F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.


As Published: 10.1145/3466752.3480070

Publisher: ACM

Persistent URL: https://hdl.handle.net/1721.1/143470

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of use: a Creative Commons Attribution-Share Alike International 4.0 License
F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption

Axel Feldmann1, Nikola Samardzic1, Aleksandar Krastev1, Srini Devadas1, Ron Dreslinski2, Christopher Peikert2, Daniel Sanchez1

1 Massachusetts Institute of Technology 2 University of Michigan

{axelf, nsamar, alexalex, devadas, sanchez}@csail.mit.edu {dreslin, cpeikert}@umich.edu

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. This organization provides so much compute throughput that data movement becomes the key bottleneck. Thus, F1 is primarily designed to minimize data movement. Hardware provides an explicitly managed memory hierarchy and mechanisms to decouple data movement from execution. A novel compiler leverages these mechanisms to maximize reuse and schedule off-chip and on-chip data movement.

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 5,400x and by up to 17,000x. These speedups counter most of FHE’s overheads and enable new applications, like real-time private deep learning in the cloud.

CCS CONCEPTS

• Computer systems organization → Parallel architectures;
• Security and privacy → Cryptography.

KEYWORDS

fully homomorphic encryption, hardware acceleration.

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 [43, 69]. 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. Fig. 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.

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

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 [2, 59]. FHE has inherent limitations—for example, data-dependent branching is impossible, since data is encrypted—so it won’t subsume all computations. Nonetheless, important classes of computations, like deep learning inference [17, 25, 26], linear algebra, and other inference and learning tasks [40] are a good fit for FHE. This has sparked significant industry and government investments [4, 9, 23] to widely deploy FHE.
Unfortunately, FHE still carries substantial performance overheads: despite recent advances [15, 25, 26, 61, 66], FHE is still $10,000 \times$ to $100,000 \times$ 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 [20, 22, 27, 65, 66, 71] target individual FHE operations, and miss important ones that they leave to software. These designs are FPGA-based, so they are small and miss the data movement issues facing an FHE 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 paper 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 MBs) 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 bottleneck 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 specialized functional units (Sec. 5) and scheduling algorithms (Sec. 4) to design a programmable FHE accelerator. We implement the main components of F1 in RTL and synthesize them in a commercial 14nm/12nm process. With a modest area budget of 151 mm², 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.

2 BACKGROUND

Fully Homomorphic Encryption allows 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 with somewhat different capabilities and performance tradeoffs. BGV [14], B/FV [13, 28], GSW [35], and CKKS [17] are popular FHE schemes.† 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: Sec. 2.1 introduces FHE’s programming model and operations, i.e., FHE’s *interface*; Sec. 2.2 describes how FHE operations are *implemented*; Sec. 2.3 presents implementation *optimizations*; and Sec. 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 Sec. 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. Data values are 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 [14], each vector consists of $N$ integers modulo an integer $t$. BGV provides three operations on these vectors: element-wise

†These scheme names are acronyms of their authors’ last names. For instance, BGV is Brakerski-Gentry-Vaikuntanathan.
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 Sec. 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 where 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 [15, 36], logistic regression [39], and genome processing [11]. These implementations are often coded by hand, but recent work has proposed FHE compilers to automate this translation for particular domains, like deep learning [25, 26].

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 [15].

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

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

is a plaintext. Each plaintext is encrypted into a ciphertext consisting of two polynomials of $N$ integer coefficients modulo some $Q \gg t$. 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 $s \in R_Q$. To encrypt a plaintext $m \in R_t$, one samples a uniformly random $a \in R_Q$, an error (or noise) $e \in R_Q$ with small entries, and computes the ciphertext $ct$ as

$$ct = (a, b = as + te + m)$$

Ciphertext $ct = (a, b)$ is decrypted by recovering $e' = te + m = b - as \mod Q$, and then recovering $m = e' \mod t$. Decryption is correct as long as $e'$ 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 $e$, the original message $m$ 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, homomorphic operations on ciphertexts increase 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 (Sec. 2.2.2) to keep this noise bounded and thereby allow unlimited operations.

2.2.1 Homomorphic operations.

Homomorphic addition of ciphertexts $ct_0 = (a_0, b_0)$ and $ct_1 = (a_1, b_1)$ is done simply by adding their corresponding polynomials: $ct_{add} = ct_0 + ct_1 = (a_0 + a_1, b_0 + b_1)$.

