A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption

A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption

ALEKSANDAR KRASTEV*, NIKOLA SAMARDZIC*, SIMON LANGOWSKI, SRINIVAS DEVADAS, and DANIEL SANCHEZ, Massachusetts Institute of Technology, USA

Fully Homomorphic Encryption (FHE) enables computing on encrypted data, letting clients securely offload computation to untrusted servers. While enticing, FHE has two key challenges that limit its applicability: it has high performance overheads (10,000× over unencrypted computation) and it is extremely hard to program. Recent hardware accelerators and algorithmic improvements have reduced FHE's overheads and enabled large applications to run under FHE. These large applications exacerbate FHE's programmability challenges.

Writing FHE programs directly is hard because FHE schemes expose a restrictive, low-level interface that prevents abstraction and composition. Specifically, FHE requires packing encrypted data into large vectors (tens of thousands of elements long), FHE provides limited operations on these vectors, and values have noise that grows with each operation, which creates unintuitive performance tradeoffs. As a result, translating large applications, like neural networks, into efficient FHE circuits takes substantial tedious work.

We address FHE's programmability challenges with the Fhelipe FHE compiler. Fhelipe exposes a simple, numpy-style *tensor* programming interface, and compiles high-level tensor programs into efficient FHE circuits. Fhelipe's key contribution is *automatic data packing*, which chooses data layouts for tensors and packs them into ciphertexts to maximize performance. Our novel framework considers a wide range of layouts and optimizes them analytically. This lets Fhelipe compile large FHE programs efficiently, unlike prior FHE compilers, which either use inefficient layouts or do not scale beyond tiny programs.

We evaluate Fhelipe on both a state-of-the-art FHE accelerator and a CPU. Fhelipe is the first compiler that matches or exceeds the performance of large hand-optimized FHE applications, like deep neural networks, and outperforms a state-of-the-art FHE compiler by gmean 18.5×. At the same time, Fhelipe dramatically simplifies programming, reducing code size by 10×–48×.

CCS Concepts: • **Software and its engineering → Compilers**; • **Security and privacy → Cryptography**.

Additional Key Words and Phrases: fully homomorphic encryption, tensors, automatic bootstrapping

ACM Reference Format:
Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption. *Proc. ACM Program. Lang.* 8, PLDI, Article 152 (June 2024), 25 pages. https://doi.org/10.1145/3656382

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) is an emerging class of encryption that allows computing directly on encrypted data. FHE enables offloading computation to untrusted servers in the cloud with cryptographic privacy. While enticing, FHE is rarely used today due to two key challenges: it has high performance overheads, and it is extremely hard to program.

*Both authors contributed equally to this work.

Authors' address: Aleksandar Krastev, alexalex@csail.mit.edu; Nikola Samardzic, nsamar@csail.mit.edu; Simon Langowski, slangows@mit.edu; Srinivas Devadas, devadas@csail.mit.edu; Daniel Sanchez, sanchez@csail.mit.edu, Massachusetts Institute of Technology, USA.

ACM 2475-1421/2024/6-ART152
https://doi.org/10.1145/3656382

Currently, FHE programs are about 10,000× slower than their unencrypted equivalents when run on a CPU. This has sparked work that has countered most of these overheads: FHE accelerator chips [45–47, 74, 75] are about 10,000× faster than CPUs; and recent GPU [42] and FPGA [5, 85] implementations achieve speedups beyond 100×. For example, deep neural networks like ResNet can be evaluated in seconds on a GPU using modern FHE libraries and optimizations [66].

As FHE infrastructure and optimizations enable large and complex applications, it becomes crucial to tackle the programmability challenges of FHE through new compiler techniques. While recent work has proposed compilers for FHE, they either leave significant performance on the table or tackle only small programs with tens of operations. Thus, large applications are still coded by hand, a tedious process that takes weeks or months of work. In this paper, we present a compiler called Fhelipe that, for the first time, translates *large applications* into efficient FHE programs that match or outperform state-of-the-art, painstakingly optimized hand-coded programs.

Why is FHE hard to program? Fig. 1 shows how FHE execution works. A client wants to compute an expensive function f (e.g., a DNN inference) on private data x. The client encrypts x and sends it to the server, which uses FHE to compute $f(x)$ directly on encrypted data x, and returns the encrypted result. To do this, the server runs f_{FHE}, an *FHE implementation of* f, i.e., an implementation using the datatypes and operations supported by FHE.

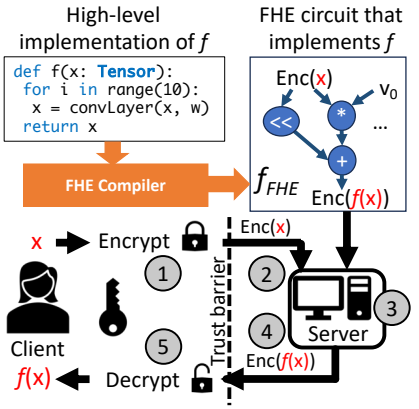

Fig. 1. Overview of FHE execution. FHE computations are circuits that are hard to write manually. An FHE compiler automatically produces an FHE circuit from a high-level program.

Because data x is encrypted, FHE programs are circuits (i.e., static dataflow graphs), as it's impossible to perform data-dependent operations (e.g., branching) on encrypted data. Though FHE is not Turing-complete, it is sufficient to implement a broad class of computations. But writing f_{FHE} by hand is hard. The goal of FHE compilers, as shown in Fig. 1, is to produce f_{FHE} from an implementation of f in a higher-level language.

Specifically, modern FHE schemes like CKKS [18] expose a restrictive and low-level *vector* interface with three characteristics that complicate programming:

(1) *Each ciphertext encrypts a huge vector*, tens of thousands of elements long.
(2) *Each encrypted vector supports limited operations*: addition, multiplication, and rotations.
(3) *Encrypted vectors have noise that grows with each operation*, especially multiplications. This requires expensive noise management operations to preserve correctness. Thus, minimizing the circuit's *multiplicative depth* is key for performance.

How do programmers cope with FHE's challenges? Using FHE's huge vectors well requires filling them with data. But applications do not have vectors with tens of thousands of elements, so the general strategy is to pack FHE vectors with *larger data structures*, like a matrix, tensor, or tile. This allows implementing many kernels, like matrix-vector multiply, as massively data-parallel procedures (e.g., operating on the whole matrix at once) despite FHE's limited operation set.

