Designing a Programmable Hardware Accelerator for Fully Homomorphic Encryption

by

Axel S. Feldmann

B.S. in Computer Science
Carnegie Mellon University, 2019

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

© Massachusetts Institute of Technology 2021. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by . . . .
Daniel Sanchez
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Designing a Programmable Hardware Accelerator for Fully Homomorphic Encryption

by

Axel S. Feldmann

Submitted to the Department of
Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Fully Homomorphic Encryption (FHE) allows computing on encrypted data, enabling secure offloading of computation to untrusted servers. Though it provides ideal security, FHE is expensive when executed in software, 4 to 5 orders of magnitude slower than computing on unencrypted data. These overheads are a major barrier to FHE's widespread adoption.

We present F1, the first FHE accelerator that is programmable, i.e., capable of executing full FHE programs. F1 builds on an in-depth architectural analysis of the characteristics of FHE computations that reveals acceleration opportunities. F1 is a wide-vector processor with novel functional units deeply specialized to FHE primitives, such as modular arithmetic, number-theoretic transforms, and structured permutations.

Due to the static nature of FHE computations, F1 uses an exposed ISA, requiring novel compilation techniques to statically schedule all compute and data movement. We design a compiler that efficiently maps FHE programs onto F1 hardware and maximizes reuse of on-chip data, helping to reduce data movement bottlenecks. The compiler leverages F1's explicitly managed scratchpad to decouple computation from data movement, a necessary ingredient in achieving high performance given the large size of FHE operands.

We evaluate F1 using cycle-accurate simulation and RTL synthesis. F1 is the first system to accelerate complete FHE programs, and outperforms state-of-the-art software implementations by gmean 6,500× and by up to 17,000×. These speedups counter most of FHE's overheads and enable new applications, like real-time private deep learning in the cloud.

Thesis Supervisor: Daniel Sanchez
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This work was conducted in collaboration with Nikola Samardzic, Alex Krastev, Nicholas Genise, Prof. Srini Devadas, Karim Eldefrawy, Prof. Ron Dreslinski, Prof. Christopher Peikert, and my research advisor Prof. Daniel Sanchez. Much of this thesis is adapted from a joinly written paper. This work would not have been possible without all of their contributions, and I have learned a lot working with them. Fully Homomorphic Encryption is not a very accessible research topic, and I am grateful to my collaborators for always taking the time to explain complex mathematical notions in simple terms that I can understand.

I am especially grateful to my advisor, Prof. Daniel Sanchez. It is very easy to be impressed by Daniel's technical ability and encyclopedic knowledge of computer architecture. However, I am most thankful for Daniel's infinite patience and constant encouragement. Working in isolation during a global pandemic has been tough, but Daniel's consistent support has made a difficult situation much easier. Whenever I have felt lost, Daniel has always had time to meet with me and discuss whatever is on my mind, be it high level research directions or tiny specific software bugs.

I am also extremely grateful to my research partner, Nikola Samardzic. Many times, when I have felt very stuck on a particular problem, Nikola has calmly stepped up and engineered a brilliant solution. Though I have never actually met Nikola in person, I feel that working on this project has brought us together, and I consider Nikola to be a dear friend.

I would also like to thank all the members of our research group. My officemates, Quan Nguyen, Yifan Yang, and Victor Ying have always been willing to help me out with both technical and non-technical problems. On top of that, they have always put up with my jokes and unrelated discussion topics. My other groupmates, Maleen Abeydeera, Prof. Joel Emer, Hyun Ryong (Ryan) Lee, and Guowei Zhang have always provided me with fascinating research discussions. I have learned so much from all of them, and our group hangouts have been a great source of joy during a relatively bleak time period.

I would like to thank my academic advisor, Prof. Vivienne Sze, who has always been supportive and given me good advice on classes and research.

I would also like to thank my undergraduate-research advisor at CMU, Prof. Nathan Beckmann, for getting me started in computer architecture research and always giving me great advice. Without him, I would not be where I am today.

Last but certainly not least, I would like to thank my parents. My parents have always provided me with unconditional support in every endeavour. They have always encouraged me to seek to better myself, take risks, and pursue my goals. And when I fail, they have always been there to catch me. Anything I have achieved in my life so far would not have been possible without their love and sacrifices.

Contents

1 Introduction **9**

1.1 Contributions 12

2 Background **13**

2.1 FHE programming model and operations 14

2.2 BGV implementation overview 15

2.3 Algorithmic insights and optimizations 18

2.4 Architectural analysis of FHE 19

2.5 FHE schemes other than BGV 22

3 F1 Architecture **23**

3.1 Vector processing with specialized functional units 24

3.2 Compute clusters 24

3.3 Memory system 25

3.4 Static scheduling 25

3.5 Distributed control 26

3.6 Register file (RF) design 26

3.7 Functional Units 27

3.7.1 Automorphism unit 27

3.7.2 Four-step NTT unit 30

3.7.3 Optimized modular multiplier 31

4 Scheduling Data and Computation **33**

4.1 Comparison with prior work 34

4.2 Translating the program to a dataflow graph 35

4.3 Compiling homomorphic operations . . . 35
4.4 Scheduling data transfers . . . 38
4.5 Cycle-level scheduling . . . 39
4.6 Limitations . . . 40

5 Results **41**
5.1 Experimental Methodology . . . 41
5.1.1 Modeled system . . . 41
5.1.2 Benchmarks . . . 42
5.1.3 Baseline systems . . . 44
5.2 Performance . . . 44
5.2.1 Benchmarks . . . 44
5.2.2 Microbenchmarks . . . 45
5.3 Architectural analysis . . . 45
5.3.1 Data movement . . . 45
5.3.2 Power consumption . . . 46
5.3.3 Utilization over time . . . 47
5.4 Sensitivity studies . . . 47
5.5 Scalability . . . 49

6 Related Work **51**
6.1 FHE accelerators . . . 51
6.2 Hybrid HE-MPC accelerators . . . 52
6.3 GPU acceleration . . . 52

7 Conclusion **53**
7.1 Future Work . . . 53

Chapter 1

Introduction

Despite massive efforts to improve the security of computer systems, security breaches are only becoming more frequent and damaging, as more sensitive data is processed in the cloud [42, 65]. Current encryption technology is of limited help, because servers must decrypt data before processing it. Once data is decrypted, it is vulnerable to breaches.

Fully Homomorphic Encryption (FHE) is a class of encryption schemes that address this problem by enabling *generic computation on encrypted data*. Figure 1-1 shows how FHE enables secure offloading of computation. The client wants to compute an expensive function f (e.g., a deep learning inference) on some private data x. To do this, the client encrypts x and sends it to an untrusted server, which computes f on this encrypted data *directly* using FHE, and returns the encrypted result to the client. FHE provides ideal security properties: even if the server is compromised, attackers cannot learn anything about the data, as it remains encrypted throughout.

FHE is a young but quickly developing technology. First realized in 2009 [33], early FHE schemes were about 10^9 times slower than performing computations on unencrypted data. Since then, improved FHE schemes have greatly reduced these overheads and broadened its applicability [3, 58]. FHE has inherent limitations—for example, data-dependent branching is impossible, since data is encrypted—so it will not subsume all computations. Nonetheless, important classes of computations, like deep learning inference [18,26,27], linear algebra [38], genomics [12], and other inference and learning tasks [40] are a good fit for FHE. This has sparked significant industry

and government investments [5, 10, 24] to widely deploy FHE.

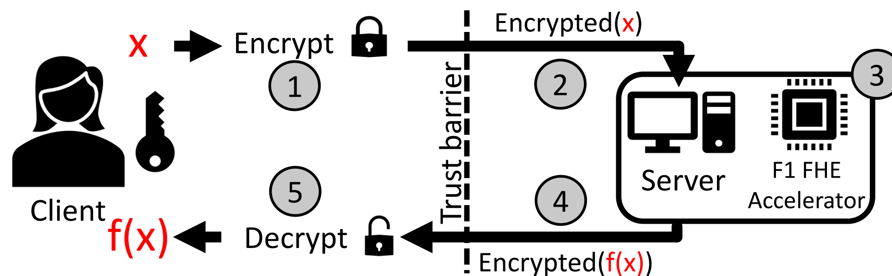

Figure 1-1: FHE allows a user to securely offload computation to an untrusted server.

Unfortunately, FHE still carries substantial performance overheads: despite recent advances [16, 26, 27, 60, 63], FHE is still 10,000× to 100,000× slower than unencrypted computation when executed in carefully optimized software. Though this slowdown is large, it can be addressed with hardware acceleration: *if a specialized FHE accelerator provides large speedups over software execution, it can bridge most of this performance gap and enable new use cases.*

For an FHE accelerator to be broadly useful, it should be programmable, i.e., capable of executing arbitrary FHE computations. While prior work has proposed several FHE accelerators, they do not meet this goal. Prior FHE accelerators [22, 23, 28, 62, 63, 67] target individual FHE operations, and miss important ones that they leave to software. These designs are designed for a Field Programmable Gate Arrays (FPGA), so they are small and miss the data movement issues facing an FHE Application specific Integrated Circuit (ASIC) accelerator. These designs also overspecialize their functional units to specific parameters, and cannot efficiently handle the range of parameters needed within a program or across programs.

In this thesis we present F1, the first programmable FHE accelerator. F1 builds on an in-depth architectural analysis of the characteristics of FHE computations, which exposes the main challenges and reveals the design principles a programmable FHE architecture should exploit.

Harnessing opportunities and challenges in FHE: F1 is tailored to the three defining characteristics of FHE:

(1) Complex operations on long vectors: FHE encodes information using very large vectors, several thousand elements long, and processes them using modular

arithmetic. F1 employs *vector processing* with *wide functional units* tailored to FHE operations to achieve large speedups. The challenge is that two key operations on these vectors, the Number-Theoretic Transform (NTT) and automorphisms, are not element-wise and require complex dataflows that are hard to implement as vector operations. To tackle these challenges, F1 features specialized NTT units and the first vector implementation of an automorphism functional unit.

(2) Regular computation: FHE programs are dataflow graphs of arithmetic operations on vectors. All operations and their dependences are known ahead of time (since data is encrypted, branches or dependences determined by runtime values are impossible). F1 exploits this by adopting *static scheduling*: in the style of Very Long Instruction Word (VLIW) processors, all components have fixed latencies and the compiler is in charge of scheduling operations and data movement across components, with no hardware mechanisms to handle hazards (i.e., no stall logic). Thanks to this design, F1 can issue many operations per cycle with minimal control overheads; combined with vector processing, F1 can issue tens of thousands of scalar operations per cycle.

(3) Challenging data movement: In FHE, encrypting data increases its size (typically by at least 50×); data is grouped in long vectors; and some operations require large amounts (tens of megabytes) of auxiliary data. Thus, we find that data movement is *the key challenge* for FHE acceleration: despite requiring complex functional units, in current technology, limited on-chip storage and memory bandwidth are the bottlenecks for most FHE programs. Therefore, F1 is primarily designed to minimize data movement. First, F1 features an explicitly managed on-chip memory hierarchy, with a heavily banked scratchpad and distributed register files. Second, F1 uses mechanisms to decouple data movement and hide access latencies by loading data far ahead of its use. Third, F1 uses new, FHE-tailored scheduling algorithms that maximize reuse and make the best out of limited memory bandwidth. Fourth, F1 uses relatively *few functional units with extremely high throughput*, rather than lower-throughput functional units as in prior work. This *reduces the amount of data that must reside on-chip simultaneously*, allowing higher reuse.

In summary, F1 brings decades of research in architecture to bear, including vector processing and static scheduling, and combines them with new techniques and

scheduling algorithms (Chapter 4), to design a programmable FHE accelerator. The main components of F1 were implemented in RTL and synthesized in a commercial 14nm/12nm process. With a modest area budget of 151 mm^2, our F1 implementation provides 36 tera-ops/second of 32-bit modular arithmetic, 64 MB of on-chip storage, and a 1 TB/s high-bandwidth memory.

We evaluate F1 using cycle-accurate simulation running complete FHE applications, and demonstrate speedups of 1,200×–17,000× over state-of-the-art software implementations. These dramatic speedups counter most of FHE's overheads and enable new applications. For example, F1 executes a deep learning inference that used to take 20 minutes in 240 milliseconds, enabling secure real-time deep learning in the cloud.