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

$$ct_k = (l_2, l_0) = (a_0 a_1, a_0 b_1 + a_1 b_0, b_0 b_1).$$

This $ct_k$ 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 $s$. More specifically, $l_2$ undergoes this key-switching process to produce two polynomials $(u_1, u_0) = \text{KeySwitch}(l_2)$. The final output ciphertext is $ct_{mul} = (l_1 + u_1, l_0 + u_0)$.

As we will see later (Sec. 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 $\sigma_k(a)$ and $\sigma_{-k}(a)$ for all positive odd $k < N$. Specifically,

$$\sigma_k(a) : a_i \mapsto (-1)^k a_{ik} \mod N \text{ for } i = 0, \ldots, N - 1,$$

where $s = 0$ if $ik \mod 2N < N$, and $s = 1$ otherwise. For example, $\sigma_2(a)$ permutes $a$’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 $205 \cdot 5 \mod 1024 = 1$).

To perform a homomorphic permutation, we first compute an automorphism on the ciphertext polynomials: $ct_a = (\sigma_k(a), \sigma_2(b))$. Just as in homomorphic multiplication, $ct_a$ is encrypted under a different secret key, requiring an expensive key-switch to produce the final output $ct_{perm} = (u_1, \sigma_k(b) + u_0)$, where $(u_1, u_0) = \text{KeySwitch}(\sigma_k(a))$.

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.

2.2.2 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, the amount of noise 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 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. This is accomplished by evaluating the decryption function homomorphically. Bootstrapping is an expensive procedure that consists of many (typically tens to hundreds) homomorphic operations. 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 computationally 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 circuits 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 32 bits shorter. 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.

### 2.2.3 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, this means that using a wide modulus to support deep programs also requires a large \( N \). For example, with 512-bit \( Q \), \( N = 16k \) is required to provide an acceptable level of security, resulting in very large ciphertexts.

### 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. Polynomial multiplication can be performed as element-wise multiplication in the NTT domain. Specifically,

\[
NTT(ab) = NTT(a) \odot NTT(b),
\]

where \( \odot \) denotes component-wise multiplication. (For this relation to hold with \( N \)-point NTTs, a negacyclic NTT [49] must be used (Sec. 5.2).)

Because an NTT requires only \( O(N \log N) \) modular operations, multiplication can be performed in \( O(N \log N) \) operations by using two forward NTTs, element-wise multiplication, and an inverse NTT.

```python
1 def keySwitch(x: RVec[L],
2 ksh0: RVec[L][L], ksh1: RVec[L][L]):
3 y = \{NTT(x[i], q) for i in range(L)}
4 u0: RVec[L] = \{0,...\}
5 ul: RVec[L] = \{0,...\}
6 for i in range(L):
7    for j in range(L):
8        xqj = (i == j) * x[i] \cdot NTT(y[i], q)
9        u0(j) += xqj * ksh0[i,j] mod q,
10       ul[j] += xqj * ksh1[i,j] mod q
11 return (u0, ul)
```

**Listing 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.

NTT. And in fact, optimized FHE implementations often store polynomials in the NTT domain rather than in their coefficient form across operations, further reducing the number of NTTs. This is possible because the NTT is a linear transformation, so additions and automorphisms can also be performed in the NTT domain:

\[
NTT(\sigma_q(a)) = \sigma_q(NTT(a))
\]

\[
NTT(a + b) = NTT(a) + NTT(b)
\]

**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. 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 chosen 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 \( R_{q_i} \), 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 \( L = \log Q / \log W = 16 \) 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 majority of work in all of our benchmarks.

Listing 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; only the \( y[i] \) polynomials (line 3) are in coefficient form.

**Computation vs. data movement:** A single key-switch requires \( L^2 \) NTTs, \( 2L^2 \) 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 (Sec. 2.2), so key-switching is dominant.

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 text 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. 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 there is no effective way to decompose or tile this operation to reduce storage needs while achieving good reuse: tiling the key-switch hint matrices on either dimension produces many long-lived intermediate values; and tiling across RVec elements is even worse because in NTTs every input element affects every output element.

**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., number of 32-bit prime factors 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 \(L\) (~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 [20, 22, 27, 52, 53, 65, 66, 71]. 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 [65] 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 [17] supports "approximate" computation on fixed-point values. B/FV [13, 28] encodes plaintexts in a way that makes modulus switching before homomorphic multiplication unnecessary, thus easing programming (but forgoing the efficiency gains of modolo switching). And GSW [35] features reduced, asymmetric noise growth under homomorphic multiplication, but encrypts a small amount of information per ciphertext (not a full \(N/2\)-element vector).

