An Architecture to Accelerate Computation on Encrypted Data

Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, and Srinivas Devadas, *Massachusetts Institute of Technology, Cambridge, MA, 02139, USA*

Ron Dreslinski and Chris Peikert, *University of Michigan, Ann Arbor, MI, 48109, USA*

Daniel Sanchez, *Massachusetts Institute of Technology, Cambridge, MA, 02139, USA*

Fully homomorphic encryption (FHE) allows computing on encrypted data, enabling secure offloading of computation to untrusted servers. Though it provides ideal security, FHE is prohibitively expensive when executed in software. These overheads are a major barrier to FHE's widespread adoption. We present F1, the first FHE accelerator that is 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 bottleneck. Thus, F1 is primarily designed to minimize data movement. F1 is the first system to accelerate complete FHE programs, and outperforms state-of-the-art software implementations by gmean 5,400x. These speedups counter FHE's overheads and enable new applications, like real-time private deep learning in the cloud.

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. 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 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 these 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,[7] early FHE schemes were about a billion times slower than performing computations on unencrypted data. Since then, improved FHE schemes have greatly reduced these overheads and broadened their applicability. 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, such as deep learning inference, linear algebra, and other machine learning tasks are a good fit for FHE. This has recently sparked significant industry and government investments to widely deploy FHE.

Unfortunately, FHE still carries substantial performance overheads: despite recent advances, 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

[*] Axel Feldmann and Nikola Samardzic contributed equally to this work.

0272-1732 © 2022 IEEE
Digital Object Identifier 10.1109/MM.2022.3170792
Date of publication 28 April 2022; date of current version 30 June 2022.

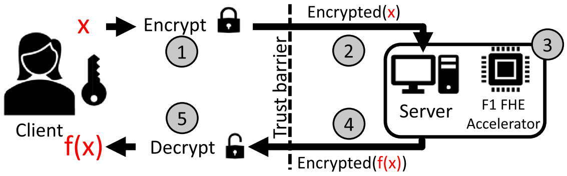


FIGURE 1. FHE allows a user to securely offload computation to an untrusted server. F1 can perform a DNN inference in 240 ms versus 20 min for prior work.

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[2,3,11,14–16] target individual FHE operations, and miss important ones that they leave to software. These designs target field-programmable gate arrays (FPGAs), so they are small and miss the data movement issues facing FHE accelerators that target an application-specific integrated circuit (ASIC) implementation. These designs also over-specialize their functional units (FUs) to specific parameters, and cannot efficiently handle the range of parameters needed within a program or across programs.

In our MICRO'21 paper[4] we introduced 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. Specifically, 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 FUs* 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 elementwise 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 FU.
2) *Regular computation:* FHE programs are dataflow graphs of arithmetic operations on vectors. All operations and their dependencies 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 cheaply issue tens of thousands of scalar operations per cycle.

FOR AN FHE ACCELERATOR TO BE BROADLY USEFUL, IT SHOULD BE PROGRAMMABLE, I.E., CAPABLE OF EXECUTING ARBITRARY FHE COMPUTATIONS.

3) *Challenging data movement:* In FHE, encrypting data increases its size (typically by about 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 FUs, 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 FUs with extremely high throughput*, rather than lower throughput FUs as in prior work. This reduces the amount of data that must reside on-chip simultaneously, allowing higher reuse.

As a result of these innovations, F1 outperforms a general-purpose multicore by gmean 5,400× and by up to 17,000× on a variety of FHE programs. 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 min in 240 ms, enabling real-time private deep learning in the cloud.

FULLY HOMOMORPHIC ENCRYPTION

FHE 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 had been performed on the plaintext values.

Over the last decade, prior work has proposed multiple *FHE schemes*, each with somewhat different capabilities and performance tradeoffs. Brakerski–Gentry–Vaikuntanathan (BGV), Brakerski/Fan-Vercauteren (B/FV), Gentry-Sahai-Waters (GSW), and Cheon-Kim-Kim-Song (CKKS) 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: we first introduce FHE's *interface*, i.e., its programming model and operations; then describe how FHE operations are *implemented*; and finally present implementation *optimizations*.