1.1 Contributions

F1 is a collaborative project. This thesis describes the full design and implementation of F1, while emphasizing the author's key contributions:

- **Design and implementation of the F1 software stack:** We implement a complete software stack that takes high level descriptions of FHE computations and compiles them down to F1 instructions issued at precise cycles. Because our computations are statically scheduled, the scheduled programs also provide the basis for our performance evaluation (Chapter 4, Chapter 5).

- **Design space exploration:** Using the F1 scheduler, we perform a design-space exploration to select efficient configurations of our accelerator (Section 5.5).

- **FHE benchmark programs:** We implement FHE programs from the literature in the F1 DSL to evaluate our accelerator's performance on relevant workloads (Section 5.1.2).

Chapter 2

Background

Fully Homomorphic Encryption allows for performing arbitrary arithmetic on encrypted plaintext values, via appropriate operations on their ciphertexts. Decrypting the resulting ciphertext yields the same result as if the operations were performed on the plaintext values "in the clear."

Over the last decade, prior work has proposed multiple *FHE schemes*, each working in somewhat different ways and providing various trade-offs. These schemes include BGV [15], B/FV [14, 29], GSW [35], and CKKS [18]. Though these schemes differ in how they encrypt plaintexts, they all use the same data type for ciphertexts: polynomials where each coefficient is an integer modulo Q. This commonality makes it possible to build a single accelerator that supports multiple FHE schemes; F1 supports BGV, GSW, and CKKS.

We describe FHE in a layered fashion: Section 2.1 introduces FHE's programming model and operations, i.e., FHE's *interface*; Section 2.2 describes how FHE operations are *implemented*; Section 2.3 presents implementation *optimizations*; and Section 2.4 performs an *architectural analysis* of a representative FHE kernel to reveal acceleration opportunities.

For concreteness, we *introduce FHE using the BGV scheme*, and briefly discuss other FHE schemes in Section 2.5.

2.1 FHE programming model and operations

FHE programs are *dataflow graphs*: directed acyclic graphs where nodes are operations and edges represent data values (inputs, outputs, or intermediate values consumed by one or more operations). All operations and dependences are known in advance, and data-dependent branching is impossible.

In FHE, unencrypted (plaintext) data values are always *vectors*; in BGV [15], each vector consists of N integers modulo an integer t. BGV provides three operations on these vectors: element-wise *addition* (mod t), element-wise *multiplication* (mod t), and a small set of particular vector *permutations*.

We stress that this is BGV's *interface*, not its implementation: it describes *unencrypted* data, and the homomorphic operations that BGV implements on that data in its encrypted form. In Section 2.2 we describe how BGV represents encrypted data and how each operation is implemented.

At a high level, FHE provides a vector programming model with restricted operations. In particular, individual vector elements cannot be directly accessed. This causes some overheads in certain algorithms; for example, summing up the elements of a vector is non-trivial, and requires a sequence of permutations and additions.

Despite these limitations, prior work has devised reasonably efficient implementations of key algorithms, including linear algebra [38], neural network inference [16, 36], logistic regression [39], and genome processing [12]. These implementations are often coded by hand, but recent work has proposed FHE compilers to automate this translation for particular domains, like deep learning [26, 27].

Finally, note that not all data must be encrypted: BGV provides versions of addition and multiplication where one of the operands is unencrypted. Multiplying by unencrypted data is cheaper, so algorithms can trade privacy for performance. For example, a deep learning inference can use encrypted weights and inputs to keep the model private, or use unencrypted weights, which does not protect the model but keeps inputs and inferences private [16].

2.2 BGV implementation overview

We now describe how BGV represents and processes encrypted data (ciphertexts). The implementation of each computation on ciphertext data is called a *homomorphic operation*. For example, the *homomorphic multiplication* of two ciphertexts yields another ciphertext that, when decrypted, is the element-wise multiplication of the encrypted plaintexts.

Data types: BGV encodes each plaintext vector as a polynomial with N coefficients mod t. We denote the plaintext space as R_t, so

$$\mathfrak{a} = a_0 + a_1 x + ... + a_{N-1} x^{N-1} \in R_t$$

is a plaintext. Each plaintext is encrypted into a ciphertext consisting of two polynomials of N integer coefficients modulo some value Q, with $Q \gg t$. Each ciphertext polynomial is a member of R_Q.

Encryption and decryption: Though encryption and decryption are performed by the client (so F1 need not accelerate them), they are useful to understand. In BGV, the *secret key* is a polynomial $\mathfrak{s} \in R_Q$. To encrypt a plaintext $\mathfrak{m} \in R_t$, one samples a uniformly random $\mathfrak{a} \in R_Q$, an *error* (or *noise*) $\mathfrak{e} \in R_Q$ with small entries, and computes the ciphertext ct as

$$ct = (\mathfrak{a}, \mathfrak{b} = \mathfrak{a}\mathfrak{s} + t\mathfrak{e} + \mathfrak{m}).$$

Ciphertext $ct = (\mathfrak{a}, \mathfrak{b})$ is decrypted by recovering $\mathfrak{e}' = t\mathfrak{e} + \mathfrak{m} = \mathfrak{b} - \mathfrak{a}\mathfrak{s} \bmod Q$, and then recovering $\mathfrak{m} = \mathfrak{e}' \bmod t$. Decryption is correct as long as $\mathfrak{e}'$ does not "wrap around" modulo Q, i.e., its coefficients have magnitude less than $Q/2$.

The security of any encryption scheme relies on the ciphertexts not revealing anything about the value of the plaintext (or the secret key). Without adding the noise term $\mathfrak{e}$, the original message $\mathfrak{m}$ would be recoverable from ct via simple Gaussian elimination. Including the noise term entirely hides the plaintext (under cryptographic assumptions) [49].

As we will see, applying homomorphic operations on ciphertexts increases their noise, so we can only perform a limited number of operations before the resulting noise

becomes too large and makes decryption fail. We later describe *noise management strategies* to keep this noise bounded and thereby allow unlimited operations.

Homomorphic addition of ciphertexts $ct_0 = (\mathfrak{a}_0, \mathfrak{b}_0)$ and $ct_1 = (\mathfrak{a}_1, \mathfrak{b}_1)$ is done simply by adding their corresponding polynomials: $ct_{\text{add}} = ct_0 + ct_1 = (\mathfrak{a}_0 + \mathfrak{a}_1, \mathfrak{b}_0 + \mathfrak{b}_1)$.

Homomorphic multiplication requires two steps. First, the four input polynomials are multiplied and assembled:

$$ct_\times = (\mathfrak{l}_2, \mathfrak{l}_1, \mathfrak{l}_0) = (\mathfrak{a}_0\mathfrak{a}_1, \mathfrak{a}_0\mathfrak{b}_1 + \mathfrak{a}_1\mathfrak{b}_0, \mathfrak{b}_0\mathfrak{b}_1).$$

This $ct_\times$ can be seen as a special intermediate ciphertext encrypted under a different secret key. The second step performs a *key-switching operation* to produce a ciphertext encrypted under the original secret key $\mathfrak{s}$. More specifically, $\mathfrak{l}_2$ undergoes this key-switching process to produce two polynomials $(\mathfrak{u}_1, \mathfrak{u}_0) = \text{KeySwitch}(\mathfrak{l}_2)$. The final output ciphertext is $ct_{\text{mul}} = (\mathfrak{l}_1 + \mathfrak{u}_1, \mathfrak{l}_0 + \mathfrak{u}_0)$.

As we will see later (Section 2.4), key-switching is an expensive operation that dominates the cost of a multiplication.

Homomorphic permutations permute the N plaintext values (coefficients) that are encrypted in a ciphertext. Homomorphic permutations are implemented using *automorphisms*, which are special permutations of the coefficients of the ciphertext polynomials. There are N automorphisms, denoted $\sigma_k(\mathfrak{a})$ and $\sigma_{-k}(\mathfrak{a})$ for all positive odd $k < N$. Specifically,

$$\sigma_k(\mathfrak{a}) : a_i \to (-1)^s a_{ik \bmod N} \text{ for } i = 0, ..., N-1,$$

where $s = 0$ if $ik \bmod 2N < N$, and $s = 1$ otherwise. For example, $\sigma_5(\mathfrak{a})$ permutes $\mathfrak{a}$'s coefficients so that a_0 stays at position 0, a_1 goes from position 1 to position 5, and so on (these wrap around, e.g., with $N = 1024$, a_{205} goes to position 1, since $205 \cdot 5 \bmod 1024 = 1$).

To perform a homomorphic permutation, we first compute an automorphism on the ciphertext polynomials: $ct_\sigma = (\sigma_k(\mathfrak{a}), \sigma_k(\mathfrak{b}))$. Just as in homomorphic multiplication, ct_σ is encrypted under a different secret key, requiring an expensive key-switch to produce the final output $ct_{\text{perm}} = (\mathfrak{u}_1, \sigma_k(\mathfrak{b}) + \mathfrak{u}_0)$, where $(\mathfrak{u}_1, \mathfrak{u}_0) = \text{KeySwitch}(\sigma_k(\mathfrak{a}))$.

We stress that the permutation applied to the ciphertext *does not* induce the

same permutation on the underlying plaintext vector. For example, using a single automorphism and careful indexing, it is possible to homomorphically *rotate* the vector of the N encrypted plaintext values.

Noise growth and management: Recall that ciphertexts have noise, which limits the number of operations that they can undergo before decryption gives an incorrect result. Different operations induce different noise growth: addition and permutations cause little growth, but multiplication incurs much more significant growth. So, to a first order, noise size is determined by *multiplicative depth*, i.e., the longest chain of homomorphic multiplications in the computation.

Noise forces the use of a large ciphertext modulus Q. For example, an FHE program with multiplicative depth of 16 needs Q to be about 512 bits. The noise budget, and thus the tolerable multiplicative depth, grow about linearly with $\log Q$.

FHE uses two noise management techniques in tandem: *bootstrapping* and *modulus switching*.

Bootstrapping [33] enables FHE computations of *unbounded* depth. Essentially, it removes noise from a ciphertext without access to the secret key (by evaluating the decryption function homomorphically). Thus, FHE programs with a large multiplicative depth can be divided into regions of limited depth, separated by bootstrapping operations.

Even with bootstrapping, FHE schemes need a large noise budget (i.e., a large Q) because *(1)* bootstrapping is expensive, and a higher noise budget enables less-frequent bootstrapping, and *(2)* bootstrapping itself consumes a certain noise budget (this is similar to why pipelining any circuit hits a performance ceiling: registers themselves add area and latency).

Modulus switching rescales ciphertexts from modulus Q to a modulus Q', which reduces the noise proportionately. Modulus switching is usually applied before each homomorphic multiplication, to reduce its noise blowup.

For example, to execute an FHE program of multiplicative depth 16, we would start with a 512-bit modulus Q. Right before each multiplication, we would switch to a modulus that is smaller by 32 bits. So, for example, operations at depth 8 use a 256-bit modulus. Thus, beyond reducing noise, modulus switching reduces ciphertext sizes, and thus computation cost.

Security and parameters: The dimension N and modulus Q cannot be chosen independently; $N/\log Q$ must be above a certain level for sufficient security. In practice, many useful computations require non-trivial depth, meaning that they must be implemented with a wide modulus (large Q). To provide acceptable security for these computations, we must in turn use a large N, resulting in very large ciphertexts. For example, with 512-bit Q, N must be at least $16K$, resulting in very large ciphertexts (2 MB).

2.3 Algorithmic insights and optimizations

F1 leverages two optimizations developed in prior work:

Fast polynomial multiplication via NTTs: Multiplying two polynomials requires convolving their coefficients, an expensive (naively $O(N^2)$) operation. Just like convolutions can be made faster with the Fast Fourier Transform, polynomial multiplication can be made faster with the Number-Theoretic Transform (NTT) [54], a variant of the discrete Fourier transform for modular arithmetic. The NTT takes an N-coefficient polynomial as input and returns an N-element vector representing the input in the *NTT domain*. This representation is useful because polynomial addition and multiplication are both simple component-wise operations. Specifically,

$$NTT(\mathfrak{a} + \mathfrak{b}) = NTT(\mathfrak{a}) + NTT(\mathfrak{b})$$

and

$$NTT(\mathfrak{a}\mathfrak{b}) = NTT(\mathfrak{a}) \odot NTT(\mathfrak{b})$$

where $\odot$ denotes component-wise multiplication.