The key challenge to pack data well is choosing a good *data layout*, i.e., how multidimensional data is packed into vectors. Even if FHE vectors start fully packed, kernels perform reductions that induce large *gaps* of empty vector elements, and leave output elements in a different order than the input. Unlike conventional SIMD processing, FHE does not allow accessing individual vector elements, and reordering elements is very expensive. Thus, programmers manually codesign kernels and data layouts to *(1)* achieve good packing, and *(2)* avoid costly conversions. *This often requires reimplementing a kernel for many different data layouts.*

As a result, finding efficient layouts manually is very tricky (Sec. 2.2). For example, Lee et al.'s state-of-the-art FHE ResNet introduces a new layout for efficient multi-channel convolutions [51–53], that relies on a complex interleaving, and builds on years of prior work on this problem [13, 31, 44].

Why are current FHE compilers insufficient? Recent FHE compilers recognize that data layouts are key, but they either leave substantial performance on the table or only handle tiny programs.

CHET [26], HECO [79], and HeLayers [6] are FHE compilers that abstract data layouts: programmers use a high-level language, and they automatically pack data into FHE vectors. But these compilers use a limited set of layouts (e.g., row-major for CHET, column-major for HECO, and the same layout throughout the program for HeLayers), which forces either inefficient packing or expensive conversions. Moreover, these compilers rely on profiling to choose layouts, which is expensive and greatly limits their search space. Though these compilers scale to large programs, their limited layouts cause *order-of-magnitude overheads* over hand-coded benchmarks.

Prior work has also proposed compilers based on program synthesis, Porcupine [23] and Coyote [58]. These are analogous to superoptimizers [59]: they produce high-performance FHE circuits, but work only on tiny kernels with tens of instructions [23, 58] and cannot scale to large applications.

To tackle these challenges, we present *Fhelipe*, a tensor compiler that translates applications written in a simple, numpy-style tensor language to efficient FHE programs. Fhelipe relies on two key novel contributions:

1. A flexible framework to represent and optimize data layouts, achieving the dual goals of maximizing packing and minimizing costly layout conversions. Our approach builds on four new techniques. First, we introduce a new *layout representation* that enables many packing choices, including arbitrary dimension orders and interleaved dimensions. This representation captures the complex layouts used in real-world programs. Second, we contribute a novel *compaction* technique that leverages our flexible layouts to pack data with little cost and achieve high utilization. Third, we contribute an *analytical layout assignment* procedure that chooses layouts throughout the program to minimize the number and cost of layout conversions. This process systematically negotiates complex choices, and avoids the limitations of prior profiling-based approaches. Fourth, we contribute novel layout conversion techniques that reduce the cost of necessary conversions.

2. Automatic end-to-end noise management, including bootstrapping: Compiling large FHE programs requires addressing one more challenge beyond layouts: to cope with noise, programs must perform auxiliary *noise management* operations to prevent data corruption. Prior compilers like EVA and HECATE [25, 55] have automated the local aspects of noise management (rescaling). However, large programs require *bootstraps*, expensive operations that reset ciphertext noise. Bootstraps often dominate execution time, and placing them well is hard because it requires reasoning about global program structure; optimal bootstrap placement is NP-hard [9], and the only prior automatic technique, FHE-booster [83], places bootstraps poorly (Sec. 8.1).

We present a new scalable algorithm for automatic bootstrap placement. We identify a set of simple heuristics that reduces this problem to dynamic programming. This lets Fhelipe efficiently place bootstraps in close to linear time.

Together, these contributions enable abstraction and composition. Even local changes to an FHE kernel often change the layout and noise of its output. Without a compiler, this requires *(1)* rewriting all downstream kernels to use compatible layouts and *(2)* performing noise management from scratch. Fhelipe automates these, drastically reducing programmer effort: for example, Lee et al.'s hand-coded ResNet takes 4,800 lines of code [52]; with Fhelipe, it takes just 100.

We implement Fhelipe targeting CKKS on both CPUs and CraterLake [75], a state-of-the-art FHE accelerator. We evaluate Fhelipe on several complex FHE programs, including neural networks, logistic regression training, and higher-dimensional tensor kernels. Fhelipe is the first compiler to

match or exceed the performance of these large and carefully hand-optimized FHE applications, with speedups of up to 12.3×; Fhelipe also outperforms CHET+, which is CHET+EVA extended with automatic bootstrapping, by gmean 18.5×.

2 BACKGROUND AND MOTIVATION

In this section, we first present the necessary background on FHE, focusing on the CKKS scheme targeted by Fhelipe; then, we show the importance of data layouts in FHE; finally, we discuss prior compilers and their limitations.

2.1 FHE Schemes

There are two types of FHE schemes: *vector* schemes like BGV and CKKS [11, 12, 18, 29], encrypt long vectors of numbers and provide arithmetic operations on them, and *scalar* schemes like FHEW/TFHE [21, 27] encrypt one value per ciphertext, typically a Boolean or a small integer.

Scalar schemes are more flexible than vector ones, but they have have much higher overheads, especially in data-parallel applications. For example, in ResNet-20, the Lattigo CPU CKKS library takes 31μs on average per application-level scalar multiply (Table 5); the TFHE-rs [86, 88] library takes 2.1s per 32-bit multiply, over 60,000× slower. For this reason, Fhelipe targets CKKS [18], the state-of-the-art vector scheme. But Fhelipe's techniques apply to all vector schemes, as they all have the same operations and performance trade-offs.

We introduce CKKS's *interface*, i.e., its datatypes and operations, without delving into its *implementation*. We present only the details needed to understand CKKS's tradeoffs between performance, security, and correctness; full implementation details are available in prior work [18, 69, 75].

Ciphertexts encrypt long vectors: In CKKS, each ciphertext encrypts a vector of n fixed-point numbers. Each ciphertext has a *scale* parameter, s, that determines the width of the fractional part of each element; typically, s is between 30–60 bits.

Internally, ciphertexts are represented using two polynomials with integer coefficients modulo some value. Typically, coefficient bitwidth w is over 1,000 bits, much larger than s.

Due to security requirements (that we detail later), n must be quite large, typically 32K or 64K. As leaving vector slots unused does not reduce operation costs, applications must find ways to fill these large vectors to avoid ineffectual work.