FHE Programming Model and Operations

In FHE, unencrypted (plaintext) data values are *vectors*. For example, in BGV, each plaintext is a vector of N integers modulo some integer t. FHE schemes provide a limited set of operations on these vectors. For example, BGV allows elementwise *addition* (mod t), elementwise *multiplication* (mod t), and a small set of particular vector *permutations*.

We stress that this is FHE's *interface*, not its implementation: it describes *unencrypted* data and the homomorphic operations that the FHE scheme provides on that data in its encrypted form.

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 nontrivial, and requires a sequence of permutations and additions.

FHE Implementation Overview

We now describe how FHE encodes and processes ciphertexts. We focus on BGV for concreteness, but other FHE schemes work similarly.

Data types: BGV encrypts each plaintext vector as a pair of degree-N polynomials with coefficients modulo an integer Q. N and Q are often large (e.g., N=16,384 coefficients with a 512-bit modulus Q), leading to very large (multimegabyte) ciphertexts.

Homomorphic operations: A homomorphic operation is the implementation of each computation on ciphertexts. For example, the homomorphic multiplication of two ciphertexts yields another ciphertext that, when decrypted, is the elementwise multiplication of the input plaintexts.

BGV implements homomorphic addition, multiplication, and permutation using operations on ciphertext polynomials. Our full paper[4] details their implementation; here, we highlight two key characteristics:

1) Homomorphic operations use a small set of operations on polynomials: polynomial addition (mod Q), polynomial multiplication (mod Q), and *automorphisms*, special structured permutations of ciphertext coefficients.
2) While homomorphic addition is cheap (ciphertext polynomials are simply added together), homomorphic multiplication and permutation are costly, as they depend on a computationally expensive subroutine called *key switching*. Key switching often dominates the performance of FHE applications. Key switching features numerous polynomial multiplications and large pieces of auxiliary data called *key-switch hints (KSHs)*. KSHs are even larger than ciphertexts, usually taking up tens of megabytes.

FHE PROVIDES A VECTOR PROGRAMMING MODEL WITH RESTRICTED OPERATIONS WHERE INDIVIDUAL VECTOR ELEMENTS CANNOT BE DIRECTLY ACCESSED.

Noise management: To prevent decryption without the secret key, FHE schemes add some random noise to ciphertexts when encrypting them. Unfortunately, this noise grows with each homomorphic operation, and especially with multiplications. To a first order, the amount of noise is determined by the program's *multiplicative depth*, i.e., its longest chain of multiplications.

Noise forces the use of large polynomials. For example, an FHE program with a multiplicative depth of 16 needs Q to be about 512 bits. To maintain security, $N/\log Q$ must be above a certain threshold, which also forces the use of large N, e.g., 16,384 coefficients for a 512-bit Q (which requires 2 MB per ciphertext).

As ciphertexts undergo homomorphic operations, they are rescaled to use a smaller modulus. This trims noise, slowing its growth. For instance, to execute an

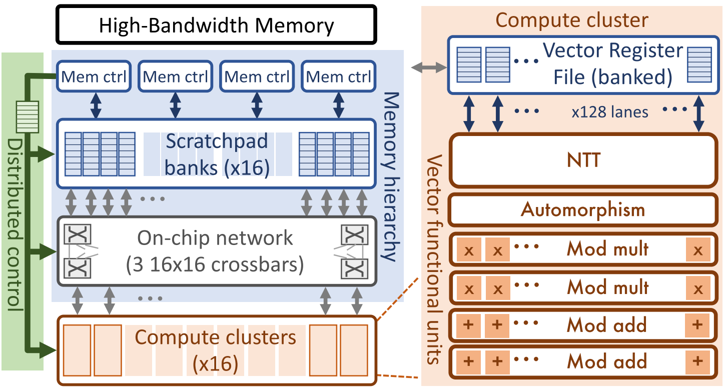


FIGURE 2. Overview of the F1 architecture.

FHE program with multiplicative depth 16, we would start with a 512-bit modulus Q. After each multiplication, we would switch to a modulus that is 32 bits smaller, until reaching a 32-bit modulus at the largest depth. Thus, beyond reducing noise, this modulus switching reduces ciphertext sizes and computation costs.