NTTs require only $O(N \log N)$ modular operations, saving substantial computation over the naive approach. Thus, FHE implementations often leave ciphertexts in the NTT domain across homomorphic operations, instead of performing forward and inverse NTTs for every multiply.

Avoiding wide arithmetic via Residue Number System (RNS) representation: FHE requires wide ciphertext coefficients (e.g., 512 bits), but wide arithmetic is expensive: the cost of a modular multiplier (which takes most of the compute) grows

quadratically with bit width in our range of interest. Sub-quadratic designs such as Karatsuba multipliers [44] exist but impose higher constant-factor overheads that make them more expensive for our targetted bit widths. Moreover, we need to efficiently support a broad range of widths (e.g., 64 to 512 bits in 32-bit increments), both because programs need different widths, and because modulus switching progressively reduces coefficient widths.

RNS representation [31] enables representing a single polynomial with wide coefficients as multiple polynomials with narrower coefficients, called *residue polynomials*. To achieve this, the modulus Q is set to be the product of L smaller distinct primes, $Q = q_1 q_2 \cdots q_L$. Then, a polynomial in R_Q can be represented as L polynomials in $R_{q_1}, \ldots, R_{q_L}$, where the coefficients in the i-th polynomial are simply the wide coefficients modulo q_i. For example, with $W = 32$-bit words, a ciphertext polynomial with 512-bit modulus Q is represented as $L = \log Q/W = 16$ polynomials with 32-bit coefficients.

All FHE operations can be carried out under RNS representation, and have either better or equivalent bit-complexity than operating on one wide-coefficient polynomial.

2.4 Architectural analysis of FHE

We now analyze a key FHE kernel in depth to understand how we can (and cannot) accelerate it. Specifically, we consider the key-switching operation, which is expensive and takes the bulk of work in homomorphic multiplications and permutations.

Listing 2.1 shows an implementation of key-switching. Key-switching takes three inputs: a polynomial `x`, and two *key-switch hint matrices* `ksh0` and `ksh1`. `x` is stored in RNS form as L residue polynomials (`RVec`). Each residue polynomial `x[i]` is a vector of N 32-bit integers modulo q_i. Inputs and outputs are in the NTT domain, and only the `y[i]` (line 3) are in coefficient form.

Computation vs. data movement: A single key-switch requires L^2 forward or inverse NTTs (which have the same cost), $2L^2$ multiplications, and $2L^2$ additions of N-element vectors. In RNS form, the rest of a homomorphic multiplication (excluding key-switching) is $4L$ multiplications and $3L$ additions (Section 2.2), so key-switching is dominant.

```
1   def keySwitch(x: RVec[L],
2         ksh0: RVec[L][L], ksh1: RVec[L][L]):
3     y = [INTT(x[i],q_i) for i in range(L)]
4     u0: RVec[L] = [0, ...]
5     u1: RVec[L] = [0, ...]
6     for i in range(L):
7       for j in range(L):
8         xqj = (i == j) ? x[i] : NTT(y[i], q_j)
9         u0[j] += xqj * ksh0[i,j] mod q_j
10        u1[j] += xqj * ksh1[i,j] mod q_j
11    return (u0, u1)
```

Listing 2.1: Key-switch implementation. `RVec` is an N-element vector of 32-bit values, storing a single RNS polynomial in either the coefficient or the NTT domain.

However, the main cost at high values of L and N is data movement. For example, at $L = 16$, $N = 16K$, each RNS polynomial (`RVec`) is 64 KB; each ciphertext polynomial is 1 MB; each ciphertext is 2 MB; and the key-switch hints dominate, taking up 32 MB. With F1's compute throughput, fetching the inputs of each key-switching from off-chip memory would demand about 10 TB/s of memory bandwidth, far beyond what is available in current technology. Thus, it is crucial to reuse these values as much as possible.

Fortunately, key-switch hints can be reused: all homomorphic multiplications use the same key-switch hint matrices, and each automorphism has its own pair of matrices. But values are so large that few of them fit on-chip.

Finally, note that decomposing or tiling these computations is difficult and may lead to performance penalties. Tiling across `RVec` elements works poorly because in NTTs every input element affects every output element. Tiling key-switch hint matrices on either dimension is more feasible, but produces many long-lived intermediate values. This approach may produce performance gains if a particular key-switch hint matrix is reused many times, but must be balanced against the increased footprint of intermediates.

Performance requirements: We conclude that, to accommodate these large operands, an FHE accelerator requires a memory system that *(1)* decouples data movement from computation, as demand misses during frequent key-switches would tank performance; and *(2)* implements a large amount of on-chip storage (over 32 MB in our example) to allow reuse across entire homomorphic operations (e.g., reusing the same key-switch hints across many homomorphic multiplications).

Moreover, the FHE accelerator must be designed to use the memory system well. First, scheduling data movement and computation is crucial: data must be fetched far ahead of its use to provide decoupling, and operations must be ordered carefully to maximize reuse. Second, since values are large, excessive parallelism can increase footprint and hinder reuse. Thus, the system should use relatively few high-throughput functional units rather than many low-throughput ones.

Functionality requirements: Programmable FHE accelerators must support a wide range of parameters, both N (polynomial/vector sizes) and L (number of RNS polynomials, i.e., width of Q). While N is generally fixed for a single program, L changes as modulus switching sheds off polynomials.

Moreover, FHE accelerators must avoid overspecializing in order to support algorithmic diversity. For instance, we have described *an* implementation of key-switching, but there are others [34, 45] with different tradeoffs. For example, an alternative implementation requires much more compute but has key-switch hints that grow with L instead of L^2, so it becomes attractive for very large L ($\sim$20).

F1 accelerates *primitive operations on large vectors*: modular arithmetic, NTTs, and automorphisms. It exploits wide vector processing to achieve very high throughput, even though this makes NTTs and automorphisms costlier. F1 avoids building functional units for coarser primitives, like key-switching, which would hinder algorithmic diversity.

Limitations of prior accelerators: Prior work has proposed several FHE accelerators for FPGAs [22, 23, 28, 52, 52, 53, 62, 63, 67]. These systems have three important limitations. First, they work by accelerating some primitives but defer others to a general-purpose host processor, and rely on the host processor to sequence operations. This causes excessive data movement that limits speedups. Second, these accelerators build functional units for *fixed parameters* N and L (or $\log Q$ for those not using RNS). Third, many of these systems build overspecialized primitives that limit algorithmic diversity.

Most of these systems achieve limited speedups, about 10× over software baselines. HEAX [62] achieves larger speedups (200× vs. a single core). But it does so by overspecializing: it uses relatively low-throughput functional units for primitive operations, so to achieve high performance, it builds a fixed-function pipeline for key-switching.

2.5 FHE schemes other than BGV

We have so far focused on BGV, but other FHE schemes provide different tradeoffs. For instance, whereas BGV requires integer plaintexts, CKKS [18] supports “approximate” computation on fixed-point values. B/FV [14, 29] encodes plaintexts in a way that makes modulus switching before homomorphic multiplication unnecessary, thus easing programming (but forgoing the associated efficiency gains). And GSW [35] features reduced, asymmetric noise growth under homomorphic multiplication, but encrypts a small amount of information per ciphertext (not a full N-element vector).

Because F1 accelerates primitives rather than whole-ciphertext operations, it supports BGV, CKKS, and GSW with the same hardware, since they use the same primitives. Accelerating B/FV would require some other primitives, so, though adding support for them would not be too difficult, our current implementation does not target it.

Chapter 3

F1 Architecture

F1 was designed in collaboration with Nikola Samardzic, Aleksandar Krastev, and Daniel Sanchez. All work on the design of functional units and register files was conducted by Nikola Samardzic and Alex Krastev. Descriptions of the functional unit and register file design are included in this thesis to fully describe F1's architecture, even though they are not the work of the author.

Figure 3-1 shows an overview of F1, which we derive from the insights in Section 2.4.

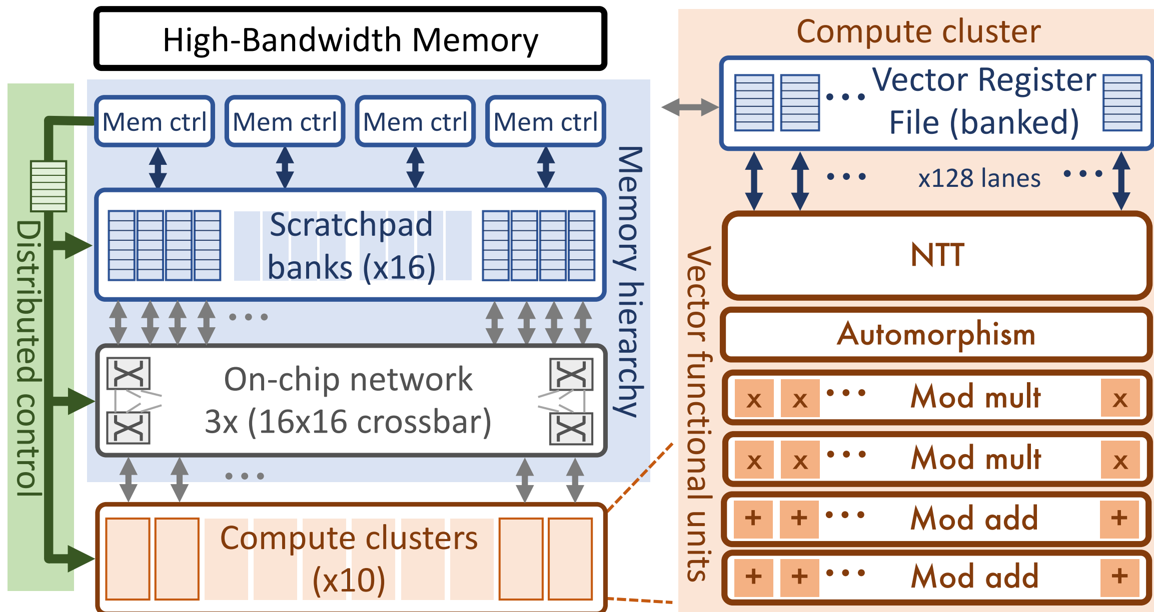

Figure 3-1: Overview of the F1 architecture.

3.1 Vector processing with specialized functional units

F1 features wide-vector execution with functional units (FUs) tailored to primitive FHE operations. Specifically, F1 implements vector FUs for modular addition, modular multiplication, NTTs (forward and inverse in the same unit), and automorphisms. Because we leverage RNS representation, these FUs use a fixed, small arithmetic word size (32 bits in our implementation), avoiding wide arithmetic.

FUs process vectors of configurable *length* N using a fixed number of *vector lanes* E. Our implementation uses $E = 128$ lanes and supports power-of-two lengths N from 1,024 to 16,384. This covers the common range of FHE polynomial sizes, so an RNS polynomial maps to a single vector. Larger polynomials (e.g., of 32K elements) can use multiple vectors.

All FUs are *fully pipelined*, so they achieve the same throughput of $E = 128$ elements/cycle. FUs consume their inputs in contiguous chunks of E elements in consecutive cycles. This is easy for element-wise operations, but hard for NTTs and automorphisms. Section 3.7 details our novel FU implementations, including the first vector implementation of automorphisms. Our evaluation shows that these FUs achieve much higher performance than those of prior work, because, as we saw in Section 2.4, *having fewer high-throughput FUs limits parallelism and thus memory footprint*.

3.2 Compute clusters

Functional units are grouped in *compute clusters*, as Figure 3-1 shows. Each cluster features several FUs (1 NTT, 1 automorphism, 2 multipliers, and 2 adders in our implementation) and a banked register file that can (cheaply) supply enough operands each cycle to keep all FUs busy. The chip has multiple clusters (16 in our implementation).

3.3 Memory system

F1 features an explicitly-managed memory hierarchy. As Figure 3-1 shows, F1 features a large, heavily-banked scratchpad (64 MB across 16 banks in our implementation). The scratchpad interfaces with both high-bandwidth off-chip memory (HBM2 in our implementation) and with compute clusters through an on-chip network.

