CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data

Nikola Samardzic
nsamar@csail.mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

Axel Feldmann
axelf@csail.mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

Aleksandar Krastev
alexalex@csail.mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

Nathan Manohar*
nmanohar@ibm.com
IBM T.J. Watson
Yorktown Heights, NY, USA

Nicholas Genise
nicholas.genise@sri.com
SRI International
Menlo Park, CA, USA

Srinivas Devadas
devadas@csail.mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

Karim Eldefrawy
karim.eldefrawy@sri.com
SRI International
Menlo Park, CA, USA

Chris Peikert
cpeikert@umich.edu
University of Michigan
Ann Arbor, MI, USA

Daniel Sanchez
sanchez@csail.mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

ABSTRACT

Fully Homomorphic Encryption (FHE) enables offloading computation to untrusted servers with cryptographic privacy. Despite its attractive security, FHE is not yet widely adopted due to its prohibitive overheads, about 10,000× over unencrypted computation. Recent FHE accelerators have made strides to bridge this performance gap. Unfortunately, prior accelerators only work well for simple programs, but become inefficient for complex programs, which bring additional costs and challenges.

We present CraterLake, the first FHE accelerator that enables FHE programs of *unbounded* size (i.e., unbounded multiplicative depth). Such computations require very large ciphertexts (tens of MBs each) and different algorithms that prior work does not support well. To tackle this challenge, CraterLake introduces a new hardware architecture that efficiently scales to very large ciphertexts, novel functional units to accelerate key kernels, and new algorithms and compiler techniques to reduce data movement.

We evaluate CraterLake on deep FHE programs, including deep neural networks like ResNet and LSTMs, where prior work takes minutes to hours per inference on a CPU. CraterLake outperforms a CPU by gmean 4,600× and the best prior FHE accelerator by 11.2× under similar area and power budgets. These speeds enable real-time performance on unbounded FHE programs for the first time.

CCS CONCEPTS

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

*Work done while an intern at SRI International and affiliated with UCLA.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06.
https://doi.org/10.1145/3470496.3527393

KEYWORDS

fully homomorphic encryption, hardware acceleration.

ACM Reference Format:
Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527393

1 INTRODUCTION

A large and increasing fraction of the world's compute runs on the cloud, which is vulnerable to data breaches. Conventional techniques to mitigate attacks offer limited security, as cloud servers must decrypt data in order to process it.

Fully homomorphic encryption (FHE) is a special type of encryption scheme that enables *computing on encrypted data directly*, without decrypting it. FHE allows a client to offload a computation to an untrusted server *without* revealing any data, as Fig. 1 illustrates. This enables the client to harness the compute power of the cloud while maintaining cryptographic privacy. Though FHE has some limitations (e.g., data-dependent branching is not possible), it is general enough to support many compelling use cases, such as privacy-preserving machine learning, secure genome analysis, private set intersection, private information retrieval, and many more [15, 27, 30, 35, 36, 43, 44].

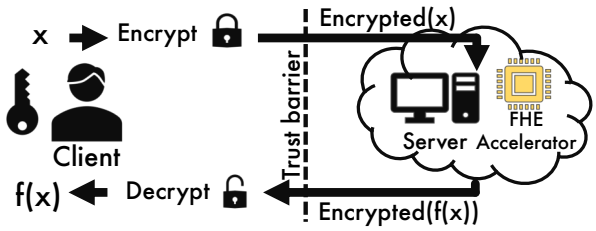

Figure 1: FHE can offload computation to the cloud securely.

Despite its ideal privacy, FHE is rarely used today because it incurs prohibitive overheads: in CPUs, FHE computations are 10,000× to 100,000× slower than equivalent unencrypted computations, even when using highly optimized FHE libraries.

Fortunately, state-of-the-art FHE schemes are well-suited to hardware acceleration. First, they are regular and structured: FHE programs operate on very long vectors, and all operations are known ahead of time. Second, FHE requires several non-SIMD operations, such as *number-theoretic transforms* (NTTs), that are inefficient on CPUs and GPUs. But these operations can be accelerated by specialized functional units, avoiding these inefficiencies. As a result, prior work has proposed FPGA and ASIC-based accelerators [18–20, 25, 60, 61, 63]. While most prior accelerators achieve limited speedups, a recent design, F1 [25], achieves speedups of about 5,000× on FHE programs.

Unfortunately, prior accelerators are efficient only on a limited subset of simple FHE computations—those of *shallow multiplicative depth*. For example, prior FHE accelerators can run neural network inference efficiently only for networks with few layers (3-6), but they cannot accelerate state-of-the-art deep neural networks (DNNs) with tens to hundreds of layers.

This limitation stems from the characteristics of FHE schemes: each ciphertext has some associated noise, which grows with each homomorphic operation, and especially with multiplications. If noise becomes too large, it garbles the message, making decryption impossible. Larger ciphertexts tolerate more noise before becoming undecryptable. However, operations on larger ciphertexts are also more expensive. To enable computations of unbounded depth, ciphertexts can be "refreshed" using a procedure called *bootstrapping* that reduces noise. But bootstrapping is expensive, so ciphertexts must be very large (10s of MBs) for bootstrapping to be efficient.

Prior FHE accelerators do not efficiently handle unbounded-depth computations because they natively support vectors of a limited size and they use algorithms that scale poorly to the large ciphertexts in high-depth programs. As a result, they can only run small FHE computations, and they do not support sufficient depth to run the full bootstrapping procedure.

In this paper we tackle this challenge through CraterLake, the first FHE accelerator to support *FHE computations of unbounded depth*. To achieve this, we contribute new algorithms, specialized functional units, hardware architecture, and compiler techniques that overcome the key challenge of deep FHE computations: its extreme data movement demands.

Deep FHE is limited by data movement: FHE schemes encode information over very long vectors of wide elements. Concretely, supporting unbounded-depth computations requires vectors of 64K elements with 1,600 bits per element. This takes 25 MB per ciphertext, 12× larger than what prior FHE accelerators target.

Moreover, prior work has employed FHE algorithms that require even larger amounts of auxiliary data. For example, multiplying 2 MB ciphertexts in F1 [25] requires 32 MB of auxiliary data, and scaling their algorithm to 25 MB ciphertexts would require over 1.4 GB of auxiliary data—far too large to fit on-chip. To tackle this challenge, our *key insight* is to adopt an FHE algorithm called *boosted keyswitching* (Sec. 3) that eliminates most of the auxiliary data, reducing this overhead from 1.4 GB to 50 MB. Boosted keyswitching also reduces computation costs. However, this new algorithm is a poor match for prior accelerators: it is dominated by simple operations where these designs have limited efficiency, and makes poor use of the specialized functional units of prior designs.

Beyond being inefficient, prior accelerators suffer from a hard-to-scale vector multicore architecture: to support the needed non-SIMD operations with reasonable cost, they implement multiple independent cores with narrower vector datapaths [25]. However, this causes excessive inter-core communication, and the high-bandwidth interconnect needed grows superlinearly with the number of cores.

Deep FHE demands new hardware techniques: To tackle these challenges, we introduce the *CraterLake* architecture (Sec. 4, Sec. 5), the first FHE accelerator that achieves high performance on unbounded FHE programs. CraterLake is a wide-vector uniprocessor with specialized functional units. The design is statically scheduled to leverage the regularity of FHE computations. We contribute several new techniques that make this possible, including:

- A new extremely wide (2,048 lanes) vector uniprocessor architecture that spreads each vector operation across the chip, departing from prior vector multicore architectures. This vector uniprocessor approach reduces the number of concurrent operations, which minimizes footprint, reducing off-chip traffic, and simplifies the compiler.
- An efficient implementation of the above architecture, which is challenging due to non-SIMD FHE operations, NTTs and automorphisms. We decompose these operations in a novel way that allows the use of a *fixed transpose network* among physically distributed groups of lanes. This reduces on-chip data movement and interconnect cost over prior approaches.
- A new functional unit that encapsulates the bulk of operations in boosted keyswitching, improving efficiency and enabling high utilization across ciphertexts of all sizes.
- A new functional unit that generates half of the required auxiliary data on the fly (reducing overheads from 50 MB to 25 MB), saving on-chip storage and memory bandwidth.
- A vector chaining technique that builds long FU pipelines to enable many concurrent operations with few register ports.

To program CraterLake, we develop a novel compiler (Sec. 6) that produces efficient code from high-level FHE programs. The compiler schedules operations to maximize reuse, decouples data movement from computation, and adapts the state-of-the-art bootstrapping algorithm to achieve high utilization [11].

We evaluate CraterLake through a combination of simulation and RTL synthesis (to find its area and power). We use a broad range of FHE benchmarks, including programs with high multiplicative depth that require bootstrapping. CraterLake outperforms a scaled-up and improved version of the state-of-the-art FHE accelerator, F1, by gmean 11.2× on these deep computations, and is 4,600× faster than a 32-core CPU. These speedups enable new use cases for FHE. For example, deep neural networks like ResNet take 23 minutes per inference on the CPU, whereas CraterLake achieves 250 *milliseconds* per inference, enabling real-time private deep learning.

2 BACKGROUND

State-of-the-art FHE schemes implement operations on *encrypted vectors*. The ciphertexts in these schemes support several *homomorphic operations*: element-wise addition, element-wise multiplication,

and rotations of vector elements. Each homomorphic operation produces a ciphertext that, when decrypted, produces the same result as if the operation had been performed on the unencrypted inputs.

Importantly, homomorphic operations have a different implementation from their unencrypted counterparts—for example, a homomorphic multiplication is not implemented using element-wise multiplication of the input ciphertexts, but a more complex sequence of operations. Therefore, it is useful to differentiate between FHE's *interface*, i.e., its supported plaintext datatypes and operations, and its *implementation*, i.e., the structure of ciphertexts and the implementation of homomorphic operations.

There are several FHE schemes, which mainly differ in their plaintext datatypes and the operations they support. For example, BGV [12] encodes vectors of integers modulo a constant, whereas CKKS [16] encodes vectors of fixed-point numbers. Despite the differences between these schemes, the commonalities in their underlying implementation [49] make it possible for the same hardware to accelerate many schemes efficiently—CraterLake supports CKKS, BGV, and GSW [29]. For concreteness, the rest of this section will focus only on CKKS, as it is the scheme best suited for machine learning tasks and has been the focus of much recent work in FHE algorithms and applications [11, 22, 23, 30, 35, 48, 57].