FHE provides a limited set of operations: Ciphertexts support only three operations, called *homomorphic operations*: elementwise *adds*, elementwise *multiplies*, and *cyclic rotates* of the underlying encrypted vectors (add and multiply allow their second operand to be unencrypted).

FHE programs are *static dataflow graphs* of these operations: since data is encrypted, there is no data-dependent control flow and all operations are known in advance.

Note that FHE provides no way to access individual vector elements. As a result, shuffling vector elements is tremendously expensive, requiring many rotates and multiplies; this is a major difference between FHE and conventional SIMD processing, as in vector instructions or GPUs.

Homomorphic operations have costs that vary with vector length n and coefficient width w:[1]

- Adds of all types, and multiplies of an encrypted and an unencrypted vector are cheap, costing $O(n \cdot w)$.
- Rotates, and multiplies of two encrypted vectors are expensive, costing $O(n \cdot w^2)$.

However, these *direct* costs tell only part of the story: operations also indirectly affect the cost of *other* operations due to noise, which we discuss next.

[1]Costs are given in aggregate *bit-complexity* per homomorphic operation; they are derived from a state-of-the-art CKKS implementation that is optimized with RNS representation, NTTs, and multi-digit keyswitching [63, 75].

Performance depends on multiplicative depth: For security, ciphertexts are encrypted with a small amount of noise. Unfortunately, homomorphic operations increase noise, and if noise becomes too large, it corrupts the underlying encrypted values. Noise grows primarily due to multiplies: each one adds about s bits of noise. Adds and rotates add negligible noise.

There are two main techniques to keep noise in check:
(1) Noise is trimmed by progressively *narrowing* w. In CKKS, w is reduced by about s bits after each multiply through two operations, rescaling and mod-switching. Thus, each ciphertext supports $l \approx w/s$ noise trims before running out of bitwidth; we call l the *level* of the ciphertext. Rescaling and mod-switching are cheap and prior work has already proposed effective ways to apply them automatically [25, 54, 55].
(2) Once w cannot be narrowed further (l reaches 0), the ciphertext must be *bootstrapped*, a procedure that lowers noise and restores the ciphertext to a high w, letting it undergo more operations. Though bootstraps enable arbitrarily deep computations, they are extremely expensive: they involve hundreds of rotates and multiplies of high-w ciphertexts. Thus, they should be minimized.

Fig. 2 shows how w evolves in a typical program: w decreases as noise is trimmed, then resets up during bootstrapping. Since multiplies consume w, the number of bootstraps depends on the application's *multiplicative depth* (i.e., the longest chain of multiplies). And since bootstraps dominate cost in most FHE programs, reducing multiplicative depth is in practice far more important than minimizing direct operation costs.

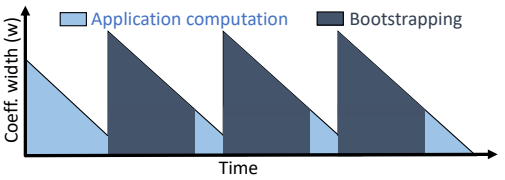

Fig. 2. Noise management affects ciphertext coefficient width (w). Narrowing w after operations trims noise, and bootstrapping produces a low-noise, high-w ciphertext.

2.2 Data Layout Is Crucial

Selecting good layouts is crucial in FHE. We first show this concretely with a simple example, then discuss how this problem affects more complex prior applications.

Consider a workload that consists of successive matrix-vector multiplications: each step multiplies an m-element vector with an $m \times m$ matrix, producing an m-element output vector for the next step. This workload arises frequently, in e.g., fully connected or recurrent neural network (RNN) layers.

FHE ciphertexts have thousands of slots, and using all of these slots is crucial for performance. For concreteness, assume m=128-element vectors and n=16K-slot ciphertexts. Placing each matrix row in a separate ciphertext would be tremendously inefficient, as each ciphertext would use only 128 of these slots. Instead, we must pack data further: we store the entire $m \times m$ matrix in one ciphertext, using all its 16K slots. Fig. 3 shows this example scaled down to m=4 and n=16.

To perform the matrix-vector multiply efficiently, we replicate vector v to match the shape of matrix A (Fig. 3b, step 1), then compute all partial products using a single FHE multiply (step 2), and finally sum each row's partial products to produce the output vector (step 3). This procedure is efficient because replicate and sum are relatively cheap in FHE, performing only $\log m$=7 rotates and adds; overall, this computes matrix-vector multiply in multiplicative depth 1 and modest overhead.

The problem is that this procedure *leaves the output in a different format than the input*: while the input x's elements are contiguous, the output y's elements have gaps of $m-1$ unused slots. These gaps cannot be removed efficiently: rotates are the only mechanism for moving elements between slots, and each of the m output elements must be rotated by a different amount. Converting y back to x's contiguous layout requires 128 rotates and masks, which would add about 10× overhead.

The right approach is to use *different* layouts for successive matrix-vector multiplies: instead of converting y to follow x's layout, we can find a different procedure that uses y as-is. In this simple example, this is achieved by alternating row-major and column-major layouts on successive

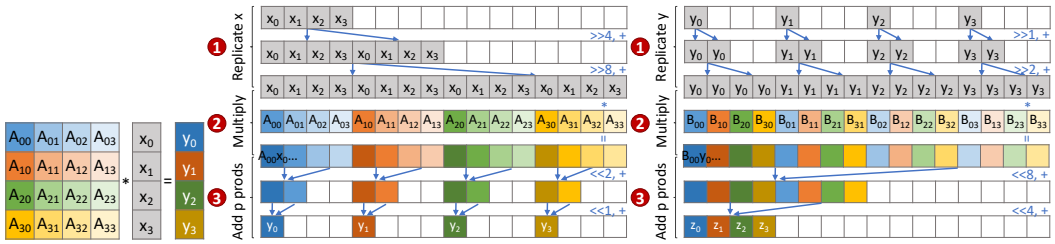

Fig. 3. Example workload consisting of successive 4×4 matrix-vector multiplies: (a) $A \cdot x = y$. (b) Row-major implementation of $A \cdot x = y$ using n=16-slot ciphertexts. The resulting ciphertext y has gaps that are expensive to remove. (c) Column-major implementation of $B \cdot y = z$, which allows using y's ciphertext from (a) as-is.

matrix-vector multiplies, as Fig. 3c shows: a column-major procedure takes an input with gaps, and produces an output with contiguous elements.