F1 uses decoupled data orchestration [59] to hide main memory latency. Scratchpad banks work autonomously, fetching data from main memory far ahead of its use. Since memory has relatively low bandwidth, off-chip data is always staged in scratchpads, and compute clusters do not access main memory directly.

The on-chip network connecting scratchpad banks and compute clusters provides very high bandwidth, which is necessary because register files are small and achieve limited reuse. We implement a single-stage bit-sliced crossbar network [57] that provides full bisection bandwidth. Banks and the network have wide ports (512 bytes), so that a single scratchpad bank can send a vector to a compute unit at the rate it is consumed (and receive it at the rate it is produced). This avoids long staging of vectors at the register files.

3.4 Static scheduling

Because FHE programs are completely regular, F1 adopts a *static, exposed microarchitecture*: all components have fixed latencies, which are exposed to the compiler. The compiler is responsible for scheduling operations and data transfers in the appropriate cycles to prevent structural or data hazards. This is in the style of VLIW [30].

This approach simplifies logic throughout the chip. For example, FUs need no stalling logic; register files and scratchpad banks need no dynamic arbitration to handle conflicts; and the on-chip network uses simple switches that change their configuration independently over time, without the buffers and arbiters of packet-switched networks.

Because memory accesses do have a variable latency, we assume the worst-case latency, and buffer data that arrives earlier (note that, because we access large chunks of data, e.g., 64 KB, this worst-case latency is not far from the average).

3.5 Distributed control

Though static scheduling is the hallmark of VLIW, F1's implementation is quite different: rather than having a single stream of instructions with many operations each, in F1 each component has an *independent instruction stream*. This is possible because F1 does not have any control flow: though FHE programs may have loops, we unroll them to avoid all branches, and compile programs into linear sequences of instructions.

This approach may appear costly. But vectors are very long, so each instruction encodes a lot of work and this overhead is minimal. Moreover, this enables a compact instruction format, which encodes a single operation followed by the number of cycles to wait until running the next instruction. This encoding avoids the low utilization of VLIW instructions, which leave many operation slots empty. Each FU, register file, network switch, scratchpad bank, and memory controller has its own instruction stream, which a control unit fetches in small blocks and distributes to components. We use 3 bits to encode the instruction opcode, 17 bits to encode operands, and a further 20 bits to encode the number of cycles until the next instruction is issued. This results in 5 byte instruction format, an insignificant overhead compared to our minimum operand size of 4KB. Across all of our benchmarks, instruction fetches consume less than 0.1% of main memory traffic.

3.6 Register file (RF) design

Each cluster in F1 requires 10 read ports and 6 write ports to keep all FUs busy. To enable this cheaply, use an 8-banked *element-partitioned* register file design [6] that leverages long vectors: each vector is striped across banks, and each FU cycles through all banks over time, using a single bank each cycle. By staggering the start of each vector operation, FUs access different banks each cycle. This avoids multiporting, requires a simple RF-FU interconnect, and performs within 5% of an ideal infinite-ported RF.

3.7 Functional Units

In this section, we describe F1's novel functional units. These include the first vectorized automorphism unit (Section 3.7.1), the first fully-pipelined flexible NTT unit (Section 3.7.2), and a new simplified modular multiplier adapted to FHE (Section 3.7.3).

3.7.1 Automorphism unit

Each residue polynomial in F1 is stored as G independent E-element hardware vectors ($N = G \cdot E$). An automorphism σ_k maps the element at index i to index $ki \bmod N$; there are N automorphisms total, two for each odd $k < N$ (Section 2.2). The key challenge in designing an automorphism unit is that these permutations are hard to vectorize: we would like this unit to consume and produce E =128 elements/cycle, but the vectors are much longer, with N up to 16 K, and elements are permuted across different hardware vectors. Moreover, we must support variable N *and* all automorphisms.

Standard solutions fail: a 16 K×16 K crossbar is much too large; a scalar approach, like reading elements in sequence from an SRAM, is too slow (taking N cycles); and using banks of SRAM to increase throughput runs into frequent bank conflicts: each automorphism "spreads" elements with a different stride, so regardless of geometry, some automorphisms will map many consecutive elements to the same bank.

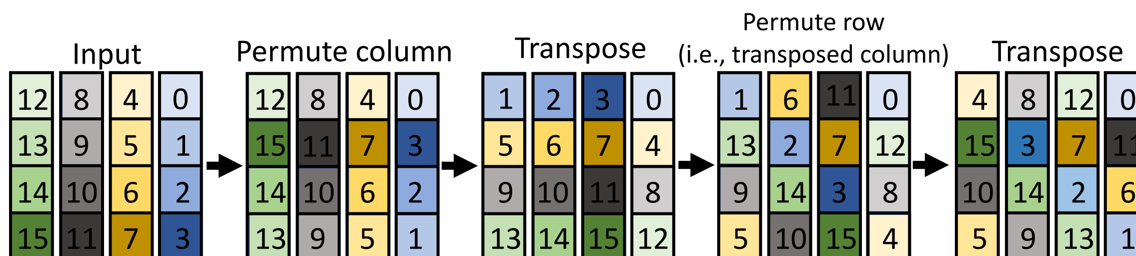

Figure 3-2: Applying σ_3 on a ciphertext of four 4-element chunks by using only permutations local to chunks.

We contribute a new insight that makes vectorizing automorphisms simple: if we interpret a residue polynomial as a $G \times E$ matrix, an automorphism can always be decomposed into two independent *column* and *row permutations*. If we transpose this matrix, both column and row permutations can be applied *in blocks of E elements*.

Figure 3-2 shows an example of how automorphism σ_3 is applied to a residue polynomial with $N = 16$ and $E = 4$ elements/cycle. Note how the permute column and row operations are local to each 4-element hardware vector. Other σ_k induce different permutations, but with the same row/column structure.

Our automorphism unit, shown in Figure 3-3, uses this insight to be both vectorized (consuming $E = 128$ elements/cycle) and fully pipelined. Given a residue polynomial of $N = G \cdot E$ elements, the automorphism unit first applies the column permutation to each E-element input. Then, it feeds this to a *transpose unit* that reads in the whole residue polynomial interpreting it as a $G \times E$ matrix, and produces its transpose $E \times G$. The transpose unit outputs E elements per cycle (outputting multiple rows per cycle when $G < E$). Row permutations are applied to each E-element chunk, and the reverse transpose is applied.

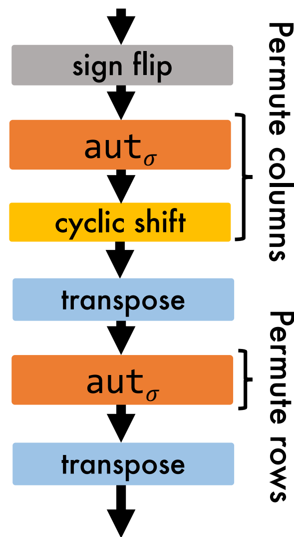

Figure 3-3: Automorphism unit.

Further, we decompose both the row and column permutations into a pipeline of sub-permutations that are *fixed in hardware*, with each sub-permutation either applied or bypassed based on simple control logic; this avoids using crossbars for the E-element permute row and column operations.

Transpose unit: Our *quadrant-swap transpose* unit transposes an $E \times E$ (e.g., 128×128) matrix by recursively decomposing it into quadrants and exploiting the identity

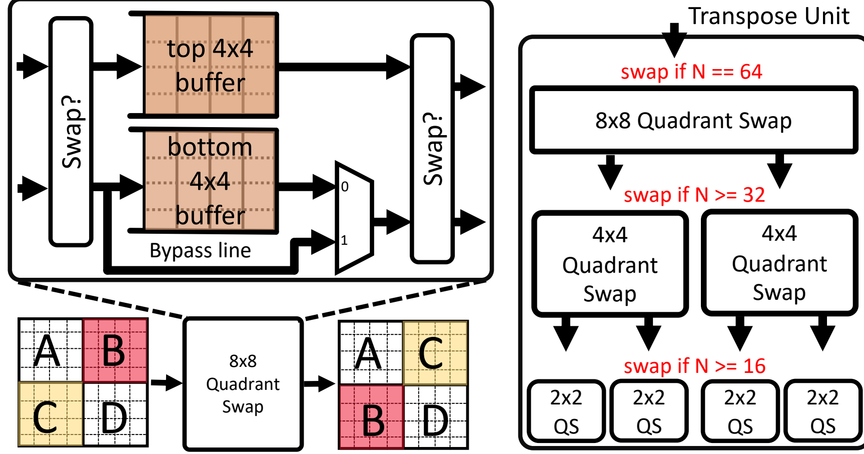

Figure 3-4: Transpose unit (right) and its component quadrant-swap unit (left).

$$\begin{bmatrix} \mathtt{A} & \mathtt{B} \\ \mathtt{C} & \mathtt{D} \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} \mathtt{A}^{\mathrm{T}} & \mathtt{C}^{\mathrm{T}} \\ \mathtt{B}^{\mathrm{T}} & \mathtt{D}^{\mathrm{T}} \end{bmatrix}.$$

The basic building block is a $K \times K$ *quadrant-swap* unit, which swaps quadrants `B` and `C`, as shown in Figure 3-4(left). Operationally, the quadrant swap procedure consists of three steps, each taking $K/2$ cycles:

1. Cycle `i` in the first step reads `A[i]` and `C[i]` and stores them in `top[i]` and `bottom[i]`, respectively.
2. Cycle `i` in the second step reads `B[i]` and `D[i]`. The unit activates the first swap MUX and the bypass line, thus storing `D[i]` in `top[i]` and outputting `A[i]` (by reading from `top[i]`) and `B[i]` via the bypass line.
3. Cycle `i` in the third step outputs `D[i]` and `C[i]` by reading from `top[i]` and `bottom[i]`, respectively. The second swap MUX is activated so that `C[i]` is on top.

Note that step 3 for one input can be done in parallel with step 1 for the next, so the unit is *fully pipelined*.

The transpose is implemented by a full $E \times E$ quadrant-swap followed by $\log_2 E$ layers of smaller transpose units to recursively transpose `A`, `B`, `C`, and `D`. Figure 3-4 (right) shows an implementation for $E = 8$. Finally, by selectively bypassing some of the initial quadrant swaps, this transpose unit also works for all values of N ($N = G \times E$ with power-of-2 $G < E$).

Prior work has implemented transpose units for signal-processing applications, either using registers [72, 74] or with custom SRAM designs [64]. Our design has three advantages over prior work: it uses standard SRAM memory, so it's dense without requiring complex custom SRAMs; it is fully pipelined; and it works for a wide range of dimensions.

3.7.2 Four-step NTT unit

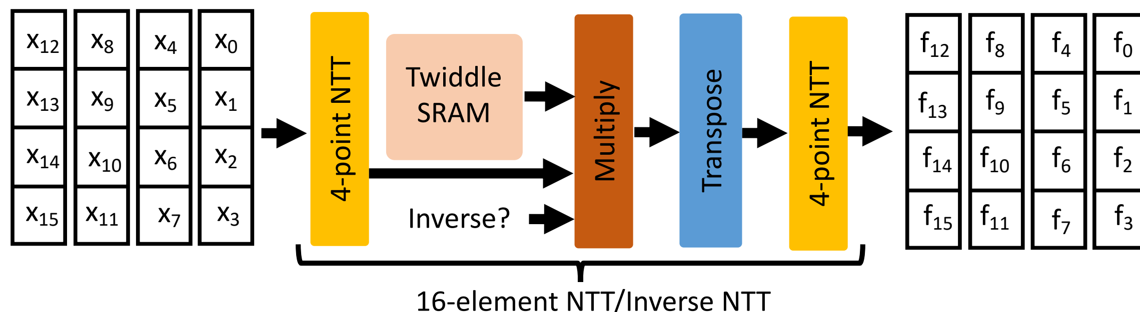

Figure 3-5: Example of a four-step NTT datapath that uses 4-point NTTs to implement 16-point NTTs.

There are many ways to implement NTTs in hardware: an NTT is like an FFT [20] but with a butterfly that uses modular multipliers. We implement N-element NTTs (from 1K to 16K) as a composition of smaller E=128-element NTTs, since implementing a full 16K-element NTT datapath is prohibitive. The challenge is that standard approaches result in memory access patterns that are hard to vectorize.