To enable computations of unbounded depth, FHE schemes can use a procedure called *bootstrapping* that refreshes the noise in the ciphertext. But bootstrapping is expensive and consumes many levels, so it must be applied infrequently. Thus, FHE schemes need a large noise budget.

Algorithmic Optimizations

F1 leverages two common optimizations in FHE.

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 NTT,[12] a variant of the discrete Fourier transform for modular arithmetic. Specifically, polynomial multiplication becomes elementwise multiplication in the NTT domain: NTT$(\mathfrak{ab})$ = NTT$(\mathfrak{a}) \odot$ NTT$(\mathfrak{b})$. (For this relation to hold with N-point NTTs, a negacyclic NTT[10] must be used.)

Because an NTT requires only $O(N \log N)$ operations, multiplication becomes $O(N \log N)$. And because the NTT is a linear transform, other operations can be performed in the NTT domain. Thus, FHE implementations often store polynomials in the NTT domain rather than in coefficient form *across operations* to reduce the number of NTTs.

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, F1 must efficiently support a broad range of widths (e.g., 64–512 bits in 32-bit increments), both because programs need different widths, and because modulus switching progressively reduces coefficient widths.

RNS representation[6] enables representing a single polynomial with wide coefficients as multiple polynomials with narrower coefficients, called *residue polynomials*. Q is chosen to be the product of L smaller distinct primes, $Q = q_1 q_2 \ldots\, q_L$. 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 ith 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. By adopting RNS, F1 can efficiently support a wide range of coefficient widths effectively.

F1 ARCHITECTURE

Figure 2 shows an overview of F1. We now highlight F1's design approach and key features.

Vector processing with specialized FUs: F1 features wide-vector execution with FUs tailored to FHE primitive 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 the 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 a residue polynomial maps to a single vector. Larger polynomials (e.g., of 32,768 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 elementwise operations, but hard for NTTs and automorphisms. The "Functional Units" section 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 having fewer high-throughput FUs reduces the amount of data that is accessed concurrently, improving reuse.

Compute clusters: FUs are grouped in *compute clusters*, as Figure 2 shows. Each cluster features several FUs (one NTT, one automorphism, two multipliers, and two adders in our implementation) and a register file that is banked to 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 Figure 2 shows, F1 features a large, highly 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[13] to hide 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 bit-sliced crossbar network 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). Avoids long staging of vectors at 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.[5]

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 variable latency, we assume the worst-case latency and buffer data that arrives earlier (since we access large chunks of data, e.g., 64 KB, this worst case latency is not far from the average).

Distributed control: Though F1 uses static scheduling, its implementation differs significantly from that of VLIW processors: instead of a single stream of instructions, each packed with many operations, in F1 each component has an *independent instruction stream*.

F1's lack of control flow makes this possible: though FHE programs can have loops, they do not have data-dependent branches. Therefore, we can unroll all loops and compile programs into linear instruction sequences.

This approach may seem costly in terms of instruction footprint. However, because operands are very long vectors and each instruction encodes a lot of work, instruction fetch overhead is well-amortized. To further reduce these overheads, we adopt a compact instruction format where each instruction encodes a single operation in addition to the number of cycles to stall until running the next instruction. Overall, instruction fetches consume less than 0.1% of memory traffic.

SCHEDULING DATA AND COMPUTATION

F1's compiler translates FHE programs to use hardware well. To achieve high utilization, we find that it is crucial to minimize off-chip data movement, the critical bottleneck. We contribute new scheduling algorithms to achieve this.

Figure 3 shows an overview of F1's compiler. The compiler takes as input an FHE program written in a high-level domain-specific language. The compiler is structured in three stages. First, the *homomorphic operation compiler* orders high-level operations to maximize reuse and translate 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 stage uses a simplified view of hardware, considering it as a scratchpad directly attached to FUs. The result is a dataflow graph that includes loads and stores from off-chip memory. Third, the *cycle-level*

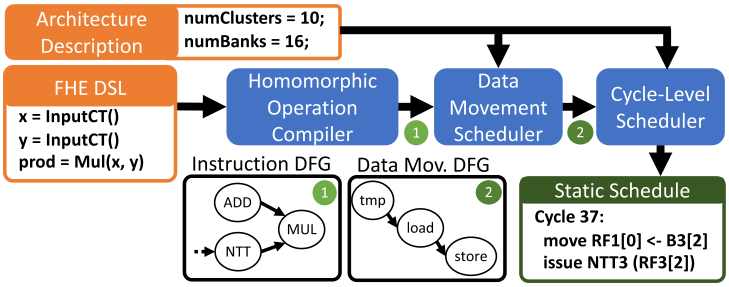


FIGURE 3. Overview of the F1 compiler.

