OnionChopper: A Modular Arithmetic Hardware Accelerator for Private Information Retrieval

by

Georgia Shay

S.B. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Georgia Shay
May 12, 2023

Certified by: Mengjia Yan
Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

OnionChopper: A Modular Arithmetic Hardware Accelerator for Private Information Retrieval

by

Georgia Shay

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Private information retrieval (PIR) is a protocol which allows a user to retrieve data from a database on a server without the server being able to deduce which records were retrieved. Due to the homomorphic cryptography systems required to make these protocols work and large amount of data processing required per user query, these algorithms tend to run much slower than needed for real-time applications such as streaming movies or voice calling. To improve these speeds to ones more tolerable for user applications, we designed OnionChopper: a small, fast, and energy efficient hardware accelerator on which to offload the heaviest computational work.

This hardware accelerator is optimized for a state-of-the-art PIR algorithm, OnionPIR, but is widely applicable due to similarities in the fundamental algorithms and cryptographies used in private information retrieval. We identified the major bottleneck operation common to OnionPIR and other PIR schemes and designed computation units to aid with that operation. We designed a near-storage accelerator with on-chip parallel computation units, SRAMs and register files for exploiting data reuse, and a near-storage connection to the SSD to exploit its high internal bandwidth to access the database. We used a space exploration tool to identify the optimal architecture and scheme of computation and data movement over that architecture. Our resulting design offers a nearly 300× speed improvement over running on a general-purpose processor for a 64GB database.

Thesis Supervisor: Mengjia Yan
Title: Assistant Professor

Acknowledgments

I would like to thank my research supervisor, Mengjia Yan, for providing constant support for this project even as it changed directions. Your advice and encouragement has been well appreciated.

I would also like to thank my research mentor Tianhao Huang. Your valuable insights on near-storage accelerators and SSDs made this project possible.

I would like to thank my dad for letting me borrow his kubernetes cluster for computational power to run experiments.

Finally, I would like to thank my family for their support and encouragement throughout this project.

Contents

1 Introduction **15**

1.1 Motivation 15

1.2 Contribution 16

1.3 Thesis Organization 17

2 Background and Related Work **19**

2.1 Private Information Retrieval 19

2.1.1 Basic Scheme 20

2.1.2 Homomorphic Encryption 22

2.1.3 OnionPIR 26

2.1.4 Other PIR Schemes 28

2.2 Timeloop 32

2.2.1 Problem 32

2.2.2 Architecture 33

2.2.3 Mapping 34

2.2.4 Mapper 35

2.2.5 Design Space Tool 37

3 Application Characteristics **39**

3.1 Disk Bandwidth Profiling 40

3.2 Runtime Profiling 40

3.3 Basic Operation 41

3.4 Applicability to Other PIR Schemes 45

3.4.1 SealPIR 45
3.4.2 MulPIR 45
3.4.3 SimplePIR 46
3.4.4 FastPIR 46

4 Hardware Accelerator Design **47**
4.1 Overview 47
4.2 Compute Unit 48
4.2.1 Multiplication 49
4.2.2 Addition 52
4.2.3 Refresh 53
4.2.4 Synthesis 54
4.2.5 Pipelining 54
4.3 Storage Hierarchy 56
4.4 Moduli 58

5 Space Exploration **61**
5.1 High-Level Strategy 61
5.2 Space Definition 62
5.2.1 Architecture 62
5.2.2 Mappings 62
5.3 Implementation 63
5.3.1 Memory Restriction 63
5.3.2 Constraints 63
5.3.3 Disk 64

6 Evaluation Results **65**
6.1 Initial Architecture Sweep 65
6.2 Refined Architecture Sweep 68
6.3 Mapping 70
6.4 Evaluation Statistics 74

6.5 Different Database Sizes 74
6.6 Different PIR Algorithms 75

7 Limitations and Future Work 81

8 Conclusion 85

A Example Timeloop Mapping 87

List of Figures

2-1 Basic PIR scheme 21
2-2 2D PIR scheme 22
2-3 Matrix C Tiling Comparison 36

3-1 OnionPIR Basic Operation 43
3-2 Generic Basic Operation 44

4-1 Overview of Hardware 48
4-2 Compute Unit 50
4-3 Multiply Unit 51
4-4 Addition Unit 53
4-5 Pipelined Adder 55
4-6 Pipelined Multiplier 55
4-7 Pipelined Comparator 56
4-8 Storage Hierarchy 57
4-9 Moduli Selector Unit 58

6-1 Initial Sweep Architecture Cycles 66
6-2 Refined Sweep Architecture Cycles 68
6-3 Refined Sweep Area vs. Energy 69
6-4 Refined Sweep (Fast Only) Area vs. Energy 69
6-5 Register File Depth vs Energy 71
6-6 Overall Optimal Architecture 72
6-7 Accelerator Speed vs Database Size for OnionPIR 75

6-8 OnionPIR Speed on a General-Purpose Processor vs Accelerator . . . 76
6-9 Accelerator Energy Consumption vs Database Size for OnionPIR . . 76
6-10 Accelerator Power Consumption vs Database Size for OnionPIR . . . 77
6-11 Database Size vs Speed Comparison 77
6-12 Database Size vs Energy Comparison 78
6-13 Database Size vs Power Comparison 79

List of Tables

2.1 Homomorphic Operation Costs 23

3.1 Cachegrind Results 41

3.2 Applicability of Basic Operation to Other PIR Schemes 45

Chapter 1

Introduction

With an increasingly privacy-conscious populus, demand has heightened for services that prevent inadvertent release of user data, even to the service itself. Private Information Retrieval (PIR) allows a user to retrieve a record from a database located on a server, without the server being able to detect which record was retrieved. PIR could be used to privately stream movies [14], for voice calling that keeps participants private [1], and even for privacy-preserving advertisement delivery [13].

1.1 Motivation

A popular method to achieve private information retrieval on a single, untrusted server is by using homomorphic encryption, a type of cryptography that allows for operations such as addition and multiplication on encrypted ciphertexts. The user, or client, sends an encrypted query to the server indicating which record of the database should be retrieved. The server then performs homomorphic computations and returns to the user an encryption of the desired record in the database.

Unfortunately, private information retrieval (PIR) methods tend to be quite slow. While PIR algorithms have been getting faster, they are bounded by some fundamental limitations. Homomorphic encryption, used in many PIR algorithms, is computationally expensive. However, its use is fundamental in many such algorithms due to the need to perform computation on a client's query without knowing what the query is. Many recent works improve upon previous algorithms by using faster homomorphic cryptographies or faster homomorphic operations.

An even more fundamental limitation is the requirement that any PIR algorithm computes upon every record in the database. Otherwise, it would be possible for the server to surmise that certain records which were not accessed must not be the records the client has queried for. PIR algorithms must, then, be at least linear in the size of the database, and practical databases can be quite large.

1.2 Contribution

We designed OnionChopper, a near-storage hardware accelerator to improve the performance of server-side computation in a PIR algorithm. We designed the accelerator targeting the OnionPIR algorithm, a state-of-the-art algorithm with low response sizes [21]. Hardware has the advantage of being able to process data in parallel and specialization allows hardware units to compute faster than with general-purpose programming instructions.

By placing the hardware near the storage for the database, the disk, computation can happen on data streamed directly from the database rather than waiting for the data to be brought in to memory, caches, and ultimately the computer's processor itself. Based on our analysis of the OnionPIR program, disk bandwidth is easily saturated by running this protocol, so this approach allows an even higher disk bandwidth to be achieved.

Inside an NVMe disk is some amount of DRAM that can be accessible to a hardware accelerator [5] [7]. The design of the accelerator itself also includes some SRAM buffers, registers, and compute units. We performed a space search to find out the most efficient configuration of these different components.

OnionChopper has performance benefits of nearly 300x compared to a general-purpose processor. Running the OnionPIR algorithm's bottleneck operation on this accelerator would result in speeds nearing the SSD's bandwidth. The majority of energy consumption comes from that required for the minimum number of possible

SSD accesses. OnionChopper is also similarly efficient on other PIR algorithms for large database sizes.

1.3 Thesis Organization

In Chapter 2, we discuss background on PIR schemes and the Timeloop tool. In Section 2.1, we discuss basic PIR algorithms, cryptographic operations used in PIR, and various improvements made in different PIR variants. Section 2.2 contains a discussion of Timeloop, the design exploration tool used to explore various architectures and data movements in our thesis.

In Chapter 3, we discuss the bottleneck operation of OnionPIR, as well as its similarities to the main operations in other PIR schemes.

In Chapter 4, we discuss the basic design of our hardware accelerator, including the computation units and storage hierarchy in the near-storage accelerator.

In Chapter 5, we discuss our methods of using Timeloop to explore the space of possible storage hierarchy designs.

In Chapter 6, we analyze the explored space of storage hierarchy designs to determine the optimal architecture. We also evaluate this architecture's speed, area, and energy consumption, compare it to a general purpose processor, and compare its performance across different PIR schemes.

In Chapter 7, we discuss limitations to our processes and analyses.

Chapter 8 provides concluding remarks.

Chapter 2

Background and Related Work

Here, we provide background on PIR, covering the security problem that PIR tries to address, PIR variations and algorithmic performance optimization techniques, and the basic cryptographic operations necessary in modern PIR algorithms. We then give background on Timeloop, a design space exploration tool which was built for designing Deep Neural Network (DNN) or tensor accelerators. In this thesis, we will use Timeloop to assist in the design space exploration of our near-storage PIR accelerator.

2.1 Private Information Retrieval

Private Information Retrieval (PIR) allows a user to retrieve a record from a database without the server hosting the database discovering which record was retrieved. The basic problem is as follows:

Suppose the server has a database D with n elements, $D_0 ... D_{n-1}$, and a user wants to retrieve the record at index i, D_i. The user can communicate a query q to the server to request this record, and the server can respond with some response r. The query q must not reveal anything about i.

Note that, no matter what algorithm the server uses to compute response r, it must ultimately process every element of the database D in some way. Otherwise, this must mean that the elements it did not process were not the record that the client requested, revealing something about the value of i. This means that a fundamental limitation of PIR algorithms is that they must be at least linear in the size of the database, n.

2.1.1 Basic Scheme

Suppose for the moment that we have a system of encryption with special functions that, when applied to ciphertexts, can add or multiply the underlying plaintexts. Let's call these operations $\oplus$ and $\otimes$, and denote $\mathcal{E}(p)$ as the encrypted ciphertext corresponding to plaintext p.

Then, we have that:

$$\mathcal{E}(p_1) \oplus \mathcal{E}(p_2) = \mathcal{E}(p_1 + p_2)$$

$$\mathcal{E}(p_1) \otimes \mathcal{E}(p_2) = \mathcal{E}(p_1 \cdot p_2)$$

We will also need to be able to multiply a ciphertext and a plaintext together. We will denote this by the $\star$ operation.

$$p_1 \star \mathcal{E}(p_2) = \mathcal{E}(p_1 \cdot p_2)$$

So, using the $\oplus$, $\otimes$, and $\star$ operator, a server can operate on the underlying plaintexts of ciphertexts without needing to (or indeed, being able to) decrypt them.

If we have a cryptosystem with these operations, we can use it to build a PIR protocol. Suppose the user wants to request record D_i, out of a database with n total elements. The user can encrypt, using a secret key, a one-hot vector encoding the index i (a vector of n elements where every element is 0, except for the element at index i, which is 1), and send this as their query q [21].

$$q = \mathcal{E}(\{\underset{0}{0}, \underset{1}{0}, \underset{2}{0}, ..., \underset{i}{1}, ..., \underset{n-3}{0}, \underset{n-2}{0}, \underset{n-1}{0}\})$$

Then, the server takes the dot product of the database vector D with the query vector q, using the $\star$ operation for multiplication and the $\oplus$ operation for addition.

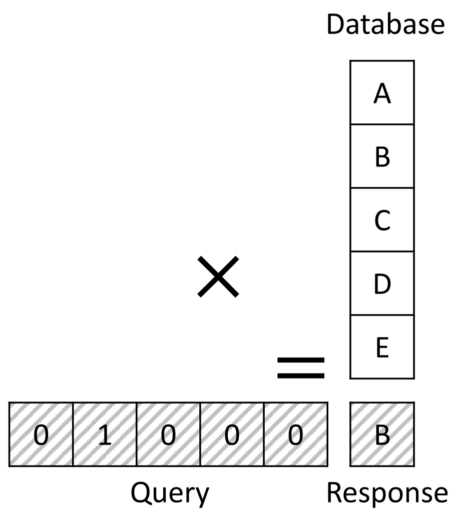

Figure 2-1: Basic PIR scheme

This results in the encryption of the element D_i, which is sent back to the client.

$$D \cdot q = \oplus \sum_{j=0}^{n-1} D_j \star q_j = \mathcal{E}(D_i)$$

The client can decrypt this to determine D_i. Since all of these operations were occurring under encryption, the server could never learn the value of i or D_i. This process is shown in Figure 2-1.

Dimensionality

To reduce the size of the query, the server can reconceptualize the database as a 2D matrix of size $\sqrt{n} \times \sqrt{n}$ [21]. To retrieve the record at row i, column j ($D_{i,j}$), the client sends two encrypted one-hot vectors, one with a 1 at index i and one with a 1 at index j.

$$q = (q_1, q_2)$$

$$q_1 = \mathcal{E}(\{\underset{0}{0}, \underset{1}{0}, \underset{2}{0}, ..., \underset{i}{1}, ..., \underset{\sqrt{n}-3}{0}, \underset{\sqrt{n}-2}{0}, \underset{\sqrt{n}-1}{0}\})$$

$$q_2 = \mathcal{E}(\{\underset{0}{0}, \underset{1}{0}, \underset{2}{0}, ..., \underset{j}{1}, ..., \underset{\sqrt{n}-3}{0}, \underset{\sqrt{n}-2}{0}, \underset{\sqrt{n}-1}{0}\})$$

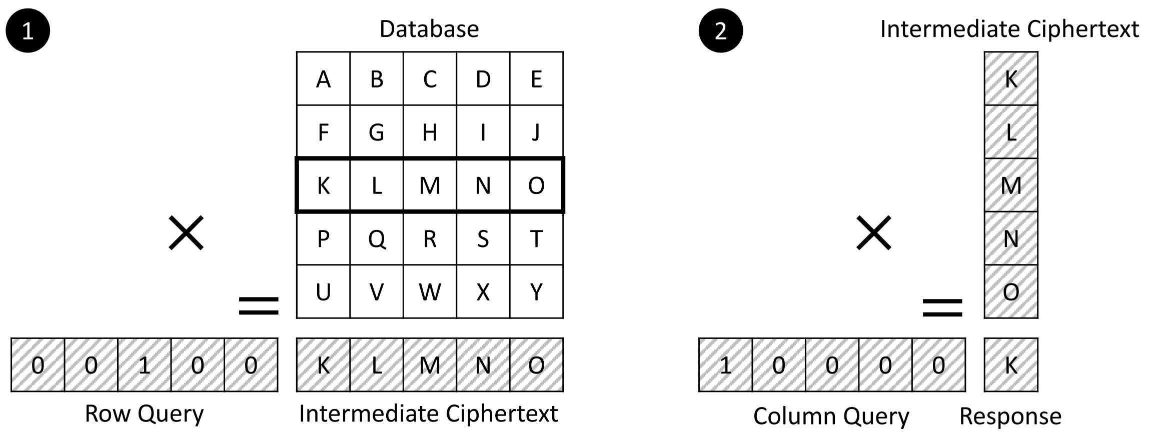

Figure 2-2: 2D PIR scheme

As a first step, the server multiplies the database $D^{\sqrt{n}\times\sqrt{n}}$ with the row part of the query, q_1. Each plaintext-ciphertext multiplication is done using the $\star$ operation.

$$A^{1\times\sqrt{n}} = q_1^{1\times\sqrt{n}} \cdot D^{\sqrt{n}\times\sqrt{n}}$$

This results in a vector containing encryptions of all of the records in row i. For the next step, the server multiplies this result with the column part of the query, q_2. Now, all multiplications will be ciphertext-ciphertext, so they will use the $\otimes$ operation.

$$r = q_2^{1\times\sqrt{n}} \cdot (A^{1\times\sqrt{n}})^T$$

This process is demonstrated in Figure 2-2. This can be extended to an arbitrary number of dimensions, with all dimension reductions after the first using the $\otimes$ operation.

2.1.2 Homomorphic Encryption

Two encryption schemes which have the necessary operations, $\oplus$, $\otimes$, and $\star$, are BFV and RGSW encryption. A crytography system which implements these types of operations is called homomorphic. We present the definitions of BFV and RGSW that are used in the implementation of OnionPIR, rather than those presented in the paper.

Operation Cost Summary

Table 2.1 is a summary of the different homomorphic operations discussed below, their computational costs (in equivalent polynomial multiplications), and the noise of the resulting ciphertext (assuming a ciphertext c or two ciphertext cx and cy, and plaintext p) [21]. B, l, and t are parameters of the cryptosystems and will be explained below.