To that end, we use the *four-step variant* of the FFT algorithm [7], which adds an extra multiplication to produce a vector-friendly decomposition. Figure 3-5 illustrates a simple four-step NTT pipeline for $E = 4$; we use the same structure with $E = 128$. The unit is fully pipelined and consumes E elements per cycle. To compute an $N = E \times E$ NTT, the unit first computes an E-point NTT on each E-element group, multiplies each group with twiddles, transposes the E groups, and computes another E-element NTT on each transpose. The same NTT unit implements the inverse NTT by storing multiplicative factors (*twiddles*) required for both forward and inverse NTTs in a small *twiddle SRAM*.

Crucially, we are able to support all values of N using a single four-step NTT

pipeline by conditionally bypassing layers in the second NTT pipeline. We use the same transpose unit implementation as with automorphisms.

The NTT unit is large: each of the 128-element NTTs requires $E(\log(E) - 1)/2{=}384$ multipliers, and the full unit uses 896 multipliers. But its high throughput improves performance over many low-throughput NTTs (Section 5.2). This is the first implementation of a fully pipelined four-step NTT unit, improving NTT performance by 1,600× over the state of the art (Table 5.3).

3.7.3 Optimized modular multiplier

Multiplier	Area [μm^2]	Power [mW]	Delay [ps]
Barrett	5,271	18.4	1,317
Montgomery	2,916	9.2	1,040
NTT-friendly	2,165	5.36	1,000
FHE-friendly (ours)	1,817	4.1	1,000

Table 3.1: Area, power, and delay of modular multipliers.

Modular multiplication computes $a \cdot b \bmod q$. This is the most expensive and frequent operation. Therefore, improvements to the modular multiplier have an almost linear impact on the computational capabilities of an FHE accelerator.

Prior work [51] recognized that a Montgomery multiplier [55] within NTTs can be improved by leveraging the fact that the possible values of modulus q are restricted by the number of elements the NTT is applied to. We notice that if we only select moduli q_i, such that $q_i = -1 \bmod 2^{16}$, we can remove a multiplier stage from [51]; this reduces area by 19% and power by 30% (Table 3.1). The additional restriction on q is acceptable because FHE requires at most 10s of moduli [34], and our approach allows for 6,186 prime moduli.

Chapter 4

Scheduling Data and Computation

We now describe F1's software stack, focusing on the new static scheduling algorithms needed to use hardware well.

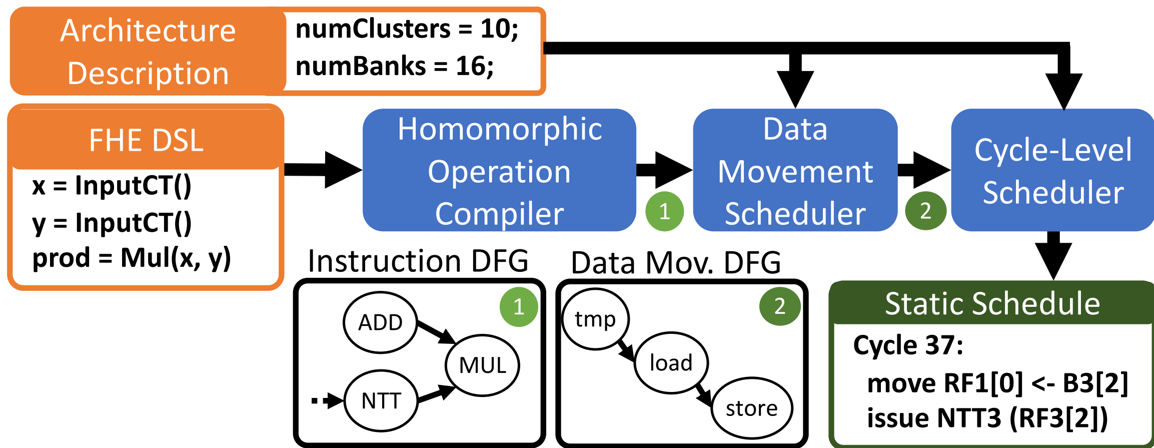

Figure 4-1: Overview of the F1 compiler.

Figure 4-1 shows an overview of the F1 compiler. The compiler takes as input an FHE program written in a high-level domain specific language (Section 4.2). The compiler is structured in three stages. First, the *homomorphic operation compiler* orders high-level operations to maximize reuse and translates the program into a *computation dataflow graph*, where operations are computation instructions but there are no loads or stores. Second, the *off-chip data movement scheduler* schedules transfers between main memory and the scratchpad to achieve decoupling and maximize reuse. This phase uses a simplified view of hardware, considering it as a scratchpad directly

attached to functional units. The result is a dataflow graph that includes loads and stores from off-chip memory. Third, the *cycle-level scheduler* refines this dataflow graph. It uses a cycle-accurate hardware model to divide instructions across compute clusters and schedule on-chip data transfers. This phase determine the exact cycles of all operations, and produces the instruction streams for all components.

This multi-pass scheduling primarily minimizes off-chip data movement, the critical bottleneck. Only in the last phase do we consider on-chip placement and data movement.

4.1 Comparison with prior work

We initially tried static scheduling algorithms from prior work [8, 13, 37, 50, 56], which primarily target VLIW architectures. However, we found these approaches ill-suited to F1 for multiple reasons. First, VLIW designs have less-flexible decoupling mechanisms and minimizing data movement is secondary to maximizing compute operations per cycle. Second, prior algorithms often focus on loops, where the key concern is to find a compact repeating schedule, e.g., through software pipelining [47]. By contrast, F1 has no flow control and we can schedule each operation independently. Third, though prior work has proposed register-pressure-aware instruction scheduling algorithms, they targeted small register files and basic blocks, whereas we must manage a large scratchpad over a much longer time horizon. Thus, many of the algorithms we tried worked poorly [37, 50, 56] for our use case.

For example, when considering an algorithm such as Code Scheduling to Minimize Register Usage (CSR) [37], we find that the schedules it produces suffer from a large blowup of live intermediate values. This large footprint causes scratchpad thrashing and results in poor performance. Furthermore, CSR is also quite computationally expensive, requiring long scheduling times for our larger benchmarks. We evaluate our approach against CSR in Section 5.4.

We also attempted to frame our schedules as a register allocation problem. Effectively, the key challenge in all of our schedules is *data movement*, not computation, so finding a register allocation which minimizes spills could provide a good basis for an efficient schedule. However, our scratchpad stores at least 1024 residue vectors

(1024 at maximum $N = 16K$, more for smaller values of N), and many of our benchmarks involve hundreds of thousands of instructions, meaning that register allocation algorithms simply could not scale to our required sizes [8, 11, 66, 73].

4.2 Translating the program to a dataflow graph

We implement a high-level domain-specific language (DSL) for writing F1 programs. To illustrate this DSL and provide a running example, Listing 4.1 shows the code for matrix-vector multiplication. This follows HELib's algorithm [38], which Figure 4-2 shows. This toy $4 \times 16K$ matrix-vector multiply uses input ciphertexts with $N = 16K$. Because accessing individual vector elements is not possible, the code uses homomorphic rotations to produce each output element.

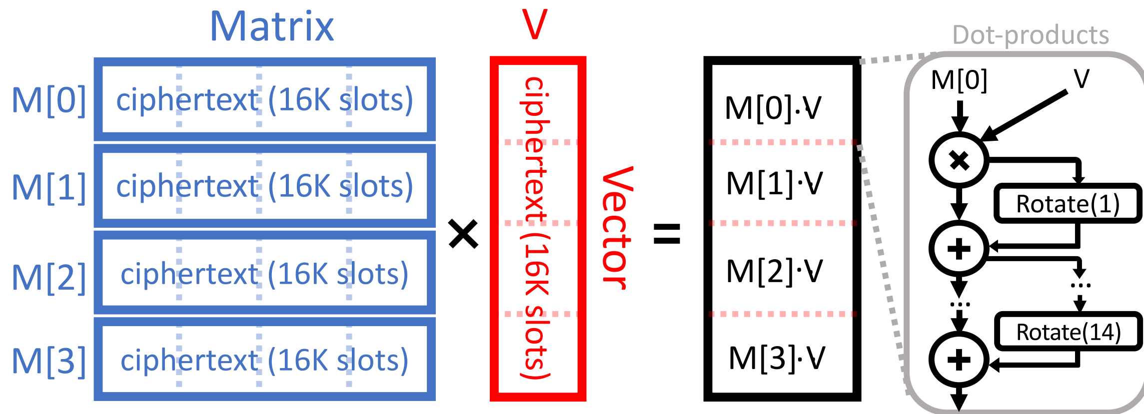

Figure 4-2: Example matrix-vector multiply using FHE.

As Listing 4.1 shows, programs in this DSL are at the level of the simple FHE interface presented in Section 2.1. There is only one aspect of the FHE implementation in the DSL: programs encode the desired noise budget ($L = 16$ in our example), as the compiler does not automate noise management.

4.3 Compiling homomorphic operations

The first compiler phase works at the level of the homomorphic operations provided by the DSL. It clusters operations to improve reuse, and translates them down to instructions.

```
1  p = Program(N = 16384)
2  M_rows = [ p.Input(L = 16) for i in range(4) ]
3  output = [ None for i in range(4) ]
4  V = p.Input(L = 16)
5
6  def innerSum(X):
7    for i in range(log2(p.N)):
8      X = Add(X, Rotate(X, 1 << i))
9    return X
10
11 for i in range(4):
12   prod = Mul(M_rows[i], V)
13   output[i] = innerSum(prod)
```

Listing 4.1: $(4 \times 16K)$ matrix-vector multiply in F1's DSL.

Ordering homomorphic operations seeks to maximize the reuse of key-switch hints, which is crucial to reduce data movement (Section 2.4). For instance, the program in Listing 4.1 uses 15 different sets of key-switch hint matrices: one for the multiplies (line 12), and a different one for *each* of the rotations (line 8). If this program was run sequentially as written, it would cycle through all 15 key-switching hints (which total 480 megabytes, exceeding on-chip storage) four times, achieving no reuse. Clearly, it is better to reorder the computation to perform all four multiplies, and then all four `Rotate(X, 1)`, and so on. This reuses each key-switch hint four times.

To achieve this, this pass first clusters *independent* homomorphic operations that reuse the same hint, then orders all clusters through simple list-scheduling. Operations that do depend on each other are not clustered together because they must be serialized. This approach generates schedules with good key-switch hint reuse, where such reuse is available in the underlying computation.

Translation: Each homomorphic operation is then compiled into instructions, using the implementation of each operation in the target FHE scheme (BGV, CKKS, or GSW). Each homomorphic operation may translate to thousands of instructions. These instructions are also ordered to minimize the amount of live intermediates. The end result is an instruction-level dataflow graph where every instruction is tagged with a priority that reflects its global order.

To illustrate how different translations can impact performance, we consider two different key-switching instruction orders in Listing 4.2. The ordering used in `keySwitch1` requires not only that `y` be precomputed, but also that `y` be stored in memory for the entire duration of the function, requiring L residue vectors of

```
1    def keySwitch1(x: RVec[L],
2          ksh0: RVec[L][L], ksh1: RVec[L][L]):
3      y = [INTT(x[i],q_i) for i in range(L)]
4      u0: RVec[L] = [0, ...]
5      u1: RVec[L] = [0, ...]
6      for j in range(L): # outer loop
7        for i in range(L):
8          xqj = (i == j) ? x[i] : NTT(y[i], q_j)
9          u0[j] += xqj * ksh0[i,j] mod q_j
10         u1[j] += xqj * ksh1[i,j] mod q_j
11     return (u0, u1)
12
13   def keySwitch2(x: RVec[L],
14         ksh0: RVec[L][L], ksh1: RVec[L][L]):
15     u0: RVec[L] = [0, ...]
16     u1: RVec[L] = [0, ...]
17     for i in range(L): # outer loop
18       yi = INTT(x[i], q_i)
19       for j in range(L):
20         xqj = (i == j) ? x[i] : NTT(yi, q_j)
21         u0[j] += xqj * ksh0[i,j] mod q_j
22         u1[j] += xqj * ksh1[i,j] mod q_j
23     return (u0, u1)
```

Listing 4.2: Key-switch implementation. `RVec` is an N-element vector of 32-bit values, storing a single RNS polynomial in either the coefficient or the NTT domain.

scratchpad space. The ordering used in `keySwitch2` computes a single value of `y` (called `yi`) on each iteration of the outer loop. This only requires a single residue vector of scratchpad space, making this implementation more scratchpad-space efficient. However, as F1 is a highly parallel system, we must also consider the implications of parallelizing these functions. If we parallelize the outer loop in `keySwitch1` and send each iteration to a different compute cluster, then only elements of `y` need to be broadcast between different clusters. On the other hand, if we parallelize the outer loop of `keySwitch2`, then all accumulations into `u0` and `u1` need to be sent across the network, as each cluster must access each element in `u0` and `u1`. The F1 compiler uses an implementation very similar to `keySwitch1`, as we find that in practice, the additional footprint introduced is not a problem, and the reduction of on-chip data movement is more beneficial. Other dataflows are possible and are being considered for future versions of the F1 compiler. We also select an instruction order for modulo reduction, which in addition to key-switching, is the only other non-trivial primitive operation that admits multiple instruction orderings.