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 stage determines the exact cycles of all operations and produces the instruction streams for all components.

This multipass scheduling primarily minimizes off-chip data movement. Only in the last stage do we consider on-chip placement and data movement.

The full paper[4] compares our approach to prior work.

FUNCTIONAL UNITS

In this section, we describe F1's novel FUs. These include the first vectorized automorphism unit, the first fully pipelined flexible NTT unit, and a new simplified modular multiplier adapted to FHE.

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 σ_k maps the element at index i to index $ki \bmod N$; there are N automorphisms total, two for each odd $k < N$. The key challenge in designing an automorphism unit is that these permutations are hard to vectorize: this unit should consume and produce $E = 128$ elements/cycle, but vectors are much longer, with N up to 16,384, and elements are permuted across different chunks. Moreover, we must support variable N *and* all automorphisms.

Standard solutions fail: a 16,384x16,384 crossbar is much too large; a scalar approach, such as 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.

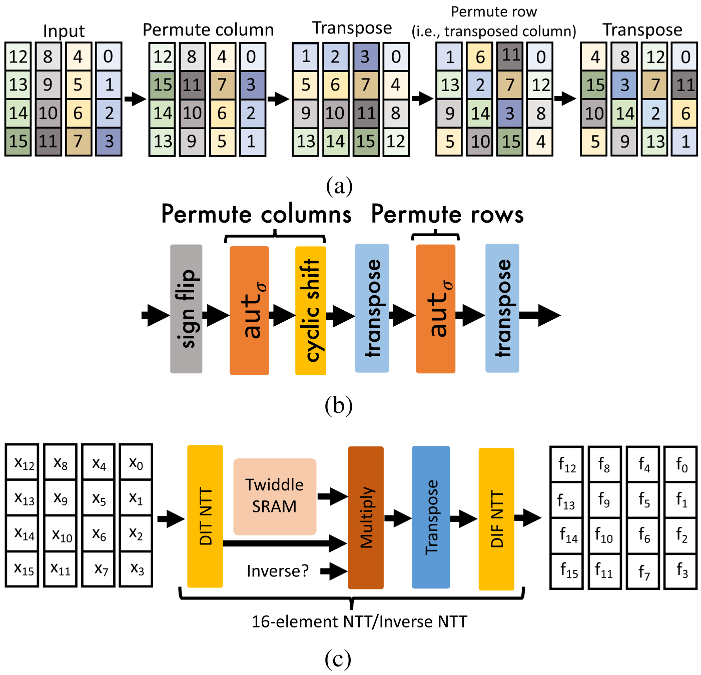
We contribute a new insight that makes vectorizing automorphisms simple: if we interpret a residue polynomial as a $G \times E$ matrix, an automorphism can always be decomposed into two independent *column* and *row permutations*. If we transpose this matrix, both column and row permutations can be applied *in chunks of E elements*. Figure 4(a) shows an example of how automorphism σ_3 is applied to a residue polynomial with $N = 16$ and $E = 4$ elements/cycle. Note that the permute column and row operations are local to each 4-element chunk. Other σ_k induce different permutations, but with the same row/column structure.

Our automorphism unit, shown in Figure 4(b), 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 subunit* that reads in the whole residue polynomial interpreting it as a $G \times E$ matrix and produces its transpose $E \times G$. The transpose subunit 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. To enable a fully pipelined design, we contribute a novel transpose subunit design, which our full paper[4] describes in detail.

Four-Step NTT Unit

An NTT is like an FFT but with a butterfly that uses modular multipliers. We implement N-element NTTs (from 1 to 16,384) as a composition of smaller E =128-element NTTs, since implementing a full 16,384-element NTT datapath is prohibitive. The challenge is that standard approaches produce hard-to-vectorize memory access patterns.