Operation	Cost	Noise Growth
BFV ciphertext-ciphertext addition	-	$O(\text{Err}(cx) + \text{Err}(cy))$
BFV ciphertext-plaintext multiplication	2	$O(\text{Err}(c) \cdot \lvert p \rvert)$
BFV ciphertext-ciphertext multiplication	$4 + 2l$	$O(t \cdot (\text{Err}(cx) + \text{Err}(cy)))$
BFV-RGSW External Product	$2l$	$O(B \cdot \text{Err}(cx) + cy)$

Table 2.1: Homomorphic Operation Costs

BFV Encryption

BFV encryption deals with integer polynomials of degree less than n (where n is one security parameter), whose coefficients are from $\mathbb{Z}_q$ (where q is another parameter). $\mathbb{Z}_q$ here is the set of integers modulo q. We say that these polynomials are members of a ring $R_q = \mathbb{Z}_q/(f(x))$, where $f(x) = x^n + 1$. After polynomial multiplications or additions are done, the coefficients often need to be reduced modulo q which is represented by $[h]_q$ for some polynomial h and modulo q. [10]

Polynomials can be represented by their coefficients in the interval 0 to $q - 1$. Plaintexts are integer polynomials drawn from R_t, and ciphertexts are pairs of polynomials drawn from R_q. The ciphertext expansion factor F is the ratio in size between ciphertexts and plaintexts, which is:

$$F = \frac{2 \log q}{\log t}$$

During the encryption process, and in setting up secret and public keys, several random polynomials must be chosen. For the purposes of proving the security of this scheme, the exact manner in which these polynomials are drawn from R_q is important. However, the details are not relevant here so I will point the reader to [10] for additional information.

To encrypt plaintexts, a secret key must first be chosen. The secret key s is a random polynomial in R_q.

To encode a plaintext message m, a random polynomial a must be sampled from R_q. A small random polynomial e, the error term, is also sampled from R_q. Let $b = -(a \cdot s + e)$, and $\Delta = \lfloor q/t \rfloor$. Then the ciphertext can be calculated as:

$$c = (b + \Delta \cdot m, a) = ([-(a \cdot s + e) + \Delta \cdot m]_q, a)$$

To decrypt a ciphertext (c_0, c_1):

$$\mu = \left\lfloor \frac{c_0 + c_1 \cdot s}{\Delta} \right\rceil = \left\lfloor \frac{-e + \Delta m}{\Delta} \right\rceil = \left\lfloor m + \frac{-e}{\Delta} \right\rceil$$

Since e is small, when μ is rounded, we obtain the original message m [26].

Under these methods of encryption and decryption it is possible to construct definitions for the homomorphic operations needed for a PIR scheme - that is $\oplus$, $\otimes$, and $\star$. Also notice that the accuracy of decryption depends on the error term, or "noise" being small. Homomorphic operations can increase the noise in a ciphertext, and if it becomes too large, the ciphertext will not be decryptable.

Homomorphic addition can be performed in this scheme by simply adding the polynomials in the ciphertexts togethers. For two ciphertexts cx and cy:

$$(cx_0, cx_1) \oplus (cy_0, cy_1) = ([cx_0 + cy_0]_q, [cx_1 + cy_1]_q)$$

Here, the normal addition operator represents polynomial addition, **which can be performed by adding each coefficient**. Thus, the $\oplus$ operation is linear in the size of the polynomial, n.

The noise in the resulting ciphertext is the sum of the noise in the two input ciphertexts.

Homomorphic plaintext-ciphertext multiplication can be performed in this scheme by multiplying each of the two ciphertext polynomials by the plaintext polynomial. For plaintext p and ciphertext c:

$$p \star c = (p \cdot c_0, p \cdot c_1)$$

Here, the normal multiplication operator represents polynomial multiplication. This can be done by performing polynomial multiplication in the standard way, but it is typically done by using a Number Theoretic Transform (NTT) [18]. Performing an NTT on two polynomials allows them to be multiplied coefficient-wise. By then performing an inverse-NTT, the resulting polynomial will be the same as if the polynomials had been multiplied in the standard way. Since many ciphertexts or plaintexts will be reused in multiple $\star$ operations, the NTT operation can be bulk-applied to all cipher and plaintexts before and after a large computation. **In this case, the $\star$ operation will effectively boil down to coefficient-wise multiplication**, making it linear in the size of the polynomial.

The noise in the resulting ciphertext is the magnitude of the plaintext p times the noise in the ciphertext c.

Homomorphic ciphertext-ciphertext multiplication is more complicated. The initial multiplication step produces not two, but three polynomials. A relinearization step is required to reformat the ciphertext into two polynomials [10]. Overall, the computational cost of this operation is equivalent to about $4 + 2l$ polynomial multiplications, where $l = \lfloor \log_T q \rfloor$ for a chosen base T [10]. l is usually 5 [21].

The noise in the resulting ciphertext is t times the sum of the noise in the two input ciphertexts.

RGSW Encryption

An RGSW plaintext is also a polynomial, as in BFV encryption. However, an RGSW ciphertext is constructed as follows. For a base parameter B and length parameter l, the gadget vector g is defined as:

$$g^{(\ell\times 1)} = (B^{\log q/\log B-1}, B^{\log q/\log B-2}, ..., B^{\log q/\log B-\ell})^T$$

The gadget matrix G is:

$$G = I_2 \vee g = \begin{bmatrix} g & 0 \\ 0 & g \end{bmatrix}$$

A ciphertext that encrypts a message m is then:

$$C = [Z + m \cdot G]_q$$

where each row of $Z \in R^{(2\ell\times 2)}$ is a BFV encryption of 0. The bottom half of the matrix C is very similar to ℓ BFV ciphertexts encrypting base B decompositions of the plaintext m, except that the factor of Δ has not been multiplied by m. The details of these decompositions are not important for this thesis, but they can be used to decrypt the ciphertext [21].

Due to the relationship between RGSW and BFV encryption, a new operation is possible: **External Product**. This operation multiplies an RGSW ciphertext cx and BFV ciphertext cy, and returns a BFV ciphertext. The computational cost is equivalent to 2ℓ polynomial multiplications. The noise of the output ciphertext is B times the noise of cx plus the noise of cy. Since the noise growth of the external product is less compared to that of normal BFV ciphertext-ciphertext multiplication, it is a good replacement for the $\otimes$ operation in private information retrieval.

2.1.3 OnionPIR

OnionPIR, like most PIR schemes, uses the basic pieces of §2.1.1 and §2.1.2. The main improvements that OnionPIR makes are to reduce the response size and computational cost compared to other modern PIR algorithms [21].

In order to reduce noise growth, other algorithms such as SealPIR do not use homomorphic ciphertext-ciphertext multiplications. Instead, once one dimension of the database matrix is processed, the resulting ciphertexts are split into F (the ciphertext expansion factor) ciphertexts. These ciphertexts are then treated as plaintexts for the next step, so that instead of ciphertext-ciphertext multiplication with the next portion of the query, plaintext-ciphertext multiplication is performed. However, when the matrix is split into d dimensions, this results in a response size of F^{d-1} ciphertexts (compared to a single ciphertext when ciphertext-ciphertext multiplications can be used). Remember that if noise growth is uncontrolled, ciphertexts cannot be decrypted, and the noise compounds over the multiplications in all d dimensions.

As a solution to this problem, OnionPIR uses the external product operation instead of BFV ciphertext-ciphertext multiplications. The query contains RGSW ciphertexts instead of BFV ciphertexts so they can be used in the external product operation with the intermediate ciphertext matrices produced from multiplying the last dimension. The lower noise growth of the external product operation allows its use without compromising the ciphertexts, so the final response can be a single ciphertext. The lower noise growth also provides headroom to increase t, reducing the ciphertext expansion factor F and further reducing the response size.

To keep computational costs low, the first dimension remains a BFV computation - specifically, a BFV plaintext-ciphertext multiplication. The first dimension is also made larger than the subsequent dimensions to ensure that the bulk of the computation is in that dimension, and the larger computational cost associated with the external product multiplications in other dimensions is not a bottleneck. One other caveat is needed for the first dimension, which is that each plaintext in the database and each ciphertext in the first query is *decomposed* into two parts. When a decomposed plaintext is multiplied with a decomposed ciphertext, the dot product is taken between them. This decomposition is to help control noise.

The reference OnionPIR implementation uses a first PIR dimension size of 256, and subsequent dimension sizes of 4 [22]. The number of dimensions for the database depends on the database size. In general, there are tradeoffs for making the number of dimensions larger. The first dimension matrix multiplication must always calculate across the entire database, and the subsequent matrix multiplications are extra computations that add to the total computation cost. Although the first dimension is the most expensive, the subsequent dimensions do add some computational cost (especially since they are ciphertext-ciphertext multiplications in OnionPIR). On the other hand, by increasing the dimension size, the request size decreases, since you can now send multiple shorter vectors instead of one long query vector. OnionPIR's practical choice was made primarily for the purpose of reducing request size. By making the number of dimensions dependent on the size of the database, the request size is logarithmic in the size of the database. Since OnionPIR keeps the other dimensions small (4 compared to the first dimensions size of 256), and each dimension reduces the size of the matrices that need to be multiplied, the bottleneck will mainly be in the first dimension and the computational cost is essentially still linear in the size of the database despite the extra computations for each dimension. Other PIR schemes make different tradeoffs for the dimensionality.

2.1.4 Other PIR Schemes

Many modern PIR schemes have similarities in the basic scheme they follow - many follow §2.1.1 closely. They also rely on homomorphic encryption schemes, most of which use similar fundamentals. Here I catalog some PIR schemes, as well as one example of a near-storage accelerator for a PIR scheme, to display some of these similarities as well as their differences and the tradeoffs each make. Many of these similarities and differences will become relevant as we apply our own near-storage accelerator to these schemes.

SealPIR

SealPIR's main contribution to single-query PIR is in reducing the query size. Instead of sending a one-hot vector (i.e. encryptions of 0s and 1s) to the server, the client instead encrypts the index i of the database record they are requesting. The plaintext the client encrypts is the monomial x^i. The server then performs an oblivious expansion procedure that creates the necessary one-hot vector from the client's encryption of i.

As mentioned in §2.1.3, SealPIR treats the database as a multi-dimensional matrix. The client sends an encryption of an index for each axis of that matrix based on which record they are requesting. After multiplying the database with the first one-hot expanded query vector, SealPIR expands each ciphertext in the resulting vector into F ciphertexts, and treats them as plaintexts for the next round of multiplication. Thus, all multiplications will be plaintext-ciphertext multiplications to keep noise growth small. The response size in SealPIR is F^{d-1} ciphertexts.

MulPIR

MulPIR introduces various techniques to reduce the request and response size of SealPIR [2]. In BFV encryption, the first polynomial in the ciphertext is a random polynomial drawn from R_q. Instead of sending this polynomial as part of the query, the client can send the random seed used to generate it, and the server can re-generate it. This reduces the query size by about half.

MulPIR also uses a technique called modulus switching on the final ciphertexts in the PIR response to make them as small as possible. This technique decreases the size of the ciphertexts at the cost of increasing the noise. Since the ciphertexts will only be used once for decryption purposes, they can be decreased in size as much as possible while still being decryptable.

MulPIR uses homomorphic ciphertext-ciphertext multiplications to reduce the response size to one ciphertext. This decreases the response size at the expense of a higher computational cost.

SimplePIR

SimplePIR uses a different encryption scheme than many others discussed in this section, but it is based on the same RLWE concepts. Here, a vector is chosen as the secret key. A matrix A and error vector e are chosen at random. For ciphertext modulus q and plaintext modulus p, the ciphertext for a vector μ is:

$$\mathcal{E}(\mu) = (A, c) = (A, As + e + \lfloor q/p \rfloor \cdot \mu)$$

.

SimplePIR considers the database as a $\sqrt{N}$ by $\sqrt{N}$ matrix D, and the client submits two encrypted query vectors which are one-hot vectors encoding the row and column indices of the desired record in the matrix. Note that unlike BFV encryption, the entire vector is encrypted rather than each element of the vector being a separate ciphertext [15].

This encryption scheme is linearly homomorphic, meaning the database D can be multiplied by query vector q straightforwardly under encryption - $\mathcal{E}(D \cdot q) = D \cdot \mathcal{E}(q)$. In other words, the $\star$ operation is simple multiplication. Both elements of the encrypted vector, A and c, are multiplied by D. This costs one matrix-matrix and one matrix-vector multiplication.

SimplePIR additionally proposes that the client reuses the matrix A over multiple queries. Then, the server can calculate $D \cdot A$ once, and the client can download this value as a "hint". Only the matrix-vector multiplications would need to occur for every subsequent query.

This scheme has very high throughput at the cost of downloading a the hint once per client.

FastPIR

FastPIR is a PIR scheme created with the goal of low latency and high throughput in mind, since it was geared towards a voice-calling application where real-time response times are critical [1]. FastPIR uses BFV ciphertext operations for their computational efficiency.

As in other PIR methods, the query is a one-hot vector for the row of the record in the database. However, FastPIR does not use one ciphertext to encode each element of these vectors. Instead, the entire one-hot vector is encoded into a single plaintext and then encrypted. If there are more database records than the size of a plaintext, multiple plaintexts are needed to encode the query. The plaintexts for the database also include multiple elements of the database per plaintext. A BFV multiply operation is used to multiply these plaintexts and ciphertexts such that the result is an encryption of the elementwise multiplication of the underlying plaintexts. By multiplying the one-hot query vector with a database vector, the result is an encryption of many zeros and the requested record. Packing multiple indices of the one-hot vector into a single plaintext reduces request size.

The database will have multiple columns of data in each record. As a result of the multiplications, there will be one encrypted vector with a single piece of data (and many zeros) per column. To save response space, the FastPIR algorithm uses a BFV rotate operation to rotate these vectors such that the data land at different indices in each vector. These vectors can then be added together to combine data from multiple columns into a single encrypted vector, which is returned to the client.

INSPIRE

The INSPIRE system is a near-storage hardware accelerator for the FastPIR algorithm [19]. The INSPIRE system makes changes to the FastPIR protocol that make it more amenable to hardware acceleration. The original FastPIR organizes vector rotations in a recursive manner, whereas INSPIRE organizes them in an iterative manner that is faster and more amenable to hardware solutions.

The INSPIRE protocol also partitions the database into "groups" and each group into "blocks", and uses queries that target these groups and blocks. The database is streamed into the hardware, where the answers are first aggregated at the block level, then combined at the group level, and finally rotations are performed to collect results from each database column. The actual homomorphic cryptographic operations that must be performed are computed on a dedicated hardware unit. This hardware unit can perform operations on the long ciphertexts required by the INSPIRE protocol by breaking down large ciphertexts into smaller ones. Another dedicated hardware unit is used to accelerate a step of the homomorphic rotations.

This hardware was able to achieve an impressive 22.9x speedup compared to the software changes to the protocol alone.

2.2 Timeloop

In designing a hardware accelerator for a PIR algorithm, there are two main considerations we made. One is the massive amount of data being processed, and the second is the optimization of the individual calculations. To help process large amounts of data, it is helpful to have a hardware accelerator with a storage hierarchy - i.e. the database lives on disk, but parts of the data are successively pulled onto DRAM, SRAM, and registers, and finally the computational units to perform the necessary cryptographic operations. Data which is reused does not necessarily have to be accessed again multiple times from the slower levels of the storage hierarchy if it exists at the faster levels of the storage hierarchy when it is needed.

Timeloop is an infrastructure capable of evaluating different such hardware architectures for a specific design problem and determining what the best way to arrange data on the different storage levels is, and in what order to access the data to do the computations. Timeloop will then report the number of cycles taken for the entire computation, the area of the architecture, and the energy taken to do the computation, among many other relevant statistics. By testing many different architecture designs, Timeloop can be used to find a good design for a particular computational problem.

2.2.1 Problem

Timeloop is designed for Deep Neural Network (DNN) problems, but it has applications in many problems that can be expressed as a nested loop. This includes various types of matrix and tensor multiplications and convolutions [24]. A problem ("workload") is expressed in timeloop using a form of einsum notation [23].

In einsum notation, summations are left out and implied by the fact that certain indices only appear on one side of the equation. For example, the matrix multiply of A and B can be written as:

$$C^{n \times m} = A^{n \times p} \cdot B^{p \times m}$$

$$C_{i,j} = \sum_{k=0}^{p} A_{i,k} \cdot B_{k,j}$$

In einsum notation, this would be:

$$C_{i,j} = A_{i,k} \cdot B_{k,j}$$

This can simplify the notation for complex convolutions and multiplications when there are many different indices.

To define such a problem in timeloop, all indices should be specified (here, these are i, j, and k). This is called the "operation space", as it defines the entire space over which computations occur.

Then, all tensors should be specified. These are the "data spaces". For each tensor, a "projection" must be specified, which defines the relation between that data space and the operation space. In other words, how the tensor is indexed by the dimensions (in this case i, j, and k) in the operation space. For tensor C, it is indexed by the pair $[i, j]$, for example.

Finally, the problem must be concretized by assigning the bounds of each dimension in the operation space. This in turn defines the size of each tensor in the data space.

2.2.2 Architecture

An architecture in Timeloop consists of a storage hierarchy "tree" of several levels, culminating in compute unit "leaves". This can be defined by specifying the type of storage appearing at each level and the number of instances of each [24].

For example, you may specify that your storage hierarchy begins with a large DRAM (where all data is initially resident). That DRAM connects to two SRAM buffer units, each of which connects to four register files, each of which connects to a compute unit. To specify this structure in Timeloop, you can specify the total number of storage units at each level.

For each type of storage, microarchitectural attributes can be specified that affect its functioning in the system. For example, the datawidth, bandwidth, and total size of the storage unit can be specified.

With smart data access patterns, such a storage hierarchy can be used to take advantage of data reuse in a certain problem space. By loading a data value into a lower tier of the storage hierarchy, expensive DRAM accesses can be avoided on every access to that same location, replacing them with faster SRAM or register file accesses. This will become even more important when exploring applications to PIR, where the top tier of memory accesses is to the Disk, which is slower than DRAM and works best if reads and writes are performed in very large chunks at a time.

2.2.3 Mapping

A mapping in Timeloop describes the way in which the operation space is split into tiles and the order in which those tiles are delivered to each level of the memory hierarchy [24]. A tile is a portion of the operation or data spaces, based on looping through just part of the bounds of one or more of the dimensions.