Algorithmic Choice: The compiler also exploits *algorithmic choice* in generating the dataflow graph. Specifically, there are multiple implementations of key-switching

(Section 2.4), and the right choice depends on L, the amount of key-switch reuse, and load on FUs. In addition to the key-switching algorithm described in detail in Section 2.4, we support an alternative method which uses key-switch hints of size $2(L + 73)$, but requires substantially mode computation [34]. The compiler leverages knowledge of operation order to estimate these and choose the right variant.

One place where this choice is particularly effective is in bootstrapping benchmarks, where we have a single long chain of automorphisms at high values of L. Each automorphism requires a different key-switch hint and we have a single chain. Therefore, there is no reuse of key-switch hints. Additionally, because we are performing computations at high L, our key-switch hints are very large (sometimes > 64 megabytes). By switching to the alternative key-switching method, we can minimize our off-chip communication and obtain speedups as well as significant energy usage reductions.

4.4 Scheduling data transfers

The second compiler phase consumes an instruction-level dataflow graph and produces an approximate schedule that includes data transfers decoupled from computation, minimizes off-chip data transfers, and achieves good parallelism. This requires solving an interdependent problem: when to bring a value into the scratchpad and which one to replace depends on the computation schedule; and to prevent stalls, the computation schedule depends on which values are in the scratchpad. To solve this problem, this scheduler uses a simplified model of the machine: it does not consider on-chip data movement, and simply treats all functional units as being directly connected to the scratchpad with no read latency or throughput constraints. It also does not consider the availability of bandwidth for store instructions.

The scheduler is greedy, scheduling one instruction at a time. It considers instructions ready if their inputs are available in the scratchpad, and follows instruction priority among ready ones. To schedule loads, we assign each load a priority

$$p(\text{load}) = \max\{p(u) | u \in users(\text{load})\},$$

then greedily issue loads as bandwidth becomes available. When issuing an instruction, we must ensure that there is space to store its result. We can often replace a dead

value. When no such value exists, we evict the value with the furthest expected time to reuse. We estimate time to reuse as the maximum priority among unissued users of the value. This approximates Belady's optimal replacement policy [9]. Evictions of dirty data add stores to the dataflow graph. When evicting a value, we add spill (either dirty or clean) and fill instructions to our dataflow graph.

4.5 Cycle-level scheduling

Finally, the cycle-level scheduler takes in the data movement schedule produced by the previous phase, and schedules all operations for all components considering all resource constraints and data dependences. This phase distributes computation across clusters and manages their register files and all on-chip transfers. Importantly, this scheduler is fully constrained by its input schedule's off-chip data movement. It does not add loads or stores in this stage, but it does move loads to their earliest possible issue cycle to avoid stalls on missing operands. All resource hazards are resolved by stalling. In practice, we find that this separation of scheduling into data movement and instruction scheduling produces good schedules in reasonable compilation times.

This stage works by iterating through all instructions in the order produced by the previous compiler phase (Section 4.4) and determining the minimum cycle at which all on-chip resources are required. We consider the availability of off-chip bandwidth, scratchpad space, register file space, functional units, read ports, and write ports. Scheduling a single instruction may require reserving numerous resources. For instance, bringing operands to a particular cluster requires reserving a read port at the data source and a write port at the data destination in addition to a landing register. Furthermore, we may need to evict a live value from the landing register, requiring even more on-chip resources.

Our highly banked scratchpad (Section 3.3) and many-ported register files (Section 3.6) make the task of on-chip scheduling substantially simpler. Each resource constraint has the possibility of stalling an instruction, a fact that is mitigated by our large number of ports.

During this final compiler pass, we also finally account for store bandwidth, scheduling stores (which result from spills) as needed. In practice, we find that this

does not hurt our performance much, as stores are infrequent across most of our benchmarks due to our global schedule and replacement policy design. After the final schedule is generated, we validate it by simulating it forward to ensure that no clobbers or resource usage violations occur.

It is important to note that because our schedules are fully static, our scheduler also doubles as a performance measurement tool. As illustrated in Figure 4-1, the compiler takes in an architecture description file detailing a particular configuration of F1. This flexibility allows us to conduct design space explorations very quickly (Section 5.5).

4.6 Limitations

Our compiler is designed to maximize reuse of on-chip data and minimize off-chip traffic for key-switch hint dominated workloads. While it performs this task successfully, it suffers some inefficiencies; we leave addressing these inefficiencies to future work.

First, min-cycle scheduling greedily makes decisions about where to place data and schedule computations one instruction at a time. This often leads to globally suboptimal schedules with unnecessary on-chip data movement. For instance, we find that NTTs are often computed in one cluster and the results are sent to a different cluster to be multiplied by key-switch hints. In most cases, this is energy-inefficient and can also lead to the addition of further stall cycles in the min-cycle scheduler.

Additionally, taking a single-instruction greedy view of scheduling disallows us from using vector chaining. This leads to additional and unnecessary register-file reads and writes which consume additional energy and introduce stall cycles.

Finally, another compiler limitation is that our key-switch hint based global scheduler assumes that key-switch hints will dominate off-chip data movement. This assumption holds true in computations at higher values of L (recall, KSH size of $O(L^2)$ for operations at a given value of L) but with low values of L, key-switch hints are not as dominant. Thus, though prioritizing key-switch hints is the right overall heuristic, it causes some additional data movement in some benchmarks (Section 5.3.1).

Chapter 5

Results

Evaluating F1 was done in collaboration with Nikola Samardzic, Aleksandar Krastev, Ron Dreslinski, and Daniel Sanchez. The work on RTL implementation and synthesis was conducted by Nikola Samardzic, Aleksandar Krastev, and Ron Dreslinski, while the work on cycle-accurate simulation was performed primarily by the author.

5.1 Experimental Methodology

5.1.1 Modeled system

We evaluate our F1 implementation from Chapter 5. We use a cycle-accurate simulator to execute F1 programs. Because the architecture is static, this is very different from conventional simulators, and acts more as a checker: it runs the instruction stream at each component and verifies that latencies are as expected and there are no missed dependences or structural hazards. We use activity-level energies from RTL synthesis to produce energy breakdowns.

F1's components were implemented in RTL, and synthesized in a commercial 14/12nm process using state-of-the-art tools. These include a commercial SRAM compiler that we use for scratchpad and register file banks.

We use a dual-frequency design: most components run at 1 GHz, but memories (register files and scratchpads) run double-pumped at 2 GHz. Memories meet this frequency easily and this enable using single-ported SRAMs while serving up to two

accesses per cycle. By keeping most of the logic at 1 GHz, we achieve higher energy efficiency. We explored several non-blocking on-chip networks (Clos, Benes, and crossbars). We use 3 16×16 bit-sliced crossbars [57] (scratchpad→cluster, cluster→scratchpad, and cluster→cluster).

Table 5.1 shows a breakdown of area by component, as well as the area of our F1 configuration, 151.43 mm^2. FUs take 42% of the area, with 31.7% going to memory, 6.6% to the on-chip network, and 19.7% to the two HBM2 PHYs (with 512 GB/s per PHY, as in Ampere GPUs [19]). We use prior work to estimate HBM2 PHY area [1, 25] and power [1, 32].

Component	Area [mm^2]	TDP [W]
NTT FU	2.27	4.80
Automorphism FU	0.58	0.99
Multiply FU	0.25	0.60
Add FU	0.03	0.05
Vector RegFile (512 KB)	0.56	1.67
Compute cluster (NTT, Aut, 2× Mul, 2× Add, RF)	3.97	8.75
Total compute (16 clusters)	**63.52**	**140**
Scratchpad (16×4 MB banks)	48.09	20.35
3×NoC (16×16 512 B bit-sliced [57])	10.02	19.65
Memory interface (2×HBM2 PHYs)	29.8	0.45
Total memory system	**87.91**	**40.45**
Total F1	**151.43**	**180.45**

Table 5.1: Area and Thermal Design Power (TDP) of F1, and breakdown by component.

This design is constrained by memory bandwidth: though it has 1 TB/s of bandwidth, the on-chip network's bandwidth is 24 TB/s, and the aggregate bandwidth between RFs and FUs is 128 TB/s. This is why maximizing reuse is crucial.

5.1.2 Benchmarks

We use several FHE programs to evaluate F1. All programs come from state-of-the-art software implementations, which we port to F1:

Logistic regression uses the HELR algorithm [40], which is based on CKKS. We compute a single batch of logistic regression training with up to 256 features, and 256 samples per batch, starting at computational depth $L = 16$; this is equivalent to the

first batch of HELR's MNIST workload. This computation features ciphertexts with large $\log Q$ ($L = 14, 15, 16$), so it needs careful data orchestration to run efficiently.

Neural network benchmarks come from Low Latency CryptoNets (LoLa) [16]. This work uses B/FV, an FHE scheme that F1 does not support, so we use BGV instead. We run two neural networks: LoLa-MNIST is a simple, LeNet-style network used on the MNIST dataset [48], while LoLa-CIFAR is a much larger 6-layer network (similar in computation to MobileNet v3 [41]) used on the CIFAR-10 dataset [46]. LoLa-MNIST includes two variants with unencrypted and encrypted weights; LoLa-CIFAR is available only with unencrypted weights. These three benchmarks use relatively low L values (their starting L values are 4, 6, and 8, respectively), so they are less memory-bound. They also feature frequent automorphisms, showing the need for a fast automorphism unit.

Due to the huge size of the LoLa-CIFAR (approx. 47 million instructions), we scheduled multiple scaled down versions (fewer output maps on the second layer), and linearly estimate the performance of the full benchmark to reduce scheduling time. We find that our reduced versions fit our linear model nearly perfectly.

Bootstrapping: We evaluate bootstrapping benchmarks for BGV and CKKS. Bootstrapping takes an $L = 1$ ciphertext with an exhausted noise budget and refreshes it by bringing it up to a chosen top value of $L = L_{max}$, then performing the bootstrapping computation to eventually obtain a usable ciphertext at a lower depth (e.g., $L_{max} - 15$ for BGV).

For BGV, we use Sheriff and Peikert's algorithm [4] for non-packed BGV bootstrapping, with $L_{max} = 24$. This is a particularly challenging benchmark because it features computations at large values of L. This exercises the scheduler's algorithmic choice component, which selects the right key-switch method to balance computation and data movement.

For CKKS, we use non-packed CKKS bootstrapping from HEAAN [17], also with $L_{max} = 24$. CKKS bootstrapping has many fewer ciphertext multiplications than BGV, greatly reducing reuse opportunities for key-switch hints.

Execution time (ms) on	CPU	F1	Speedup
LoLa-CIFAR Unencryp. Wghts.	1.2×10^6	**241**	$5,011\times$
LoLa-MNIST Unencryp. Wghts.	$2,960$	**0.17**	$17,412\times$
LoLa-MNIST Encryp. Wghts.	$5,431$	**0.36**	$15,086\times$
Logistic Regression	$8,300$	**1.15**	$7,217\times$
BGV Bootstrapping	—[2]	**1.8**	—[2]
CKKS Bootstrapping	$1,554$	**1.3**	$1,195\times$
gmean speedup			6,471×

[1]LoLa's release did not include MNIST with encrypted weights, so we reimplemented it in HELib.
[2]BGV bootstrapping in HELib crashes for this input, and we could not find an alternative implementation or fix HELib. We expect F1's speedup to be at least 3,000×.