2.1 FHE Interface

FHE schemes implement operations on vectors of values. In CKKS, each vector element is a *fixed-point* complex number with a configurable number of bits. (Programs that do not use complex arithmetic can zero out the imaginary part.)

Since values are encrypted, FHE does not permit data-dependent branching or indirection. Thus, all operations and dependencies are known ahead of time, and FHE programs can be represented using *static dataflow graphs*.

Homomorphic operations in CKKS include element-wise addition, element-wise multiplication, and cyclic rotations. These operations are *approximate* in CKKS, inducing a small and controllable amount of error. However, this error can be made arbitrarily small at the cost of reduced performance. This error is acceptable in practice as machine learning applications are insensitive to it.

FHE exposes a vector programming model with a restricted set of operations; in particular, FHE does not provide access to individual vector elements. This makes it challenging to implement some operations that are trivial in plaintext: For example, implementing a convolutional layer of a neural network requires the careful replication of filter weights. The lack of non-linear functions introduces other difficulties. For example, the ReLU activation function must be approximated using a high-degree polynomial [47]. As a result, faithfully replicating deep neural networks in FHE, as done by a recent ResNet implementation [48], comes at a high compute cost. Instead, recent work has proposed neural network structures that are tailored to FHE and achieve lower overheads while maintaining similar accuracy to some unencrypted networks [13]. We evaluate CraterLake on both styles of neural networks.

Finally, not all data needs to be encrypted: additions and especially multiplications are much cheaper in FHE if one of the operands is unencrypted. This allows algorithms to trade privacy for performance. For example, running a neural network using unencrypted weights is faster; it still ensures the privacy of inputs and results, but does not protect weights [13].

2.2 FHE Implementation

We now describe how CKKS represents and operates on encrypted data (i.e., ciphertexts); other schemes (e.g., BGV) have a similar structure.

Encryption: A ciphertext holds an encrypted vector of plaintext values. To create a ciphertext, the vector of plaintext values is first encoded, or *packed*, in a polynomial; this polynomial is then *encrypted*. CKKS packs a plaintext vector of $n = N/2$ complex fixed-point numbers into a degree-$(N-1)$ polynomial:

$$(c_0, c_1, ..., c_n) \xmapsto{pack} \mathfrak{m} = k_0 + k_1 x + ... + k_{N-1} x^{N-1}$$

$\mathfrak{m}$ is then encrypted into a ciphertext. Each ciphertext consists of $\mathfrak{ct}_0, \mathfrak{ct}_1$—two *ciphertext polynomials* with coefficients modulo a *ciphertext modulus* Q. Specifically, we encrypt $\mathfrak{m}$ under a *secret key* $\mathfrak{s}$ by sampling a uniformly random $\mathfrak{a}$ and a small *error* $\mathfrak{e}$ ($\mathfrak{s}$, $\mathfrak{a}$, and $\mathfrak{e}$ are also polynomials):

$$\mathfrak{m} \xmapsto{encrypt} \mathfrak{ct} = (\mathfrak{ct}_0, \mathfrak{ct}_1) = (\mathfrak{a}, \mathfrak{a} \cdot \mathfrak{s} + \mathfrak{e} + \mathfrak{m})$$

The above process produces a *fully-packed* ciphertext, i.e., one that encodes as many plaintext values as possible. It is possible (though almost always less efficient) to pack fewer values, producing a partially packed or unpacked (single-element) ciphertext.

Homomorphic operations are implemented through several modular-arithmetic operations on ciphertext polynomials, i.e., vectors of coefficients. Specifically:

- *Homomorphic addition* of two ciphertexts simply requires modular addition of their ciphertext polynomials: $\mathfrak{ct}_{\text{add}} = \mathfrak{a} + \mathfrak{b} = (\mathfrak{a}_0 + \mathfrak{b}_0, \mathfrak{a}_1 + \mathfrak{b}_1)$.
- *Homomorphic multiplication* is implemented using polynomial multiplications and additions; multiplying two polynomials requires convolving their coefficients.
- *Homomorphic rotation* rotates the vector encrypted in a ciphertext. Implementing it requires performing an *automorphism* on the ciphertext polynomials, a structured permutation where, for automorphism k, each input index i is mapped to output index ik mod N. There are N possible automorphisms; each automorphism induces a simpler, cyclic rotation in the plaintext.

On top of this, homomorphic multiplications and rotations also require a procedure called *keyswitching*, which is needed so that the final ciphertext stays encrypted by the same secret key as the input. Keyswitching is expensive, and in practice *takes over 90% of all operations*. Keyswitching is central to CraterLake, so we discuss it in detail in Sec. 3.

2.3 Challenges of Deep FHE Computation

FHE ciphertexts include some *noise* or *error* to ensure cryptographic privacy [49]. Noise compounds during homomorphic operations, which adds overheads. Noise increases primarily during ciphertext multiplications; each ciphertext can tolerate only a fixed amount of noise before decryption becomes impossible. Therefore, we say that the *multiplicative depth* that a ciphertext can tolerate is the ciphertext's *multiplicative budget*.

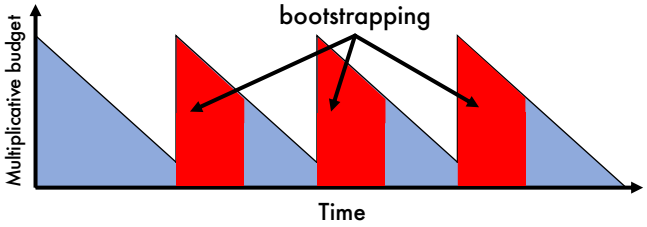

Figure 2: Multiplicative budget over time. Bootstrapping is used to refresh a ciphertext's multiplicative budget.

Supporting a high multiplicative budget requires using ciphertexts with wide coefficients and a large ciphertext modulus Q. For example, ciphertexts with 512-bit coefficients have a multiplicative budget of about 16. After each multiplication, the ciphertext is *rescaled* to use a smaller modulus (e.g., dropping 32 bits). This trims the noise and makes computation more efficient over time, as narrower coefficients are cheaper to operate on. Ciphertexts run out of multiplicative depth when their coefficients become too narrow to support further operations (e.g., 32 bits). In CKKS, the specific number of bits to drop per operation is not fixed, but depends on the precision that the application requires.

A computation with high multiplicative depth can be supported by simply using ciphertexts with sufficiently high multiplicative budgets, but this adds major overheads. First, it requires using very wide coefficients, which take more storage per plaintext element and make computations more complex. Moreover, wide coefficients induce a second hurdle: they force the use of larger vectors. This is because, for security, $N/\log Q$ must be above a certain threshold. For instance, a multiplicative budget of 16 requires Q of about 512 bits and N=16K (i.e., 2 MB per ciphertext), and a multiplicative budget of 32 requires Q of about 1,024 bits and N=32K (i.e., 8 MB per ciphertext). Though larger vectors can pack more plaintext elements, this quickly results in vectors so large that they cannot fit on-chip. Overall, ciphertext size grows quadratically with multiplicative budget, and compute cost cubically (inducing linear and quadratic overheads per plaintext element, respectively).

Bootstrapping: FHE schemes limit the overheads of deep computation through a procedure called bootstrapping that refreshes the multiplicative budget of a ciphertext. Bootstrapping enables computations of arbitrary depth by separating them into regions of limited depth. But bootstrapping is an expensive and deep computation, so it should happen infrequently.

Fig. 2 illustrates a typical evolution of a ciphertext's multiplicative budget during program execution: computation proceeds until the ciphertext runs out of budget, then bootstrapping is applied to refresh the ciphertext. For example, in our LSTM benchmark, computation starts with a multiplicative budget of 57 and bootstrapping consumes the highest 35 levels (in red in Fig. 2), leaving 22 levels for application computation (in blue in Fig. 2).

Ciphertext sizes needed for deep FHE: We now show that CraterLake supports the ciphertext sizes required for deep FHE, and why prior work falls short. Fig. 3 reports the cost per homomorphic operation (in scalar multiplies per homomorphic multiply, y-axis) of two deep programs, as a function of the maximum ciphertext size used (x-axis). This determines bootstrapping frequency: using larger ciphertexts requires less frequent bootstrapping. Fig. 3 also

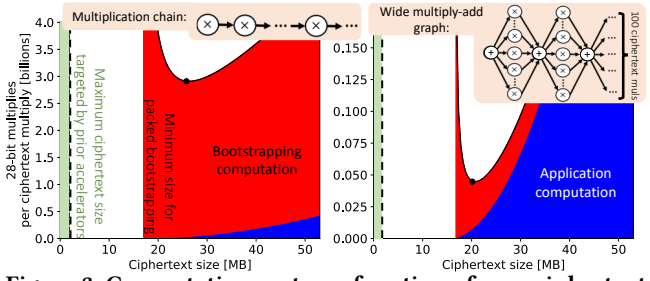

Figure 3: Computation cost as a function of max ciphertext size for narrow (left) and wide (right) deep FHE programs.

breaks down cost by that used for application computation (blue) and bootstrapping (red).

The left plot is for a serial chain of multiplies, the worst case for bootstrapping cost, as the amount of computation between bootstrappings is minimal. Consequently, bootstrapping dominates. By contrast, the right plot is for a wide graph with 100 multiplies per depth, which converge to a single output after each level. This amortizes bootstrapping across many operations, a best-case scenario.

Crucially, the optimal maximum ciphertext size (shown with a black dot) is in a narrow range for both extremes, between 20 MB (right) and 26 MB (left). This is because *both* application computation and bootstrapping become more expensive with ciphertext size, so regardless of which dominates, once bootstrapping is infrequent enough, moving to larger ciphertexts only hurts performance.

Thus, 20–26 MB max ciphertexts are the sweet spot for most deep programs, which fall between these extremes. In practice, bootstrapping placement is NP-hard [9], because real FHE programs are not as regular. But all our benchmarks show a similar tradeoff curve to these synthetic programs.

Prior FHE accelerators cannot efficiently support ciphertexts this large. For example, F1 [25] *becomes inefficient past 2 MB*, and other accelerators [60] are limited to even smaller values. This is insufficient to run even bootstrapping itself. (Although F1 supports *unpacked* bootstrapping of ciphertexts that encode only a single element, this is >1,000× slower per element and thus impractical for full applications, as Sec. 9 shows.)