Recall our matrix example:

$$C_{i,j} = A_{i,k} \cdot B_{k,j}$$

Suppose we instantiate this with concrete bounds for i, j, k, denoted $i = 16$, $j = 8$, $k = 4$.

For instance, in our matrix example, a tile could be the resulting operation and data spaces from using only half of all values of i. Thus, we would use all values from $i = 0$ to 7 in one tile, and the second tile would contain all values form $i = 8$ to 15.

In terms of the data spaces, this tile would contain half the matrix C, half the matrix A, and the entire matrix B.

In a Timeloop mapping, you can specify that on the highest level of storage (say, the DRAM level), the operation space would be split into tiles in this manner to be delivered to the next level of storage (say, the SRAM level). In timeloop, this is specified by stating that this level of storage has factor "2" for the "*i*" dimension, and factor "1" for all other dimensions (since the other dimensions are not being split).

In addition to specifying which tiles must be delivered to the next level of storage, a timeloop mapping must also specify in what order. For the above example, there is only one dimension across which the tiling is occuring, so tiles will simply be delivered in order across that dimension. However, data can be tiled across multiple dimensions at once.

For example, we can tile in both the i and j dimensions (say, with a factor of 2 again). Now, there are four tiles to deliver to the next level, and we have two choices as to how to deliver them.

Option 1 would be:

```
for i' in range(2):
    for j' in range(2):
        deliver tile i', j'
```

Option 2 would be:

```
for j' in range(2):
    for i' in range(2):
        deliver tile i', j'
```

The ordering of these tiles in Matrix C are show in Figure 2-3.

So, at every level of storage, a timeloop mapping specifies both the tiling factors of all dimensions - this determines *how many* tiles there are and how they are defined across dimensions - and the ordering of the dimensions - this determines *how* tiles are delivered to the next level of storage.

A full example of a mapping is located in Appendix A.

2.2.4 Mapper

Timeloop can iterate through all legal mappings for an architecture and check which are the best on certain metrics, such as cycle count or energy [24]. This is accom-

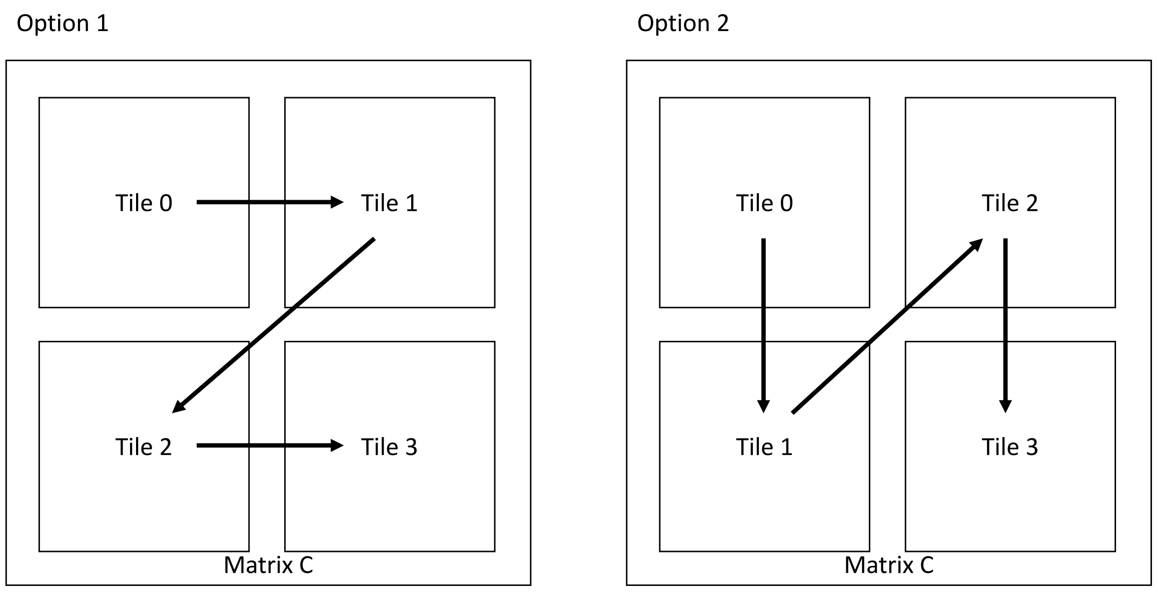

Figure 2-3: Matrix C Tiling Comparison

plished by checking every possible reordering of the dimensions at each storage level, and every possible dimension factor at each level (so long as their product is the overall dimension bound). Each mapping must be checked for validity by enforcing that data for a tile being stored at each level can actually fit in the storage allotted at that level.

Each mapping is evaluated by determining how many cycles it will take to deliver tiles and do the computation itself. Delivering tiles and doing computations can happen in parallel across different storage levels, assuming double buffering. Cycle counts are limited by the number of computations that must be done, the size of tiles to deliver, and the bandwidths of various storage components.

Each mapping is also evaluated on the energy consumed by these operations. Each primitive operation, such as an individual read/write from a storage unit, or an individual computation from a compute unit, is evaluated through a program called Accelergy, which takes in the architecture specifications (such as size, datawidth, etc), and determines the energy for these operations. Timeloop then determines the overall energy by adding up the number of these operations which must occur over the course of the entire calculation.

Timeloop can iterate through every possible mapping for small mapspaces, but for larger mapspaces, there are also flags to stop the search early if a certain number of consecutive invalid or suboptimal mappings have been found. Timeloop also has options to search the mapspace linearly, randomly, or a mix of the two. When finished, the timeloop mapper reports the statistics for the best mapping.

2.2.5 Design Space Tool

Timeloop is also able to assist in exploring design spaces through the design-space tool [8]. This tool allows for sweeping over various architectural variables, such as the sizes and numbers of storage units. For each architecture, it runs the timeloop mapper and finds the best mapping. This allows the user to easily compare architectures for speed and energy on the same problem.

This will be especially relevant for the our case of PIR, where it is not immediately clear what architectures will yield the best results. Timeloop's automation will make it easy to iterate over many architectures at once.

Chapter 3

Application Characteristics

To determine the best design of a hardware accelerator, first we explored the runtime characteristics of OnionPIR when running with large database. We profiled the disk bandwidth while OnionPIR was running to determine if it was saturated by the application, as this has implications for the design of hardware accelerators as well. We found that with modest parallelization, disk bandwidth would be saturated on modern SSDs, so the best choice of hardware accelerator is a near-storage hardware accelerator.

We also determined the specific operation which was taking the most runtime. This operation turned out to be the query-database and query-intermediate ciphertext multiplications that occur during OnionPIR. These are effectively matrix-vector multiplications, but where each element of the matrix and vector are RGSW or BFV ciphertexts. Those ciphertexts turn out to be matrices of their own due to the definition of RGSW ciphertexts and the decomposition occuring in OnionPIR. Inside those matrices are several polynomials.

Finally, we explored the extensibility of the OnionPIR bottleneck operation to other PIR schemes, as this informs if the hardware accelerator would be more widely applicable. Several PIR schemes also share a main operation which is also a "matrix-of-matrix-of-many-polynomial" multiplication, making this operation possibly extensible to other protocols.

3.1 Disk Bandwidth Profiling

In order to run OnionPIR on large database sizes, where we would like to optimize its performance, we first had to modify the OnionPIR source code. The source code was originally created to keep the entire database in memory. We modified this to store the database instead in files on disk, using the `mmap` command to map the database into virtual memory [20].

To profile OnionPIR, we ran OnionPIR with a database size larger than physical memory and ran the `iostat` tool to determine how much disk bandwidth was taken up by the program [17]. Our tests revealed that approximately 650MB/s disk bandwidth was used by OnionPIR.

In comparison, we also ran the disk benchmarking tool `fio` [11]. We used the `fio` tool to test the speed of the disk on our machine by running a sequential read test. This revealed a maximum disk bandwidth of around 1000MB/s. The disk on this machine is a modern SSD.

OnionPIR is currently a single-threaded program, with many opportunities for parallelism. This test reveals that if OnionPIR is parallelized, it will quickly saturate the disk bandwidth and fail to see continued benefit from further parallelism. Because of this, we chose to design a near-storage hardware accelerator. A near-storage accelerator can take advantage of the internal bandwidth of the SSD, rather than being limited to the external bandwidth. In addition, by placing the computation directly near disk, the data does not have to travel through the entire memory hierarchy of caches and DRAM to reach the CPU before computation can begin. Since disk bandwidth can easily become a limiting factor in OnionPIR, a near-storage accelerator makes sense in our design.

3.2 Runtime Profiling

We profiled OnionPIR using the callgrind tool, which constructs a callgraph and counts the costs of various functions running within a program [6]. This allows for determination of where a program spends most of its time.

We profiled OnionPIR running on a Linux virtual machine that was allocated 8GB memory, using a 32GB database. This ensured that OnionPIR would be tested in an environment where it would be required to read from disk, as would likely be the case in a large-scale production application of OnionPIR.

We found that OnionPIR spent 75.55% of the time in the external product function. See Table 3.1 for the results. This function is used in both the first dimension and subsequent dimensions to do the underlying matrix multiplications. In the first dimension, it is used to perform the ciphertext-plaintext multiplications between the first query vector and the database matrix. In subsequent dimensions, it is used to perform the external product ciphertext-ciphertext multiplications between an RGSW query vector and an intermediate ciphertext matrix. 99.4% of the external product multiplication effort is spent in the first dimension.

Percentage of Cycles	Function
42.69%	Polynomial Multiplication
21.34%	Polynomial Addition
11.52%	Other
75.55%	External Product (Total)

Table 3.1: Cachegrind Results

3.3 Basic Operation

The responsibility of the external product function is to do the RGSW-BFV ciphertext-ciphertext external product multiplications that occur during the matrix-vector multiplications of OnionPIR [22].

In dimensions past the first, the vector being multiplied is an RGSW one-hot vector, and the matrix is an intermediate matrix of BFV ciphertexts. Recall the parameter ℓ for the RGSW cryptosystem - the size of an RGSW ciphertext matrix is $2\ell \times 2$, with each element of the matrix being a polynomial. The other parameter for the cryptosystem is the base B. In order to use the RGSW external product operation, the BFV ciphertexts are decomposed, base B, into $\ell \cdot 2 = 2\ell$ polynomials.

This setup is done in between each successive dimension computation. From here, the external product operation between the RGSW ciphertext c_1 (a $2\ell \times 2$ matrix) and the BFV ciphertexts (effectively a $2\ell \times 1$ matrix) c_2 is the matrix multiplication $c_1^T \cdot c_2$.

Each element within the matrix that needs to be multiplied and added during this matrix multiplication is a polynomial. The polynomial additions can be accomplished by adding the coefficients element-wise modulo q (the ciphertext modulus), and the multiplications can be accomplished by multiplying the coefficients element-wise modulo q if an NTT is applied as a pre-processing step and an inverse NTT as a post-processing step.

However, each ciphertext or plaintext matrix element is actually not a single polynomial, but multiple polynomials, one per ciphertext modulus. Thus far, we have been assuming one modulus q for the entire system. However, OnionPIR uses multiple coefficient moduli, storing one polynomial for each. Thus, each multiplication and addition operation must be repeated for each polynomial within the matrix element.

The first dimension does not use ciphertext-ciphertext multiplications, since the database consists of plaintexts. However, it can still use the same external product function. The BFV ciphertexts for this dimension are decomposed as well, except instead of decomposition base B they are decomposed into 2 parts, forming a 2×2 matrix. The plaintexts are also decomposed in a similar way, forming a 2×1 matrix. These match the dimensions of the RGSW external product operations if ℓ were set to 1. This means that the operations occurring in the first dimension and subsequent dimensions are exactly the same, just operating over different matrices and vectors.

For simplicity, we will assume that the query vector ciphertexts are pre-transformed to be $2 \times 2\ell$ matrices. Then, what we have as the bottleneck operation is a matrix-vector multiplication, where each element of the matrix and vector are themselves matrices and vectors, and where elements of those matrices and vectors are lists of several polynomials. To keep this as generalizable as possible, we will not make any assumptions about the dimensions of these matrices. With this in mind, this operation can be defined by the following einsum.

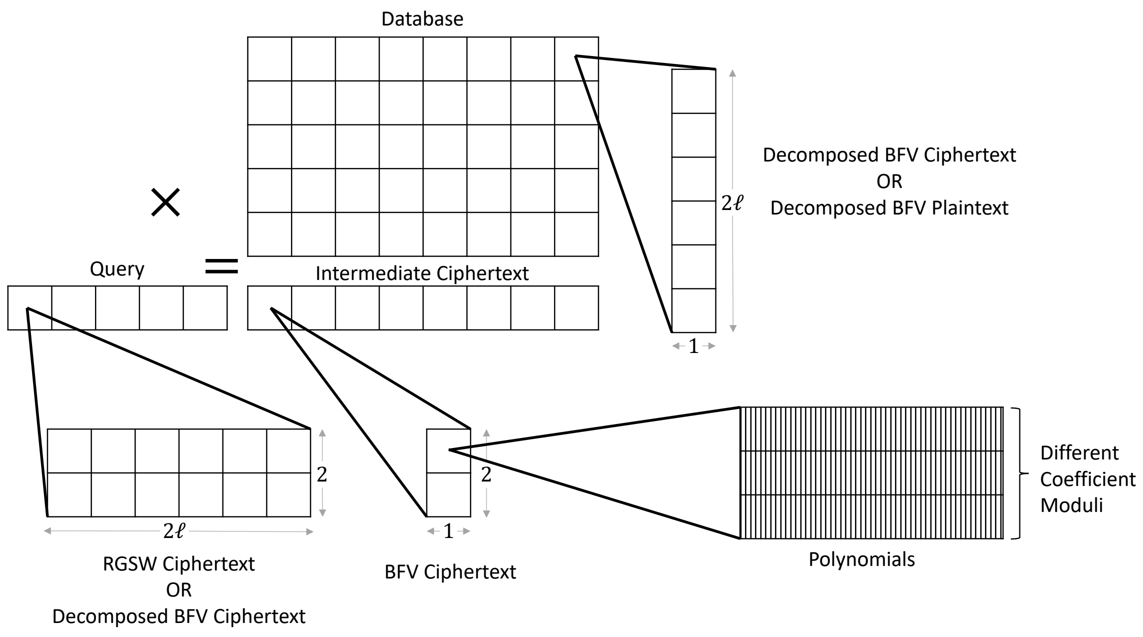

Figure 3-1: OnionPIR Basic Operation

$$C_{m,n,p,q,c,d} = A_{m,j,p,k,c,d} \cdot B_{j,n,k,q,c,d}$$

Here:

- m represents the row in matrices A or C
- n represents the column in matrices B or C
- p represents the sub-row (row in the sub-matrix) in A or C
- q represents the sub-column (column in the sub-matrix) in B or C
- c represents the coefficient modulus
- d represents the polynomial coefficient
- j is a summation index which is the column in A and row in B
- k is a summation index which is the sub-column in A and sub-row in B

This operation is shown in Figure 3-1 for OnionPIR and in general in Figure 3-2.

Let the bounds for these dimensions be M, N, P, Q, C, D, J, and K respectively. In OnionPIR, the polynomials typically have degree 4096, so $D = 4096$. There are typically 2 coefficient moduli, so $C = 2$. Since the sub-matrices are $2 \times 2\ell$ and $2\ell \times 1$,

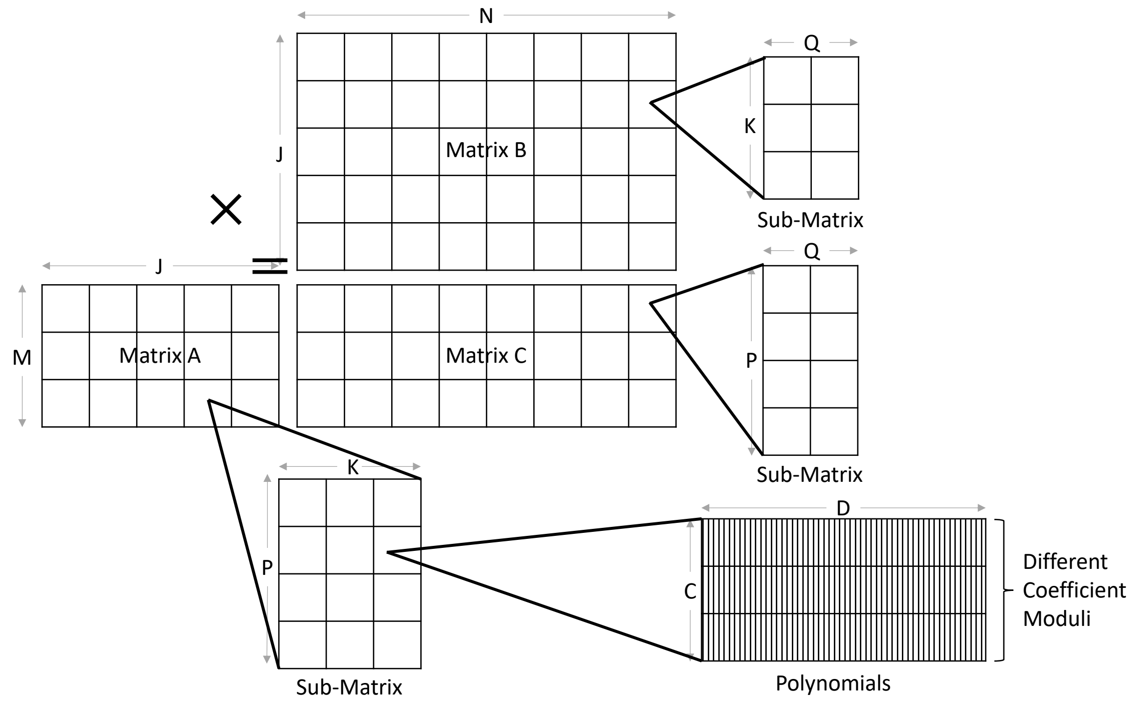

Figure 3-2: Generic Basic Operation

and ℓ is 1 for the first dimension operation, $P = 2$, $Q = 1$, and $K = 2$. K would change for different values of ℓ, such as in subsequent dimensions where $\ell = 5$ (and $K = 10$). A will be a query vector, so $M = 1$. N and J depend on the size of the database and which dimension the operation is being applied in. As an example, for a 64GB database, the first dimension will have $J = 256$ and $N = 2048$.