Stitching these layouts manually is tedious: changing the layout of a tensor requires rewriting all downstream computation. And this example only scratches the surface of the issues in finding good layouts in complex applications.

For a more complex example, consider deep learning. In 2018, GAZELLE [44] proposed a clever layout for convolutional layers that allowed packing across input or output channels, but not both. In 2022, Lee et al. [52] invented a more complex layout that allows full packing, even with striding, and improves ResNet performance by about 10×. This layout uses a tricky interleaving of elements (see Sec. 5.6 and [52, Fig.3, Fig.5]). Similarly, LoLa [13] achieves large speedups over CryptoNets [31] through the use of efficient data layouts.

In summary, experts spend substantial effort to find data layouts, and the key contribution in many FHE application papers is a new layout. Fhelipe is the first compiler that finds and systematically optimizes these layouts, matching or outperforming these manual implementations.

2.3 Prior FHE Compilers

Prior FHE compilers automate important aspects of FHE programming, but have key limitations. Table 1 summarizes the main differences among these compilers.

Tensor compilers: *CHET* [26] is a domain-specific compiler for neural networks. CHET abstracts the data layout of FHE programs, like Fhelipe. However, CHET is very different from Fhelipe: First, it provides a limited interface that supports only a few coarse-grained operations (e.g., convolutions and fully connected layers); by contrast, Fhelipe exposes a general tensor programming interface that enables a broad set of applications beyond the specific neural nets targeted by CHET. Second, CHET considers only row-major layouts; Fhelipe considers a far wider range of layouts, where dimensions can be in arbitrary orders and interleavings. Third, CHET compares layout choices by profiling, which limits it to evaluate only four layout combinations per program; instead, Fhelipe selects layouts analytically, without profiling, and systematically produces programs that combine hundreds of different layouts in linear time. As Sec. 8 shows, CHET's limited layouts cause large overheads: gmean 18.5× in neural networks and up to 7,600× in tensor applications.

Other tensor compilers improve some aspects of CHET, but share many of its limitations. *HECO* [79] is a more general compiler than CHET that works by automatically vectorizing scalar loop nests to FHE. But like CHET, HECO uses a fixed layout (column-major) for all tensors. AHEC [16] compiles a range of machine learning frameworks down to different hardware backends, but it also does not optimize layouts.

Table 1. Prior FHE compilers have key limitations. By contrast, Fhelipe supports automatic layout assignment, automatic bootstrap placement, programs with millions of operations, and a general tensor-based interface.

	Auto Layouts	Many Layouts	Auto Bootstrap	Large Programs	General Interface
CHET [26],	✓	✗	✗	✓	✗
HECO [79], HeLayers [6]	✓	✗	✗	✓	✓
EVA [25], HECATE [55], ELASM [54]	✗	✗	✗	✓	✓
Porcupine[23]	✗	✗	✗	✗	✓
Coyote[58]	✓	✓	✗	✗	✓
Fhelipe	✓	✓	✓	✓	✓

nGraph-HE2 [10] and SEALion [77] are also compilers for neural networks, but they use only one ciphertext slot per inference and rely on *batching* to use more slots. Batching many inferences together makes vectorization easy, but it is impractical: using all slots requires batching n=32K inferences. Individual clients cannot provide this many inferences, and on large networks like ResNet-20, batched activations take 100s of GB of memory [52, 75].

Finally, *HeLayers* [6] is a compiler for neural networks that supports a wider range of layouts than row- or column-major. But these are only a small subset of Fhelipe's layouts, which limits packing. Moreover, HeLayers picks a single layout for the whole program, and uses profiling to select it. Thus, HeLayers has similar overheads to CHET: its evaluation reports similar performance on single inferences, and HeLayers outperforms CHET substantially only when batching is used [6].

Vector compilers: *EVA* [25], *HECATE* [55], and *ELASM* [54] abstract key details of CKKS, preventing several correctness and security bugs. Their main contributions are efficient techniques for inserting *rescaling* (managing noise by trimming coefficients, Sec. 2.1). However, these compilers expose a vector interface that leaves data layouts to programmers. Their techniques are orthogonal to our contributions, and in fact, Fhelipe adopts EVA's *waterline rescaling*.

Alchemy [24], E3 [20], Marble [81], and T2 [33] are also vector compilers, but they do not optimize rescaling.

Program synthesis: *Porcupine* [23] and *Coyote* [58] use program synthesis techniques to optimize tiny FHE kernels. Like Fhelipe, Coyote optimizes layouts, whereas Porcupine leaves them to programmers. These techniques work well on small programs, but are limited to programs with only *tens* of scalar operations: they are so expensive that they would take years to compile any real-world application. For example, Coyote [58] takes ≈10s per scalar operation. Extrapolating linearly (generous, since its search algorithms are superlinear), ResNet-20 (120M operations) would take 3.8 years. By contrast, Fhelipe leverages scalable optimizations to compile programs with millions of operations in seconds (e.g., 15s for ResNet-20).

Automatic bootstrapping: All compilers mentioned so far target small applications and do not perform bootstrapping. FHE-Booster [83] is recent work that automates bootstrapping. However, FHE-Booster uses score-based heuristics that achieve limited speedups and incur pathologies, as we show in Sec. 8. Moreover, these heuristics have superlinear runtime, and take many minutes or fail to complete in some circuits [82]. By contrast, Fhelipe uses heuristics to reduce the problem to dynamic programming, which it solves optimally in close to linear time.

DaCapo [19] is concurrent work that, like Fhelipe, automates bootstrapping by applying heuristics to reduce the search space and then optimizing in polynomial time.

Scalar FHE compilers: Several prior compilers target scalar FHE schemes [14, 15, 22, 32, 50, 80, 87], like TFHE. These compilers have different objectives from vector ones: scalar schemes encrypt one value per ciphertext, so packing is unnecessary, and they bootstrap after every operation. However, as Sec. 2.1 discussed, scalar schemes have much higher overheads than vector ones.

3 FHELIPE OVERVIEW

Fig. 4 shows an overview of Fhelipe. Fhelipe takes as input a program written in a simple *tensor language* (Sec. 4) that hides FHE details.

Fhelipe first parses the program to produce a *dataflow graph* (DFG) of tensor operations, which is refined in successive passes. Then, Fhelipe assigns *layouts* to tensors (Sec. 5), inserting layout conversions when needed. Next, Fhelipe applies noise-management techniques: waterline rescaling first, and then automatic bootstrapping (Sec. 6).