As we will see, scaling to large ciphertexts is not merely a matter of scaling up hardware; it requires new algorithms and a new hardware organization to support these algorithms and to cope with the huge footprint of ciphertexts.

2.4 Implementation Optimizations

We use two common FHE implementations' optimizations:

Residue Number System (RNS) representation [26] allows representing each of the wide coefficients of a ciphertext polynomial as L *residue polynomials* with narrow coefficients. This is achieved by choosing the wide modulus Q to be a product of L smaller factors, $Q = q_1 q_2 ... q_L$, called *small moduli*. Then, x mod Q is uniquely represented as $(x \bmod q_1, x \bmod q_2, ..., x \bmod q_L)$.

RNS representation reduces overall operation cost, and allows supporting many coefficient widths with a single narrow width in hardware. CraterLake exploits this by using 28-bit elements (algorithm-related limits prevent using arbitrarily narrow coefficients, as Sec. 5.5 explains). For example, a ciphertext polynomial with 1,500-bit Q is stored using L=54 28-bit residue polynomials.

Number-Theoretic Transform (NTT) is a modular-arithmetic variant of the Fast Fourier Transform. The NTT is an $O(N \log N)$ operation that makes polynomial multiplications efficient: in the NTT domain, the convolution required for polynomial multiplication becomes element-wise multiplication. FHE implementations often store polynomials in the NTT domain. CraterLake includes NTT functional units.

2.5 Prior Accelerators and Their Limitations

Prior work has proposed multiple FHE accelerators. From these, F1 [25] is the closest to CraterLake. F1 is a statically scheduled processor with clusters of vector functional units specialized to FHE. In particular, F1 introduces new high-throughput vector functional units for NTTs and automorphisms, all-to-all operations that cannot be efficiently performed with conventional SIMD datapaths.

Though F1 is programmable and can accelerate full computations, it targets shallow computations. Specifically, F1 is tailored to a keyswitching algorithm that does not scale to high multiplicative budgets L (Sec. 3). As a result, F1 is inefficient when using a more scalable keyswitching algorithm: It has an inappropriate mix of functional units, and even with the right mix, it would be dominated by simple operations that would require over 100 register file ports for the FUs to be fully utilized. Moreover, F1's organization incurs excessive communication and is hard to scale to larger systems.

As a result, CraterLake introduces a fundamentally different design, needed for deep computations: it adopts a new, simpler hardware organization and data tiling approach that reduces communication and scales to the large ciphertexts required, and it is tailored to use an efficient keyswitching algorithm, which requires new functional units and optimizations.

Besides F1, prior work proposed accelerators for FPGAs [18, 20, 24, 50, 51, 60, 61, 63]. These systems are not programmable: they accelerate a small number of primitives and use fixed parameters (N and L), so they cannot execute full applications; they suffer from excessive data movement; and they achieve limited speedups.

3 BOOSTED KEYSWITCHING

Keyswitching is the dominant computation in FHE, especially for ciphertexts with high multiplicative budgets (L). Thus, we use keyswitching to drive CraterLake's design.

Keyswitching consists of a large number of operations on residue polynomials and requires a large auxiliary operand called a *keyswitch hint* (KSH); the KSH adds pressure on memory bandwidth and on-chip storage.

Prior accelerators are optimized for the *standard* keyswitching algorithm, which is inefficient for deep computations. By contrast, we target the keyswitching algorithm proposed by Gentry et al. [28, Section 3.1]. In fact, there are multiple variants of this algorithm [33, Section 5.3.4], that we collectively refer to as *boosted keyswitching*. FHE libraries targeting deep computations use boosted keyswitching [1, 28, 33, 53].

The key innovation in boosted keyswitching is to expand the input polynomial to use wider coefficients. This simplifies the KSH and its application. Boosted keyswitching variants differ in how much they expand the input, which introduces a tradeoff between performance and security. We first analyze the most efficient variant (which expands the input the most), then discuss the performance and security tradeoffs of different variants.

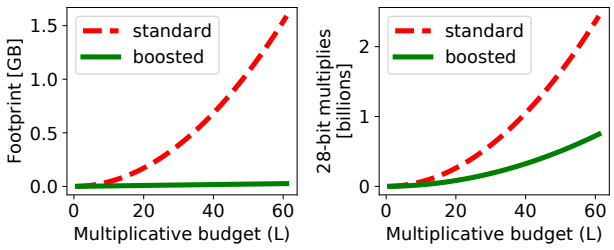

Figure 4: Comparison of footprint and compute for standard and boosted keyswitching.

```
1  def boostedKeySwitch(p[0:L]):
2    pTmp[0:L] = INTT(p[0:L])
3    pTmp[L:2L] = changeRNSBase(pTmp[0:L], [L:2L])
4    p[L:2L] = NTT(pTmp[L:2L])
5    for i = 0, 1:
6      prod_i[0:2L] = p[0:2L] * KSH_i[0:2L]
7      tmp_i[0:2L] = INTT(prod_i[L:2L], [0:L])
8      mDTmp_i[0:L] = changeRNSBase(tmp_i[L:2L], [0:L])
9      modDown_i[0:L] = NTT(mDTmp_i[0:L])
10     ks_i[0:L] = prod_i[0:L] + modDown_i[0:L]
11   return (ks_0[0:L], ks_1[0:L])
12
13 def changeRNSBase(x[0:L], destModIdxList):
14   for srcModIdx in [0:L]:
15     for destModIdx in destModIdxList:
16       C = constant[srcModIdx][destModIdx]
17       result[destModIdx] += x[srcModIdx] * C
18   return result
```

Listing 1: Boosted keyswitching implementation (1-digit).

Fig. 4 compares the memory footprint and compute cost (measured in scalar multiplications) of standard and boosted keyswitching as a function of L (the number of residue polynomials, proportional to the bitwidth of Q). Both algorithms have similar costs for small values of L, but costs grow much more quickly with L for standard keyswitching.

In particular, keyswitch hints are the size of two ciphertexts in the boosted algorithm. This footprint reduction is the most important factor to CraterLake. For instance, at N=64K and L=60, a keyswitch hint takes 52.5 MB instead of 1.7 GB for the standard algorithm. This enables holding KSHs on-chip and allows for high reuse. Fig. 4 also shows that boosted keyswitching reduces computational costs across the range of multiplicative budget.

Listing 1 shows the implementation of boosted keyswitching. Keyswitching takes a ciphertext polynomial and the keyswitching hint (KSH) as inputs, and produces two ciphertext polynomials as output. This variant of boosted keyswitching expands the input polynomial to use 2× wider coefficients; this expansion reduces the KSH sizes and their application. In RNS representation, this is accomplished through `changeRNSBase()`, which is used to both *expand* the L-residue input into a $2L$-residue intermediate and later to *shrink* the output back to L residues.

Table 1 compares the operations used by boosted and standard keyswitching. Whereas standard keyswitching has L^2 NTTs, boosted keyswitching uses only $O(L)$ NTTs: a 10× reduction for L=60. To achieve this, boosted keyswitching incurs about 50% more

Ops	Boosted keyswitching changeRNSBase	+	other	Standard keyswitching
Mult	$3L^2$	+	$4L$	$2L^2$
Add	$3L^2$	+	$2L$	$2L^2$
NTT			$6L$	L^2
At L=60:				
Mult	10,800	+	240	7,200
Add	10,800	+	120	7,200
NTT			360	3,600

Table 1: Operation breakdown for boosted vs. standard keyswitching as a function of multiplicative budget (L), and for L=60. Boosted keyswitching operations are split by whether they happen within or outside `changeRNSBase()`.

multiplies and adds than standard keyswitching. However, trading off fewer NTTs for more multiplies and adds is highly beneficial because NTTs are much more complex, requiring $O(N \log N)$ scalar multiplies and adds.

Previous accelerators cannot perform boosted keyswitching efficiently because they are designed to execute all multiplies and adds separately, resulting in an overwhelming amount of register file port pressure. However, the bulk of these operations take place in `changeRNSBase()` (Table 1), and are structured as sequence of multiply and accumulate operations (Listing 1) [7]. CraterLake exploits this by introducing a novel `changeRNSBase()` functional unit, *CRB*, that buffers the intermediate sums and thereby reduces the register file pressure *by a factor of L* (i.e., up to 60×).

Additionally, Listing 1 shows that most intermediate variables are consumed immediately after being produced and can then be discarded. CraterLake exploits this by building *configurable pipelines* of functional units, further reducing register file pressure (Sec. 5.4).

3.1 Performance-Security Tradeoffs in Boosted Keyswitching

As Sec. 2.3 explained, the security level of FHE depends on $N/logQ$: the ratio between the number of polynomial coefficients and the width of each coefficient. By expanding the input polynomial by a factor of two, the above boosted keyswitching algorithm increases the maximum $logQ$ by 2×. This would require either doubling N or using half of the levels to achieve the same security level as standard keyswitching.

Since boosted keyswitching is much more efficient than standard algorithm, this is a worthwhile tradeoff. Moreover, other boosted keyswitching variants offer finer control over this tradeoff. Specifically, boosted keyswitching variants are parameterized by the number of so-called *digits* t. The variant described above is 1-digit keyswitching. In t-digit keyswitching, *(1)* the input polynomial is expanded by a factor $t/(1+t)$; *(2)* keyswitch hint footprint is proportional to $1+t$; and *(3)* compute operations outside of `changeRNSBase` also increase, e.g., multiplications grow by a factor $1+t$; however, this is a minor effect, because `changeRNSBase` dominates the number of operations (Table 1), and operations within `changeRNSBase` do not grow with the number of digits.

Concretely, using t=2, 3, or 4 digits increases the maximum $logQ$ by 1.5×, 1.33×, and 1.25×, respectively. Thus, higher-digit keyswitching variants reduce the N required for a given level of security, or increase the number of levels allowed between bootstrappings vs. 1-digit keyswitching. The main drawback of these variants is that keyswitch hints grow quickly with the number of digits (by a factor $t+1$), so whereas in 1-digit keyswitching each KSH is the size of 2 ciphertexts, with 2–4-digit keyswitching each KSH takes 3–5 ciphertexts. This makes these variants more memory-bound, especially if the larger KSHs reduce on-chip reuse.