It is important to note here that the products and sums implied by this einsum are not integer multiplication and addition, but integer multiplication and addition *modulo specific coefficient moduli.* Which modulus to perform an operation under depends on the value of c. Each computation in the einsum takes the product of some element of A and some element of B, adds the value for some element of C, and stores this back into that element of C. This operation is called a "multiply-and-accumulate".

3.4 Applicability to Other PIR Schemes

We find that our operation is similar to the main operations found in some other PIR schemes. This hardware accelerator may therefore improve the speeds of not just OnionPIR, but other PIR schemes as well.

PIR Scheme	M	N	P	Q	J	K	C	D
SealPIR	1	$\sqrt{\text{records}}$	2	1	$\sqrt{\text{records}}$	1	2	4096
MulPIR	-	-	-	-	-	-	-	-
SimplePIR	$\sqrt{\text{records}}$	1	1	1	$\sqrt{\text{records}}$	1	1	1
FastPIR	1	columns	2	1	ptxts/column	1	2	4096

Table 3.2: Applicability of Basic Operation to Other PIR Schemes

3.4.1 SealPIR

SealPIR uses BFV ciphertexts and plaintexts and does not use decomposition. Each BFV ciphertext-plaintext multiplication, since it occurs under NTT, can be trivially viewed as a matrix multiplication between the BFV ciphertext (a 2×1 matrix of polynomials) and the BFV plaintext (a 1×1 matrix of polynomials). Once SealPIR has performed the setup before each dimension, which consists of splitting the ciphertexts into F ciphertexts to be treated as plaintexts, the main operation per dimension is a matrix-vector multiplication of ciphertexts and plaintexts, so our operation can be applied. Based on the ciphertext-plaintext matrix sizes, $P = 2$, $Q = 1$, and $K = 1$. The degree of the polynomials in SealPIR is 4096, so $D = 4096$. $M = 1$ since A is used for the query vector. N depends on which dimension is being processed and the size of the database, as does J. For the first dimension, they will be the square root of the database size. [3].

3.4.2 MulPIR

MulPIR uses BFV ciphertext-ciphertext multiplications. These are more complex than the RGSW-BFV external product of BFV ciphertext-plaintext operations, and involve relinearization to reduce the ciphertext size from 3 to 2 polynomials after the initial multiplication. As such, there is no easy way to use our operation in MulPIR, which relies on the ability to do elementwise multiplication and addition over the polynomial coefficients [2].

3.4.3 SimplePIR

SimplePIR's main operation is a matrix-vector multiplication [15]. Thus, our operation is straightforwardly applicable by setting all sub-matrix dimensions to 1. That is, $P = 1$, $Q = 1$, $K = 1$. In addition, there are no polynomials and only one coefficient modulus, so $D = 1$ and $C = 1$. The value of $N = 1$, since B will be a vector. The values of M and J will depend on the size of the matrix and vector being multiplied in SimplePIR, i.e. the size of the database.

SimplePIR will likely not benefit (and may be hindered by) any optimizations to our design specific to the fact that submatrices and polynomials exist within the matrices being multiplied.

3.4.4 FastPIR

In FastPIR, each database column contains several BFV plaintext vectors. The query is also several BFV ciphertext vectors, which together make up a one-hot vector selecting an row index from the database. The plaintexts from the database column are multiplied by the ciphertexts from the query vector and added together to form the result for a column. This happens for each column, making up a matrix multiplication across the entire database. The BFV plaintexts and ciphertexts, as discussed in §3.4.1, can be considered sub-matrices for the purposes of our operation. For a database with c columns and p plaintexts per database column, we can set $P = 2$, $Q = 1$, $K = 1$, $M = 1$, $N = c$, $J = p$.

Chapter 4

Hardware Accelerator Design

4.1 Overview

Our proposed hardware design is a near-storage accelerator operating at 2GHz located in an SSD. Our design is pictured in Figure 4-1.

Our accelerator relies on a storage hierarchy starting with an SSD and DRAM unit. Beyond the DRAM will be a hierarchy of SRAMs and register files that aims to take advantage of data reuse. Each unit in the storage hierarchy fans out into potentially multiple other storage units in the hierarchy level below it; exactly how high this fan-out is for each level is a detail to be determined in §6.

The leaves of this storage hierarchy are computational units that perform a multiply-and-accumulate operation with modulo. This is the main computation that is needed in the bottleneck operation of OnionPIR. We designed a 6-stage computation unit that can be pipelined further to achieve the desired clock time.

Since each computation depends not just on the data being operated on, but also on a modulus, a separate small structure is kept that stores modulus information. In OnionPIR and other PIR algorithms, there are typically very few coefficient moduli. Based on the tiling and loop ordering, the moduli selector unit will be responsible for selecting which moduli to apply to each computation. We anticipate very lower energy and area overhead for this structure.

The host will be responsible for setting up the dataflows within this hardware by

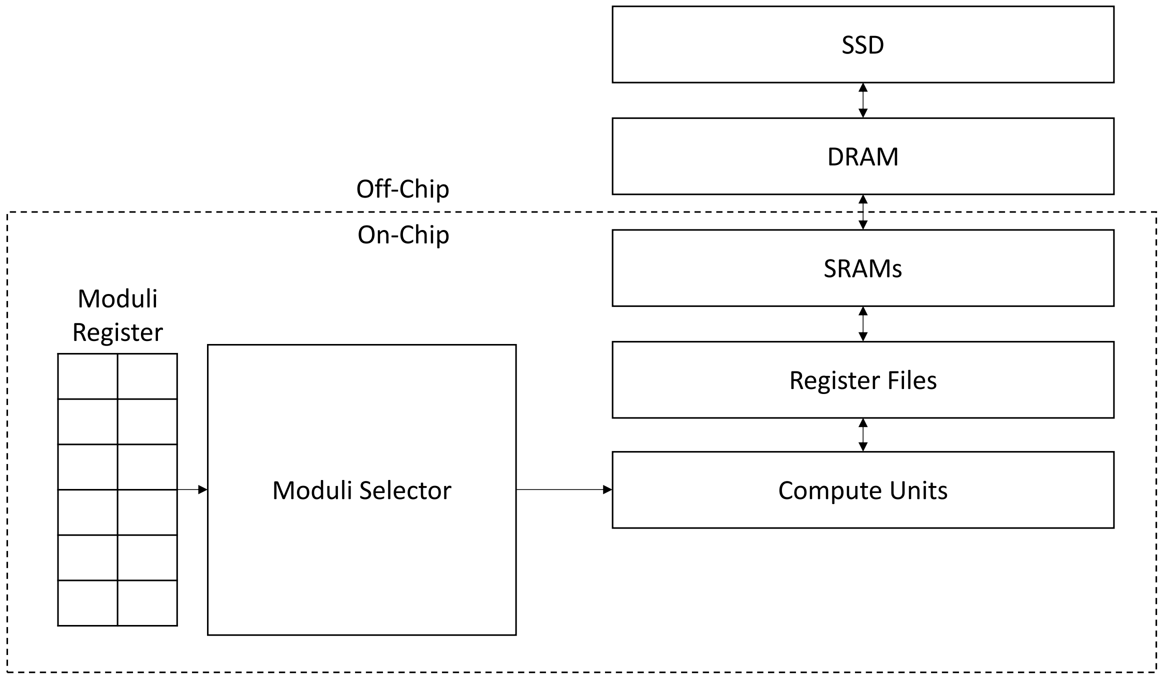

Figure 4-1: Overview of Hardware

sending a pre-determined dataflow schedule to a hardware scheduler unit that selects which data will be transferred between which storage units at what times during the computation.

4.2 Compute Unit

The compute unit needs to perform a multiply-and-accumulate, modulo some number which is different potentially every cycle (depending on the ordering of the loops to compute the einsum). The compute unit has inputs a, b, c, n, and m, an accumulator register, and an output. Each cycle, the compute unit calculates a new product (a times b for some previous inputs a and b, due to pipelining) and adds it to the accumulator register (modulo n). The compute unit outputs the value of the accumulator register. The input m will be explained in §4.2.1.

The compute unit also has a reset signal and a refresh signal. The reset signal simply resets all register values to zero. The refresh signal starts a new sum with a new modulo, beginning with the current product plus c, in the accumulator register, rather than adding to the current sum.

The compute unit will have two overall stages; one for the "multiply" step and one for the "accumulate" (addition) step. The compute unit is shown in Figure 4-2.

4.2.1 Multiplication

Algorithm

To multiply two numbers in hardware, we use the Barrett Reduction algorithm described in [4]. We will be multiplying two 64-bit numbers (a and b) and reducing the result modulo a third 64-bit number, n.

For simplicity, let the intermediate 128-bit product $p = a \cdot b$. Our goal is to find the result r, where:

$$r = p \mod n$$

We can write this using the definition of modulo, and floor division:

$$q := \lfloor p/n \rfloor$$

$$r = p - q \cdot n$$

Where we are now subtracting from p a whole number of ns to find the remainder on division by n. The first step, dividing p by n, is expensive in hardware, so we instead choose to replace division by n with multiplication by m and division by 2^k (accomplished with a shift) such that $m/2^k$ is as close to $1/n$ as possible. This is achieved by setting $m = \lfloor 2^k/n \rfloor$. We set the value of k to 128 in order to find the modulo of a 128-bit number p.

Now, our modulo calculation is:

$$p := a \cdot b$$

$$q := \lfloor (p \cdot m) >> k \rfloor$$

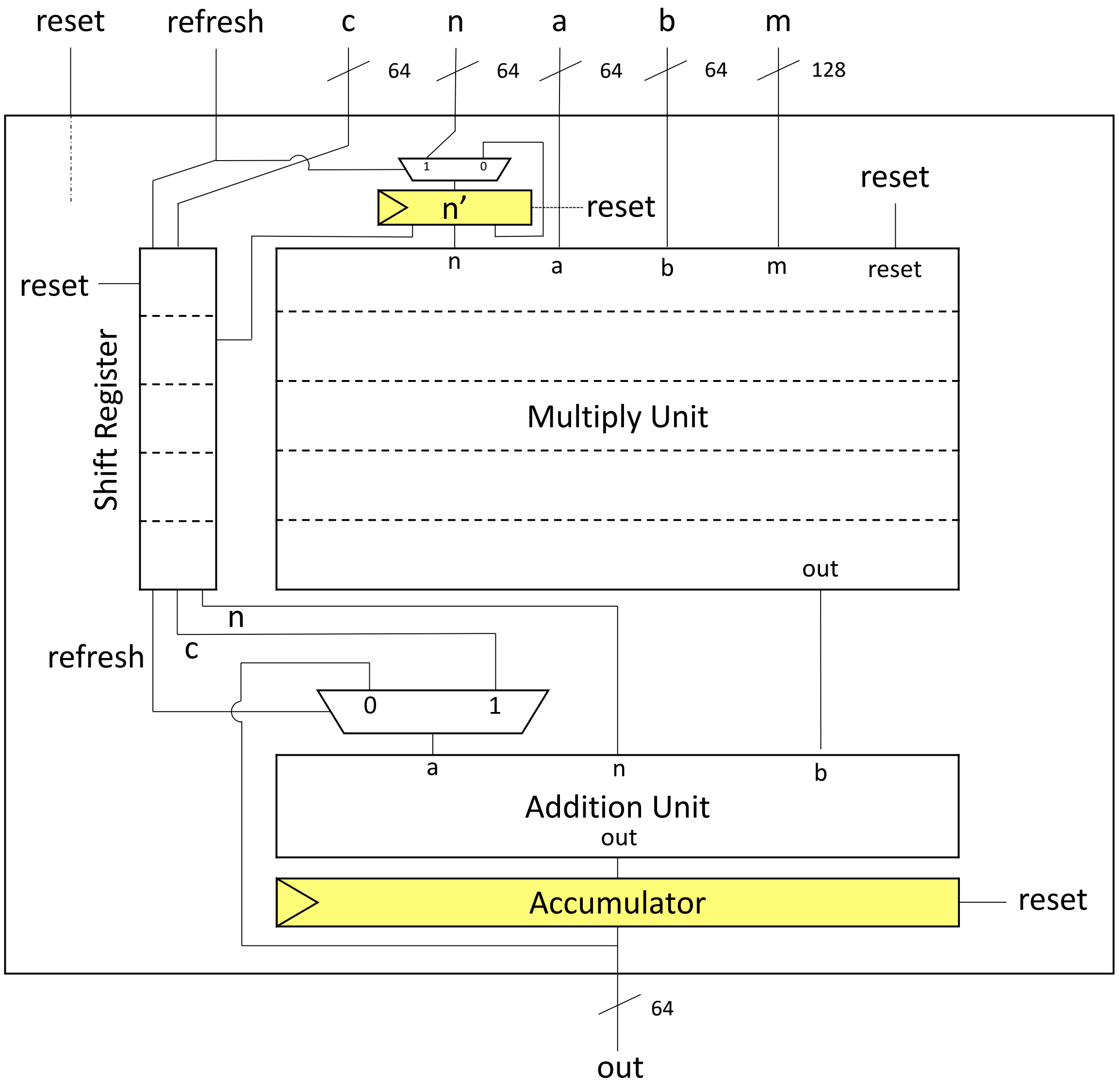

Figure 4-2: Compute Unit

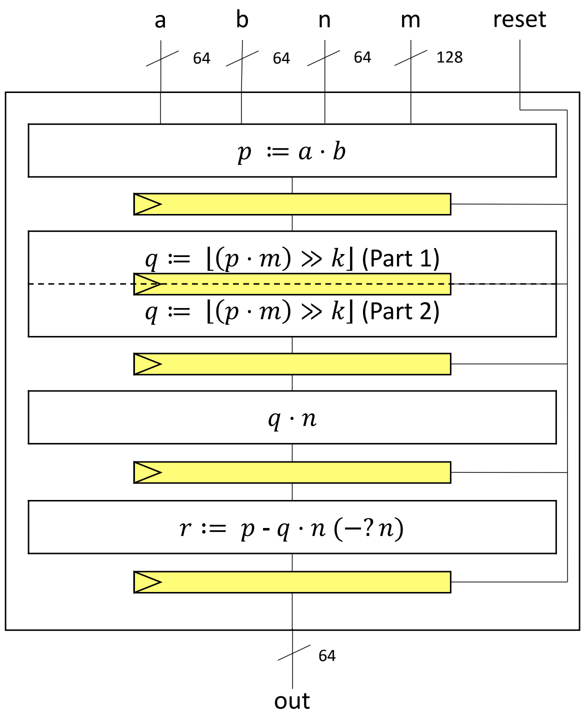
a
b
n
m
reset
64
64
64
128
$p := a \cdot b$
$q := \lfloor (p \cdot m) \gg k \rfloor$ (Part 1)
$q := \lfloor (p \cdot m) \gg k \rfloor$ (Part 2)
$q \cdot n$
$r := p - q \cdot n\ (-?n)$
64
out

Figure 4-3: Multiply Unit

$$r = p - q \cdot n$$

However, since m is only an approximation, q may be 1 too small. Thus, it is possible that to complete the modulo operation, an extra subtraction by n is needed.

The full operation is then:

$$p := a \cdot b$$

$$q := \lfloor (p \cdot m) >> k \rfloor$$

$$r = p - q \cdot n$$

$$r = \left\{ \begin{array}{ll} r - n, & \text{if } r \geq n \\ r, & \text{if } r < n \end{array} \right\}$$

Hardware

The multiplication unit contains four basic stages, displayed in Figure 4-3. The first stage calculates the initial product $a \cdot b$. This product is 128 bits. The second stage calculates the product $p \cdot m$. This product is 256 bits. It is shifted by k, which is equal to 128, so only the top 128 bits of this product need to be saved. The third stage calculates the product $q \cdot n$. Even though q is 128 bits and n is 64 bits, this product will be strictly less than p (which is 128 bits), so we can safely store only the lower 128 bits of this product. The final stage subtracts this $q \cdot n$ product from p, as well as an additional n if necessary.

As an additional optimization, the largest stage - the 128 bit by 128 bit multiplication of p and m - can be split into two stages. Each of these 128 bit numbers is split into an upper and lower half. In the first stage, the product of every pair of halves is taken. These are all 64 by 64 bit multiplications. In the second stage, these products are summed at the appropriate bit locations to form the final product.

Additional pipelining is discussed in §4.2.5.

4.2.2 Addition

Modulo addition is simple to implement in hardware. Our goal is to implement the operation $r = a + b \mod n$ in hardware, assuming that a and b are already reduced modulo n (i.e. both are less than n). This can be accomplished by simply performing the sum, and subtracting n if the result is greater than or equal to n:

$$s := a + b$$
$$r = \left\{ \begin{array}{ll} s - n, & \text{if } s \geq n \\ s, & \text{if } s < n \end{array} \right\}$$

The addition unit does not require the extra input m. This operation can be directly implemented in hardware in a single stage with a comparator, multiplexer, and adder, as shown in Figure 4-4.

