, as the model complexity or the size of the inference dataset increases, offloading ML inference to an accelerator becomes the optimal choice. A wrong decision to not offload to an accelerator in this case can result in 70× lower throughput. These results highlight the need for incorporating performance monitoring mechanisms into accelerator-based systems that can measure the actual accelerator performance during runtime.

There are several analytical models that aim to provide accurate predictions of accelerator performance and power consumption (Altaf and Wood, 2017), (Hill and Janapa Reddi, 2019), (Sriraman and Dhanotia, 2020)). These models are based on the observation that the execution time and energy consumption of an accelerator are primarily determined by the communication and memory access overheads involved in transferring data between the accelerator and the main memory. These models provide a valuable tool for designers and developers of hardware accelerators and allow them to quickly and accurately evaluate the performance and power consumption of their designs without the need for complex simulation or profiling tools. However, these high-level models are not well-suited for making offloading decisions at runtime as they do not consider the specific characteristics of the executed application, system state, or other factors that may affect the accelerator's performance.

To make offloading decisions at runtime, advanced techniques are necessary that can dynamically analyze the application, the system, and the available resources to determine the optimal offloading strategy. One potential future direction is to design runtime schedulers by extending the existing analytical models or developing an ML-based model in accelerator-based systems. This approach will involve considering different knobs/tunable parameters at different levels, such as application-level parameters, platform/hardware level parameters, dataset size, and model complexity parameters, to decide on accelerator offloading based on end-to-end application performance.

4.2.2 In-Pipeline Hardware Accelerators

As we discussed in section 1.1.1, using accelerators is not always the best solution due to the overhead associated with it. In such scenarios, runtime schedulers can schedule operations on the CPU. However, to achieve better end-to-end performance and energy efficiency, the concept of in-pipeline acceleration can be used. In-pipeline hardware accelerators are specialized hardware units that are integrated into the pipeline of a processor to accelerate specific operations. These accelerators are designed to perform specific tasks much faster than a general-purpose CPU, which can lead to significant performance improvements in specific applications. This approach reduces the overhead associated with moving data between the processor and accelerator and can result in improved end-to-end performance and energy efficiency.

The design of an in-pipeline accelerator requires an ISA extension that can accelerate specific computations required by the target application. Developing a custom ISA extension requires a thorough understanding of the application's algorithms and computations. The following steps are involved in designing a custom ISA extension:

- Design an instruction set extension that accelerates the specific computations required by the application.

- Develop a compiler backend that can generate machine code for the new instructions.

- Modify the processor's microarchitecture to support the new instructions, which may require changes to the pipeline, register file, and other components of the processor.

Numerous works have focused on accelerating ML in the processor pipeline by extending open-source RISC-V ISA with a domain-specific set of instructions (Azad et al., 2020) (Louis et al., 2019) (Garofalo et al., 2020a) (Assir et al., 2021) (Garofalo et al., 2020b). However, research in HE is still ongoing, and there is a lack of research in ISA extension and in-pipeline accelerators for HE. A potential future direction is to use open-source ISAs, such as RISC-V, and extend them with a set of custom instructions to address the bottlenecks of HE client-side and server-side operations in the CPU pipeline to improve performance and energy efficiency.

Based on the profiling results we presented in section 3, en/decryption and error sampling operations are the main performance bottleneck in client-side HE operations, i.e., message-to-ciphertext and ciphertext-to-message conversion. To improve the end-to-end performance, this thesis proposes RISE as a loosely-coupled accelerator that is implemented outside the processor core. However, it would be interesting to investigate the possibilities of mixed hardware/software co-design through ISA extension and tighter integration of the accelerator with the processor core to determine the most energy-efficient solution. Additionally, server-side operations such as dot product computations and polynomial evaluations can also benefit from a custom ISA extension.

4.2.3 Coherent Accelerators Using Compute Express Link (CXL)

One of the primary bottlenecks in accelerator-based systems is the data transfer process, which can limit the overall performance of the system. To address this issue, a new open standard interface called CXL (Sharma, 2022) has been designed to enable high-speed communication between CPUs and accelerators. CXL uses a cache-coherent protocol that allows accelerators to directly access memory without requiring the CPU to act as an intermediary. This significantly reduces communication overhead and can improve the end-to-end performance in accelerator-based systems. Potential future directions for CXL-based accelerator systems are as follows.