Table 5.2: Performance of F1 and CPU on full FHE benchmarks: execution times in milliseconds and F1's speedup.

5.1.3 Baseline systems

We compare F1 with a CPU system running the baseline programs (a 4-core, 8-thread, 3.5 GHz Xeon E3-1240v5). Since prior accelerators do not support full programs, we also include microbenchmarks of single operations and compare against HEAX [62], the fastest prior accelerator.

5.2 Performance

5.2.1 Benchmarks

Table 5.2 compares the performance of F1 and the CPU on full benchmarks. It reports execution time in milliseconds for each program (lower is better), and F1's speedup over the CPU (higher is better). F1 achieves dramatic speedups, from 1,195× to 17,412× (6,471× gmean). CKKS bootstrapping has the lowest speedups as it's highly memory-bound; other speedups are within a relatively narrow band, as compute and memory traffic are more balanced.

These speedups greatly expand the applicability of FHE. Consider deep learning: in software, even the simple LoLa-MNIST network takes seconds per inference, and a single inference on the more realistic LoLa-CIFAR network takes *20 minutes*. F1 brings this down to 241 *milliseconds*, making real-time deep learning inference practical: when offloading inferences to a server, this time is comparable to the roundtrip latency between server and client.

	$N = 2^{12}$, $\log Q = 109$			$N = 2^{13}$, $\log Q = 218$			$N = 2^{14}$, $\log Q = 438$		
	F1	vs. CPU	vs. HEAX$_\sigma$	**F1**	vs. CPU	vs. HEAX$_\sigma$	**F1**	vs. CPU	vs. HEAX$_\sigma$
NTT	**12.8**	17,148×	1,600×	**44.8**	10,736×	1,733×	**179.2**	8,838×	1,866×
Automorphism	**12.8**	7,364×	440×	**44.8**	8,250×	426×	**179.2**	16,957×	430×
Homomorphic multiply	**60**	48,640×	172×	**300**	27,069×	148×	**2,000**	14,396×	190×
Homomorphic permutation	**40**	17,488×	256×	**224**	10,814×	198×	**1,680**	6,421×	227×

Table 5.3: Performance on microbenchmarks: F1's **reciprocal throughput, in nanoseconds per ciphertext operation** (lower is better) and speedups over CPU and HEAX$_\sigma$ (HEAX augmented with scalar automorphism units) (higher is better).

5.2.2 Microbenchmarks

Table 5.3 compares the performance of F1, the CPU, and HEAX$_\sigma$ on four microbenchmarks: the basic NTT and automorphism operations on a single ciphertext, and homomorphic multiplication and permutation (which uses automorphisms). We report three typical sets of parameters. We use microbenchmarks to compare against prior accelerators, in particular HEAX. But prior accelerators do not implement automorphisms, so we extend each HEAX key-switching pipeline with an SRAM-based, scalar automorphism unit. We call this extension HEAX$_\sigma$.

Table 5.3 shows that F1 achieves large speedups over HEAX$_\sigma$, ranging from 172×to 1,866×. Moreover, F1's speedups over the CPU are even larger than in full benchmarks. This is because microbenchmarks are pure compute, and thus miss the data movement bottlenecks of FHE programs.

5.3 Architectural analysis

To gain more insights into these results, we now analyze the F1's data movement, power consumption, and compute.

5.3.1 Data movement

5-1a shows a breakdown of off-chip memory traffic across data types: key-switch hints (KSH), inputs/outputs, and intermediate values. KSH and input/output traffic is broken into compulsory and non-compulsory (i.e., caused by limited scratchpad capacity). Intermediates, which are always non-compulsory, are classified as loads or stores.

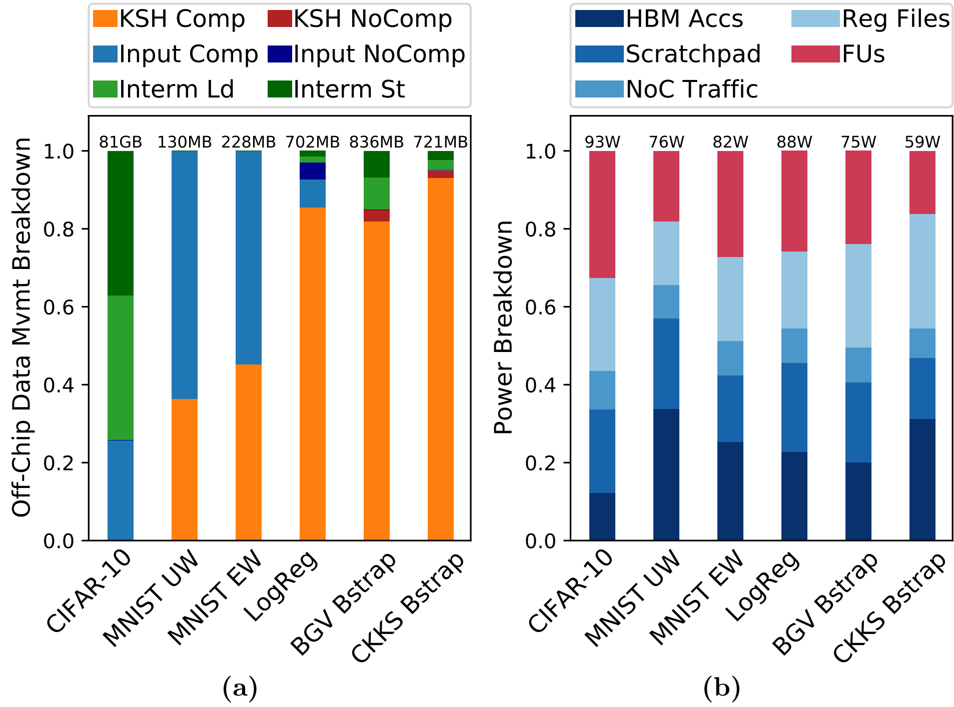

Figure 5-1: Per-benchmark breakdowns of **(a)** data movement and **(b)** average power for F1.

5-1a shows that key-switch hints dominate in high-depth workloads (LogReg and bootstrapping), taking up to 94% of traffic. Key-switch hints are also significant in the LoLa-MNIST variants. This shows why scheduling should prioritize them. Second, due our scheduler design, F1 approaches compulsory traffic for most benchmarks, with non-compulsory accesses adding only 5-18% of traffic. The exception is LoLa-CIFAR, where intermediates consume 75% of traffic. LoLa-CIFAR has very high reuse of key-switch hints, and exploiting it requires spilling intermediate ciphertexts.

5.3.2 Power consumption

5-1b reports average power for each benchmark, broken down by component. This breakdown also includes off-chip memory power (Table 5.1 only included the on-chip component). Results show reasonable power consumption for an accelerator card. Overall, computation consumes 20-30% of power, and data movement dominates.

5.3.3 Utilization over time

F1's average FU utilization is about 30%. However, this doesn't mean that fewer FUs could achieve the same performance: benchmarks have memory-bound phases that weigh down average FU utilization. To see this, Figure 5-2 shows a breakdown of FU utilization over time for LoLa-MNIST Plaintext Weights. Figure 5-2 also shows off-chip bandwidth utilization over time (black line). The program is initially memory-bound, and few FUs are active. As the memory-bound phase ends, compute intensity grows, utilizing a balanced mix of the available FUs. Finally, due to decoupled execution, when memory bandwidth use peaks again, F1 can maintain high compute intensity.

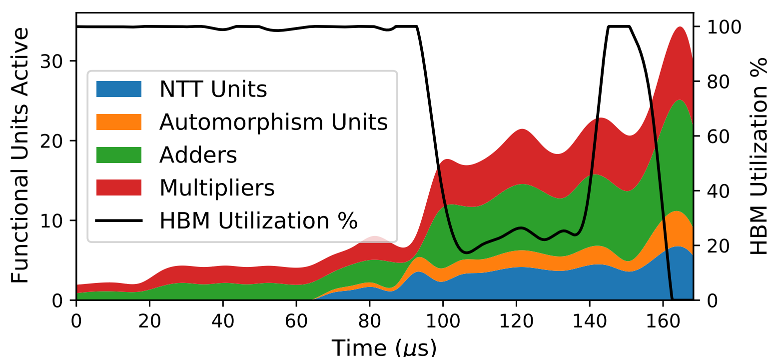

Figure 5-2: Functional unit and HBM utilization over time for the LoLa-MNIST PTW benchmark.

5.4 Sensitivity studies

To understand the impact of our FUs and scheduling algorithms, we evaluate F1 variants without them. Table 5.4 reports the *slowdown (higher is worse)* of F1 with: *(1)* low-throughput NTT FUs that follow the same design as HEAX (processing one stage of NTT butterflies per cycle); *(2)* low-throughput automorphism FUs using a serial SRAM memory, and *(3)* Goodman's register-pressure-aware scheduler [37].

For the FU experiments, our goal is to show the importance of having high-throughput units. Therefore, the low-throughput variants use many more (NTT or automorphism) FUs, so that aggregate throughput across all FUs in the system is the

Benchmark	LT NTT	LT Aut	CSR
LoLa-CIFAR Unencryp. Wghts.	3.5×	12.1×	—[1]
LoLa-MNIST Unencryp. Wghts.	5.0×	4.2×	1.1×
LoLa-MNIST Encryp. Wghts.	5.1×	11.9×	7.5×
Logistic Regression	1.7×	2.3×	11.7×
BGV Bootstrapping	1.6×	1.1×	5.4×
CKKS Bootstrapping	1.1×	1.2×	2.7×
gmean speedup	2.5×	5.5×	4.2×

[1]CSR is intractable for this benchmark.

Table 5.4: Speedups of F1 over alternate configurations: LT NTT/Aut = Low-throughput NTT/Automorphism FUs; CSR = Code Scheduling to minimize Register Usage [37].

same. Also, the scheduler accounts for the characteristics of these FUs. In both cases, performance drops substantially, by gmean 2.5× and 5.5×. This is because achieving high throughput requires excessive parallelism, which hinders data movement, forcing the scheduler to balance both.

Finally, the scheduler experiment uses register-pressure-aware scheduling as the off-chip data movement scheduler instead, operating on the full dataflow graph. This algorithm was proposed for VLIW processors and register files; we apply it to the larger scratchpad. The large slowdowns show that prior capacity-aware schedulers are ineffective on F1.

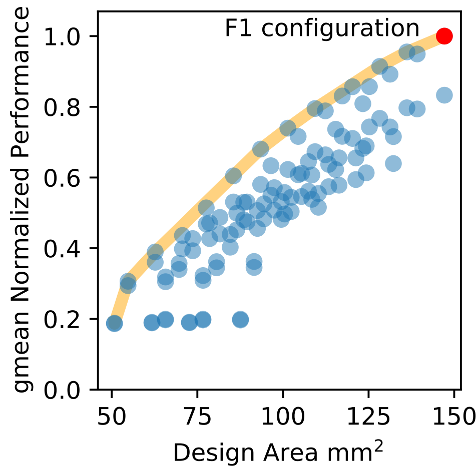

Figure 5-3: Performance vs. area across F1 configurations.

5.5 Scalability

Finally, we study how F1's performance changes with its area budget: we sweep the number of compute clusters, scratchpad banks, HBM controllers, and network topology to find the most efficient design at each area. Figure 5-3 shows this Pareto frontier, with area in the x-axis and performance in the y-axis. This curve shows that, as F1 scales, it uses resources efficiently: performance grows about linearly through a large range of areas.

Chapter 6

Related Work

We now discuss related work not covered so far.

6.1 FHE accelerators

Prior work has proposed accelerators for individual FHE operations, but not full FHE computations [21, 22, 23, 28, 52, 52, 53, 62, 63, 67]. These designs target FPGAs and rely on a host processor; Section 2.4 discussed their limitations. Early designs accelerated small primitives like NTTs, and were dominated by host-FPGA communication. State-of-the-art accelerators execute a full homomorphic multiplication independently: Roy et al. [63] accelerate B/FV multiplication by 13× over a CPU; HEAWS [67] accelerates B/FV multiplication, and uses it to speed a simple benchmark by 5×; and HEAX [62] accelerates CKKS multiplication and key-switching by up to 200×. These designs suffer high data movement (e.g., HEAX does not reuse key-switch hints) and use fixed pipelines with relatively low-throughput FUs.