Because F1 accelerates primitive operations rather than full homomorphic 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.

### 3 F1 ARCHITECTURE

Fig. 2 shows an overview of F1, which we derive from the insights in Sec. 2.4.

**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. Sec. 5 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. This is important because, as we saw in Sec. 2.4, having fewer high-throughput FUs reduces parallelism and thus memory footprint.
Figure 2: Overview of the F1 architecture.

**Compute clusters:** Functional units are grouped in compute clusters, as Fig. 2 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).

**Memory system:** F1 features an explicitly managed memory hierarchy. As Fig. 2 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 [60] 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 [58] 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.

**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 processors [30].

Static scheduling 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).

**4 SCHEDULING DATA AND COMPUTATION**

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

Fig. 3 shows an overview of the F1 compiler. The compiler takes as input an FHE program written in a high-level domain specific language (Sec. 4.1). 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...
4.1 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 2 shows the code for matrix-vector multiplication.

This follows HELib’s algorithm [38], which Fig. 4 shows. This toy 4×16K 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.

Listing 2: (4 × 16K) matrix-vector multiply in F1’s DSL.

As Listing 2 shows, programs in this DSL are at the level of the simple FHE interface presented in Sec. 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.

4.2 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.

Ordering homomorphic operations seeks to maximize the reuse of key-switch hints, which is crucial to reduce data movement (Sec. 2.4). For instance, the program in Listing 2 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 MB, 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. This generates schedules with good key-switch hint reuse.

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 intermediates. The end result is an instruction-level dataflow graph where every instruction is tagged with a priority that reflects its global order.

The compiler exploits algorithmic choice. Specifically, there are multiple implementations of key-switching (Sec. 2.4), and the right choice depends on L, the amount of key-switch reuse, and load on FUs. The compiler leverages knowledge of operation order to estimate these and choose the right variant.

4.3 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.

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

\[ p(\text{load}) = \max(p(u)|u \in \text{users(\text{load}))}, \]
	hen 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 \[8\]. 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.

4.4 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 dependencies. 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 (Sec. 4.3) and determining the minimum cycle at which all required on-chip resources are available. We consider the availability of off-chip bandwidth, scratchpad space, register file space, functional units, and ports.

During this final compiler pass, we 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 Fig. 3, 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 (Sec. 8.4).

5 FUNCTIONAL UNITS

In this section, we describe F1’s novel functional units. These include the first vectorized automorphism unit (Sec. 5.1), the first fully-pipelined flexible NTT unit (Sec. 5.2), and a new simplified modular multiplier adapted to FHE (Sec. 5.3).

5.1 Automorphism unit

Because F1 uses \( E \) vector lanes, each residue polynomial is stored and processed as \( G \) groups, or chunks, of \( E \) elements each (\( N = G \cdot E \)). An automorphism \( \sigma_k \) maps the element at index \( i \) to index \( ki \mod N \); there are \( N \) automorphisms total, two for each odd \( k < N \) (Sec. 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 chunks. Moreover, we must support variable \( N \) and all automorphisms.

Standard solutions fail: a 16 Kx16 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 the banking scheme, some automorphisms will map many consecutive elements to the same bank.

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 chunks of \( E \) elements. Fig. 5 shows an example of how automorphism \( \sigma_3 \) 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 \( E \)-element chunk. Other \( \sigma_k \) induce different permutations, but with the same row/column structure.

Our automorphism unit, shown in Fig. 6, 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 \( E < G \)). Row permutations are applied to each \( E \)-element chunk, and the reverse transpose is applied.

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

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

The basic building block is a $K \times K$ quadrant-swap unit, which swaps quadrants B and C, as shown in Fig. 7 (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 outputing $A[i]$ (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. Fig. 7 (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 [76, 78] or with custom SRAM designs [68]. Our design has three advantages over prior work: it uses standard SRAM memory, so it is dense without requiring complex custom SRAMs; it is fully pipelined; and it works for a wide range of dimensions.

5.2 Four-step NTT unit

There are many ways to implement NTTs in hardware: an NTT is like an FFT [19] 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.

Figure 8: Example of a four-step NTT datapath that uses 4-point NTTs to implement 16-point NTTs.