- **CXL Performance Optimization:** Investigate new methods for optimizing the performance of CXL-based accelerator systems, taking into account factors such as data transfer times, accelerator utilization, and memory bandwidth. This could involve developing new algorithms, scheduling techniques, and resource management strategies that are optimized for CXL. To evaluate the effectiveness of CXL in reducing data transfer and accelerator communication overhead, several factors need to be considered such as application characteristics (the data size, data access pattern, and the amount of computation performed by the accelerator) and workload distribution (a workload that requires frequent data transfers between the CPU and accelerator, such as iterative algorithms, may benefit more from CXL).

- **CXL Accelerator Design:** CXL provides a shared memory space between the CPU and the accelerator, which can lead to contention for shared resources such as memory bandwidth and cache capacity. CXL also enables the use of advanced features such as cache coherence, which further increases the complexity of designing accelerator micro-architecture. Maximizing the performance of CXL-based accelerators requires careful design and management of cache coherence protocols to minimize the number of cache misses, optimize data locality, and minimize the impact of memory access latency by utilizing pipelining and parallelism in accelerator micro-architecture. Overall, the use of CXL requires careful consideration of accelerator micro-architecture to ensure that the benefits of the technology are fully realized and that the accelerator can operate efficiently in a shared memory environment.

- **CXL for Edge Computing:** Investigate the use of CXL for edge computing, exploring the design of low-power, low-cost CXL-based accelerators for use in IoT devices, autonomous robots, and other embedded systems. While CXL was originally designed for use in data centers, it is now being explored for its potential applications in edge computing. Edge computing involves processing data closer to the source of its generation, which reduces latency and increases the speed of data analysis. By using CXL technology in edge computing systems, it may be possible to further enhance the performance and efficiency of these systems.

4.3 Final Remarks

In summary, we strongly believe that to make informed decisions about the usage of accelerators, it is necessary to evaluate the accelerator-based system as a whole and consider all the accelerator taxes involved. In this regard, in the first part of our research, we emphasize the need for a holistic, end-to-end analysis of the workloads using AI and HE applications. In the second part of our research, we propose a robust and scalable software-hardware framework for accelerator evaluation using BlackParrot, an open-source RISC-V based SoC design. Our framework can be used by accelerator designers and system architects to perform an end-to-end performance analysis of accelerators by carefully accounting for the interaction of the accelerator with the rest of the system. In the third part of our research, we present RISE, a full RISC-V SoC, consisting of BlackParrot core and an efficient custom-designed accelerator tailored for accelerating end-to-end HE client-side operations.

References

Agrawal, R., Bu, L., and Kinsy, M. A. (2020). A Post-Quantum Secure Discrete Gaussian Noise Sampler. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 295–304. IEEE.

Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., and Lauter, K. (2021). Homomorphic Encryption Standard. In *Protecting Privacy through Homomorphic Encryption*, pages 31–62. Springer.

Alkim, E., Ducas, L., Pöppelmann, T., and Schwabe, P. (2016). Post-Quantum Key Exchange—A New Hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343.

Altaf, M. S. B. and Wood, D. A. (2017). Logca: A high-level performance model for hardware accelerators. *ACM SIGARCH Computer Architecture News*, 45(2):375–388.

Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., et al. (2020). Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21.

Apple Inc. (2020). Apple unleashes M1. `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`.

ARM (2017). LittleFS. `https://github.com/ARMmbed/littlefs`.

Asanović, Krste. (2016). The Rocket Chip Generator. `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

Askarihemmat, M., Wagner, S., Bilaniuk, O., Hariri, Y., Savaria, Y., and David, J.-P. (2023). BARVINN: Arbitrary precision DNN accelerator controlled by a RISC-V CPU. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pages 483–489.