Achieving a given level of security efficiently requires carefully trading off the keyswitching variant used and frequency of bootstrapping. FHE programs also use multiple keyswitching variants over time: higher-digit keyswitching may be necessary when $logQ$ is large, but 1-digit keyswitching can be used when ciphertexts become narrower, since a 2× expansion does not affect the maximum $logQ$ of the computation. Given these tradeoffs, FHE accelerators should support different keyswitching variants efficiently. For example, to achieve 80-bit security with N=64K in our evaluation, we use 2-digit keyswitching for multiplicative budgets $L > 52$ and 1-digit keyswitching elsewhere; we also show how to achieve higher security levels, e.g., 128 bits (Sec. 9.4).

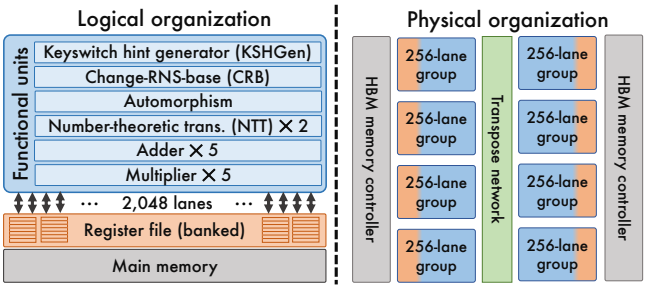

Figure 5: Overview of the CraterLake architecture.

4 ARCHITECTURE OVERVIEW

Fig. 5 shows an overview of the CraterLake architecture. We first describe its key elements, and then explain why this is the right architecture for FHE by considering design alternatives.

4.1 Logical Organization

Vector FUs and data types: CraterLake is a vector processor with specialized functional units (FUs) tailored to FHE operations. CraterLake includes fast vector FUs for modular additions, modular multiplications, NTTs, and automorphisms, which are adapted from F1 [25]. In addition, CraterLake contributes two novel FUs: a Change-RNS-Base unit (CRB) that accelerates the bulk of boosted keyswitching, and a Keyswitch hint generator (KSHGen) that generates half of each keyswitch hint on the fly, reducing memory traffic and on-chip storage.

CraterLake implements a *single* set of vector FUs that process vectors of a *configurable length* N, which can be any power of 2 from 2,048 to 65,536 in our implementation. Each vector represents one residue polynomial, so vector elements have a fixed, narrow width (28 bits in our implementation). CraterLake has a large number of *vector lanes* E, 2,048 in our implementation. All FUs are *fully pipelined*, consuming and producing E=2,048 elements/cycle. Each vector is fed to an FU in N/E consecutive cycles.

Memory system: CraterLake's on-chip storage is organized as a single-level *256 MB* register file shared by all FUs. While this amount of storage might appear excessive, it fits *just 10 ciphertexts* at the largest parameters CraterLake targets ($N_{\max}$=64K, $L_{\max}$=60), and smaller register files severely limit performance, as we show in Sec. 9.3. The register file uses an *element-partitioned* design [6] to efficiently emulate 12 read and write ports. Still, this is only half of the 24 input ports of our FUs. We bridge this gap by allowing FUs to be *chained* to form multi-FU pipelines that execute more complex operations.

CraterLake uses high-bandwidth main memory (HBM2E in our implementation). Memory controllers interface directly with register file banks. The system uses decoupled data orchestration [56] to hide memory latency: memory transfers are performed independently of compute operations, staging ciphertexts in the register file ahead of their use.

Static control: CraterLake hardware is *statically scheduled* to leverage the regularity of FHE operations. All operations have a fixed latency, and the compiler is responsible for scheduling all operations and memory transfers to respect all data dependencies. This avoids the need for hardware to implement any dynamic control, like backpressure or stalling logic, and enables CraterLake to support very wide vector FUs with minimal control overheads.

4.2 Physical Organization

Implementing an E=2,048-lane vector processor naively would result in prohibitive on-chip traffic. CraterLake addresses this by splitting its lanes into G=8 *lane groups*. Each lane group is E_G=256 elements wide and occupies a physically distinct region of the chip, as Fig. 5 shows. As lane groups contain both FU lanes and register file banks, the majority of data movement can be performed locally within each group.

Splitting lanes into groups is challenging because NTTs and automorphisms have all-to-all dependencies between vector elements, requiring communication among lane groups. Luckily, F1 [25] showed that these dependencies can be mapped to *transposes* (one per NTT and two per automorphism). But F1's implementation of transposes does not scale to this many lanes. To address this challenge, we contribute a new transpose implementation that performs all data movement between lane groups through a simple *fixed permutation network* that consists of only wires and registers, without any network switches. Supporting CraterLake's compute throughput requires this network to have a total bandwidth of $4E$ elements per cycle (29 TB/s).

4.3 Comparison to Prior Work

As CraterLake implements a single set of E=2,048-lane FUs, all of its lane groups operate in tandem on different parts of the *same* residue polynomial. The only operations that require communication among lane groups are NTTs and automorphisms, which need E and $2E$ elements per cycle of bandwidth, respectively (Sec. 5.3). This is at most 29 TB/s for the 2 NTTs and 1 automorphism unit in CraterLake; each homomorphic multiplication and rotation transfer $8NL$ and $10NL$ words among lane groups, respectively.

In contrast, prior work implements multiple independent *compute clusters* (analogous to our lane groups), and assigns all elements of each residue polynomial to a compute cluster [25, 60]. This makes each NTT and automorphism local to a cluster, but each keyswitch requires all-to-all communication of residue polynomials at a rate of GE elements per cycle, where G is the number of clusters; in total, each homomorphic operation transfers $3GNL$ words among clusters. Thus, this approach scales poorly. Specifically, for G=8 (as we use in CraterLake), it requires double the peak bandwidth of our approach (57 TB/s), and incurs over 2.4× more traffic per homomorphic operation. More importantly, this approach requires a complex network between clusters, which is 16× larger than our fixed permutation network (Sec. 8).

Additionally, having all lane groups operate in lockstep reduces on-chip storage requirements, as we can dedicate the whole chip to a single homomorphic operation at a time. By contrast, using compute clusters well often requires overlapping multiple homomorphic operations, which adds to footprint.

Finally, our approach makes the compiler's job easier: it only needs to decide on a single polynomial operation to run at a time, instead of needing to orchestrate for parallelism by scheduling multiple operations across clusters. This keeps the compiler simple and utilization high.

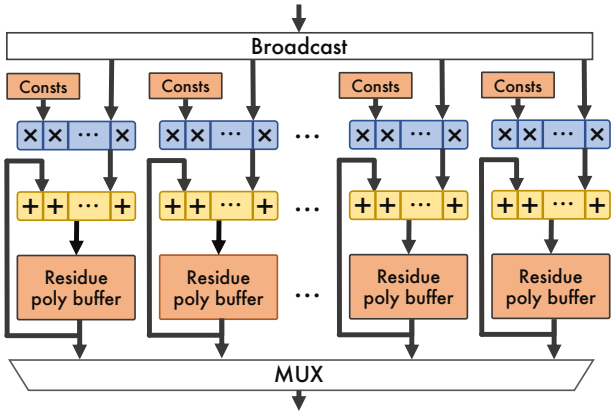

Figure 6: Microarchitecture of the change RNS Base (CRB) unit.

5 MICROARCHITECTURE

This section introduces CraterLake's novel FUs, CRB (Sec. 5.1) and KSHGen (Sec. 5.2); presents the transpose network needed by NTT and automorphism FUs (Sec. 5.3); describes our implementation of FU chaining to reduce register file pressure (Sec. 5.4); and introduces new optimizations for modular multipliers, which dominate area and energy (Sec. 5.5).

5.1 Change-RNS-Base (CRB) Unit

As Sec. 3 discussed, boosted keyswitching is dominated by change-RNS-base operations. The CRB unit, shown in Fig. 6, consists of many parallel multiply-and-accumulate pipelines that spatially unroll the inner loop of `changeRNSBase()` (Listing 1). The CRB unit exploits the high internal reuse in `changeRNSBase()` to allow much higher throughput than independent multipliers and adders communicating through the register file. This is the same insight behind DNN accelerators like the TPU [41] and Tensor Cores in GPUs [17].

The CRB unit first receives as input L residue polynomials, and then produces L output residue polynomials, both at E elements/cycle. Each input polynomial is broadcast to all pipelines, with each pipeline producing exactly one of the output polynomials. The CRB unit is double-buffered so that it can simultaneously produce the output of one operation, and receive the input for the next one.

We size the CRB unit to handle the largest ciphertexts that we find in deep applications, N_{max}=64K and L_{max}=60. This results in 60 parallel pipelines with 26.25 MB of total buffers. Smaller ciphertexts leave some of the CRB pipelines unused.

The CRB unit is by far CraterLake's largest FU: it consists of 120K scalar multipliers and adders, consuming 34% of on-chip area. Despite its size, the CRB unit is easy to lay out in hardware as it performs only element-wise operations. In return, the CRB unit reduces the time it takes to perform keyswitching from $O(L^2)$ to $O(L)$. This is essential for achieving high utilization across different ciphertext sizes, as the runtime of all other operations in FHE grows linearly with L.

5.2 KeySwitch Hint Generator (KSHGen)

As half of each keyswitch hint (KSH) is pseudo-random, it can be generated on the fly from a small seed, halving KSH storage and bandwidth. While this optimization has been previously implemented in software [32], we propose the first hardware KeyswitchHint Generator (KSHGen).

The KSHGen generates numbers uniformly distributed modulo some prime by sampling random bits from a cryptographic PRNG [10], and then performing rejection sampling. The challenge is that rejection sampling has *variable throughput*, which plays poorly with static scheduling.

We address this in two ways: First, we reduce the probability of rejection by sampling additional random bits per generated word. Second, we introduce small buffers (16 words deep) that hide the occasional rejections. As these buffers are refilled between generating different KSHs, the probability any of them runs empty is negligible. Additionally, since software controls the seeds, it can test and avoid the few that fail to produce outputs at-speed. The KSHGen unit is cheap and improves performance by up to 2.5× (Sec. 9).

5.3 Transpose Network and FUs

CraterLake's lane groups communicate using a *fixed permutation network*, i.e., a network that connects specific input/output pairs *without any control logic*. This approach reduces network area over the full-crossbar approach of F1 by 16×, with a 2.4× reduction in network bandwidth for keyswitching.