To that end, we use the four-step variant of the FFT algorithm [6], which adds an extra multiplication to produce a vector-friendly decomposition. Fig. 8 illustrates our 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 butterfly. We use the same transpose unit implementation as with automorphisms.

Our four-step pipeline supports negacyclic NTTs (NCNs), which are more efficient than standard non-negacyclic NTTs (that would require padding, Sec. 2.3). Specifically, we extend prior work [49, 62, 67] in order to support both forward and inverse NCNs using the same hardware as for the standard NTT. Namely, prior work shows how to either (1) perform a forward NCN via a standard decimation-in-time (DIT) NTT pipeline, or (2) perform an inverse NCN via a standard decimation-in-frequency (DIF) NTT pipeline. The DIF and DIT NTT variants use different hardware; therefore, this approach requires separate pipelines for forward and inverse NCNs. Prior work [49] has shown that separate pipelines can be avoided by adding a multiplier either before or after the NTT: doing an inverse NCN using a DIT NTT requires a multiplier unit before the NTT, while doing a forward NCN using a DIF NTT requires a multiplier unit after the NTT.

We now show that both the forward and inverse NCN can be done in the same standard four-step NTT pipeline, with no additional hardware. This is because the four-step NTT already has a multiplier and two NTTs in its pipeline. We set the first NTT to be decimation-in-time and the second to be decimation-in-frequency (Fig. 8). To do a forward NTT, we use the forward NCN implementation via DIT NTT for the first NTT; we modify the contents of the Twiddle SRAM so that the multiplier does the pre-multiplication necessary to implement a forward NCN in the second NTT (which is DIF and thus requires the pre-multiplication). Conversely, to do an inverse NTT, we modify the Twiddle SRAM contents to do the post-multiplication necessary to implement an inverse NCN in the first NTT (which is DIT); and we use the inverse NCN implementation via DIF NTT for the second NTT.

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 (Sec. 8). This is the first implementation of a
We use a cycle-accurate simulator to execute F1 programs. Because we have implemented F1’s components in RTL, and synthesize them 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 $q_i = -1 \mod 2^{16}$, we can remove a multiplier stage from [51]; this reduces area by 19% and power by 30% (Table 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.

### 5.3 Optimized modular multiplier

Modular multiplication computes $a \cdot b \mod q$. 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 $q_i = -1 \mod 2^{16}$, we can remove a multiplier stage from [51]; this reduces area by 19% and power by 30% (Table 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.

### 6 F1 IMPLEMENTATION

We have implemented F1’s components in RTL, and synthesize them 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 enables 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 $3\times16$ bit-sliced crossbars [58] (scratchpad$\rightarrow$cluster, cluster$\rightarrow$scratchpad, and cluster$\rightarrow$cluster).

Table 2 shows a breakdown of area by component, as well as the area of our F1 configuration, $151.4 \text{ mm}^2$. 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. We assume 512 GB/s bandwidth per PHY; this is similar to the NVIDIA A100 GPU [18], which has 2.4 TB/s with 6 HBM2E PHYs [56]. We use prior work to estimate HBM2 PHY area [24, 63] and power [32, 63].

This design is constrained by memory bandwidth: though it has $16 \times 16$ bits 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.

### 7 EXPERIMENTAL METHODOLOGY

**Modeled system:** We evaluate our F1 implementation from Sec. 6. 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.

**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) [15]. This work uses B/FV, an FHE scheme that F1 does not support, so we use CKKS 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 [42]) 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.

- **DB Lookup** is adapted from HELib’s $\text{BGV\_country\_db\_1\_lookup}$ [41]. A BGV-encrypted query string is used to traverse an encrypted key-value store and return the corresponding value. The original implementation uses a low security level for speed of demonstration, but in our version, we implement it at $L = 17$, $N = 16K$ for realism. We also parallelize the CPU version so it can effectively use all available cores. DB Lookup is both deep and wide, so running it on F1 incurs substantial off-chip data movement.