Assir, I. A., Iskandarani, M. E., Sandid, H. R. A., and Saghir, M. A. (2021). Arrow: A risc-v vector accelerator for machine learning inference. *arXiv preprint arXiv:2107.07169*.

Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehlé, D. (2021). CRYSTALS-Kyber (version 3.02). `https://pq-crystals.org/kyber/resources.shtml`.

Azad, Z., Delshadtehrani, L., Zhou, B., Joshi, A., Gilani, F., Lim, K., Petrisko, D., Jung, T., Wyse, M., Guarino, T., et al. (March 2019). The BlackParrot Processor: An Open-Source Industrial-Strength RV64G Multicore Processor. Technical report.

Azad, Z., Louis, M. S., Delshadtehrani, L., Ducimo, A., Gupta, S., Warden, P., Reddi, V. J., and Joshi, A. (2020). An end-to-end risc-v solution for ml on the edge using in-pipeline support. In *Boston area Architecture (BARC) Workshop*.

Azad, Z., Sen, R., Park, K., and Joshi, A. (2021). Hardware Acceleration for DBMS Machine Learning Scoring: Is It Worth the Overheads? In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 243–253. IEEE.

Azad, Z., Yang, G., Agrawal, R., Petrisko, D., Taylor, M., and Joshi, A. (2022). RACE: RISC-V SoC for En/decryption Acceleration on the Edge for Homomorphic Computation. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 1–6.

Badawi, A. A., Veeravalli, B., Lin, J., Xiao, N., Kazuaki, M., and Mi, A. K. M. (2021). Multi-GPU Design and Performance Evaluation of Homomorphic Encryption on GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32:379–391.

Bai, J., Lu, F., Zhang, K., et al. (2019). ONNX: Open Neural Network Exchange. `https://github.com/onnx/onnx`.

Baldi, P., Sadowski, P., and Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9.

Balkind, J., Chang, T.-J., Jackson, P. J., Tziantzioulis, G., Li, A., Gao, F., Lavrov, A., Chirkov, G., Tu, J., Shahrad, M., et al. (2020). OpenPiton at 5: A Nexus for Open and Agile Hardware Design. *IEEE Micro*, 40(4):22–31.

Banerjee, U., Ukyab, T. S., and Chandrakasan, A. P. (2019). Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *arXiv preprint arXiv:1910.07557*.

Barrett, P. (1986). Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer.

Bernstein, D. J. (2008). ChaCha, a Variant of Salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Lausanne, Switzerland.

Bootland, C., Castryck, W., Iliashenko, I., and Vercauteren, F. (2020). Efficiently Processing Complex-Valued Data in Homomorphic Encryption. *Journal of Mathematical Cryptology*, 14:55–65.

Bouraoui, A., Jamoussi, S., and Hamadou, A. B. (2022). A comprehensive review of deep learning for natural language processing. *International Journal of Data Mining, Modelling and Management*, 14(2):149–182.

Brakerski, Z. (2012). Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Annual Cryptology Conference*, pages 868–886. Springer.

Buch, M., Azad, Z., Joshi, A., and Reddi, V. J. (2021). AI Tax in Mobile SoCs: End-to-end Performance Analysis of Machine Learning in Smartphones. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 96–106. IEEE.

Camastra, F. and Vinciarelli, A. (2015). *Machine learning for audio, image and video analysis: theory and applications*. Springer.

Chen, H., Laine, K., and Player, R. (2017). Simple encrypted arithmetic library-SEAL v2. 1. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*, pages 3–18. Springer.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014a). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284.

Chen, X., Yang, B., Yin, S., Wei, S., and Liu, L. (2022). CFNTT: Scalable Radix-2/4 NTT Multiplication Architecture with an Efficient Conflict-free Memory Mapping Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 94–126.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., et al. (2014b). Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE.

Chen, Z., Wang, J., He, H., and Huang, X. (2014c). A fast deep learning system using GPU. In *2014 IEEE international symposium on circuits and systems (ISCAS)*, pages 1552–1555. IEEE.

Cheon, J. H., Kim, A., Kim, M., and Song, Y. (2017). Homomorphic Encryption for Arithmetic of Approximate Nnumbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer.