NTTs and automorphisms are the only FUs with dependencies between elements of the same vector. F1 approaches these operations by laying out the N-element input vector as a $\sqrt{N} \times \sqrt{N}$ matrix. Then, an NTT over the whole vector can be expressed as a set of row-wise NTTs followed by a set column-wise ones. A similar decomposition exists for automorphisms. Therefore, by introducing a fully-pipelined transpose unit, F1 efficiently maps these dataflows to a $\sqrt{N}$-lane vector processor [25].

CraterLake differs from F1 in that it has E=2,048 lanes, 8× the maximum $\sqrt{N}$=256 it targets. F1's approach is unsuitable for CraterLake as it results in monolithic 2,048-lane pipelines, with excessive

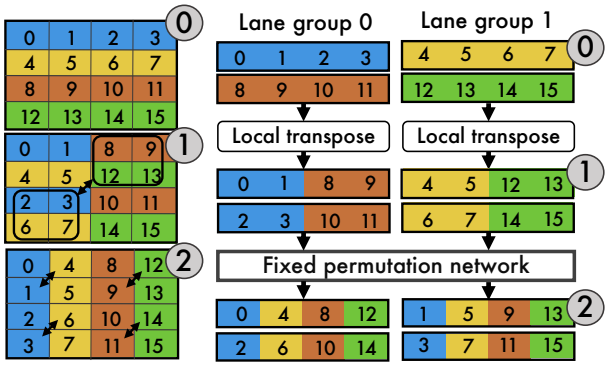

Figure 7: Example of CraterLake's two-level transpose: transposing a 16-element vector across two lane groups.

communication between lanes. Instead, CraterLake partitions its lanes into G=8 groups of E_G=256 lanes each. All dependencies between lane groups are satisfied using a novel, spatially distributed transpose network that can process E=2,048 elements per cycle.

CraterLake distributes the rows of the $E_G \times E_G$ matrix across lane groups in a round-robin fashion (Fig. 7, step 0). Then, it splits the matrix into $G \times G$ *blocks*, and decomposes the transpose into two steps: *(1)* transposing the $(E_G/G) \times (E_G/G)$ *block-matrix*, and *(2)* transposing all $G \times G$ blocks.

Fig. 7 illustrates transposing a 4×4 matrix (E_G=4) across G=2 lane groups. The right side of the figure shows how the steps are executed in hardware, while the left side shows their effect on the $E_G \times E_G$ matrix.

Step 1: As lane group i is responsible for row i of *all* $G \times G$ blocks, the block-matrix transpose can be performed locally: each lane group performs a block-level transpose on the $1 \times G$ subblocks it holds. CraterLake implements this step by using a *separate* fully-pipelined transpose unit *in each lane group* (using the same transpose unit design as F1).

Step 2: When transposing each $G \times G$ block, group i starts off storing its i-th *row*, and must end up storing its i-th *column*. While this requires moving elements between lane groups, the exchange follows a fixed pattern: group i sends to group j the elements in the j-th columns of all $1 \times G$ subblocks it holds. CraterLake implements this using a fixed permutation network among lane groups.

CraterLake handles vectors with $N < N_{max}$ similarly to F1: vectors are laid out as $N/E_G \times E_G$ matrices, and transposed only within $N/E_G \times N/E_G$ blocks. This requires only adjusting step 1, which is performed locally.

5.4 Vector Chaining

While the CRB substantially reduces register file (RF) port pressure, achieving high utilization requires a large number of FUs, as shown in Fig. 5. If these FUs always operated on registers, keeping them busy would require dozens of RF ports.

To tackle this challenge, we allow FUs to be *chained*, so that the output of an FU can be consumed by another FU without going through the register file. This is similar to Cray-1's vector chaining [62], except that chained values are not written to the register file, saving write ports. Chaining works well because boosted keyswitching is amenable to pipelining: most operands are consumed immediately after being produced.

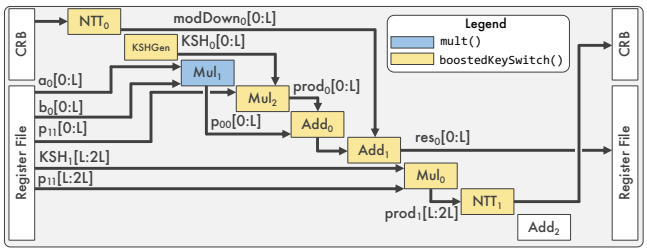

Figure 8: Illustration of how vector chaining is used for part of a homomorphic multiply.

For efficiency, we tailor the allowed inter-FU chaining options to those needed by homomorphic operations. For example, Fig. 8 shows how FU chaining is used to form a pipeline that implements a part of homomorphic multiplication. While this pipeline chains 10 FUs (beginning and ending in the CRB), it uses only 5 read and 1 write RF ports, instead of the 24 it would need without chaining. Overall, vector chaining reduces register file traffic by 3.5× during keyswitching.

We find that supporting four large pipelines with a few variants suffices to chain most operations. Chaining adds few inter-FU paths: on average, each output is connected to 3 inputs (including the RF), resulting in a cheap implementation.

5.5 Bitwidth and Multiplier Optimizations

CraterLake's FUs are dominated by scalar modular multipliers. Thus, an efficient multiplier design is crucial. RNS moduli q_i must come from a set of restricted, *NTT-friendly* primes. F1's FHE-specific multipliers exploit this to simplify logic [25].

CraterLake improves F1's design in two ways: First, since multiplier area and power scale *quadratically* with bitwidth, CraterLake adopts a narrower, 28-bit datapath (F1 uses 32 bits). We cannot reduce bitwidth any further because then there would not be enough NTT-friendly moduli to support the deep benchmarks CraterLake targets (we need $2L_{\max}$=120 small moduli). Second, we pipeline each multiplier to its energy-optimal point.

Together, these approaches improve area per bit by 1.6× and energy per bit by 1.3× over F1's multipliers. Without these optimizations, power draw would limit throughput.

6 COMPILER

The CraterLake compiler translates FHE programs written in a high-level language. It primarily seeks to minimize off-chip data movement by maximizing on-chip reuse, and to decouple memory accesses and computation, fetching off-chip operands ahead of their use to prevent stalls. The compiler produces a cycle-by-cycle configuration for all chip resources. We now describe the compiler's organization, in order of transformations applied to an input program:

1. Input FHE programs: We use a Python-embedded DSL to describe FHE programs, similar to the compiler of F1 [25].

2. Ordering of homomorphic operations: The input program is first translated to a dataflow graph of homomorphic operations. These operations are then ordered to maximize reuse of operands using a standard tiling analysis [38, 55, 68] similar to Timeloop [55].

Ordering to maximize reuse is critical: because operands are so large, CraterLake's on-chip storage can only hold a few of them. For example, for N=64K and L=60, each ciphertext is 26 MB, so on-chip storage can hold just shy of 10 ciphertexts.

3. Compiling homomorphic operations: Once ordered, homomorphic operations are translated one-by-one and scheduled to run on the accelerator.

The compiler first translates each homomorphic operation into a sequence of simpler operations on ciphertext polynomials. To achieve high utilization, the compiler implements keyswitching as a sequence of up to five FU pipelines, leveraging vector chaining (Sec. 5.4). Thus, each keyswitching operation is expressed as a sequence of up to five complex operations. All other ciphertext polynomial computations are translated to individual multiplication, addition, and automorphism operations, which use a single FU and read and write to the register file.

The compiler then schedules memory accesses and compute operations cycle-by-cycle. Off-chip loads are scheduled greedily: any time the memory is free, the scheduler traverses operations in order and fetches the first operand that it finds is off-chip. If this operand requires evicting a live value, that value is written back first. We follow Belady's MIN [8] and evict the operand reused the furthest. The load is deferred if the victim operand is used earlier than the loaded operand, or if the loaded operand would have to be evicted before its use. Each operation is then scheduled on the earliest cycle where its input operands and its FU (or FUs for keyswitch pipelines) are available.

This procedure produces a cycle-by-cycle schedule of all operations and data transfers. This schedule is then transformed into the configuration streams for all components of the chip.

Optimized bootstrapping: Since bootstrapping uses the largest ciphertexts, maximizing its reuse is crucial. We use a state-of-the-art bootstrapping algorithm that recursively decomposes its key kernels, analogously to FFTs [14]. We decompose the computation into many partitions, each small enough to fit on chip (a 4×4 tile). This decomposition makes bootstrapping consume some extra levels, but it achieves much higher performance overall by allowing on-chip reuse.

Comparison with prior work: CraterLake's compiler builds on F1's, and shares key features: it uses high-level programs as inputs, and seeks to minimize and decouple off-chip data movement, which are critical. However, CraterLake's compiler is substantially simpler and achieves higher utilization. This is because CraterLake has a simpler interface than F1: CraterLake exposes a single set of wide-vector functional units, similar to a vector uniprocessor, whereas F1 organizes narrower FUs into independent compute clusters and has two levels of on-chip storage (per-cluster register files and a shared scratchpad), similar to a vector multicore. Thus, F1's compiler must distribute a single homomorphic operation among multiple clusters, and must often overlap many homomorphic operations. This distributed design creates complex scheduling problems, e.g., trading off load balance and utilization for reuse.

7 IMPLEMENTATION

We implemented CraterLake's components in RTL, and synthesized them in a commercial 14/12nm process using state-of-the-art tools.

Component	Area [mm^2]
CRB FU	158.8
NTT FU	28.1
Automorphism FU	9.0
KSHGen FU	3.3
Multiply FU	2.2
Add FU	0.8
Total FUs (CRB, 2×NTT, Aut, KSHGen, 5×Mul, 5×Add)	**240.5**
Register file (256 MB)	192.0
On-chip interconnect	10.0
Mem. PHYs (2×HBM2E)	29.8
Total CraterLake	**472.3**

Table 2: Area breakdown of CraterLake by component.

These include a commercial SRAM compiler that we use for register file banks.

We target a configuration with area and power budgets similar to a modern GPU or server CPU. This design requires careful optimizations to limit power. We target a 1 GHz frequency for most components, and pipeline them to their energy-optimal points, using high-Vt cells and clock gating. Register file banks run double-pumped at 2 GHz. This enables using single-ported SRAMs while serving up to two accesses per cycle and bank.