Finally, Fhelipe lowers tensor operations to CKKS homomorphic operations on vector ciphertexts. The resulting circuit can be executed by a variety of backends: Fhelipe currently supports the Lattigo CPU FHE library [63] and the CraterLake FHE accelerator [75]. Adding other backends (e.g., other CPU [1, 3, 7, 34] and GPU [43] libraries) would be easy: for scale, the Lattigo backend is only 400 lines of code (2% of the codebase).

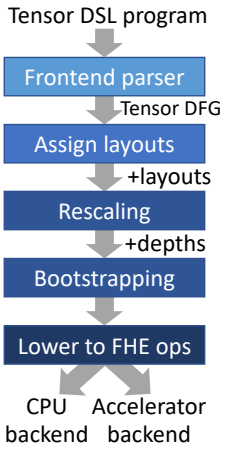

Fig. 4. Fhelipe overview.

4 FHELIPE PROGRAMMING INTERFACE

Fhelipe's input language represents data as *tensors*: multidimensional arrays of fixed-point numbers. Table 2 details the language's native operations. This is a Python-embedded DSL, providing usual conveniences like functions and loops. Overall, this interface is similar to numpy [39] and other tensor languages, like those provided by PyTorch [68] and TensorFlow [4]. Listings 1 and 2 show two basic examples: matrix-vector multiply and convolution. Note that, while tensors have a logical shape, their ciphertext layout is left *unspecified*. For instance, Listing 1 can be synthesized using both the row-major and the column-major layouts from Sec. 2.2 (and many others).

Because Fhelipe enables composability, programmers can reuse procedures like these to build more complex ones. We implement a simple standard library that we reuse across applications. It

Table 2. Fhelipe's native operations on tensors.

Operation	Description
`t + u`	Elementwise add
`t * u`	Elementwise multiply
`t.shift(dim: int, by: int)`	Shift along *dim*
`t.rotate(dim: int, by: int)`	Cyclic shift along *dim*
`t.extend(dim: int, size: int)`	Zero-pad *dim* up to *size*
`t.shrink(dim: int, size: int)`	Shrink *dim* down to *size*
`t.stride(dim: int, by: int)`	Discard indices $i \not\equiv 0 \pmod{by}$; *by* must be a power of 2
`t.drop_dim(dim: int)`	Discard a dimension of size 1
`t.insert_dim(dim: int)`	Insert a dimension of size 1
`t.reorder_dim(p: List[int])`	Permute dimensions (e.g., a transpose)
`t.sum(dim: int)`	Sum along dimension *dim*, discarding it
`t.replicate(dim: int, n: int)`	Copy tensor *n* times, forming a new dimension *dim*

```
def mv_mul(m: Tensor, v: Tensor) -> Tensor:
 v_ext = v.replicate(dim=1, n=m.shape[1])
 products = m * v_ext
 return products.sum(dim=0)
```

Listing 1. Multiplication of $N \times M$ matrix `m` with M-column vector `v`.

```
def conv2d(img: Tensor, wgts: Tensor):
 C, H, W = img.shape
 K, C, R, S = wgts.shape
 # img_r and wgts_r both contain R · S
  #  tensors with shape K × C × H × W.
 img_r = [
  img.shift(dim=0,by=i).shift(dim=1,by=j)
     .replicate(dim=3, n=K)
      for i in range(-(S//2), (S+1) // 2)
      for j in range(-(R//2), (R+1) // 2)
 ]
 wgts_r = replicate_conv_wgts(wgts, H, W)
 return sum(i*w for i, w in
            zip(img_r, wgts_r)).sum(dim=2)
```

Listing 2. 2D convolution on image (`img`) with weights (`wgts`). There are C input and K output channels; channels are $H \times W$; filters are $R \times S$.

includes common kernels (e.g., convolutional and fully connected layers) and non-linear functions (e.g., ReLU and sigmoid), which in FHE are approximated using polynomials.

5 AUTOMATIC DATA PACKING

Fhelipe's key contribution is a framework to represent, analyze, and choose tensor layouts that efficiently pack data into FHE's enormous vectors.

FHE has unique restrictions and optimization goals that are not present in unencrypted computation. Prior work, including tensor compilers [4, 68], tensor optimizers [65, 72, 84, 89], and automatic vectorization techniques [17, 60, 70, 78], optimizes data layouts to use SIMD datapaths and systolic arrays well, reduce data transformations and shuffles, and tile to reduce data movement. By contrast, FHE requires optimizing layouts to pack much larger vectors and minimize multiplicative depth, while coping with operations that create large gaps and avoiding expensive layout conversions. New techniques are necessary to reason about these tradeoffs and choose appropriate layouts.

Fhelipe's layout framework combines four novel contributions. First, we introduce a *flexible layout representation* (Sec. 5.1) that supports arbitrary dimension orders, interleavings of dimensions, and gaps. This representation generalizes the wide range of packing choices proposed in prior FHE applications, enables new optimizations, and reduces data transformations. Second, we introduce a *compaction* procedure (Sec. 5.2) that leverages our flexible layouts to keep ciphertexts highly packed. Third, we present a layout assignment algorithm (Sec. 5.4) that operates analytically and without profiling, by reasoning about the work added by conversions induced by incompatible layouts. Fourth, novel FHE permutation algorithms (Sec. 5.5) reduce the cost of needed conversions.

These contributions open a wide range of layouts and enable the compiler to optimize them quickly. We showcase these new capabilities with two end-to-end examples (Sec. 5.6).

5.1 Flexible Layout Representation

To motivate the need for flexible layouts, consider again the matrix-vector multiply example in Sec. 2.2. If we restricted all tensors to a row-major layout, we would miss the efficient implementation that alternates row-major and column-major layouts. To enable this, layouts need two key ingredients. First, they need to support arbitrary dimension orders (e.g., row-major and column-major in this case). Second, they need to support *gaps*, runs of empty slots that arise during tensor operations like striding or summing. For instance, in Fig. 3b the $m \times m$ output vector y has a stride of m, with $m - 1$ gaps between each element, due to the summing of partial products. To avoid conversions, we must track and cope with these gaps.

Fhelipe's layout representation is even more flexible than just allowing arbitrary dimension orders and gaps: it allows arbitrary permutations of the *bits* of each index.