Choquette, J., Gandhi, W., Giroux, O., Stam, N., and Krashinsky, R. (2021). Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35.

Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., et al. (2018). Serving dnns in real time at datacenter scale with project brainwave. *iEEE Micro*, 38(2):8–20.

Cong, J., Fang, Z., Gill, M., and Reinman, G. (2015). Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 380–387. IEEE.

Cousins, D. B., Rohloff, K., and Sumorok, D. (2017). Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor. *IEEE Transactions on Emerging Topics in Computing*, 5:193–206.

Deng, L. and Li, X. (2013). Machine learning paradigms for speech recognition: An overview. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(5):1060–1089.

Dua, Dheeru and Graff, Casey (2017). UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml`.

Duong-Ngoc, P., Kwon, S., Yoo, D., and Lee, H. (2023). Area-Efficient Number Theoretic Transform Architecture for Homomorphic Encryption. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(3):1270–1283.

Duong-Ngoc, P., Tan, T. N., and Lee, H. (2021). Configurable Butterfly Unit Architecture for NTT/INTT in Homomorphic Encryption. In *2021 18th International SoC Design Conference (ISOCC)*, pages 345–346. IEEE.

Esperanto Technologies (2019). Dromajo - Esperanto Technology's RISC-V Reference Model. `https://github.com/chipsalliance/dromajo`.

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., et al. (2018). A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE.

Garofalo, A., Rusci, M., Conti, F., Rossi, D., and Benini, L. (2020a). Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. *Philosophical Transactions of the Royal Society A*, 378(2164):20190155.

Garofalo, A., Tagliavini, G., Conti, F., Rossi, D., and Benini, L. (2020b). Xpulpnn: Accelerating quantized neural networks on risc-v processors through isa extensions. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 186–191. IEEE.

Gentry, C., Halevi, S., Peikert, C., and Smart, N. P. (2012). Ring Switching in BGV-Style Homomorphic Encryption. In *International Conference on Security and Cryptography for Networks*, pages 19–37. Springer.

Giri, D., Mantovani, P., and Carloni, L. P. (2018). Accelerators and coherence: An SoC perspective. *IEEE Micro*, 38(6):36–45.

Guillen, O. M., Pöppelmann, T., Mera, J. M. B., Bongenaar, E. F., Sigl, G., and Sepulveda, J. (2017). Towards post-quantum security for IoT endpoints with NTRU. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 698–703. IEEE.

Gupta, N., Jati, A., Chauhan, A. K., and Chattopadhyay, A. (2020). PQC Acceleration Using GPUs: Frodokem, Newhope, and Kyber. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):575–586.

Halevi, S. and Shoup, V. (2020). Design and implementation of HElib: a homomorphic encryption library. *Cryptology ePrint Archive*.

Heron, S. (2009). Advanced encryption standard (AES). *Network Security*, 2009(12):8–12.

Hill, M. and Janapa Reddi, V. (2019). Gables: A roofline model for mobile socs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 317–330.

Huynh, L. N., Lee, Y., and Balan, R. K. (2017). Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95.

Johnston, Jeff and Fitzsimmons, T (2011). Newlib. `http://sourceware.org/newlib`.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12.

Jung, W., Lee, E., Kim, S., Kim, J., Kim, N., Lee, K., Min, C., Cheon, J. H., and Ahn, J. H. (2021). Accelerating Fully Homomorphic Encryption through Architecture-Centric Analysis and Optimization. *IEEE Access*, 9:98772–98789.

Keccak Team (2018). SHA3 (Keccak). `https://keccak.team/hardware.html`.

Kuon, I. and Rose, J. (2010). *Quantifying and exploring the gap between FPGAs and ASICs*. Springer Science & Business Media.

Kurt Rohloff (2018). The PALISADE Lattice Cryptography Library. `https://palisade-crypto.org/software-library`.

Kurth, A., Vogel, P., Capotondi, A., Marongiu, A., and Benini, L. (2017). HERO: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA. *arXiv preprint arXiv:1712.06497*.