We use HBM2E main memory, and assume 512 GB/s bandwidth per PHY (similar to NVIDIA's A100 GPU [17], which has 2.4 TB/s with 6 PHYs [54]). We use prior work to estimate the HBM2E PHY area [21, 58] and power [58].

Table 2 shows CraterLake's area and its breakdown by component. Overall, our CraterLake configuration requires 472 mm^2. FUs take 51% of the area, with 41% going to the register file, 6% to the two HBM2E PHYs, and 2% to the on-chip interconnect. As we will see in the evaluation, this design stays within a power budget of 320 W, in line with GPUs and server CPUs.

Finally, while these figures could be considered high, note that we are not using a leading-edge fabrication node: based on published logic and SRAM scaling factors [69], on current TSMC 5 nm technology, CraterLake would consume a modest 157 mm^2 area and 146 W peak power.

Host communication: We assume that CraterLake is implemented as an accelerator. The needed CPU-accelerator bandwidth required to stream inputs and outputs in our benchmarks is 50 GB/s on average and 130 GB/s at most, so a commodity PCIe 5 interface suffices to achieve full throughput. CPU-accelerator latency is not an issue, as these are bulk transfers and can be overlapped with computation.

8 EXPERIMENTAL METHODOLOGY

Modeled system: We evaluate our CraterLake implementation from Sec. 7 using a cycle-accurate simulator to execute CraterLake programs. We use activity-level energies from synthesized components to produce energy breakdowns.

Benchmarks: We use several FHE programs to evaluate CraterLake. All programs come from state-of-the-art software implementations, and use the CKKS scheme. To show that CraterLake is efficient on unbounded computations, we use four *deep* benchmarks, which have a high multiplicative depth and require bootstrapping. We also use four *shallow* benchmarks, with low multiplicative depth and no bootstrapping, to show that CraterLake is also efficient at low depths.

We meet 80-bit security for all benchmarks by using a combination of 2-digit and 1-digit keyswitching (Sec. 3.1). We also use non-sparse keys and the most recent bootstrapping techniques [11] to maximize multiplicative budget without losing precision. We later evaluate variants with 128-bit security and beyond (Sec. 9.4). We use the LWE estimator [5] to derive security parameters.

Deep benchmarks include:

(1) LSTM is a Long-Term Short-Term (LSTM) NLP benchmark [57]. This benchmark boils down to computing $h_{i+1} = \sigma(W_0 h_i + W_1 x_i)$ many times. σ is an activation function approximated by a degree-3 polynomial, and $W_0 h_i$, $W_1 x_i$ are 128×128 matrix-vector multiplies. This computation is multiplicatively deep and requires 50 bootstrappings per inference.

(2) ResNet-20 is an FHE implementation [48] of the ResNet-20 DNN. We benchmark an inference on a single encrypted image.

(3) Logistic regression uses the HELR algorithm [36], which is based on CKKS. We compute many batches of logistic regression training with 256 features, and 256 samples per batch, starting at computational depth L=38. This benchmark is different from the one reported in F1, as it performs multiple logistic regression iterations. F1 reported performance on only a single iteration, thereby avoiding frequent bootstrapping that is necessary for running multiple training iterations.

(4) Fully-packed bootstrapping takes an L=3 and N=64K ciphertext with an exhausted multiplicative budget and refreshes it by bringing it up to L=57, then performs the bootstrapping computation to obtain a usable ciphertext at a lower budget. The *fully-packed* version implies the ciphertext uses all $N/2$=32K available slots. Bootstrapping costs grow with the number of slots (both in multiplicative depth and compute).

We use the state-of-the-art fully packed bootstrapping algorithm [53], and use Lattigo's implementation [2] as the baseline. We tune CraterLake's bootstrapping implementation to maximize performance as discussed in Sec. 6.

For consistency, *we also use this bootstrapping algorithm in all benchmarks that require bootstrapping*. This is important, because this algorithm is not yet widely implemented in other libraries, so the original ResNet-20 and LogReg used much slower bootstrapping algorithms. In fact, the cost of older bootstrapping algorithms grows very quickly with the number of plaintext elements encoded in each ciphertext, so the baselines used *partially packed ciphertexts* (e.g., packing 128 elements per N=64K ciphertext) to reduce overall overheads. But with efficient bootstrapping, using fully packed ciphertexts is more efficient. For instance, we modify ResNet-20 to pack all channels into a single ciphertext before bootstrapping. This reduces the number of bootstrappings by 38× and improves performance on all hardware platforms by about 10×.

Shallow benchmarks come from F1 [25]. They include three neural networks from Low-Latency CryptoNets (LoLa) [13]. LoLa-MNIST is a simple, LeNet-style network used on the MNIST dataset [46], while LoLa-CIFAR is a 6-layer network (similar in computation to MobileNet v3 [37]) used on the CIFAR-10 dataset [45]. LoLa-MNIST includes two variants with unencrypted and encrypted

Execution time (ms) on	CraterLake	F1+	CPU	vs. F1+	vs. CPU
ResNet-20	249.45	2,693	23 min	10.8×	5,519×
Logistic Regression	119.52	639	356 s	5.34×	2,978×
LSTM	138.00	2,573	859 s	18.6×	6,225×
Packed Bootstrapping	3.91	58.3	17.2 s	14.9×	4,398×
deep gmean speedup				**11.2×**	**4,611×**
Unpacked bootstrapping	0.10	0.21	877	2.04×	8,612×
CIFAR Unencryp. Wghts.	50.50	94.1	187 s	1.86×	3,695×
MNIST Unencryp. Wghts.	0.14	0.13	561	0.97×	4,152×
MNIST Encryp. Wghts.	0.24	0.22	1369	0.88×	5,621×
shallow gmean speedup				**1.34×**	**5,220×**

Table 3: Performance of CraterLake, F1+, and CPU on full FHE benchmarks.

Speedup vs.	KSHGen	CRB/chain	Network
ResNet-20	2.0×	20.0×	1.7×
Logistic Regression	1.3×	8.8×	1.2×
LSTM	2.5×	34.5×	1.3×
Packed Bootstrapping	2.0×	27.4×	1.3×
deep gmean speedup	**1.9×**	**20.2×**	**1.3×**
Unpacked Bootstrapping	1.9×	3.7×	1.0×
CIFAR Unencrypt. Wghts.	1.0×	3.7×	2.0×
MNIST Unencrypt. Wghts.	1.1×	1.3×	1.5×
MNIST Encryp. Wghts.	1.1×	1.0×	1.3×
shallow gmean speedup	**1.2×**	**2.0×**	**1.4×**

Table 4: Speedups of CraterLake over configurations without KSHGen, CRB or chaining, or the fixed network.

weights; LoLa-CIFAR is available only with unencrypted weights. These benchmarks do not use bootstrapping and their max L is between 4 and 8.

Finally, we also benchmark *unpacked bootstrapping*, which bootstraps a ciphertext that packs a single element. This makes it shallower ($L \leq 23$) and less computationally demanding, but performance per slot is a lot worse than fully packed bootstrapping. Thus, unpacked bootstrapping is not used much in practice. We include it because it is the bootstrapping benchmark used in F1 [25].

Compared systems: We compare CraterLake with a CPU system. We use a 32-core, 64-thread, 3.5 GHz AMD Ryzen Threadripper PRO 3975WX; at 420mm^2 in a mix of 7nm and 12nm technology, this CPU has a comparable transistor count and power budget (280 W TDP) to CraterLake.

We also compare performance to prior accelerators, in particular to F1 [25]. For fairness, we evaluate *F1+*, a version of F1 that is scaled to a 256 MB 32-bank scratchpad, 32 compute clusters with 256 lanes each, and 1 MB register file per cluster. This makes F1+ have the same or higher throughput on basic operations as CraterLake. However, F1+ takes 636 mm^2, 35% more than CraterLake, because its network scales poorly: F1+'s on-chip network takes 160 mm^2, 16× more than CraterLake's fixed permutation network. This large overhead shows that CraterLake's novel hardware organization is crucial to scale to larger chips.

Finally, although F1 is tailored to standard keyswitching, boosted keyswitching becomes more efficient for $L > 14$. Thus, F1+ uses the most efficient keyswitching algorithm at each level. In short, *these changes allow comparing the F1 and CraterLake architectures* without the confounding factors of different hardware budgets or subpar algorithms.

9 EVALUATION

9.1 Performance

Table 3 compares the performance of CraterLake, CPU, and F1+ on deep and shallow benchmarks. It shows execution times and CraterLake's speedups over the CPU and F1+.

CraterLake achieves very large speedups over the CPU implementations, ranging from 2,978× to 8,612×. Speedups are similar across deep and shallow benchmarks, showing that CraterLake provides robust gains across FHE program types.

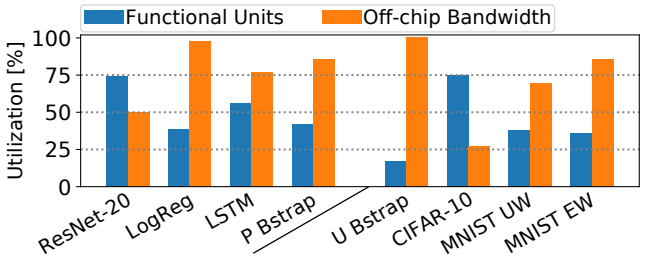

Figure 9: Utilization of functional units and main memory bandwidth.

CraterLake achieves large speedups over F1+ on deep benchmarks, outperforming it by gmean 11.2× and up to 18.6×. By contrast, CraterLake and F1+ achieve comparable speeds on shallow benchmarks, with a gmean speedup of only 1.34×. These stark differences demonstrate that *prior FHE accelerators are efficient only on shallow computations, and falter on deep ones due to the limitations that we have discussed.*

On the MNIST shallow benchmarks, CraterLake is slightly slower than F1+ because standard keyswitching is efficient in this range and CraterLake spends a large fraction of area on the CRB, which has low utilization at low L, whereas F1+ has higher NTT, add, and multiply throughput. But the high speedups on deep programs show that most compute happens at high L, so CraterLake optimizes for the common case.

9.2 Architectural Analysis

To understand these results in more depth, we examine CraterLake's resource utilization and power consumption.