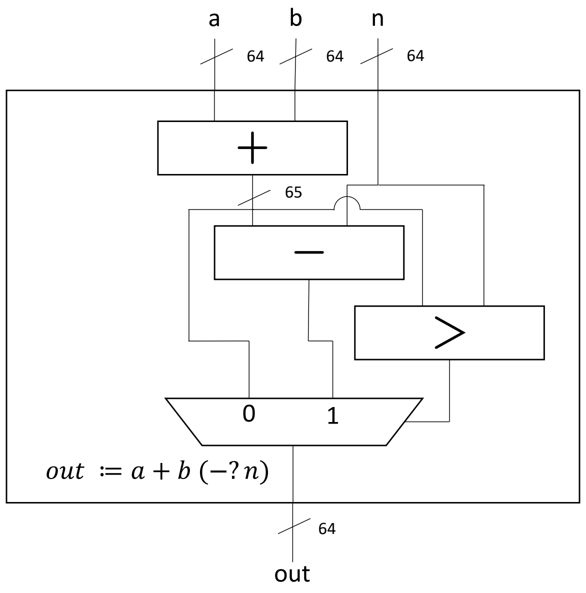

Figure 4-4: Addition Unit

4.2.3 Refresh

To ensure the computation unit can continually process inputs every cycle without pause, we add a refresh input. This input indicates that the current inputs are to start a new sum with a new moduli. The refresh input is passed into a shift register, copying its value across the many stages of computation in the multiplication unit. When the multiplication result is ready, the refresh register is checked.

If it is set, the c input (also passed through a shift register to get to this point) is passed into the addition stage. If not, the accumulator is passed into the addition stage. The other input to the addition stage is the product computed by the multiplication stage. Either way, the result of the addition stage is passed into the accumulator register. If the refresh input is set, a new coefficient moduli will also be used for the multiplication and addition.

The benefit of this scheme depends on the ultimate loop ordering. If different locations in the C array are accessed on every cycle, then the refresh input will always be true, and the accumulator register will be of little use. However, if the same location in the C array is accessed repeatedly, the accumulator register can be used to accumulate a sum for that location in C, only needing to provide a new value for the c input and read the output of the computation unit when the location in C is changed. This can reduce the need for reads and writes at the lowest level of the storage hierarchy, which may save a small amount of energy.

4.2.4 Synthesis

We used verilator [27] to simulate this modulo multiply-and-accumulate unit, and yosys [30] to synthesize it. We found the critical path of the computation unit to be 3442.00 ps. This is equivalent to a clock speed of 291MHz. This computation unit is slower than DRAM speeds. DDR5 can operate at speeds of 3.2GHz [25].

4.2.5 Pipelining

To improve the speed of the computation unit, the additions and multiplications can be highly pipelined into individual full-adder units. A full-adder unit is a hardware adder that takes in two 1 bit numbers and a 1 bit carry and outputs a 1 bit sum and a 1 bit carry. These 1-bit adders can be chained together to create larger adders, and those larger adders can be chained together to create multipliers.

Typically, adders and multipliers are not actually created by chaining together full adders linearly, but logarithmically to improve delay. No matter how they are chained together, though, a pipeline stage could be shortened to be the length of only a single full adder. An example of this type of pipeline for a linear adder is shown in Figure 4-5 and for a linear multiplier is shown in Figure 4-6. For an N-bit linear adder, this type of pipelining produces N stages. For an N-bit linear multiplier (which produces a $2N$-bit output), this type of pipelining produces $3N-4$ stages.

The comparators used in our design can also be pipelined in a similar way, with individual 1-bit comparator units being separated into pipeline stages, as shown in Figure 4-7.

Since our computation unit is made primarily of full adders (through the adders and multipliers), comparators, and multiplexers, it is a prime candidate for pipelining in this way. A single full adder, as synthesized through yosys is 153.58 ps. A single

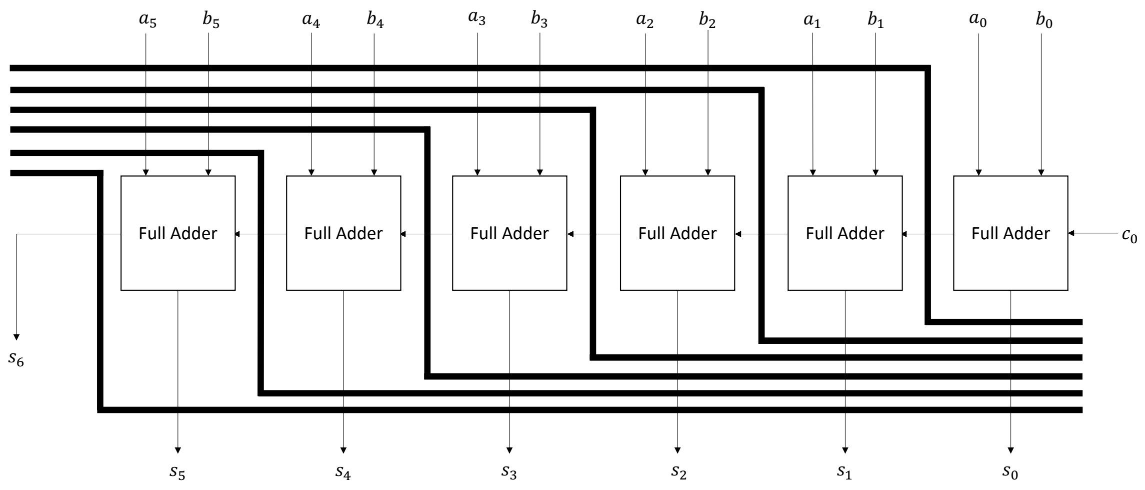

Figure 4-5: Pipelined Adder

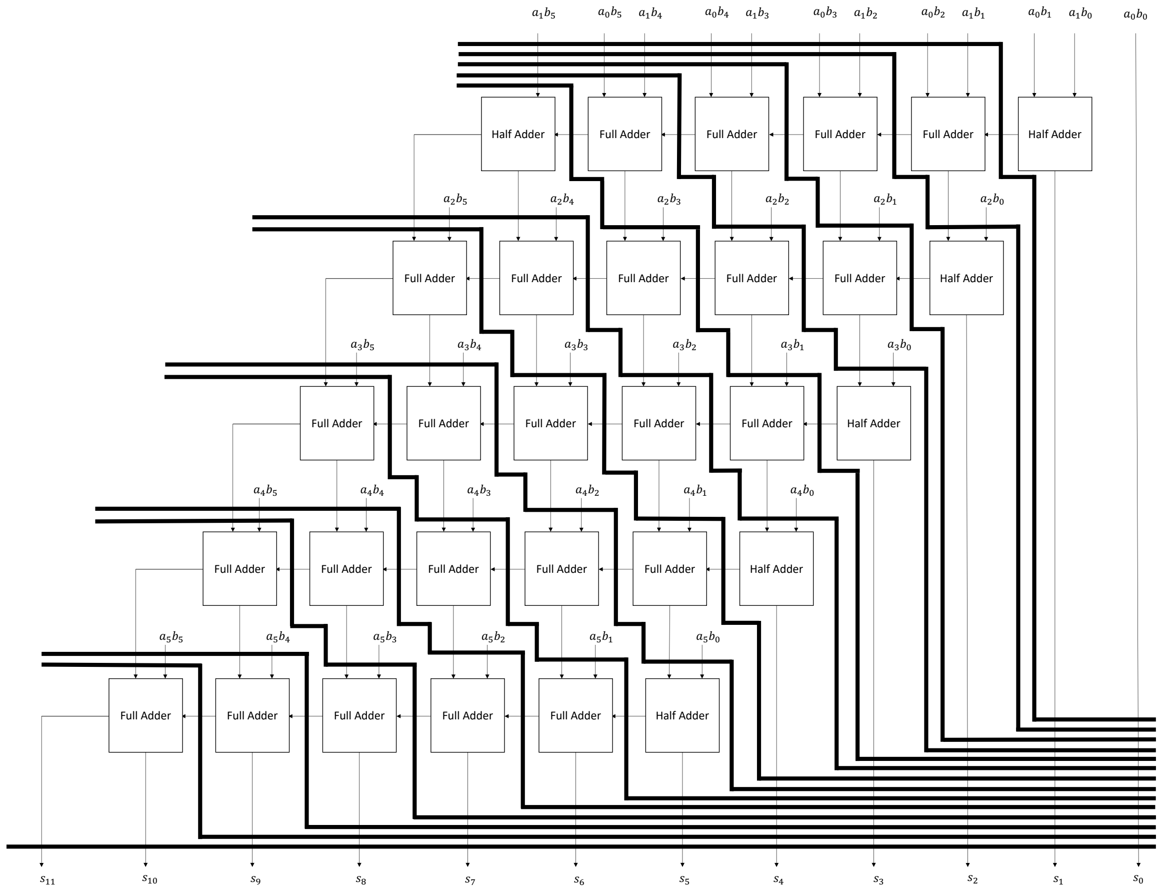

Figure 4-6: Pipelined Multiplier

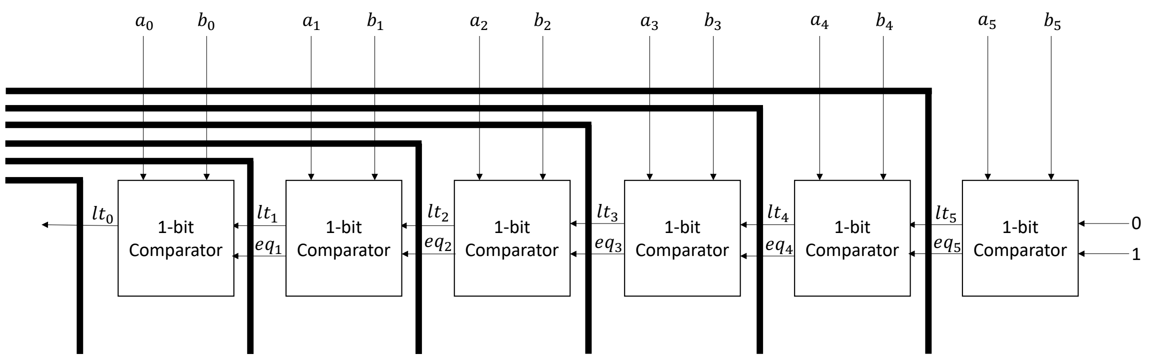

Figure 4-7: Pipelined Comparator

1-bit comparator unit is 100.32 ps. To be conservative, we will say that the entire computation unit can be pipelined in this manner to a clock period of 500ps.

This will greatly increase the pipeline depth. For instance, if using a linear multiplier, the 128-bit by 128-bit multiplication stage, which used to be a single cycle, will now take 380 (albeit, smaller) cycles. However, if the entire system - i.e. including the storage hierarchy - is also well pipelined, the latency of the computation unit should not matter as much as the clock period, which we have reduced through this pipelining exercise.

By reducing the clock period to 500ps, we allow the system to run at 2GHz.

4.3 Storage Hierarchy

The storage hierarchy is shown in Figure 4-8. The top level of the storage hierarchy is an NVME SSD. We make the assumption that it has an approximately 10GB/s internal bandwidth. This assumption is justified by using the SSD simulator MQSim [28]. We simulated a sequential read workload with average request size of 16KB using an SSD with 16 flash channels, with the assumption that the channel transfer rate has approximately doubled since the simulator's publication. This achieved external bandwidths of 1.9GB/s, which is similar to those seen in modern SSDs.

The next level of the storage hierarchy is the DRAM. SSDs typically contain on-chip DRAM, which is used in duties such as block mapping in the flash translation

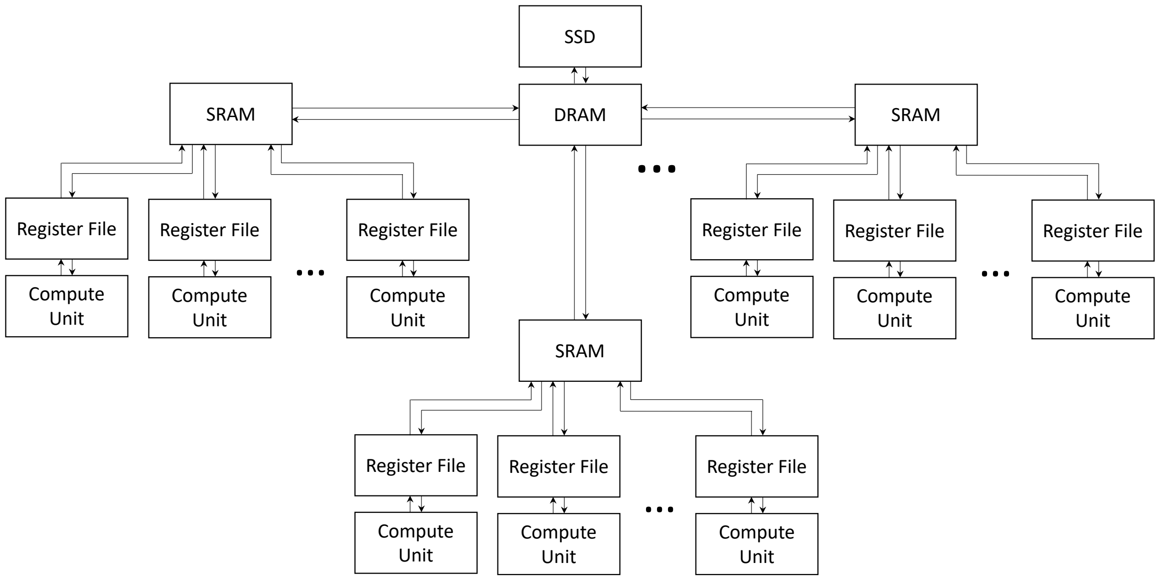

Figure 4-8: Storage Hierarchy

layer. Past research has proposed repurposing this DRAM for other uses, and having the host take over the mapping duties [5] [7]. Our design will take advantage of this, and assume access to a 4GB on-chip DRAM store. We assume that the DRAM will be DDR5.

The next level of the storage hierarchy is SRAM. The DRAM level may fan out to several SRAMs. Since the SRAM will expect fills of tiles from the DRAM level, the SRAM should be constructed to at least match the bandwidth of the DRAM level. DDR5 can serve four 32-bit words per cycle, or 128 bits [9]. The SRAMs are also required to fill tiles to the next level of storage, which is potentially many register files. A reasonable datawidth for the SRAM is 256 bits, as this exceeds the DRAM read bandwidth, and it can serve several 64-bit words per cycle to the the register files it fans out to, if those register files happen to require consecutive data.

The next level of the storage hierarchy is register files. The SRAM level may fan out to several register files. Register files are responsible for accepting tiles from the SRAM level, and accepting reads and writes from the computation units. To that end, the register file is provided with 3 read ports (to read values from elements of A, B, and C each cycle), and 2 write ports (one for fills from SRAM, and one for writes

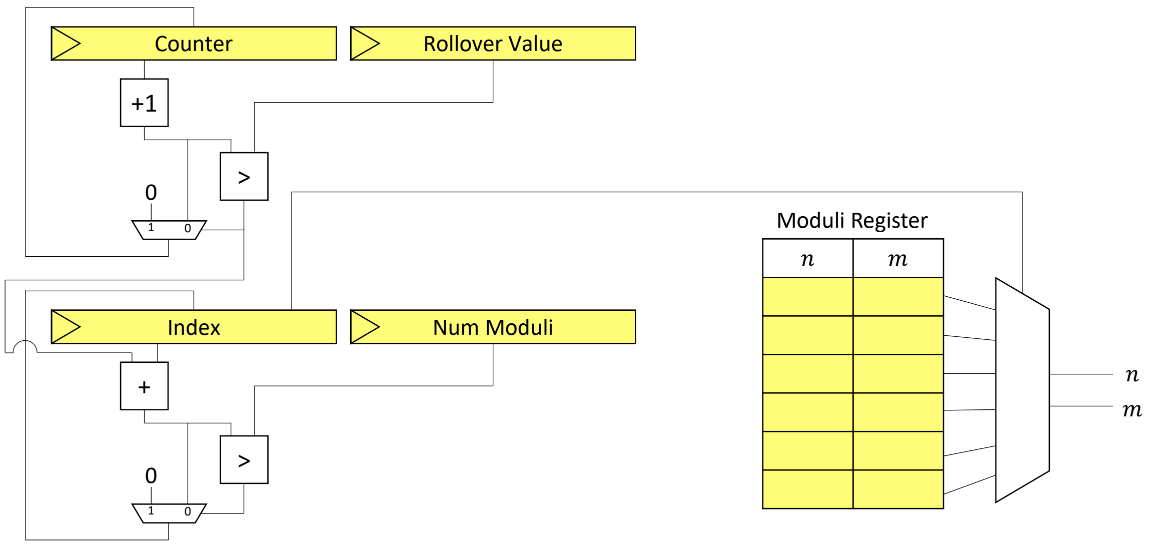

Figure 4-9: Moduli Selector Unit

from the computation unit back to C). Each register file is connected to exactly one computation unit, which it delivers data to.

4.4 Moduli

The moduli register and selector unit is shown in Figure 4-9. The moduli register contains the set of moduli available to the entire matrix calculation, both n and pre-computed m values. This is 192 bits of storage per moduli. Only a modest amount of storage would be required to support the vast majority of use cases.