Lecun, Y., Kavukcuoglu, K., Culurciello, E., et al. (2011). Large-scale FPGA-based convolutional networks. *Machine Learning on Very Large Data Sets*, pages 1–26.

Li, C. and Liu, L. (2021). A High Speed NTT Accelerator for Lattice-based Cryptography. In *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)*, pages 85–89. IEEE.

Li, H., Fan, X., Jiao, L., Cao, W., Zhou, X., and Wang, L. (2016). A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE.

Li, Y., Louri, A., and Karanth, A. (2022). SPACX: Silicon photonics-based scalable chiplet accelerator for DNN inference. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 831–845. IEEE.

Louis, M. S., Azad, Z., Delshadtehrani, L., Gupta, S., Warden, P., Reddi, V. J., and Joshi, A. (2019). Towards deep learning using tensorflow lite on risc-v. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, volume 1, page 6.

Microsoft Corp. (2021). Project Zipline. `https://github.com/opencomputeproject/Project-Zipline`.

Microsoft Corp. (2023). What is SQL Server Machine Learning Services with Python and R? `https://learn.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-ver16`.

Microsoft Research (2018). Microsoft SEAL (release 3.5). `https://github.com/Microsoft/SEAL`.

Morris Dworkin (2015). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`.

Nakandalam, S., Saur, K., Yu, G.-I., Karanasos, K., Curino, C., Weimer, M., and Interlandi, M. (2020). Taming model serving complexity, performance and cost: A compilation to tensor computations approach.

Nannipieri, P., Di Matteo, S., Zulberti, L., Albicocchi, F., Saponara, S., and Fanucci, L. (2021). A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms. *IEEE Access*, 9:150798–150808.

Natarajan, D. and Dai, W. (2021). SEAL-Embedded: A Homomorphic Encryption Library for the Internet of Things. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 756–779.

Nawrocka, A., Kot, A., and Nawrocki, M. (2018). Application of machine learning in recommendation systems. In *2018 19th International Carpathian Control Conference (ICCC)*, pages 328–331. IEEE.

Nickolls, J. and Dally, W. J. (2010). The GPU computing era. *IEEE micro*, 30(2):56–69.

NVIDIA (2018). NVIDIA RAPIDS. `https://developer.nvidia.com/rapids`.

Nvidia Corp. (2023). NVIDIA AGX Systems. `https://www.nvidia.com/en-us/deep-learning-ai/products/agx-systems/`.

Omar, S., Ngadi, A., and Jebur, H. H. (2013). Machine learning techniques for anomaly detection: an overview. *International Journal of Computer Applications*, 79(2).

OpenSBI Community (2021). OpenSBI. `https://github.com/riscv/opensbi`.

Paludo, R. and Sousa, L. (2022). NTT Architecture for a Linux-Ready RISC-V Fully-Homomorphic Encryption Accelerator. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2669–2682.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Peikert, C. et al. (2016). A decade of lattice cryptography. *Foundations and Trends® in Theoretical Computer Science*, 10(4):283–424.

Petazzoni, T. and Electrons, F. (2012). Buildroot: a nice, simple and efficient embedded Linux build system. In *Embedded Linux System Conference*, volume 2012.

Petrisko, D., Gilani, F., Wyse, M., Jung, D. C., Davidson, S., Gao, P., Zhao, C., Azad, Z., Canakci, S., and Veluri, B. (2020). BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro*, 40(4):93–102.

Power, J., Hestness, J., Orr, M. S., Hill, M. D., and Wood, D. A. (2014). gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36.

Psallidas, F., Zhu, Y., Karlas, B., Interlandi, M., Floratou, A., Karanasos, K., Wu, W., Zhang, C., Krishnan, S., Curino, C., and Weimer, M. (2019). Data Science through the looking glass and what we found there. *CoRR*.

Qualcomm Technologies, Inc. (2020). Snapdragon 888 5G Mobile Platform. `https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-888-5g-mobile-platform`.

Reis, D., Takeshita, J., Jung, T., Niemier, M., and Hu, X. S. (2020). Computing-in-Memory for Performance and Energy-Efficient Homomorphic Encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.