Utilization: Fig. 9 reports the average utilization of FUs and main memory bandwidth for each application. FU utilization is reported as the fraction of cycles that FUs are consuming input values, averaged across all FUs. It reflects *issue rate*—for example, a utilization of 66% means that 10 out of the 15 FUs are consuming new inputs. Bandwidth utilization is simply the fraction of cycles memory is active, e.g., 70% utilization denotes an average bandwidth of 700 GB/s.

Overall, CraterLake achieves high utilization of both memory and compute, denoting a balanced system. Thanks to the CRB, CraterLake's FU mix is balanced across all ciphertext sizes, so FU utilization is always high unless memory limits throughput. Some

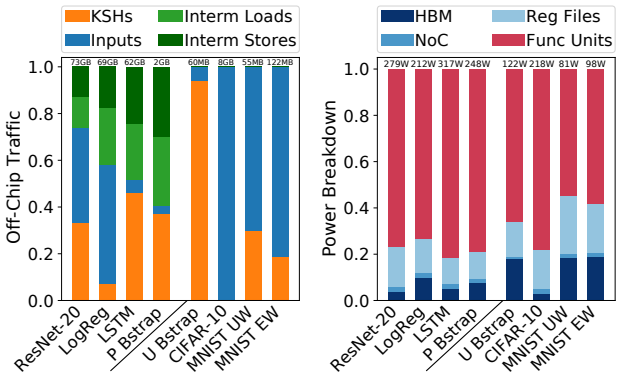

Figure 10: Per-benchmark breakdown of (a) data movement and (b) average power for CraterLake.

workloads, like unpacked bootstrapping, saturate on memory bandwidth, but most others see compute utilization near or above 50% (near 100% utilization is not achievable as all programs experience some memory-bound phases).

By contrast, F1+'s utilization is much lower, especially on deep benchmarks, where average FU utilization is 10%, owing to its inadequate FU mix and lack of CRB.

Data movement: Fig. 10a breaks down memory traffic by keyswitch hint (KSH), inputs, and intermediate loads and stores. We see that deep benchmarks incur a manageable amount of intermediate traffic, while shallow benchmarks have a sufficiently low footprint to cause no eviction of intermediates.*

Power consumption. Fig. 10b shows the power breakdown for CraterLake and the total power consumption for each benchmark. The figure includes both chip and HBM power. Power stays within a 320 W envelope, and is higher for deep benchmarks, primarily due to higher FU/memory utilization and higher internal CRB utilization. FUs dominate power across benchmarks, consuming 50-80%. This shows the importance of CraterLake's FU energy optimizations, and demonstrates that its architecture is far more efficient: F1 was dominated by data movement energy, and F1+ consumes gmean 18× more energy than CraterLake on our deep benchmarks. CraterLake's efficiency and performance benefits yield a gmean 201× improvement in performance per Joule over F1+.

9.3 Sensitivity Studies

On-chip storage: Fig. 11 shows CraterLake's performance as register file grows from 100 to 350 MB. Each line shows the performance of a different application, *normalized to its performance at the default size, 256 MB*. While shallow benchmarks are insensitive to storage size, most deep benchmarks suffer severely from a smaller register file, incurring slowdowns of up to 5.5×. This shows that CraterLake's large on-chip storage is crucial for deep benchmarks. Finally, adding more on-chip storage leads to diminishing returns: only packed bootstrapping sees significant improvements past 256 MB, reaching a 1.5× speedup with a 300 MB register file.

*For some of the shallow benchmarks from F1, the F1 paper [25] reports higher input traffic than Fig. 10a. This is because F1's evaluation used plaintexts that are the same size as ciphertexts (each plaintext fits in half the size of a ciphertext).

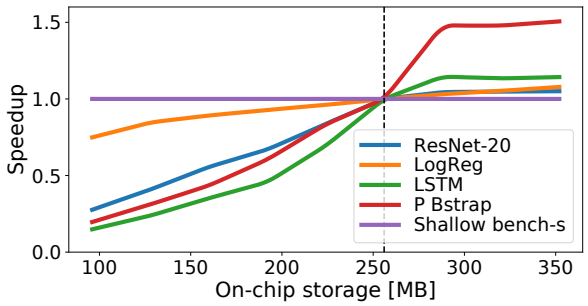

Figure 11: Gmean performance of CraterLake on deep benchmarks as a function of on-chip register file size.

Execution time (ms) for	128-bit	vs. 80-bit	200-bit	vs. 80-bit
ResNet-20	321.26	1.29×	588.70	2.36×
Logistic Regression	121.91	1.02×	123.10	1.03×
LSTM	223.56	1.62×	596.16	4.32×
Packed Bootstrapping	6.33	1.62×	17.01	4.35×
gmean slowdown		**1.36×**		**2.60×**

Table 5: Performance of CraterLake at 128-bit security and at 200-bit (which uses N=128K instead of 64K) with comparisons to performance at 80-bit security.

Effect of CraterLake's features: Table 4 shows the impact of our innovations by reporting the *slowdown* of alternative configurations. *KSHGen* omits the KSHGen FU and stores full KSHs in memory. This hurts performance noticeably, especially in deep benchmarks, by gmean 1.9× and by up to 2.5×. *CRB/chain* omits the CRB and the vector chaining optimizations. Performance is even worse than for F1+, because the system becomes bottlenecked on register file ports and CraterLake has 50% of the NTT and 40% of the multiply/add throughput of F1+. *Network* replaces CraterLake's fixed transpose network and polynomial tiling design with F1+'s crossbar-based network and residue polynomial tiling design. Performance is up to 2× worse *even though the F1+ network is 16× larger*, because residue polynomial tiling incurs 2.4× more traffic than CraterLake's tiling approach (Sec. 4.3).

9.4 Performance vs. Target Security Level

The benchmarks presented so far meet an 80-bit security level, which is often considered sufficient [25, 31, 34, 39, 40]. We now analyze the impact of meeting higher levels of security: 128-bit security, which is also a commonly used target [4, 48, 60] and can be efficiently achieved with N=64K, and 200-bit security, a very conservative target that requires using larger polynomials.

128-bit security: Table 5 shows CraterLake's performance on our benchmarks when adjusted for 128-bit security (128-bit column), as well as the slowdown compared to performance for 80-bit security.

To reach 128 bits of security with a polynomial degree of N=64K, we bootstrap twice as often as with 80 bits of security, i.e., target half the number of usable levels after bootstrapping. This allows using boosted keyswitching variants with a relatively small number of digits: we use 1-digit keyswitching for L <32, 2-digit keyswitching for 32≤ L <43, and 3-digit keyswitching for L ≥43. As we bootstrap twice as often, we never go beyond L=51.

Table 5 (left) shows that 128-bit security adds modest overheads: a 1.36× gmean slowdown, and a worst-case slowdown of 1.62×. These overheads stem from the higher memory footprint of multi-digit keyswitching and the more frequent bootstrapping. Importantly, though bootstrapping is twice as frequent, slowdowns are below 2×, because both bootstrapping and useful computation happen at lower L values, lowering their cost. This is a concrete example of the tradeoff that Fig. 3 in Sec. 2.3 illustrates.

Beyond 128-bit security: Effectively supporting significantly more than 128-bit security requires using larger polynomials. Here, we evaluate a 200-bit security target, which requires doubling N from 64K to 128K. Note that this security target is very conservative, and not used in FHE benchmarks; we use it to study performance over a wide range of security levels.

CraterLake as evaluated so far supports N up to 64K natively, but larger vectors would require multiple passes through FUs and forgo FU chaining. We thus evaluate a different CraterLake configuration that supports N up to 128K natively. This requires modest hardware changes, chiefly *(1)* the buffers in the CRB need to double, from 26.25 MB to 52.5 MB; and *(2)* NTTs require an additional butterfly stage. These changes consume 27.4 mm^2 of additional area, i.e., less than 6% of chip area.

Table 5 (right) reports the performance of deep benchmarks for 200-bit security and N=128K. Since N=128K ciphertexts have double the slots of N=64K ciphertexts, we normalize performance per element. This is because doubling N allows doubling the number of slots (plaintext elements) per ciphertext, which enables more computations to happen in parallel. For example, consider the ResNet benchmark: starting from the original benchmark, which uses N=64K ciphertexts and performs one inference at a time, it is easy to construct a ResNet benchmark that uses N=128K and performs two inferences in parallel. In addition to doubling N, CraterLake achieves 200-bit security by using higher-digit keyswitching.

Table 5 shows that the 200-bit security target imposes additional overheads, incurring a 2.6× gmean slowdown over 80-bit security, and a worst-case slowdown of 4.35×. Most of these slowdowns are caused by the fact that using N=128K doubles the footprint of ciphertexts and KSHs over N=64K. This limits reuse opportunities and adds off-chip traffic. While doubling the register file (to 512 MB) would erase most of these overheads, it would add significant area.

10 ADDITIONAL RELATED WORK

We now present related work not discussed so far.

HE-MPC accelerators are hampered by communication: To avoid the overheads of FHE, recent work has proposed accelerators for private deep learning that combine shallow homomorphic encryption (HE) with multi-party computation (MPC): Gazelle [43] and Cheetah [59]. These systems require very frequent communication with the client, essentially after every single level of multiplication. So while they reduce accelerator overheads, they are limited by high client-server communication and client encryption/decryption overheads. Delphi [52] shows that each DNN inference takes gigabytes of traffic, which dominates performance. However, the above accelerators do not consider this traffic. CHOCO [64] shows that, even after accelerating client operations, communication costs dominate.

By dramatically accelerating bootstrapping, CraterLake avoids the high communication costs of HE-MPC and shallow HE designs. To avoid bootstrapping, these prior approaches require the *client* to receive, re-encrypts, and resend ciphertexts that have exhausted their multiplicative budgets. In our benchmarks, avoiding each bootstrapping would require transferring over 13 MB between client and server (as the client must resend the ciphertext with a reasonable noise budget). Even if we ignore client computation and network latency, on a 100 Mbps connection this would require over 1 *second* per ciphertext. By contrast, CraterLake bootstraps this ciphertext in 3.9 ms, 256× faster. Bootstrapping also allows the client to send narrow inputs (e.g., with 32 instead of 1,500 bits per coefficient), which the server can bootstrap before computation. This greatly reduces encryption and network overheads.