We have shown that accelerating FHE programs requires a different approach: data movement becomes the key constraint, requiring new techniques to extract reuse across homomorphic operations; and fixed pipelines cannot support the operations of even a single benchmark. Instead, F1 achieves flexibility and high performance by exploiting wide-vector execution with high-throughput FUs. This lets F1 execute not only full applications, but different FHE schemes.

6.2 Hybrid HE-MPC accelerators

Recent work has also proposed ASIC accelerators for some homomorphic encryption primitives in the context of oblivious neural networks [43, 61]. These approaches are very different from FHE: they combine homomorphic encryption with multi-party computation (MPC), executing a single layer of the network at a time and sending intermediates to the client, which computes the final activations. Gazelle [43] is a low-power ASIC for homomorphic evaluations, and Cheetah [61] introduces algorithmic optimizations and a large ASIC design that achieves very large speedups over Gazelle.

These schemes avoid high-depth FHE programs, so server-side homomorphic operations are cheaper. But they are limited by client-side computation and client-server communication. CHOCO [68] shows that client-side computation costs are substantial, and when they are accelerated, network latency and throughput overheads dominate (several seconds per DNN inference). By contrast, F1 enables offloading the full inference using FHE, avoiding frequent communication.

6.3 GPU acceleration

Finally, prior work has also used GPUs to accelerate different FHE schemes, including GH [70, 71], BGV [69], and B/FV [2]. Though GPUs have plentiful compute and bandwidth, they lack modular arithmetic, their pure data-parallel approach makes non-element-wise operations like NTTs expensive, and their small on-chip storage adds data movement. As a result, GPUs achieve only modest performance gains. For instance, Badawi et al. [2] accelerate B/FV multiplication using GPUs, and achieve speedups of around 10× to 100× over single-thread CPU execution.

Chapter 7

Conclusion

FHE has the potential to enable computation offloading with guaranteed security. But FHE's high computation overheads currently limit its applicability to narrow cases (simple computations where privacy is paramount). F1 tackles this challenge, accelerating full FHE computations by over 3-4 orders of magnitude. This enables new use cases for FHE, like secure real-time deep learning inference.

F1 is the first FHE accelerator that is programmable, i.e., capable of executing full FHE programs. In contrast to prior accelerators, which build fixed pipelines tailored to specific FHE schemes and parameters, F1 introduces a more effective design approach: it accelerates the *primitive* computations shared by higher-level operations using novel high-throughput functional units, and hardware and compiler are co-designed to minimize data movement, the key bottleneck. This flexibility makes F1 broadly useful: the same hardware can accelerate all operations within a program, arbitrary FHE programs, and even multiple FHE schemes. In short, our key contribution is to show that, for FHE, we can achieve ASIC-level performance without sacrificing programmability.

7.1 Future Work

This work opens up a number of directions for future work:

- *Improving on-chip scheduling:* Our simple min-cycle on-chip scheduler resolves all hazards by stalling and introducing dead cycles. This leads to lower than

desired functional unit utilization. It may be possible to obtain better utilization (and by extension higher speedups) by using a more powerful on-chip scheduling algorithm with more deliberate data and computation placements.

- *Main-memory model:* The current version of the F1 compiler assumes worst-case access times. Having a more accurate memory model could allow the compiler to be more aggressive in scheduling computation to obtain larger speedups, but may also require some hardware stalling mechanism when loads take longer than expected.
- *Global scheduling for low-L computations:* Our current work assumes that key-switch hint reuse is critical to all FHE computations. However, for computations at low values of L, key-switch hints are not as dominant. Better scheduling algorithms are possible for computations where key-switch hint reuse is not as dominant.
- *Hybrid HE-MPC:* F1 is designed around fully offloading entire FHE computations. However, as described in Section 6.2, HE-MPC accelerators are also an active area of research. Future work could investigate if some variant of F1 can work well within this model.

Bibliography

[1] "HBM2E and GDDR6: Memory solutions for AI," *Rambus Inc. White Paper*, 2020.

[2] A. Q. A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IEEE Transactions on Emerging Topics in Computing*, 2019.

[3] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter *et al.*, "Homomorphic encryption security standard," *Toronto, ON, Canada*, 2018.

[4] J. Alperin-Sheriff and C. Peikert, "Practical bootstrapping in quasilinear time," in *Proceedings of the Annual Cryptology Conference (CRYPTO)*, 2013.

[5] D. Altavilla, "Intel and Microsoft Collaborate on DARPA Program that Pioneers A New Frontier Of Ultra-Secure Computing," https://www.forbes.com/sites/davealtavilla/2021/03/08/intel-and-microsoft-collaborate-on-darpa-program-that-pioneers-a-new-frontier-of-ultra-secure-computing/?sh=60db31567c1a archived at https://perma.cc/YYE6-5FT4.

[6] K. Asanovic, "Vector microprocessors," Ph.D. dissertation, EECS Department, University of California, Berkeley, 1998, archived at https://perma.cc/AHR3-AMLG. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/6404.html

[7] D. H. Bailey, "FFTs in external of hierarchical memory," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, 1989.

[8] G. Barany, "Register reuse scheduling," in *9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9), Chamonix, France*, 2011.

[9] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, 1966.

[10] F. Bergamaschi, "IBM Releases Fully Homomorphic Encryption Toolkit for MacOS and iOS," https://www.ibm.com/blogs/research/2020/06/ibm-releases-fully-homomorphic-encryption-toolkit-for-macos-and-ios-linux-and-android-coming-soon/ archived at https://perma.cc/U5TQ-K49C.

[11] D. A. Berson, R. Gupta, and M. L. Soffa, "URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures." in *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, 1993.

[12] M. Blatt, A. Gusev, Y. Polyakov, and S. Goldwasser, "Secure large-scale genome-wide association studies using homomorphic encryption," *Proceedings of the National Academy of Sciences*, vol. 117, no. 21, 2020.

[13] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM (JACM)*, vol. 46, no. 2, 1999.

[14] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Proceedings of the Annual Cryptology Conference (CRYPTO)*, 2012.

[15] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, 2014.

[16] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2019.

[17] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2018.

[18] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2017.

[19] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *micro21*.

[20] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, 1965.

[21] D. B. Cousins, K. Rohloff, C. Peikert, and R. Schantz, "An update on SIPHER (Scalable Implementation of Primitives for Homomorphic EncRyption) - FPGA implementation using Simulink," in *Proceedings of the IEEE Conference on High Performance Extreme Computing (HPEC)*, 2012.

[22] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an FPGA-accelerated homomorphic encryption co-processor," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, 2017.

[23] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An FPGA co-processor implementation of homomorphic encryption," in *Proceedings of the IEEE Conference on High Performance Extreme Computing (HPEC)*, 2014.

[24] DARPA, "DARPA Selects Researchers to Accelerate Use of Fully Homomorphic Encryption," https://www.darpa.mil/news-events/2021-03-08 archived at https://perma.cc/6GHW-2MSN.

[25] S. Dasgupta, T. Singh, A. Jain, S. Naffziger, D. John, C. Bisht, and P. Jayaraman, "8.4 Radeon RX 5700 Series: The AMD 7nm Energy-Efficient High-Performance GPUs," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2020.

[26] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2020.

[27] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2019.

[28] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Trans. Computers*, vol. 64, no. 6, 2015.

[29] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, 2012.

[30] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proc. of the 10th annual Intl. Symp. on Computer Architecture (ISCA-10)*, 1983.

[31] H. L. Garner, "The residue number system," in *Papers presented at the the March 3-5, 1959, western joint computer conference*, 1959.

[32] W. Ge, M. Zhao, C. Wu, and J. He, "The design and implementation of ddr phy static low-power optimization strategies," in *Communication Systems and Information Technology*, 2011.

[33] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Department of Computer Science, Stanford University, 2009.

[34] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Proceedings of the Annual Cryptology Conference (CRYPTO)*, 2012.

[35] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Proceedings of the Annual Cryptology Conference (CRYPTO)*, 2013.

[36] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[37] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 1988.

[38] S. Halevi and V. Shoup, "Algorithms in helib," in *Proceedings of the Annual Cryptology Conference (CRYPTO)*, 2014.

[39] K. Han, S. Hong, J. H. Cheon, and D. Park, "Efficient logistic regression on large encrypted data." *IACR Cryptol. ePrint Arch.*, vol. 2018, 2018.

[40] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019.

[41] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019.

[42] IBM, "Cost of a Data Breach Report," Tech. Rep., 2020.

[43] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[44] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Proceedings of the USSR Academy of Sciences*, vol. 145, no. 2, 1962.

[45] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang, "Secure logistic regression based on homomorphic encryption: Design and evaluation," *JMIR medical informatics*, vol. 6, no. 2, 2018.

[46] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[47] M. S. Lam, "Software pipelining," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1988.

[48] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *IEEE*, vol. 86, no. 11, 1998.

[49] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2010.

[50] L. Marchal, B. Simon, and F. Vivien, "Limiting the memory footprint when dynamically scheduling dags on shared-memory platforms," *Journal of Parallel and Distributed Computing*, vol. 128, 2019.

[51] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture," in *2019 22nd Euromicro Conference on Digital System Design (DSD).* IEEE, 2019.

[52] A. C. Mert, E. Öztürk, and E. Savaş, "Design and Implementation of Encryption/Decryption Architectures for BFV Homomorphic Encryption Scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[53] V. Migliore, C. Seguin, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, G. Gogniat, and R. Tessier, "A high-speed accelerator for homomorphic encryption using the karatsuba algorithm," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, 2017.

[54] R. T. Moenck, "Practical fast polynomial multiplication," in *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, 1976.

[55] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, 1985.

[56] E. Ozer, S. Banerjia, and T. M. Conte, "Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures," in *Proc. of the 31st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-31)*, 1998.

[57] G. Passas, M. Katevenis, and D. Pnevmatikatos, "Crossbar NoCs are scalable beyond 100 nodes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 4, 2012.

[58] C. Peikert, "A decade of lattice cryptography," *Foundations and Trends in Theoretical Computer Science*, vol. 10, no. 4, 2016.

[59] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proc. of the 24th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019.

[60] Y. Polyakov, K. Rohloff, and G. W. Ryan, "Palisade lattice cryptography library user manual," *Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep*, vol. 15, 2017.

[61] B. Reagen, W. Choi, Y. Ko, V. Lee, G.-Y. Wei, H.-H. S. Lee, and D. Brooks, "Cheetah: Optimizations and methods for privacy preserving inference via homomorphic encryption," 2021.

[62] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020.

[63] S. S. Roy, F. Turan, K. Järvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *Proc. of the 25th IEEE intl. symp. on High Performance Computer Architecture (HPCA-25)*, 2019.

[64] Q. Shang, Y. Fan, W. Shen, S. Shen, and X. Zeng, "Single-port sram-based transpose memory with diagonal data mapping for large size 2-d dct/idct," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 11, 2014.

[65] Z. M. Smith, E. Lostri, and J. A. Lewis, "The Hidden Costs of Cybercrime," Center for Strategic and International Studies, Tech. Rep., 2020.

[66] S. A. A. Touati, "Register saturation in instruction level parallelism," *International Journal of Parallel Programming*, vol. 33, 2005.

[67] F. Turan, S. Roy, and I. Verbauwhede, "HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA," *IEEE Transactions on Computers*, 2020.

[68] M. van der Hagen and B. Lucia, "Practical encrypted computing for iot clients," *arXiv preprint arXiv:2103.06743*, 2021.

[69] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using gpu," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014.

[70] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using gpu," in *Proceedings of the IEEE Conference on High Performance Extreme Computing (HPEC)*. IEEE, 2012.

[71] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 64, no. 3, 2013.

[72] Y. Wang, Z. Ma, and F. Yu, "Pipelined algorithm and modular architecture for matrix transposition," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 4, 2018.

[73] W. Xu and R. Tessier, "Tetris: a new register pressure control technique for VLIW processors," *ACM SIGPLAN Notices*, vol. 42, no. 7, 2007.

[74] B. Zhang, Z. Ma, and F. Yu, "A novel pipelined algorithm and modular architecture for non-square matrix transposition," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.