Layout definition: Consider a tensor with dimension indices i, j, k, ..., each with a different number of bits I, J, K, ... Let the string $S = (i_{I-1}, ..., i_0, j_{J-1}, ..., j_0, k_{K-1}, ...)$ be the concatenation of the individual bits of all indices. Then, a *layout* of this tensor is any permutation of the elements of S, with *gap bits* (denoted with G) interleaved arbitrarily.

For example, in Fig. 5, t's layout is (i_1, i_0, j_1, j_0), i.e., row-major. This encodes the mapping of tensor indices to slots in the vector: for instance, element t_{23} (row i=2_{10}=10_2, col j=3_{10}=11_2) maps to slot $i_1 i_0 j_1 j_0$=1011_2=11_{10} in the underlying vector. Similarly, in Fig. 3b, x's layout is (i_1, i_0), and y's layout is (i_1, i_0, G, G), denoting stride-4 gaps.

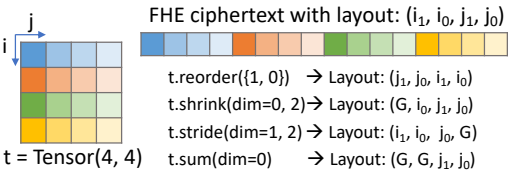

Fig. 5. Fhelipe's layouts implement complex tensor operations as simple changes of layout, without moving elements between ciphertext slots.

Fig. 5 shows the flexibility of this representation: it allows performing many operations by

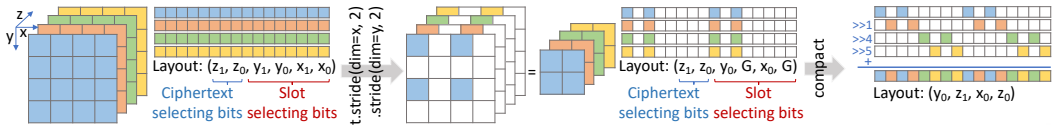

Fig. 6. Example showing ciphertext-selecting and slot-selecting bits, and how compaction removes gaps.

simply changing the tensor layout (the permutation of index bits), *without changing the underlying ciphertexts*. Fig. 5 shows that transposing two dimensions simply swaps their layout bits; and shrinking a dimension, striding by a power of two, and sum-reducing simply introduce gap bits.

Additionally, allowing the bits of each dimension to be out of order reduces the costs of more complex transformations, like compaction, which we will see in detail later.

Restrictions: The key limitation of this layout format is that dimensions whose size is not a power of two need padding (e.g., a 3×3 matrix would have one element unused between rows). However, padding non-power-of-two dimensions is a good choice overall: it simplifies layout conversions, and it is a natural fit for FHE vectors, which are power-of-two sized for performance reasons.

Slot- and ciphertext-selecting bits: As ciphertexts have only n slots, larger tensors must be stored across multiple ciphertexts. By convention, the lowest-order $\log_2 n$ bits of S encode the slot of each element, and the remaining $(|S| - \log_2 n)$ bits encode its specific ciphertext. We call these bits *slot-selecting* and *ciphertext-selecting*, respectively.

For example, Fig. 6 (left) shows a 4×4×4 tensor with layout $(z_1, z_0; y_1, y_0, x_1, x_0)$. With n=16-slot ciphertexts, bits z_1 and z_0 are ciphertext-selecting, and the rest are slot-selecting. We use ";" to denote the boundary between ciphertext- and slot-selecting bits.

Since we can access ciphertexts individually (unlike slots within the ciphertext), ciphertext-selecting bits are more flexible than slot-selecting bits: they can be reordered for free and gaps can be discarded with no overhead.

5.2 Compaction

As we have seen, common operations like striding and summing introduce gaps in slot-selecting bits. Gaps cause low utilization, as they leave many ciphertext slots unused. With restrictive layouts (e.g., row- or column-major), eliminating these gaps would require an expensive format conversion (Sec. 2.2). But our flexible layouts make eliminating gaps cheap, by *converting ciphertext-selecting bits into slot-selecting bits*, replacing the gap bits. We call this process *compaction*.

Fig. 6 shows compaction at work. A 4×4×4 tensor that takes four 16-slot ciphertexts is strided in the x and y dimensions, creating a 4×2×2 tensor with two gap bits in its layout. Compaction merges these four ciphertexts into a single ciphertext, by filling gap bits with ciphertext-selecting bits. This produces a tensor with layout (y_0, z_1, x_0, z_0).

Fig. 6 (right) shows that compaction is relatively cheap: ciphertexts have the same gap pattern, so they can be combined using only one rotate and add each, without consuming any levels. This also shows why we allow the bits of a dimension to be out of order: cheap compaction would have been impossible if z's bits had to be contiguous.

Compaction's compute cost is quickly recouped: compacting by $N\times$ takes N-1 rotates and adds, and reduces downstream computation by $N\times$. Thus, compaction *breaks even after a single rotate or multiply*, and brings large savings when followed by more expensive operations like polynomials or bootstraps. As a result, compaction lets us eliminate all inefficiencies due to gaps: compaction is automatically performed on any tensor that spans multiple ciphertexts, and gaps are unavoidable for small tensors that already fit within a single ciphertext.

5.3 Implementation of Tensor Operations

Fhelipe's layout representation makes it easy to perform operations efficiently on all tensor layouts:
Reshapes only change the tensor's layout string (`extend`, `shrink`, `stride`, `drop_dim`, `insert_dim`, `reorder_dim`), but do not modify the underlying data.
`replicate` and `sum` perform a logarithmic sequence of rotates and adds. Both operations rotate by the powers of 2 corresponding to the bits of the replicated or summed dimension: in `replicate`, each rotate-add doubles the number of replicated copies; in `sum`, each rotate-add produces partial sums on subsets of double the size.
`shift`, `rotate`, and layout conversions require reordering the elements of the underlying FHE vectors; Fhelipe implements them using a unified approach for permutations, described in Sec. 5.5.
+ and * perform elementwise adds or multiplies on the underlying FHE vectors. This requires both inputs to have the same layout, which is ensured by conversions inserted during Fhelipe's layout assignment pass (Sec. 5.4).