GPUs are inefficient on FHE: Prior work has studied the use of GPUs to accelerate FHE [3, 42, 65–67]. While the data-parallel nature of GPUs may seem a good fit for FHE, these efforts achieve modest speedups over multicore CPUs. This is because GPUs lack modular arithmetic, cannot implement all-to-all operations like NTTs and automorphisms efficiently, and their on-chip memories are too small to enable sufficient reuse (Fig. 11), despite their use of HBM. Specifically, state-of-the-art GPU approaches carefully tune algorithms to achieve high off- and on-chip bandwidth utilization [42]; however, this prior work [42] is 200× slower than CraterLake. This shows that CraterLake's high reuse is crucial: to achieve the same throughput as CraterLake, a GPU would need over 100 TB/s of memory bandwidth.

11 CONCLUSION

For widespread adoption of FHE, accelerators must efficiently support deep computations. CraterLake is the first accelerator to achieve this. By adopting state-of-the-art algorithms and using them to design CraterLake, we target a new regime of FHE not explored by prior approaches. Through new architectural and compiler techniques, CraterLake addresses the overheads of deep computations and provides an order-of-magnitude speedups over prior accelerators, which scale poorly to the accelerator sizes required to process very large ciphertexts, and are inefficient on the algorithms needed by deep computations. As a result, CraterLake enables new applications for FHE, such as real-time inference using deep neural networks like ResNet or LSTMs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, Hyun Ryong Lee, Quan Nguyen, Yifan Yang, Victor Ying, Shabnam Sheikhha, Fares Elsabbagh, Robert Durfee, Nithya Attaluri, and Joel Emer for feedback on the paper; we thank Joon-Woo Lee for helping us set up the ResNet benchmark, and Ronald Dreslinski for help with synthesis. This research was funded in part by the Defense Advanced Research Projects Agency (DARPA) under contract number Contract No. HR0011-21-C-0035, and by a Wistron research grant. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] 2020. HEAAN software library. https://github.com/snucrypto/HEAAN.

[2] 2020. Lattigo. https://github.com/ldsec/lattigo.

[3] Ahmad Qaisar Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. 2021. Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. *IEEE Transactions on Emerging Topics in Computing* 9, 2 (2021).

[4] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard.* Technical Report. HomomorphicEncryption.org.

[5] Martin R Albrecht, Benjamin R Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W Postlethwaite, Fernando Virdia, and Thomas Wunderer. 2018. Estimate all the {LWE, NTRU} schemes!. In *Proceedings of the International Conference on Security and Cryptography for Networks (SCN).*

[6] Krste Asanovic. 1998. *Vector Microprocessors.* Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[7] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. 2016. A full RNS variant of FV-like somewhat homomorphic encryption schemes. In *Proceedings of the International Conference on Selected Areas in Cryptography (SAC).*

[8] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966).

[9] Fabrice Benhamouda, Tancrède Lepoint, Claire Mathieu, and Hang Zhou. 2017. Optimization of bootstrapping in circuits. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).*

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. 2018. KangarooTwelve: Fast hashing based on Keccak. In *Proceedings of the 16th International Conference on Applied Cryptography and Network Security (ACNS).*

[11] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT).*

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014).

[13] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. 2019. Low latency privacy preserving inference. In *Proceedings of the International Conference on Machine Learning (ICML).*

[14] Hao Chen, Ilaria Chillotti, and Yongsoo Song. 2019. Improved bootstrapping for approximate homomorphic encryption. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT).*

[15] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT).*

[17] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and innovation. *IEEE Micro* 41, 2 (2021).

[18] David Bruce Cousins, John Golusky, Kurt Rohloff, and Daniel Sumorok. 2014. An FPGA co-processor implementation of Homomorphic Encryption. In *Proceedings of the IEEE Conference on High Performance Extreme Computing (HPEC).*

[19] David Bruce Cousins, Kurt Rohloff, Chris Peikert, and Rick Schantz. 2012. An update on SIPHER (Scalable Implementation of Primitives for Homomorphic EncRyption) - FPGA implementation using Simulink. In *Proceedings of the IEEE Conference on High Performance Extreme Computing (HPEC).*

[20] D. B. Cousins, K. Rohloff, and D. Sumorok. 2017. Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor. *IEEE Transactions on Emerging Topics in Computing* 5, 2 (2017).

[21] Sal Dasgupta, Teja Singh, Ashish Jain, Samuel Naffziger, Deepesh John, Chetan Bisht, and Pradeep Jayaraman. 2020. Radeon RX 5700 Series: The AMD 7nm Energy-Efficient High-Performance GPUs. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC).*

[22] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[23] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[24] Yarkin Doröz, Erdinç Öztürk, and Berk Sunar. 2015. Accelerating fully homomorphic encryption in hardware. *IEEE Trans. Comput.* 64, 6 (2015).

[25] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *Proceedings of the 54th annual IEEE/ACM international symposium on Microarchitecture (MICRO-54).*

[26] Harvey L Garner. 1959. The residue number system. In *Papers presented at the the March 3-5, 1959, Western Joint Computer Conference.*

[27] Craig Gentry and Shai Halevi. 2019. Compressible FHE with Applications to PIR. In *Proceedings of the Theory of Cryptography Conference (TCC).*

[28] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Proceedings of the 32nd Annual Cryptology Conference (CRYPTO).*

[29] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Proceedings of the 33rd Annual Cryptology Conference (CRYPTO).*

[30] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the International Conference on Machine Learning (ICML).*

[31] Shai Halevi and Victor Shoup. 2018. Faster homomorphic linear transformations in HElib. In *Proceedings of the 38th Annual International Cryptology Conference (CRYPTO).*

[32] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481.

[33] Shai Halevi and Victor Shoup. 2020. *HElib design principles.* Technical Report.

[34] Shai Halevi and Victor Shoup. 2021. Bootstrapping for HElib. *Journal of Cryptology* 34, 1 (2021).

[35] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2018. Efficient Logistic Regression on Large Encrypted Data. Cryptology ePrint Archive, Report 2018/662.

[36] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic regression on homomorphic encrypted data at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33.

[37] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Adam Hartwig. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV).*

[38] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. 2021. CoSA: Scheduling by constrained optimization for spatial accelerators. In *Proceedings of the 48th annual International Symposium on Computer Architecture (ISCA-48).*

[39] Malika Izabachène, Renaud Sirdey, and Martin Zuber. 2019. Practical fully homomorphic encryption for fully masked neural networks. In *Proceedings of the International Conference on Cryptology and Network Security (CANS).*

[40] Jyun-Neng Ji and Ming-Der Shieh. 2019. Efficient comparison and swap on fully homomorphic encrypted data. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS).*

[41] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44).*

[42] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)* (2021).

[43] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18).*

[44] Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. 2020. Semi-parallel logistic regression for GWAS on encrypted data. *BMC Medical Genomics* 13, 7 (2020).

[45] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images.* Technical Report. University of Toronto.

[46] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998).

[47] Junghyun Lee, Eunsang Lee, Joon-Woo Lee, Yongjune Kim, Young-Sik Kim, and Jong-Seon No. 2021. Precise approximation of convolutional neural networks for homomorphically encrypted data. *arXiv preprint arXiv:2105.10879* (2021).

[48] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. 2021. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *arXiv preprint arXiv:2106.07229* (2021).

[49] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On ideal lattices and learning with errors over rings. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT).*

[50] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and implementation of encryption/decryption architectures for BFV homomorphic encryption

scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2019).

[51] Vincent Migliore, Cédric Seguin, Maria Mendez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, Guy Gogniat, and Russell Tessier. 2017. A High-Speed Accelerator for Homomorphic Encryption using the Karatsuba Algorithm. *ACM Transactiosn on Embedded Computer Systems (TECS)* 16, 5s (2017).

[52] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference service for neural networks. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*.

[53] Christian Mouchet, Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and J Hubaux. 2020. Lattigo: A multiparty homomorphic encryption library in Go. In *8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*.

[54] NVIDIA. 2021. NVIDIA DGX Station A100 system architecture. https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-station-a100-system-architecture-white-paper.pdf.

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

[56] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. 2019. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the 24th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*.

[57] Robert Podschwadt and Daniel Takabi. 2020. Classification of encrypted word embeddings using recurrent neural networks. In *Workshop on Privacy in Natural Language Processing (PrivateNLP @ WSDM)*.

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

[59] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S Lee, and David Brooks. 2021. Cheetah: Optimizations and methods for privacy preserving inference via homomorphic encryption. In *Proceedings of the 27th IEEE international symposium on High Performance Computer Architecture (HPCA-27)*.

[60] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An architecture for computing on encrypted data. In *Proceedings of the 25th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*.

[61] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *Proceedings of the 25th IEEE international symposium on High Performance Computer Architecture (HPCA-25)*.

[62] Richard M Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (1978).

[63] Furkan Turan, Sujoy Roy, and Ingrid Verbauwhede. 2020. HEAWS: An accelerator for homomorphic encryption on the Amazon AWS FPGA. *IEEE Trans. Comput.* (2020).

[64] McKenzie van der Hagen and Brandon Lucia. 2022. Client-optimized algorithms and acceleration for encrypted compute offloading. In *Proceedings of the 27th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII)*.

[65] Wei Wang, Zhilu Chen, and Xinming Huang. 2014. Accelerating leveled fully homomorphic encryption using GPU. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*.

[66] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. 2012. Accelerating fully homomorphic encryption using GPU. In *Proceedings fo the IEEE conference on High Performance Extreme Computing (HPEC)*.

[67] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. 2013. Exploring the feasibility of fully homomorphic encryption. *IEEE Trans. Comput.* 64, 3 (2013).

[68] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. 2020. Interstellar: Using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the 25th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*.

[69] Geoffrey Yeap, S. S. Lin, Y. M. Chen, H. L. Shang, P. W. Wang, H. C. Lin, Y. C. Peng, J. Y. Sheu, M. Wang, X. Chen, B. R. Yang, C. P. Lin, F. C. Yang, Y. K. Leung, D. W. Lin, et al. 2019. 5nm CMOS Production Technology Platform featuring full-fledged EUV, and High Mobility Channel FinFETs with densest 0.021um2 SRAM cells for Mobile SoC and High Performance Computing Applications. In *Proceedings of the 2019 IEEE International Electron Devices Meeting (IEDM)*.