To that end, we use the *four-step variant* of the FFT algorithm, which adds a round of multiplication to produce a vector-friendly decomposition. Figure 4(c) illustrates our four-step NTT pipeline for $E = 4$; $E = 128$ uses the same structure. 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



Input

12	8	4	0
13	9	5	1
14	10	6	2
15	11	7	3

Permute column

12	8	4	0
15	11	7	3
14	10	6	2
13	9	5	1

Transpose

1	2	3	0
5	6	7	4
9	10	11	8
13	14	15	12

Permute row (i.e., transposed column)

1	6	11	0
13	2	7	12
9	14	3	8
5	10	15	4

Transpose

4	8	12	0
15	3	7	11
10	14	2	6
5	9	13	1

(a)

(b)

x_{12}	x_8	x_4	x_0
x_{13}	x_9	x_5	x_1
x_{14}	x_{10}	x_6	x_2
x_{15}	x_{11}	x_7	x_3

f_{12}	f_8	f_4	f_0
f_{13}	f_9	f_5	f_1
f_{14}	f_{10}	f_6	f_2
f_{15}	f_{11}	f_7	f_3

(c)

FIGURE 4. F1 contributes new automorphism and NTT FUs. (a) Example decomposing automorphism σ_3 into row and column permutations. (b) Vectorized automorphism FU. (c) Four-step NTT FU.

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.

Our four-step pipeline supports negacyclic NTTs, which are more efficient than standard NTTs (which would require padding). Our contribution is to support both forward and inverse negacyclic NTTs using the same amount of hardware as a standard NTT (see the our full paper[4] work for more details).

The NTT unit is large, requiring 1,024 multipliers. But its high throughput improves performance over many low-throughput NTTs. 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.

Optimized Modular Multipliers

Multipliers dominate F1's compute area and power. We contribute a new modular multiplier design that is restricted to work only with moduli relevant to FHE. This reduces area by 30% and power by 19% over the state-of-the-art multiplier design.[4]

F1 EVALUATION

Synthesis Results

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

We evaluate an F1 chip with 16 compute clusters, a 64-MB scratchpad, and a 1-TB/s main memory implemented with two HBM2 PHYs. We target a 1-GHz frequency for logic (register files and scratchpads run at 2 GHz to reduce ports). This configuration takes an area of 151 mm^2 and a TDP of 180 W. FUs take 42% of the area, with 32% going to memories, 6% to the on-chip network, and 20% to the two HBM2 PHYs.

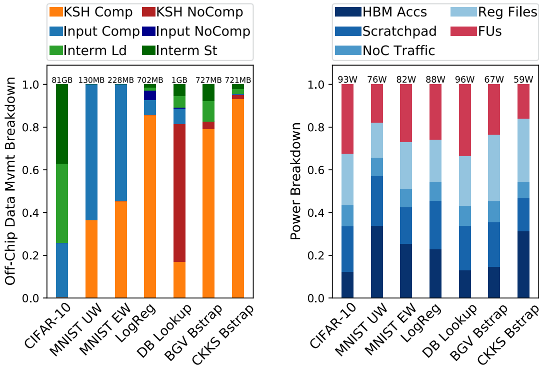
This design is constrained by memory bandwidth: though it has 1 TB/s of bandwidth, the on-chip network's bandwidth is 24 TB/s, and the aggregate bandwidth between RFs and FUs is 128 TB/s. This is why maximizing reuse is crucial.

Performance Evaluation Methodology

We use a cycle-accurate simulator to execute F1 programs. We use activity-level energies from RTL synthesis to produce energy breakdowns. We compare F1 with a CPU system running the baseline programs (a 4-core, 8-thread, 3.5-GHz Xeon E3-1240v5).

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**	$7,217\times$
DB Lookup	$29,300$	**4.36**	$6,722\times$
BGV Bootstrapping	$4,390$	**2.40**	$1,830\times$
CKKS Bootstrapping	$1,554$	**1.30**	$1,195\times$
gmean speedup			$5,432\times$

(a)



(b) (c)

FIGURE 5. (a) Performance results. (b) Per-benchmark breakdowns of off-chip data movement. (c) Power consumption.