- **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

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Area [\text{mm}^2]</th>
<th>Power [mW]</th>
<th>Delay [ps]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrett</td>
<td>5,271</td>
<td>18.40</td>
<td>1,317</td>
</tr>
<tr>
<td>Montgomery</td>
<td>2,916</td>
<td>9.29</td>
<td>1,040</td>
</tr>
<tr>
<td>NTT-friendly</td>
<td>2,165</td>
<td>5.36</td>
<td>1,000</td>
</tr>
<tr>
<td>FHE-friendly (ours)</td>
<td>1,817</td>
<td>4.10</td>
<td>1,000</td>
</tr>
</tbody>
</table>

**Table 1: Area, power, and delay of modular multipliers.**

<table>
<thead>
<tr>
<th>Component</th>
<th>Area [\text{mm}^2]</th>
<th>TDP [W]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTT FU</td>
<td>2.27</td>
<td>4.80</td>
</tr>
<tr>
<td>Automorphism FU</td>
<td>0.58</td>
<td>0.99</td>
</tr>
<tr>
<td>Multiply FU</td>
<td>0.25</td>
<td>0.60</td>
</tr>
<tr>
<td>Add FU</td>
<td>0.03</td>
<td>0.05</td>
</tr>
<tr>
<td>Vector RegFile (512 KB)</td>
<td>0.56</td>
<td>1.67</td>
</tr>
<tr>
<td>Compute cluster (NTT, Aut, 2× Mul, 2× Add, RF)</td>
<td>3.97</td>
<td>8.75</td>
</tr>
<tr>
<td><strong>Total compute (16 clusters)</strong></td>
<td><strong>63.52</strong></td>
<td><strong>140.0</strong></td>
</tr>
<tr>
<td>Scratchpad (16×4 MB banks)</td>
<td>48.09</td>
<td>20.35</td>
</tr>
<tr>
<td>3×NoC (16×16 512 B bit-sliced [58])</td>
<td>10.02</td>
<td>19.65</td>
</tr>
<tr>
<td>Memory interface (2×HBM2 PHYs)</td>
<td>29.80</td>
<td>0.45</td>
</tr>
<tr>
<td><strong>Total memory system</strong></td>
<td><strong>87.91</strong></td>
<td><strong>40.45</strong></td>
</tr>
<tr>
<td><strong>Total F1</strong></td>
<td><strong>151.4</strong></td>
<td><strong>180.4</strong></td>
</tr>
</tbody>
</table>

**Table 2: Area and Thermal Design Power (TDP) of F1, and breakdown by component.**
Table 3: Performance of F1 and CPU on full FHE benchmarks: execution times in milliseconds and F1’s speedup.

<table>
<thead>
<tr>
<th>Microbenchmark</th>
<th>Execution time (ms)</th>
<th>CPU</th>
<th>F1</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoLa-CIFAR Unencryp. Wghts.</td>
<td>1.2 × 10^5</td>
<td>200</td>
<td>241</td>
<td>5,011×</td>
</tr>
<tr>
<td>LoLa-MNIST Unencryp. Wghts.</td>
<td>2,960</td>
<td>0.17</td>
<td>17</td>
<td>412×</td>
</tr>
<tr>
<td>LoLa-MNIST Encryp. Wghts.</td>
<td>5,431</td>
<td>0.36</td>
<td>15</td>
<td>086×</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>8,300</td>
<td>1.15</td>
<td>7</td>
<td>217×</td>
</tr>
<tr>
<td>DB Lookup</td>
<td>29,300</td>
<td>4.36</td>
<td>6</td>
<td>722×</td>
</tr>
<tr>
<td>BGV Bootstrapping</td>
<td>4,390</td>
<td>2.40</td>
<td>1</td>
<td>830×</td>
</tr>
<tr>
<td>CKKS Bootstrapping</td>
<td>1,554</td>
<td>1.30</td>
<td>1</td>
<td>195×</td>
</tr>
</tbody>
</table>

Table 4: Performance on microbenchmarks: F1’s reciprocal throughput, in nanoseconds per ciphertext operation (lower is better) and speedups over CPU and HEAX_σ (HEAX augmented with scalar automorphism units) (higher is better).

<table>
<thead>
<tr>
<th>Microbenchmark</th>
<th>N = 2^12, log Q = 109</th>
<th>F1 vs. CPU</th>
<th>vs. HEAX_σ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NTT</td>
<td>12.8</td>
<td>17,148×</td>
<td>1,600×</td>
</tr>
<tr>
<td>Automorphism</td>
<td>12.8</td>
<td>7,364×</td>
<td>440×</td>
</tr>
<tr>
<td>Homomorphic multiply</td>
<td>6.00</td>
<td>48,640×</td>
<td>172×</td>
</tr>
<tr>
<td>Homomorphic permutation</td>
<td>4.00</td>
<td>17,488×</td>
<td>256×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Microbenchmark</th>
<th>N = 2^13, log Q = 218</th>
<th>F1 vs. CPU</th>
<th>vs. HEAX_σ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NTT</td>
<td>12.8</td>
<td>17,148×</td>
<td>1,600×</td>
</tr>
<tr>
<td>Automorphism</td>
<td>12.8</td>
<td>7,364×</td>
<td>440×</td>
</tr>
<tr>
<td>Homomorphic multiply</td>
<td>6.00</td>
<td>48,640×</td>
<td>172×</td>
</tr>
<tr>
<td>Homomorphic permutation</td>
<td>4.00</td>
<td>17,488×</td>
<td>256×</td>
</tr>
</tbody>
</table>