Just as the storage hierarchy is responsible for delivering the data values a and b to the computation unit, the moduli selector unit is responsible for delivering the values n and m. This will follow an extremely predictable pattern based on the loop ordering and tiling. With any loop ordering or tiling, the moduli will be cycled through. How often moduli need to be switched depends on the exact loop ordering and tiling. Based on this pattern, the moduli selector unit need only consist of an index register to keep track of the current modulus, a counter register to determine when to rollover to the next modulus, and a multiplexer to select the current modulus values as output. The moduli selector unit will also need registers to keep track of the counter value that should initialize a rollover, and the number of coefficient moduli, both of which should be initialized at the beginning of a computation. The size of the counter register can be decided by the expected range of repeats of the same modulus given a specific loop ordering and tiling, once an optimal looping ordering is found in §6.

Chapter 5

Space Exploration

5.1 High-Level Strategy

Now that we have specified our computational problem and architecture, there are several variables left undefined. Parts of the architecture storage hierarchy are unspecified, and the way in which the einsum problem actually runs on that hierarchy is also not defined. In particular, the sizes and fanouts of units in the storage hierarchy, and the loop ordering and tiling of the einsum problem must be defined.

At a high level, we will define a range of reasonable values for the storage level sizes and fanouts. For each possible architecture, we will run the timeloop tool to obtain the best mapping (i.e. loop ordering and tiling). From the Timeloop tool, we obtain details about the area of the architecture, the energy required to run the mapping on the architecture, and the number of cycles it will take. Since PIR is primarily limited by speed, we will compare architectures by their cycle counts, and for those with the same or similar cycle counts, we will compare their area-energy products to see which architectures are best.

5.2 Space Definition

5.2.1 Architecture

For each of the unspecified architecture attributes, we swept through a range of potential values that it could take on. From the minimum to maximum value, we stepped exponentially, multiplying by two each time. We swept through the following values:

- Size of each SRAM: 32KB - 64MB
- Size of each Register File: 32 - 4096 Registers (64 bits each)
- Total number of SRAMs: 1 - 32
- Total number of Register Files/Compute Units: 1 - 1024

From this large search space, it was clear that the optimal number of SRAMs is 2, and the optimal number of Register Files/Compute Units is 2. We also noticed that the most optimal designs did not require much area, so expanded a new search space to include smaller SRAMs and register files. These decisions will be discussed further in §6. This refined search space is as follows:

- Size of each SRAM: 2KB - 64MB
- Size of each Register File: 4 - 4096 Registers (64 bits each)
- Total number of SRAMs: Fixed at 2
- Total number of Register Files/Compute Units: Fixed at 2

5.2.2 Mappings

For loop reorderings and tilings, there is a large space to explore. We used Timeloop to explore this space in a random way, stopping after finding 500 consecutive valid mappings which are suboptimal (i.e. the cycle count is higher than the current best mapping, or cycle count is equivalent with a higher energy). We found this gave reasonable results and increasing this number did not find much better mappings. Timeloop frequently finds invalid mappings - this happens when the mapping, for example, implies that more data must be stored in a storage unit than there is space available. These are skipped over and do not count towards this suboptimal total.

5.3 Implementation

Architecture space sweeps are implemented in timeloop's design-space tool [8]. This tool takes in the existing architecture definition and constructs new architectures for every possible combination of the architecture attributes you are sweeping over. For each architecture, the tool runs timeloop's normal mapper to find the best mapping. In order to use this tool for our use case, we had to add several features.

5.3.1 Memory Restriction

First, our architecture search space was at many times during this research quite large. The original design space tool constructed each architecture in memory before applying the timeloop mapper to each one in turn. We made a modification to construct these architectures on-the-fly so that only one is kept in memory at a time, allowing it to work for much larger architecture spaces.

5.3.2 Constraints

Timeloop's architecture allows specification of the total number of instances of each storage unit. Our architecture search space includes sweeping over the number of instances of both SRAMs and register files. This allows for absurd cases where the number of SRAMs is more than the number of register files, so the SRAMs cannot fanout to the register files.

To control for these cases, we added user-specified constraints into the design space tool and specified that the number of instances of SRAMs must be less than or equal to the number of instances of register files.

These constraints skip evaluating any architecture that violates them. For our purposes, these are invalid architectures, but these constraints can also be used for architectures that a user wishes not to explore but that are otherwise within the search space.

5.3.3 Disk

Timeloop natively supports the highest level of the storage hierarchy being an unlimited-size DRAM. With the large amount of data required in PIR, the highest level of storage in our application must be an SSD disk. We added a Disk component to Timeloop, to be available in the same unlimited capacity that DRAM was, and allowed DRAM to be limited in size.

We also added energy estimations for the new Disk component to Accelergy [31], from which Timeloop derives its energy and area calculations. The DRAM area estimation in Accelergy was set to 0 (since it was an unlimited DRAM with no real footprint), and the energy estimation did not depend on the size of the DRAM. We modified these to use the CACTI program [29] to estimate DRAM area and energy based on the specified size of DRAM and DRAM type (e.g. DDR5, DDR4, LPDDR, etc).

Chapter 6

Evaluation Results

After collecting and analyzing data from Timeloop, we found the best architecture to have two SRAMs of size 4KB, each of which feeds into a single register file of size 32. This optimal architecture is paired with an optimal mapping that iterates over the J and N indices at the register file level.

The OnionChopper accelerator performs very well on OnionPIR for large databases, with speeds close to disk bandwidth speeds and the majority of energy consumption deriving from necessary SSD accesses. Our accelerator presents an almost 300-fold improvement over speeds on a general-purpose processor. In addition, these benefits carry over to other PIR algorithms, with most differences in speed and energy consumption occurring at small database sizes.

6.1 Initial Architecture Sweep

We performed an initial architecture sweep over a large space of register file and SRAM sizes and number of instances. We primarily wanted to optimize for speed, so the first result we examined was the number of cycles taken by different architectures to compute the results.

As seen in Figure 6-1, the number of cycles taken by the different architectures is clustered mostly around 1.7×10^{10} and 1.3×10^{10}. Investigating further, the reason for this becomes clear: these two clusters are both defined by the bandwidth limits

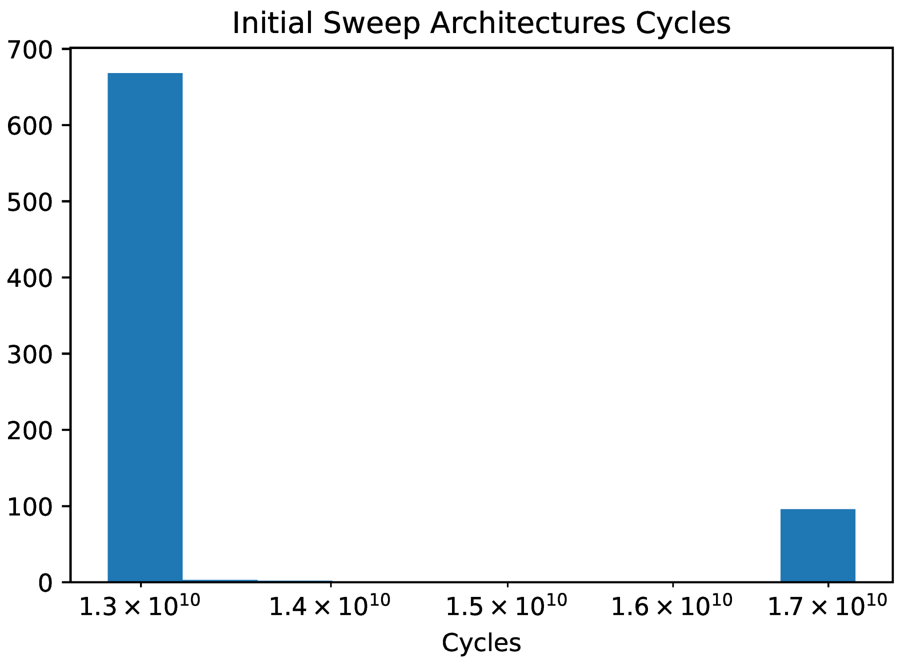

Figure 6-1: Initial Sweep Architecture Cycles

of a compute unit and the SSD respectively.

If one compute unit is operating by itself, the number of cycles it will take (minus any startup costs associated with the pipeline depth) is equal to the total number of computations required. Examining our einsum computation:

$$C_{m,n,p,q,c,d} = A_{m,j,p,k,c,d} \cdot B_{j,n,k,q,c,d}$$

A computation is needed for every value of m, n, p, q, j, k, c, and d. The bounds on these values for our chosen 64GB OnionPIR database are $M = 1$, $N = 2048$, $P = 2$, $Q = 1$, $J = 256$, $K = 2$, $C = 2$, and $D = 4096$. This brings the total number of computations to $M \cdot N \cdot P \cdot Q \cdot J \cdot K \cdot C \cdot D = 17,179,869,184$. This explains this cluster of architectures on the graph in Figure 6-1, as these are simply the architectures with only one compute unit.

As soon as there are multiple compute units, the architecture becomes capable of processing computations at multiple times this rate. However, it quickly becomes limited by another factor: the bandwidth of the SSD. Even with optimal loop tiling, the SSD must still deliver all of the data in the A, B, and C matrices to the rest of the storage hierarchy at least once.

The size of A is $M \cdot J \cdot P \cdot K \cdot C \cdot D = 8,388,608$ elements. The size of B is $J \cdot N \cdot K \cdot Q \cdot C \cdot D = 8,589,934,592$ elements. The size of C is $M \cdot N \cdot P \cdot Q \cdot C \cdot D = 33,554,432$ elements. The total size is $|A| + |B| + |C| = 8,631,877,632$ elements. Each element has 64 bits, or 8 bytes. The total size of all data that must be served is therefore $69,055,021,056$ bytes, or 64.3125 GB. The SSD can deliver 10 GB/s, so at a clock speed of 2GHz, this amount of data will take at least $12,862,500,500$ cycles. This explains the cluster of architectures around 1.3×10^{10} on the graph in Figure 6-1, as these are the architectures with multiple compute units that are limited instead by the bandwidth of the SSD.

In fact, any architecture with at least 2 compute units will be limited by the SSD, since upon splitting the computational work in half the amount of cycles necessary to devote to it is less than the amount of cycles required to bring in the data to process. Any additional compute units are purely wasteful. There are then truly only two types of storage hierarchies to consider: those with two SRAMs, feeding into one compute unit and register file each, or those with one SRAM that feeds into two compute units and register files.

We examined the data, comparing similar such architectures with the same register file sizes and *total* SRAM sizes. That is, for example, comparing an architecture with 64KB in one SRAM or split into two 32KB SRAMs. We found that the energy used by the SRAMs is smaller in the case where the SRAMs are split in half. Although the number of cycles required for the entire system remains the same, the number of cycles required for the SRAMs is higher when combined. This is because one SRAM cannot service any requests in parallel, whereas when split, both SRAMs can process requests at the same time (e.g. reads from their respective register file units). This results in the SRAMs spending a higher proportion of time idling rather than spending energy.

Based on this initial analysis, we concluded that the optimal design has 2 SRAMs and 2 Register Files/Compute Units. Taking an initial look at the area and energy

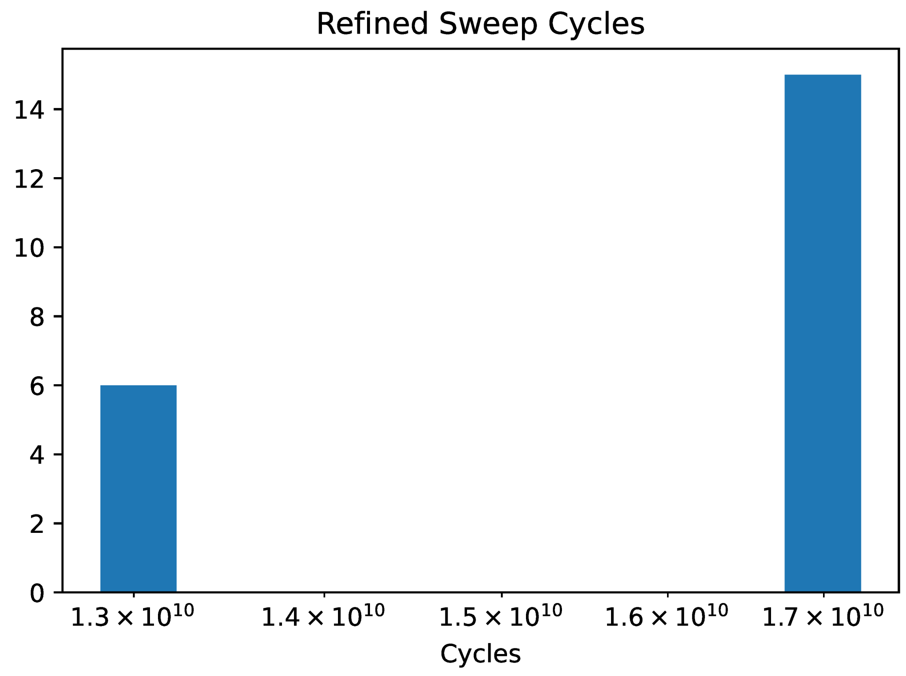

Figure 6-2: Refined Sweep Architecture Cycles

of the architectures, it was clear that even the most low-area architectures (caused by setting the sizes of the register files and SRAMs to low values) obtained some of the lowest energy values. Noting this, we refined our search space to include some even lower area options as we narrowed the scope to storage hierarchies containing 2 SRAMs and 2 Register Files.

6.2 Refined Architecture Sweep

Examining the data for only this refined sweep in Figure 6-2, we found that some architectures still had bottlenecked performance, despite being allotted two compute units. This is likely due to small register files leading to poor reuse, necessitating many repeated SRAM accesses. For the same reason, these architectures also consumed more energy than typical of their peers of the same size, as revealed in Figure 6-3.

Removing these from the set of considered architectures, we find a more consistent trend between energy and area in Figure 6-4.

In general, the architectures with larger areas have larger area impacts. Based on

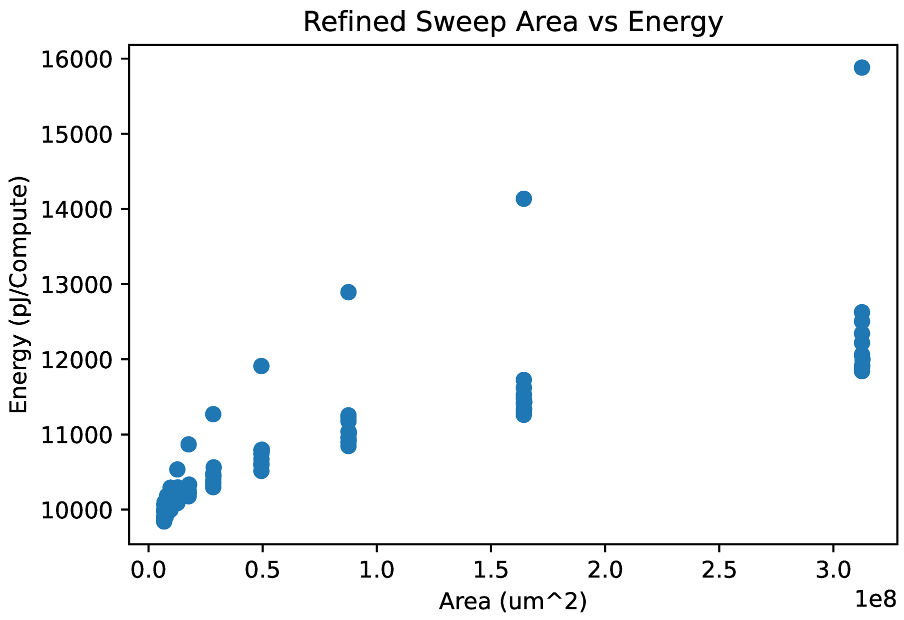

Figure 6-3: Refined Sweep Area vs. Energy

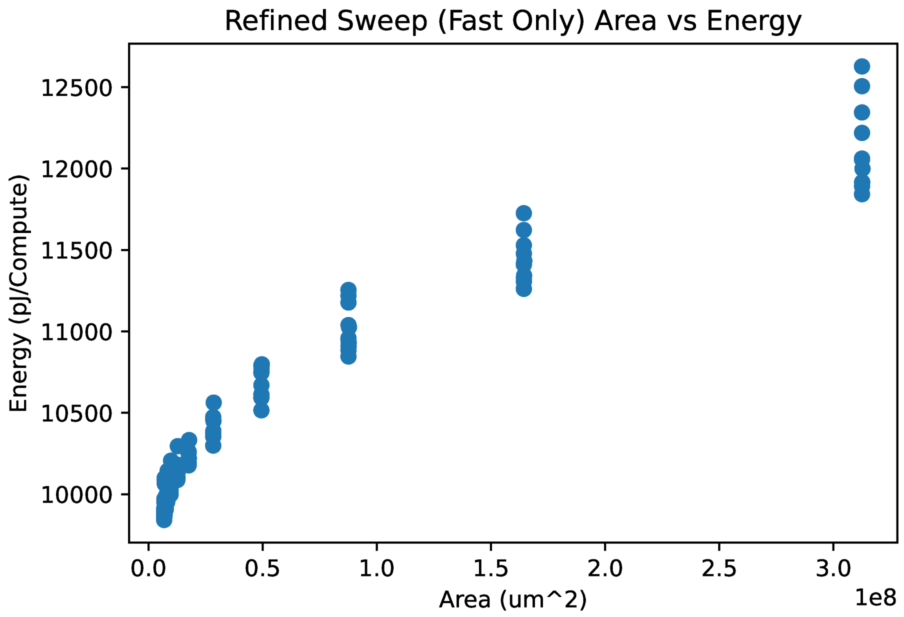

Figure 6-4: Refined Sweep (Fast Only) Area vs. Energy

the trends observed, this means that by making our architectures larger (primarily by increasing the size of the SRAM), we are for the most part not reducing energy consumption caused by inefficiencies in data reuse, only increasing energy due to the larger energy consumption per access. This means that even at the smallest SRAM sizes tested, we are adequately providing a buffer for reuse of data that prevents unnecessary DRAM accesses. Further decreasing SRAM size is not reasonable, as at the smallest size it is already only 2KB with 64 rows. Fewer than 64 rows is not supported by the Accelergy plugin used to compute the energy estimates for the SRAM.