We evaluate F1 on several existing FHE benchmark applications:

Neural networks are adapted from Low-Latency Cryptonets,[1] using the CKKS scheme. LoLa-MNIST is a simple LeNet-style network used on the MNIST data set. LoLa-CIFAR is a larger six-layer network with similar computation to MobileNet v3, used on the CIFAR-10 dataset. LoLa-MNIST has two variants with encrypted and unencrypted weights; LoLa-CIFAR is only available with unencrypted weights.

Logistic regression uses HELR,[9] a CKKS-based algorithm. It implements logistic regression training with up to 256 features and 256 samples per batch.

Database lookup is adapted from HELib's database lookup example as a BGV benchmark.

Bootstrapping: We implement unpacked bootstrapping for both BGV and CKKS. Bootstrapping refreshes noise in an $L = 1$ ciphertext by bringing it to a top value of $L = L_{\max}$, then performing the bootstrapping computation to obtain a usable ciphertext at a lower depth (e.g., $L_{\max} - 15$ for BGV). The unpacked variants process ciphertexts that encode a single element each, which simplifies bootstrapping.

Due to F1's programmability, each program is executed with the security level at which it was originally designed. For example, LoLa neural networks have 128-bit security, while HELR logistic regression and our bootstrapping benchmarks have 80-bit security.

Performance Results

Figure 5(a) 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× (5,432× gmean). CKKS bootstrapping has the lowest speedups as it is 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 min. F1 brings this down to 240 ms, making real-time deep learning inference practical: when offloading inferences to a server, this time is comparable to the roundtrip latency between the server and the client.

Prior accelerators do not support full FHE programs, so we can only compare F1 against them on microbenchmarks. Our full paper[4] includes a microbenchmark-based comparison with HEAX[14]; F1 outperforms HEAX by 172×–1,866×.

Architectural Analysis

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

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

Due to our scheduler design, F1 approaches compulsory traffic for most benchmarks, with noncompulsory access adding only 5%–18% of traffic. The exception is LoLa-CIFAR, where intermediates consume 75% of traffic. LoLa-CIFAR has a very high reuse of KSH, and exploiting it requires spilling intermediate ciphertexts. Figure 5(b) shows that KSH dominate in high-depth workloads (LogReg, DB lookup, and bootstrapping), taking up to 94% of traffic. KSHs are also significant in the LoLa-MNIST variants. To maximize performance, our scheduler is designed to prioritize their reuse.

Power consumption: Figure 5(c) reports the average power for each benchmark, broken down by component. Results show reasonable power consumption for an accelerator card. Overall, computation consumes 20%–30% of power, and data movement dominates.

CONCLUSIONS AND FUTURE WORK

FHE enables computation offloading with guaranteed security, which other techniques cannot match. For example, secure enclaves, such as Intel's SGX, have much lower overheads than FHE, but are vulnerable to attacks.[8,17] By never decrypting data, FHE provides cryptographic security guarantees. 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 OPENS UP EXCITING AVENUES FOR FUTURE WORK. WE BELIEVE THAT ALGORITHM-HARDWARE CO-DESIGN IS A KEY OPPORTUNITY.

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 FUs, 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.

F1 opens up exciting avenues for future work. We believe that algorithm-hardware co-design is a key opportunity. Prior FHE optimizations have mostly come from the cryptographic community, with a focus on improving FHE performance on CPUs. But specialization transforms the optimization landscape: whereas CPUs are mainly compute-bound, specialization makes data movement the key challenge, requiring new algorithmic optimizations. Moreover, there is ample room for further hardware improvements. For example, large applications that need frequent bootstrapping require using very large ciphertexts, about an order of magnitude larger than those in our benchmarks, to amortize bootstrapping costs. These computations demand further hardware techniques to cope with their extreme footprints.

FHE today has many parallels to deep learning a decade ago. Back then, neural networks were considered impractical over alternative machine learning algorithms, which provided similar accuracy with much cheaper computation. But around 2011, hardware acceleration made deep neural networks practical, bringing them into the mainstream. This unleashed a revolution that drove a decade of investment into improving deep learning at all layers of the stack, from algorithms to architectures. Similarly, hardware acceleration can bring FHE into the mainstream, as we have shown with F1. We hope this will spark a virtuous cycle that will further improve performance and efficiency, ultimately making FHE an integral part of the secure computer systems of the future.