Riazi, M. S., Laine, K., Pelton, B., and Dai, W. (2020). 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*, pages 1295–1309.

Ribeiro, L., da Silva Lima, J. P., de Queiroz, R. J., Bezerra, A., Chagas, J. P., da Silva, F. Q., Santos, A. L., Roberto, J., and Junior, R. (2021). Saber Post-Quantum Key Encapsulation Mechanism (KEM): Evaluating Performance in Mobile Devices and Suggesting Some Improvements. In *Third PQC Standardization Conference*, pages 07–09.

Roy, S. S., Vercauteren, F., Mentens, N., Chen, D. D., and Verbauwhede, I. (2014). Compact ring-LWE cryptoprocessor. In *Cryptographic Hardware and Embedded Systems–CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings 16*, pages 371–391. Springer.

Samajdar, A., Zhu, Y., Whatmough, P., Mattina, M., and Krishna, T. (2018). Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883*.

Sankaradas, M., Jakkula, V., Cadambi, S., Chakradhar, S., Durdanovic, I., Cosatto, E., and Graf, H. P. (2009). A massively parallel coprocessor for convolutional neural networks. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 53–60. IEEE.

Shao, Y. S., Xi, S. L., Srinivasan, V., Wei, G.-Y., and Brooks, D. (2016). Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE.

Sharma, D. D. (2022). Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 5–12. IEEE.

Sriraman, A. and Dhanotia, A. (2020). Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 733–750, New York, NY, USA. Association for Computing Machinery.

Su, Y., Yang, B., Yang, C., and Tian, L. (2020). FPGA-based Hardware Accelerator for Leveled Ring-LWE Fully Homomorphic Encryption. *IEEE Access*, 8:168008–168025.

Su, Y., Yang, B.-L., Yang, C., Yang, Z.-P., and Liu, Y.-W. (2022). A Highly Unified Reconfigurable Multicore Architecture to Speedup NTT/INTT for Homomorphic Polynomial Multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.

Tan, L. and Jiang, J. (2019). Hardware and software for digital signal processors. *Digital Signal Processing*, pages 727–784.

Thummaluru, S. R., Ameen, M., and Chaudhary, R. K. (2019). Four-port MIMO Cognitive Radio System for Midband 5G Applications. *IEEE Transactions on Antennas and Propagation*, 67(8):5634–5645.

Turan, F., Roy, S. S., and Verbauwhede, I. (2020). HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA. *IEEE Transactions on Computers*, 69:1185–1196.

Usman, M. A., Usman, M. R., and Shin, S. Y. (2018). An Intrusion Oriented Heuristic for Efficient Resource Management in End-to-End Wireless Video Surveillance Systems. In *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE.

Wang, W., Huang, X., Emmart, N., and Weems, C. (2014). VLSI Design of a Large-Number Multiplier for Fully Homomorphic Encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22:1879–1887.

Waterman, A., Lee, Y., Patterson, D., Asanovic, K., level Isa, V. I. U., Waterman, A., Lee, Y., and Patterson, D. (2014). The RISC-V instruction set manual. *Volume I: User-Level ISA', version*, 2.

Xi, S., Yao, Y., Bhardwaj, K., Whatmough, P., Wei, G.-Y., and Brooks, D. (2020). SMAUG: End-to-end full-stack simulation infrastructure for deep learning workloads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–26.

Xin, G., Han, J., Yin, T., Zhou, Y., Yang, J., Cheng, X., and Zeng, X. (2020). VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture. *IEEE transactions on circuits and systems I: regular papers*, 67(8):2672–2684.

Ye, Z., Cheung, R. C., and Huang, K. (2022). PipeNTT: A Pipelined Number Theoretic Transform Architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(10):4068–4072.

Yoon, I., Cao, N., Amaravati, A., and Raychowdhury, A. (2019). A 55nm 50nJ/encode 13nJ/decode Homomorphic Encryption Crypto-Engine for IoT Nodes to Enable Secure Computation on Encrypted Data. In *2019 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4. IEEE.

CURRICULUM VITAE

[redacted]

[redacted]

[redacted]

[redacted]