Small changes to the size of the register file do not impact the area much but may impact the reuse patterns. For the smallest SRAM sizes, we investigate this impact. Data is plotted in Figure 6-5.

We find there is a general trend of larger energies with larger register files. We also find that there is a slight dip in the energy consumption around a register file depth of 16 or 32, where both smaller or larger register files consume more energy. Smaller register files likely have poor reuse, leading to higher energy consumption. Larger register files have more energy consumption due to their larger footprint and multiplexers. Note that any fewer than 8 registers, and the cycle count becomes too high, so these architectures have been eliminated.

For the overall area-energy product, the optimal architecture turns out to have a small SRAM size (32768 bits, or 4KB), and a 32-register register file. This is consistent with the analysis above, a 4KB SRAM is one of the smallest tested and 32 registers is at a low point in the graphs in Figure 6-5.

The overall optimal architecture is shown in Figure 6-6.

6.3 Mapping

The Timeloop tool, in addition to outputting statistics that were useful in determining the optimal architecture, also determined the optimal mapping on which to run that architecture. In fact, the number of cycles and the energy the computation took to

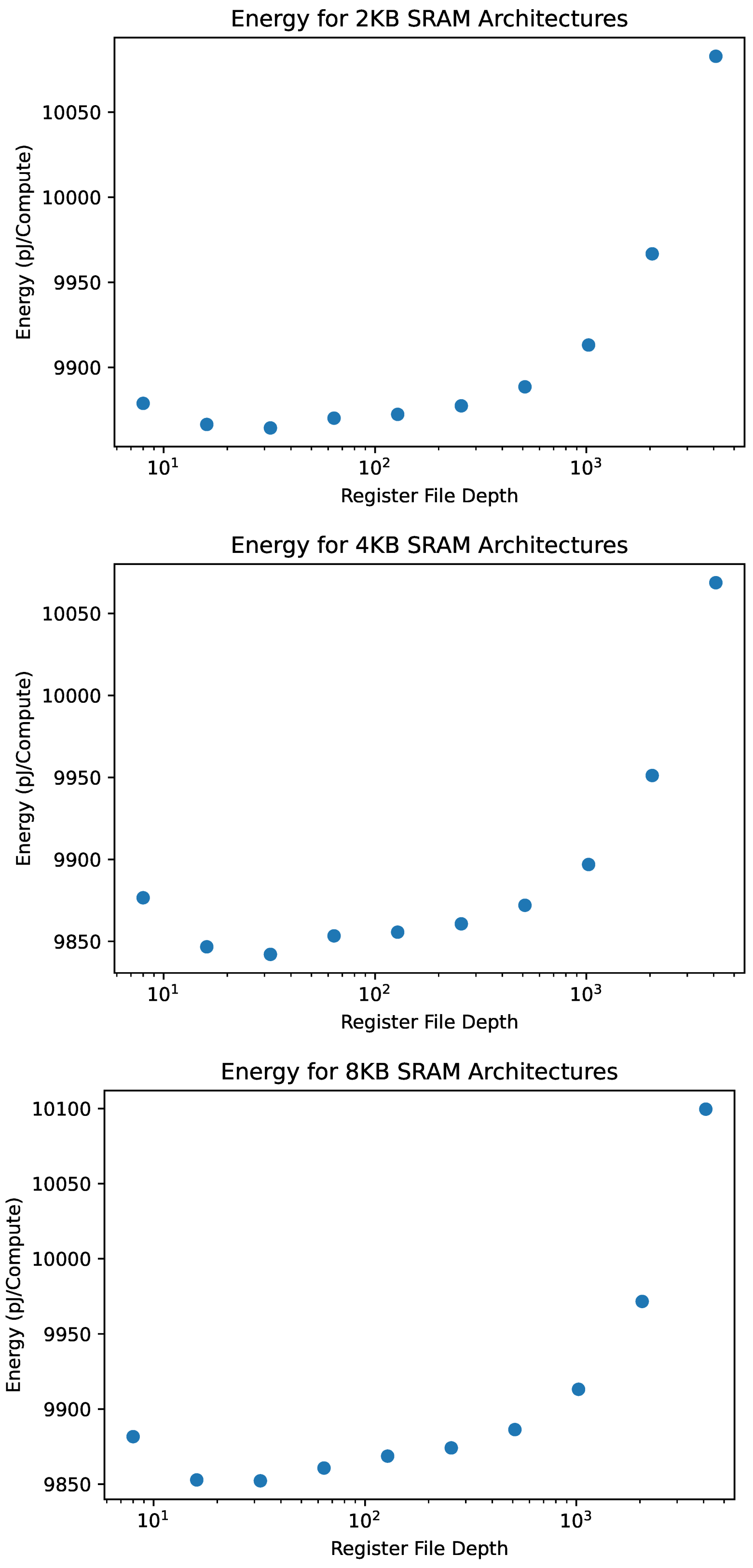

Figure 6-5: Register File Depth vs Energy

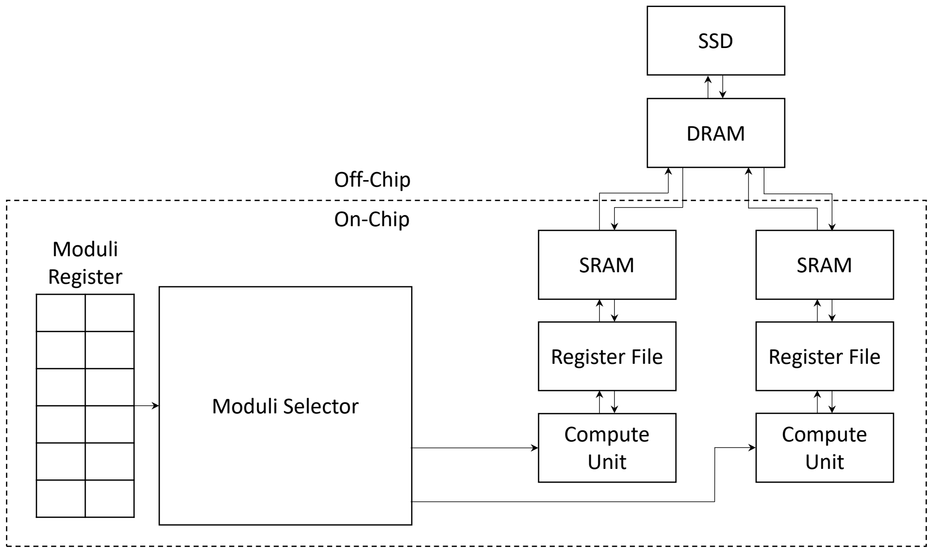

Figure 6-6: Overall Optimal Architecture

run on a given architecture are dependent on the loop ordering and tiling (or mapping) of the problem on the architecture.

With 8 different loop variables, and large loop bounds, there was a very large search space for tilings and loop orderings, and it was not reasonable to explore the entire space. Thus, there was a certain amount of randomness in whether or not the optimal mapping found by Timeloop was truly the optimal mapping.

We tried adding the additional constraints to Timeloop to perform either an "output-stationary" or "input-stationary" mapping. For our architecture, this means that at the register-file level, the mapping repeatedly accesses either the same output (or input) element, and only changes the other matrix elements involved in computations. These types of mappings tend to be more efficient, so may lead Timeloop to find a better answer by exploring a smaller space.

Timeloop's answers in this smaller space were mostly on-par with its answers in the full mapping space, and it did not find any more efficient mappings.

The best mapping was found in the original space, which we present here. The number of elements present from each array at each level is listed at the top of the tiling loops for that level.

```
SSD [ A: 8388608 B: 8589934592 C: 33554432 ]
----------------------------------------------------------
| for D in [0:32)

DRAM [ A: 262144 B: 268435456 C: 1048576 ]
----------------------------------------------------------
|   for C in [0:2)
|     for K in [0:2)
|       for D in [0:128)
|         for J in [0:8)
|           for N in [0:512)
|             for J in [0:2) (Spatial)

SRAM [ A: 32 B: 64 C: 8 ]
----------------------------------------------------------
|               for J in [0:2)
|                 for N in [0:2)
|                   for P in [0:2)

Register File [ A: 8 B: 16 C: 2 ]
----------------------------------------------------------
|                     for J in [0:8)
|                       for N in [0:2)
```

As another reminder of the problem:

$$C_{m,n,p,q,c,d} = A_{m,j,p,k,c,d} \cdot B_{j,n,k,q,c,d}$$

Tiling of D (the coefficients of the polynomials) is concentrated at the DRAM level and SSD level. Tiling of N (the columns of B and C) is also concentrated at the

DRAM level. Tiling of J (the columns of A and rows of B) occurs across the storage hierarchy. Tiling of the coefficient moduli occurs at the outer loop of the DRAM level, separating the two polynomials in each pair.

For this particular mapping, the coefficient moduli change very rarely. Thus, for the moduli selector, a large counter would be required to control the rollover of the moduli index. The counter needs to be able to count up to on the order of the number of computations. A reasonable assumption is that this number should fit in a 64 bit integer, so we should allocate 64 bits for the coefficient moduli counter.

6.4 Evaluation Statistics

For a 64GB database, our design took 12,833,318,209 cycles. At 2GHz, this is equivalent to 6.4 seconds. This time will not be affected much by latencies in the systems; even though timeloop does not account for the startup latencies, even very deep pipelines will be dwarfed by the total number of cycles taken. Running on a MacBook Pro laptop with a 2.9 GHz 6-Core Intel Core i9 and 32 GB memory in an Ubuntu virtual machine allocated 2 processors with 8GB of memory, the same calculation takes 29 minutes, an improvement of 271.9×.

Our design used 169.09 J. This means that it was consuming 26.35W during the computation, 33.43% of which was used by the compute units and 64.99% of which is used by the SSD.

Our design is 6.87mm^2 in area. The compute units contribute 99.54% of this area. Most of the remaining area is taken up by the SRAMs, with a negligible amount from the register files. The DRAM and SSD are not counted in this total.

6.5 Different Database Sizes

We evaluated the OnionChopper architecture on various database sizes for the key bottleneck operation in the OnionPIR algorithm. This is plotted in Figure 6-7 and 6-8. We found that it maintained a high speed on all database sizes and in particular,

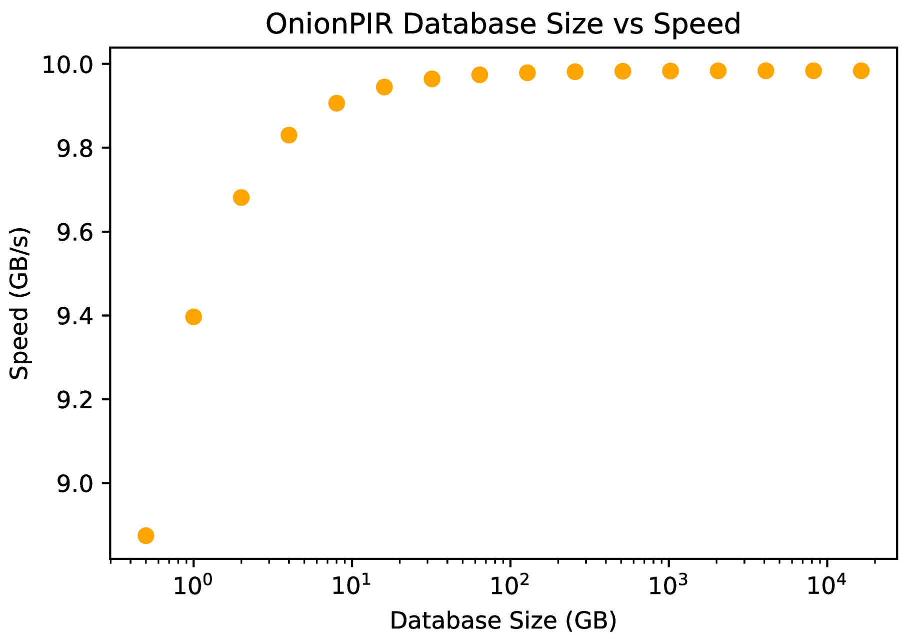

Figure 6-7: Accelerator Speed vs Database Size for OnionPIR

speeds are essentially equivalent to the SSD bandwidth once the database size is greater than the DRAM size. Speeds are worse for smaller database sizes, but even for the smallest size tested (512 MB), the speed was still 89% of the maximum speed on this architecture. OnionChopper maintains its advantage over general-purpose processor speeds at all database sizes tested.

We also found similar trends for the energy consumed per gigabyte, plotted in Figure 6-9. It is lowest for larger database sizes, and approximately 8% worse on the smallest database size tested. Power consumption, plotted in Figure 6-10, is lower for smaller database sizes and levels out as size increases. This is because the computations are slower on smaller database sizes, so energy is used less quickly.

6.6 Different PIR Algorithms

We also evaluated our OnionChopper architecture on the main operations in other PIR algorithms using Timeloop. Similar trends with respect to database size were found. SealPIR, SimplePIR, and FastPIR also approach SSD bandwidth speeds for large databases on our architecture. This is demonstrated in Figure 6-11.