To simplify compaction, Fhelipe keeps gap elements set to 0. Thus, sequences of operations that introduce gaps (`shrink`, `stride`, `sum`) must mask out discarded elements. This is done by multiplying with an unencrypted vector of 0 and 1 values.

5.4 Analytical Layout Assignment

Fhelipe assigns tensor layouts analytically, without any profiling. It uses a forward pass that assigns initial layouts, enhanced with *backtracking* to reconsider and improve layout choices.

The forward pass traverses the dataflow graph in topological order, starting from program inputs. It assigns initial layouts using a simple procedure: *(1)* each program input is assigned row-major layout, *(2)* each unary operation consumes its input in its current layout, which determines the layout of its output, and *(3)* each operation that produces gaps (`stride`, `shrink`, `sum`) performs *compaction* (Sec. 5.2) to fill them when possible.

So far, this requires no layout conversions. However, binary operations (+ and *) need both inputs to have the same layout. So, when the pass reaches a binary operation with mismatched input layouts, it initiates *backtracking* to insert a layout conversion.

Backtracking independently considers converting each of the two inputs. For each, backtracking inserts a conversion at the input, and then attempts to *hoist* that conversion earlier in the program, where it may become cheaper or unnecessary. Each step of hoisting moves the conversion from the output of an operation to its inputs by deterministically finding input layouts that would produce the output in the desired layout. Hoisting proceeds greedily while the cost of the conversion decreases or stays the same (in terms of rotate groups, Sec. 5.5). At the end, backtracking is left with two options for resolving the mismatch; it picks the cheaper one.

Backtracking works well because the cost of layout conversions varies drastically. In the best case, the conversion can be hoisted all the way to a program input, where it can be completely avoided by just changing the input's initial layout. But there are also other cases where the conversion can be made cheaper. For example, consider a matrix-vector multiply $A \cdot x = y$ (as in Fig. 3) where A and x have mismatched layouts: x is in row-major format (as in Fig. 3b), but A is in column-major format (like B in Fig. 3c). At the element-wise multiply, converting either input—A or the replicated x vector—is equally expensive. But hoisting the conversion before x's replication makes it much cheaper, as it requires permuting m=128 times fewer elements.

Our implementation traverses only linear chains of operations (fan-in and fan-out of 1) to keep runtime linear. While more sophisticated implementations are possible (e.g., backtracking through the entire graph), we find that this suffices to find efficient layouts for all the applications we study.

5.5 Permutations

Even with a good layout assignment, layout conversions are sometimes needed. Fhelipe implements layout conversions and tensor shifts using a unified framework for permuting vector elements.

Single-stage permutations: In principle, any permutation can be implemented as a dataflow graph with multiplicative depth 1. Let a rotate *group* be a subset of vector elements that the permutation rotates by the same amount. Then, we implement a single-stage permutation by *(1)* isolating each group into a separate vector via masking, *(2)* rotating each group vector individually, and *(3)* summing the rotated vectors. Unfortunately, permutations can have $g = O(n)$ rotate groups, resulting in $O(n)$ vector operations taking $O(n^2)$ time.

Decomposed permutations: To reduce runtime, we decompose permutations into a sequence of stages. Each stage is a permutation with a small number of rotate groups. Picking the number of stages is non-trivial: adding stages reduces the number of operations but increases multiplicative depth, as each stage performs masking. Thus, using too many stages can hurt performance by forcing more frequent bootstrapping. Empirically, we find that limiting stages to g_{max}=16 groups balances work and depth, and works well in practice.

For a given number of stages k, the best general permutation algorithm requires $n^{2/k}$ groups per stage [34]. But this is far worse than the theoretical lower bound of $g^{1/k}$, especially when $g \ll n$. To avoid this inefficiency, we propose decomposition algorithms, described below, that exploit the structure in the permutations induced by each operation.

Decomposing layout conversions: Layout conversions reorder the bits of the layout, with a conversion that moves b bits requiring $g = 2^b$ groups. Fhelipe decomposes conversions so that each stage (except the last) moves $b_{max} = \log_2(g_{max}) = 4$ bits of the layout. Each stage can be constructed greedily so that it reduces b by at least $b_{max} - 1$, thus reducing g by at least a factor of $(g_{max}/2)$. As a result, Fhelipe's layout conversions perform at most 2× more rotates than the theoretical lower bound.

Decomposing shifts: A tensor shift by s moves index i to index $\equiv i + s \pmod{2^I}$. Fhelipe decomposes shifts so that the first j stages move i to index $\equiv i + s \pmod{2^{I_j}}$: the first j stages add the first I_j bits of $i + s$. Fhelipe chooses I_j greedily under the constraint that no stage exceeds g_{max} groups.

We compare our tensor shift algorithm against the lower bound empirically (obtaining an analytical bound is hard because the behavior of shifts has a complex dependence on both the layout and the shift amount). For n=32K, Fhelipe's shift algorithm performs on average 1.68× (standard deviation 14.0%, max 3.10×) more rotates than the theoretical lower bound. We compute this by randomly sampling 200,000 layouts and simulating all possible shift amounts (results don't change after about 2,000 samples).

5.6 Putting It All together: Layouts in ResNet and LogReg

We show Fhelipe's layouts in action through two examples:

1. CNN layer with striding: A convolutional neural network (CNN) layer processes an *input activation* tensor with C channels of $H \times W$ elements, and produces an *output activation* tensor with K channels of $P \times Q$ elements. The convolution *weights* are $K \times C$ filters of $R \times S$ elements.

```
1 def convLayer(img: Tensor,
2       wgts: Tensor, s: int):
3   x = conv2d(img, wgts)
4   y = x.stride(dim=0, by=s) \
5         .stride(dim=1, by=s)
6   return ReLU(y)
```

Listing 3. CNN layer with striding and ReLU activations.

A key part of CNNs are *bottleneck layers*, where each $P \times Q$ output channel is smaller than each $H \times W$ input channel, and the output has more channels than the input ($K > C$). *Striding* is a common way to achieve this: striding by s discards all but every s-th row and column, so each output channel is $H/s \times W/s$.

Listing 3 shows the Fhelipe code for a CNN layer with striding, representative of ResNet. This code reuses `conv2d` from Listing 2. Fig. 7 shows Fhelipe's implementation when the input activation is $2 \times 4 \times 4$, weights are $4 \times 2 \times 2 \times 2$, and the stride is s=2, resulting in $4 \times 2 \times 2$ output activations. For simplicity, each ciphertext holds n=16 elements.