8.2 Architectural analysis
To gain more insights into these results, we now analyze F1’s data movement, power consumption, and compute.

Data movement: Fig. 9a 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.

Fig. 9a shows that key-switch hints dominate in high-depth workloads (LogReg, DB Lookup, 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 to 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.

Table 4 shows that F1 achieves large speedups over HEAX_σ, 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.
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 same. Also, the scheduler accounts for the characteristics of these FUs. In both cases, performance drops substantially, by gmean 2.6× and 3.3×. 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 [37] 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.

### 8.4 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. Fig. 11 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.

### 9 RELATED WORK

We now discuss related work not covered so far.
FHE accelerators: Prior work has proposed accelerators for individual FHE operations, but not full FHE computations [20–22, 27, 52, 53, 65, 66, 71]. These designs target FPGAs and rely on a host processor; Sec. 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. [66] accelerate B/FV multiplication by $13 \times$ over a CPU; HEAWS [71] accelerates B/FV multiplication, and uses it to speed a simple benchmark by 5×; and HEAX [65] accelerates CKKS multiplication and key-switching by up to $200 \times$. 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.

Hybrid HE-MPC accelerators: Recent work has also proposed ASIC accelerators for some homomorphic encryption primitives in the context of oblivious neural networks [44, 64]. 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 [44] is a low-power ASIC for homomorphic evaluations, and Cheetah [64] 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: Cheetah and Gazelle use ciphertexts that are up to $\sim 40 \times$ smaller than those used by F1; however, they require the client to re-encrypt ciphertexts every time they are multiplied on the server to prevent noise blowup. CHOCO [72] shows that client-side computation costs for HE-MPC 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. As a result, a direct comparison between these accelerators and F1 is not possible.

F1’s hardware also differs substantially from Cheetah and Gazelle. First, Cheetah and Gazelle implement fixed-function pipelines (e.g., for output-stationary DNN inference in Cheetah), whereas F1 is programmable. Second, Cheetah, like HEAX, uses many FUs with relatively low throughput, whereas F1 uses few high-throughput units (e.g., 40× faster NTTs). Cheetah’s approach makes sense for their small ciphertexts, but as we have seen (Sec. 8.3), it is impractical for FHE.

GPU acceleration: Finally, prior work has also used GPUs to accelerate different FHE schemes, including GH [74, 75], BGV [73], and B/FV [1]. 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. [1] accelerate B/FV multiplication using GPUs, and achieve speedups of around $10 \times$ to $100 \times$ over single-thread CPU execution (and thus commensurately lower speedups over multicore CPUs, as FHE operations parallelize well).

10 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. accelerating primitive FHE computations with novel high-throughput functional units.

ACKNOWLEDGMENTS

We especially thank Nicholas Genise and Karim Eldedfrawy for detailed explanations of state-of-the-art FHE techniques and benchmarking. Unfortunately, publication restrictions and approval delays prevented Nicholas and Karim from becoming authors of this paper. An extended version of this work [29] includes their contributions. We also thank the anonymous reviewers, Maleen Abeydeera, Hyun Ryong Lee, Quan Nguyen, Yifan Yang, Victor Ying, Guowei Zhang, and Joel Emer for feedback on the paper; Tutu Ajayi, Austin Rovinski, and Peter Li for help with the HDL toolchain setup; Shai Halevi, Wei Dai, Olli Saarikivi, and Madan Musuvathi for email correspondence; and Luka Dojciclovic for feedback on the codebase. This research is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-21-C-0035. Any opinions, findings and conclusions or recommendations expressed in this research are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA). Nikola Samardzic was partially supported by the Jae S. and Kyuho Lim Graduate Fellowship at MIT.

REFERENCES


Rambus Inc. 2020. White paper: HBM2E and GDDR6: Memory Solutions for AI.