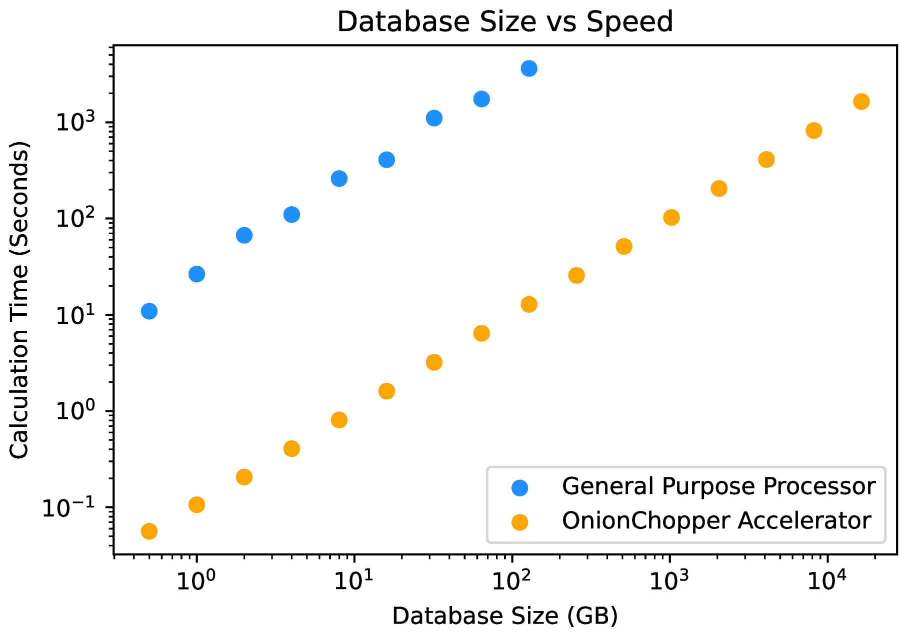

Figure 6-8: OnionPIR Speed on a General-Purpose Processor vs Accelerator

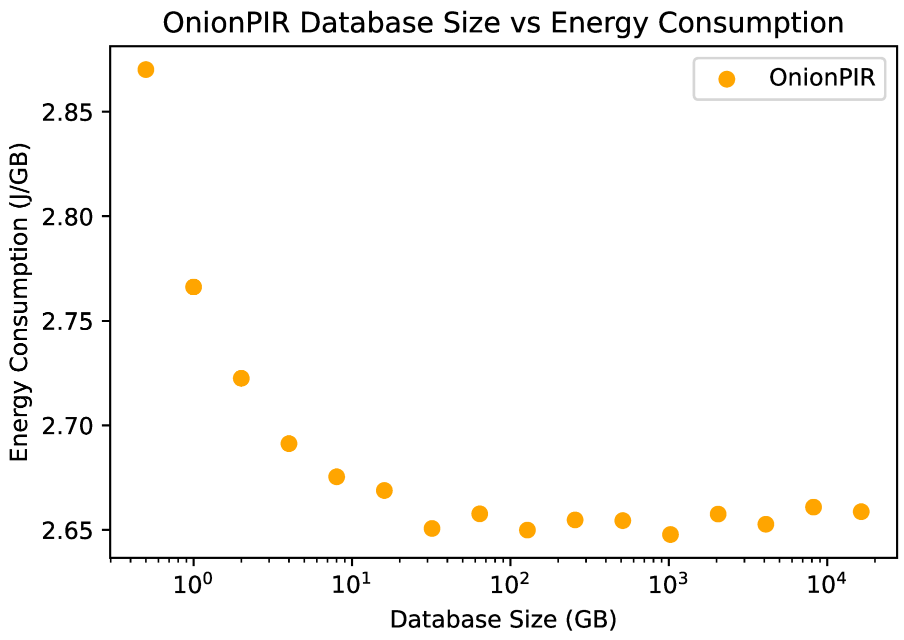

Figure 6-9: Accelerator Energy Consumption vs Database Size for OnionPIR

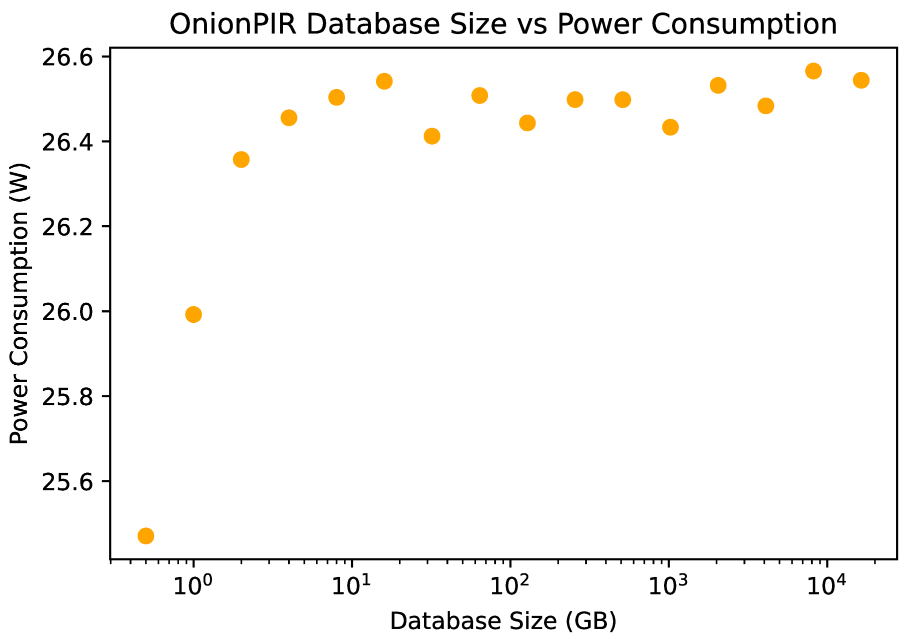

Figure 6-10: Accelerator Power Consumption vs Database Size for OnionPIR

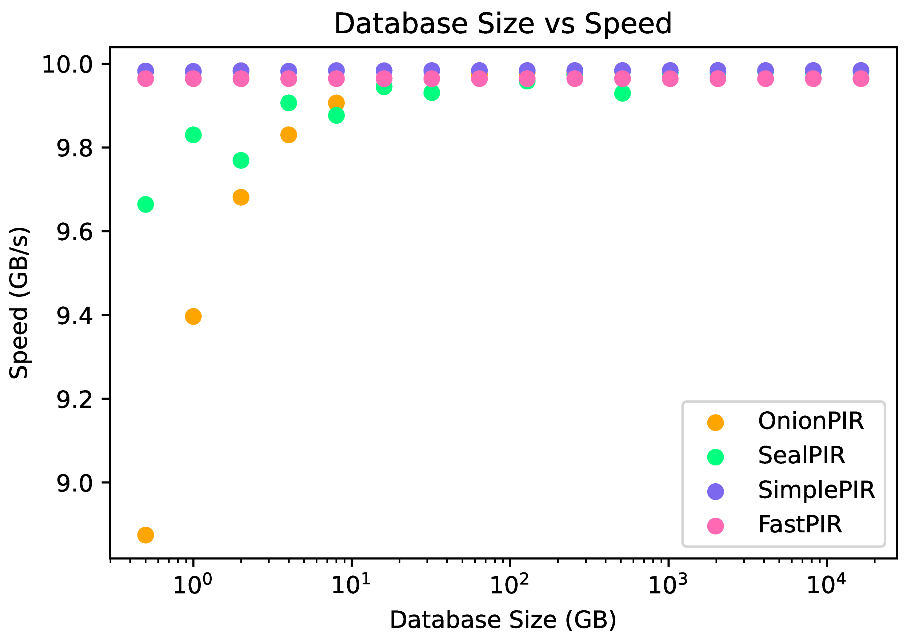

Figure 6-11: Database Size vs Speed Comparison

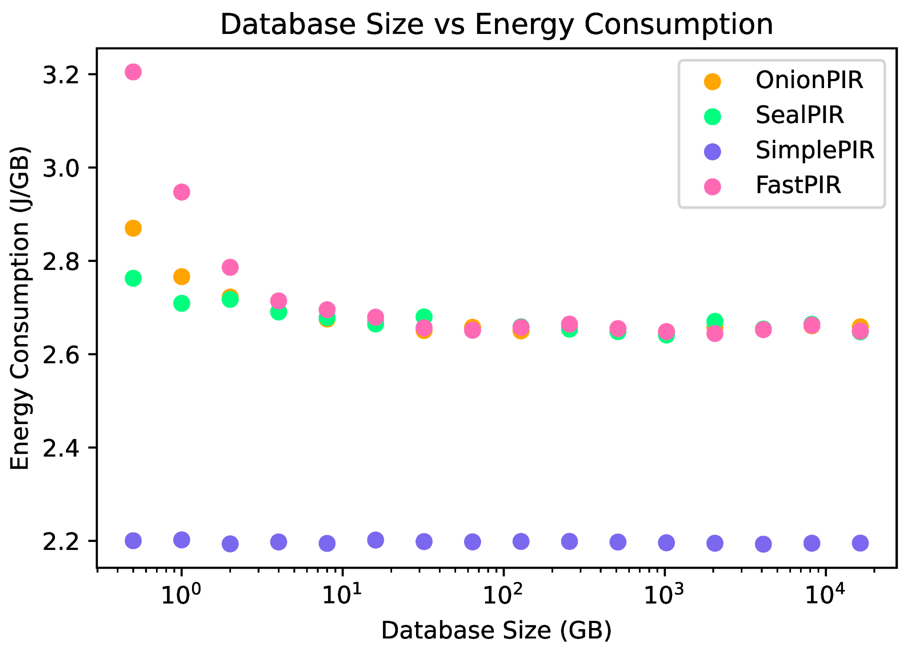

Figure 6-12: Database Size vs Energy Comparison

With regards to energy consumption, our architecture performs well on all of the PIR algorithms tested. This can be seen in the plots in Figure 6-13 and Figure 6-12. SimplePIR uses the most dissimilar algorithm to the other three, requiring fewer overall computations per gigabyte of the database, and thus is able to achieve a lower energy footprint by reducing the energy requirement of the compute units. The other three achieve a similar performance on large database sizes, with FastPIR taking more energy than the others on smaller database sizes likely due to the more sizable differences in its algorithm and OnionPIR compared to SealPIR.

These results prove that our architecture can be used for other PIR algorithms on large databases without sacrificing speed or much in the way of energy consumption.

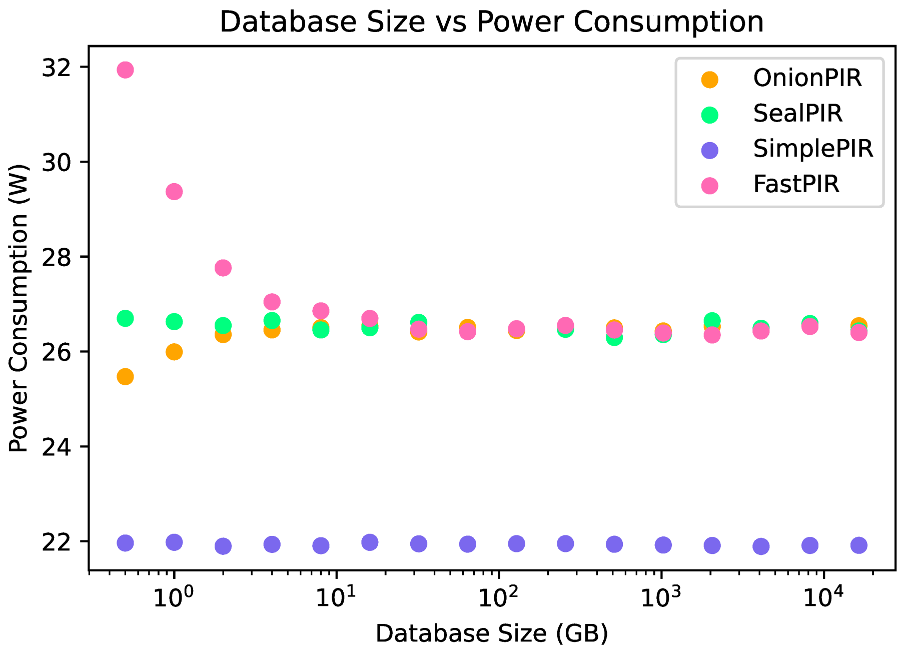

Figure 6-13: Database Size vs Power Comparison

Chapter 7

Limitations and Future Work

There are several factors that our work does not take into account. One such factor is the modeling of the Disk level of the storage hierarchy. For the DRAM and SRAM levels, Timeloop considers the total number of accesses to these storage units and assumes that the number of cycles these accesses will take is equal to the number of accesses. Then, if the bandwidth of these storage units does not permit this high of an access rate, the number of cycles is appropriately increased. Timeloop does not take into account whether the accesses are sequentially or randomly distributed. We extended Timeloop with a Disk level and used the same modelling. For Disk, the difference between sequential and random accesses can have a significant effect on delay and breaks the assumption that these accesses can be served with the same bandwidth. This is an area for future research to determine if this has an effect on the optimal tiling of data from the SSD level, or if it degrades the modelled performance of the accelerator.

Our modeling of the energy consumption from disk accesses is also an overestimate. We used data from the power consumption of NVME SSDs when sequentially reading and writing [16] [12]. This overestimates the energy consumption by including energy consumed by the entire system. However, since most of the various architectures and mappings explored used the same number of SSD accesses, this energy did not need to be overly precise for our comparison purposes.

Timeloop assumes our storage hierarchy and compute units are pipelined and double-buffered or use buffets [24]. To achieve the speeds suggested by the results in §6.4, these double buffers or buffets would need to be implemented in order to guarantee that all levels of the storage hierarchy can operate in parallel. If double buffering is used, this would use additional area and some additional energy as storage units would need to be duplicated so that one can be independently filled with the next tile as the other is drawing from the current tile.

The loop tilings and reorderings present a very large search space, due to the large number of loop dimensions (8), and large number of factorings of each dimension (leading to a large number of possible tilings). It was not practical to search through the entire space for every architecture candidate, or even for a single architecture. Our search may therefore have likely not identified the single best mapping in the search space for each architecture. This also means that the best architecture identified by our search may not have truly been the best. Based on our analysis of trends in §6.1 and §6.2, it is likely that at least a similar architecture in the space is the true optimum according to our criteria. However, further research may reveal a better architecture.

We designed our architecture based on a 10GB/s SSD. Further research is required to determine how the speed of the SSD affects the best architecture design and its throughput.

We optimized primarily for OnionPIR on a 64GB database. As seen in §6.5 and §6.6, this extends well to other database sizes and other PIR algorithms. However, because of our optimization approach, the speed and energy consumption does get worse by a small amount on small database sizes. Further research is required to design a system that works just as well on small and large database sizes.

An area for future work may be in exploring changes to the design of the accelerator for a protocol which batches several client requests and processes them at the same time, thus requiring fewer overall accesses to the database. This would involve matrix multiplications, so our accelerator is likely already a good fit, but future research may reveal different optimizations specific to the new dimension sizes and reuse patterns involved.

Our modeling of the computation unit in Accelergy assumes full pipelining of all multipliers, adders, and comparators in each stage. We do not consider the interaction between each stage, which leads to many extra registers between stages where computations could be started earlier. This increases our energy and area estimates. We may also be overly-aggressive with our pipelining in general. It may be possible to meet the 2GHz target speed without pipelining each full adder and comparator unit down to a single stage, which could reduce the energy and area requirements. Since the compute units comprise a reasonable amount of the energy consumption and the vast majority of the area requirements in our system, future research could focus on this area for improvement.

We did not implement the fine details of this accelerator, nor test it in hardware. Details such as the setup of the moduli registers and initialization of a computation require some hardware components and software interaction. The impact of these operations was not modeled as part of the latency of computation, although it should be mostly constant regardless of the size of the database. Since our accelerator has not been physically implemented, we cannot be sure it behaves exactly as modelled.

Chapter 8

Conclusion

We designed OnionChopper, a near-storage accelerator for the OnionPIR protocol. Our accelerator consists of a storage hierarchy and compute units which together stream data from an SSD disk to do arithmetic calculations modulo various integers. For the main bottleneck operation in this protocol, our accelerator improves the speed drastically, allowing the operation to run at speeds equivalent to the SSD bandwidth. OnionChopper is extensible to any large database size, as well as other PIR protocols, all achieving similar speeds. Our accelerator is also quite energy efficient, with a majority of the energy usage coming from the necessary SSD accesses, and the accelerator itself eliminating the need for any repeated SSD accesses to the same element. During computation on a 64GB database, 26.45W is consumed. Our accelerator is small, fitting within 6.87mm^2.

Realtime applications such as streaming or voice calling require a low server response time and a low server response size in order to work effectively. OnionPIR provides a small response size, and OnionChopper can greatly improve the response time to make such realtime applications feasible. Since OnionChopper is also extensible to other schemes, new PIR protocols based on the same concepts of homomorphic encryption that improve upon request size, response size, response time, or other aspects of the protocol may also be able to take advantage of this accelerator to gain an automatic boost in response time.

Appendix A

Example Timeloop Mapping

The following is an example of a Timeloop Mapping for a matrix multiplication problem:

$$C_{i,j} = A_{i,k} \cdot B_{k,j}$$

Suppose we instantiate this with concrete bounds for i, j, k, denoted $i = 16$, $j = 8$, $k = 4$. Also suppose we have an architecture with a DRAM main memory, an SRAM global buffer, and 4 register files.

A possible mapping could be as follows:

```
mapping:
  - target: MainMemory
    type: temporal
    factors: I=2 J=1 K=2
    permutation: JKI

  - target: GlobalBuffer
    type: temporal
    factors: I=4 J=2 K=1
    permutation: JKI
```

```
- target: GlobalBuffer
  type: spatial
  factors: I=1 J=4 K=1
  permutation: JKI

- target: RegisterFile
  type: temporal
  factors: I=2 J=1 K=2
  permutation: JKI
```

This mapping says that main memory will deliver 4 tiles to the SRAM, in the $i \rightarrow k$ order, with factors of 2 for both i and k. For each tile the SRAM gets, the SRAM will deliver 8 tiles to the register files, in the $i \rightarrow j$ order, with a factor of 4 for i and 2 for j. There are multiple register files, so each tile that gets delivered can further be divided into four tiles, in this case along the j dimension. Finally, for each tile it gets, the register file will deliver 4 tiles to the computation unit, with a factor of 2 in the i dimension and 2 in the k dimension, in the $i \rightarrow k$ order. Each of these tiles will be enough to perform one single computation - i.e. one element from A, one element from B, and one element from C.

Bibliography

[1] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 313–329. USENIX Association, July 2021.

[2] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–Computation trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1811–1828. USENIX Association, August 2021.

[3] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979, 2018.

[4] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings on Advances in Cryptology—CRYPTO '86*, page 311–323, Berlin, Heidelberg, 1987. Springer-Verlag.

[5] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The linux Open-Channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, February 2017. USENIX Association.

[6] Callgrind: A call-graph generating cache and branch prediction profiler, 2022.

[7] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. Lightstore: Software-defined network-attached key-value drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 939–953, New York, NY, USA, 2019. Association for Computing Machinery.

[8] NVIDIA Corporation. Timeloop github repository. `https://github.com/NVlabs/timeloop`, 2019.

[9] Ddr5 sdram. `https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr5/ddr5_sdram_core.pdf?rev=3210029bd0f44952852e9f7134d83315`, 2020.

[10] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. https://eprint.iacr.org/2012/144.

[11] Fio - flexible i/o tester rev. 3.33. https://fio.readthedocs.io/en/latest/fio_doc.html, 2017.

[12] Firecuda 510 ssd datasheet. https://www.seagate.com/content/dam/seagate/migrated-assets/www-content/datasheets/pdfs/firecuda-510-ssd-DS1999-5-2101US-en_US.pdf, Jan 2021.

[13] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1591–1601, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. Cryptology ePrint Archive, Paper 2015/489, 2015. https://eprint.iacr.org/2015/489.

[15] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022. https://eprint.iacr.org/2022/949.

[16] Jacob Hicks. How many watts does an ssd use? https://devicetests.com/how-many-watts-does-an-ssd-use, Dec 2022.

[17] Iostat(1) - linux man page. https://linux.die.net/man/1/iostat.

[18] Kim Laine. Simple encrypted arithmetic library 2.3.1. https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf, Nov 2017.

[19] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. Inspire: In-storage private information retrieval via protocol and architecture co-design. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 102–115, New York, NY, USA, 2022. Association for Computing Machinery.

[20] Mmap(2) - linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html, Mar 2021.

[21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. Cryptology ePrint Archive, Paper 2021/1081, 2021. https://eprint.iacr.org/2021/1081.

[22] Onionpir. https://github.com/mhmughees/Onion-PIR, 2022.

[23] Angshuman Parashar, Jenny Huang, Joel Emer, Nellie Wu, Michael Gilbert, Po-An Tsai, and Tanner Andrulis. Overview.

[24] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[25] Scott Schlachter and Brian Drake. Introducing micron® ddr5 sdram: More than a generational update. `https://www.micron.com/-/media/client/global/documents/products/white-paper/ddr5_more_than_a_generational_update_wp.pdf?la=en`, 2019.

[26] Microsoft SEAL (release 3.5). `https://github.com/Microsoft/SEAL`, April 2020. Microsoft Research, Redmond, WA.

[27] Wilson Snyder. Verilator user's guide. `https://verilator.org/guide/latest/`, 2022.

[28] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A framework for enabling realistic studies of modern Multi-Queue SSD devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, Oakland, CA, February 2018. USENIX Association.

[29] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. Cacti 5.1. `https://www.hpl.hp.com/techreports/2008/HPL-2008-20.pdf`, Apr 2008.

[30] Claire Wolf. Yosys open synthesis suite. `https://yosyshq.net/yosys/`.

[31] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.