ACKNOWLEDGMENTS

The authors would like to thank Nicholas Genise and Karim Eldefrawy for detailed explanations of state-of-the-art FHE techniques and benchmarking. They would also like to thank the anonymous reviewers, Maleen Abeydeera, Hyun Ryong Lee, Quan Nguyen, Yifan Yang, Victor Ying, Guowei Zhang, and Joel Emer for feedback on this article; Tutu Ajayi, Austin Rovinski, and Peter Li for help with synthesis tools; Shai Halevi, Wei Dai, Olli Saarikivi, and Madan Musuvathi for email correspondence; and Luka Dojcilovic for feedback on the codebase. This work was supported in part by DARPA under Contract HR0011-21-C-0035. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of DARPA. Nikola Samardzic was partially supported by the Jae S. and Kyuho Lim Graduate Fellowship at MIT.

REFERENCES

1. A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 812–821.
2. D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An FPGA co-processor implementation of homomorphic encryption," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2014, pp. 1–6.
3. Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Trans. Comput,*, vol. 64, no. 6, pp. 1509–1521, Jun. 2015.

4. A. Feldmann *et al.*, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 238–252.
5. J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proc. 10th Annu. Int. Symp. Comput. Archit.*, 1983, pp. 140–150.
6. H. L. Garner, "The residue number system," in *Proc. Western Joint Comput. Conf.*, 1959, pp. 146–153.
7. C. Gentry *et al.*, *A Fully Homomorphic Encryption Scheme*. Stanford, CA, USA: Stanford Univ. Press, 2009.
8. J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel SGX," in *Proc. 10th Eur. Workshop Syst. Secur.*, 2017, pp. 1–6.
9. K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 9466–9471.
10. V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, 2010, pp. 1–23.
11. A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 2, pp. 353–362, Feb. 2020.
12. R. T. Moenck, "Practical fast polynomial multiplication," in *Proc. 3rd ACM Symp. Symbolic Algebr. Comput.*, 1976, pp. 136–148.
13. M. Pellauer *et al.*, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 137–151.
14. M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 1295–1309.
15. 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. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 387–398.
16. F. Turan, S. Roy, and I. Verbauwhede, "HEAWS: An accelerator for homomorphic encryption on the amazon AWS FPGA," *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1185–1196, Aug. 2020.
17. J. Van Bulck *et al.*, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 991–1008.

AXEL FELDMANN is a Ph.D. student in the Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA. His research focuses on efficiently mapping programs onto novel hardware. Contact him at axelf@csail.mit.edu.

NIKOLA SAMARDZIC is a Ph.D. student in the Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA. His research focuses on speeding up fully homomorphic encryption. Contact him at nsamar@csail.mit.edu.

ALEKSANDAR KRASTEV is an undergraduate student at Massachusetts Institute of Technology, Cambridge, MA, 02139, USA. Contact him at alexalex@csail.mit.edu.

SRINIVAS DEVADAS has been with the Electrical Engineering and Computer Science faculty, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA, since 1988. His research interests include computer architecture, computer security, and applied cryptography. He is a Fellow of ACM and IEEE. Contact him at devadas@csail.mit.edu.

RON DRESLINSKI is an associate professor at the University of Michigan, Ann Arbor, MI, 48109, USA, researching computer architecture and VLSI. He is a Senior Member of IEEE. Contact him at rdreslin@umich.edu.

CHRIS PEIKERT is a professor of computer science and engineering at the University of Michigan, Ann Arbor, MI, 48109, USA. His research interests include cryptography, especially lattice-based, postquantum, and homomorphic cryptosystems. Peikert received a Ph.D. degree in computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA. Contact him at cpeikert@umich.edu.

DANIEL SANCHEZ is an associate professor EECS Department at Massachusetts Institute of Technology, Cambridge, MA, 02139, USA. His research interests include scalable memory hierarchies, architectural support for parallelization, and accelerators for sparse computations and secure computing. Sanchez received a Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA. He is a Member of IEEE. Contact him at sanchez@csail.mit.edu.