The dataflow graph in Fig. 7 shows the tensor operations. Some operations list the corresponding Fhelipe code and ciphertext operations. The drawings on the right show the tensors, layouts, and ciphertext contents like in Fig. 6. Dashed lines separate graph regions with different layouts (labeled 1 – 5). We emphasize which layout bits change on the right.

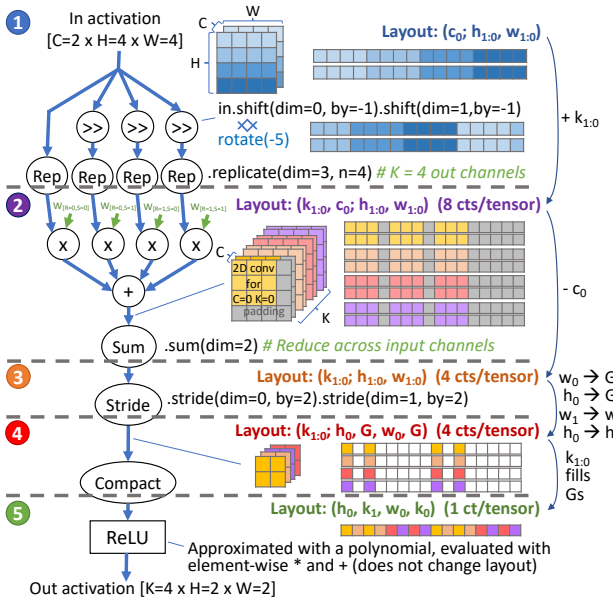

Fig. 7. Fhelipe detailed layouts in strided convolution, for a simplified example with 16-element ciphertexts. Fhelipe's compaction produces a fully packed output.

The input tensor is in row-major layout and takes 2 ciphertexts 1. Convolution first shifts the input to produce 4 tensors, each aligned with an element of the 2×2 filters. This does not change layouts. Then, each tensor is replicated K=4 times. This adds $k_{1:0}$ as ciphertext-selecting bits, so each tensor now takes 8 ciphertexts 2. Next, the tensors are multiplied by the weights and added together; layouts remain the same. Last, the combined tensor is summed along its C=2 dimension: as c_0 is ciphertext-selecting, this involves summing ciphertexts pair-wise and does not introduce gaps 3.

Striding is challenging in FHE, because it creates gaps. Here, striding with s=2 discards every other element of h and w, replacing their least significant layout bits h_0 and w_0 with gaps G 4. Note how h_1 and w_1 in the input become h_0 and w_0 in the output. Fhelipe automatically fills these gaps through compaction 5: ciphertext-selecting bits k_1 and k_0 fill the G bits, reducing the tensor from 4 to 1 ciphertext (just as in Fig. 6). Lee et al.'s multiplexed-convolutions layout is a specific implementation of this technique for ResNet [52]; however, Fhelipe generalizes this technique, and is the first to apply it automatically.

Finally, a non-linear activation function (ReLU) produces the output activations. In FHE, non-linear functions are approximated with polynomials, which consist only of element-wise adds and multiplies. These element-wise operations do not change the layout. Fhelipe allows implementing any polynomial; prior work has proposed a range of accurate [53] and cheap [41, 62, 67] activation functions for FHE.

Due to compaction, the output layout (h_0, k_1, w_0, k_0) has the bits of dimension k in non-consecutive slot-bits. Thus, the next layer must use that layout for its input. With Fhelipe, this happens automatically and without any conversions: the shifts in Fig. 7 induce different rotations (and masking if needed); Fhelipe automatically chooses the right format for filter weights; and the sum-reduction on C is done with rotates and adds.

2. LogReg: Fig. 8 shows Fhelipe's layouts in logistic regression training, an end-to-end FHE application (Sec. 7). Here, we show full tensor sizes and ciphertext layouts instead of using reduced

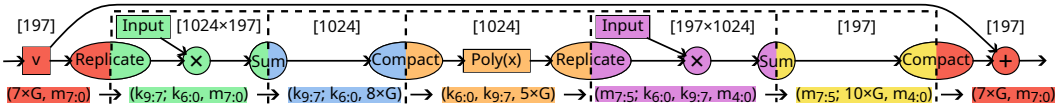

Fig. 8. Fhelipe layouts in LogReg. Fhelipe's flexible layouts and assignment algorithm keep tensors fully packed and avoid costly layout conversions.

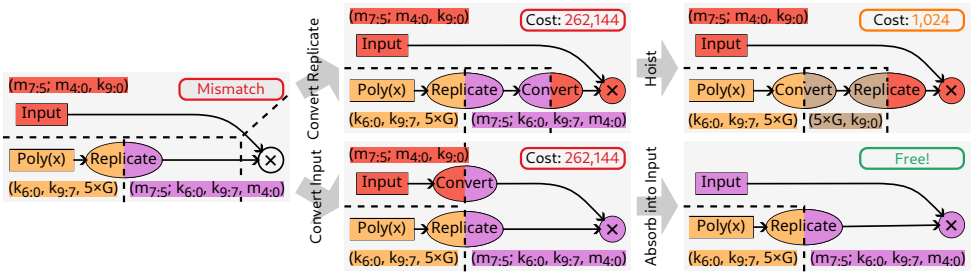

Fig. 9. Backtracking resolves layout mismatches by considering converting each operand, and then hoisting these conversions to reduce their cost. Here, the conversion can hoisted to an input, eliminating it completely.

dimensions. We show one LogReg iteration, which consists of two matrix-vector multiplies with a non-linear activation in between. There are k=1024 data points with m=197 features; ciphertexts have n=32K slots. Colors in Fig. 8 denote different layouts, which are listed at the bottom.

First, note how Fhelipe's flexible layouts again enable fully using ciphertexts without expensive conversions. As in Sec. 2.2, v starts in row-major, `replicate` fills gaps, and `sum` creates new ones. But here, matrices are large and take 8 ciphertexts. So, after each matrix-vector multiply, Fhelipe compacts the vector back to 1 ciphertext. Like in the CNN layer, efficient compactions require complex layouts that are not just a reordering of dimensions.

Flexible layouts are crucial for performance. Using only row-major layouts forces a conversion after each `sum`, resulting in a 22.5× slowdown (Sec. 8.2). Further, the prior state-of-the-art solution [37], which simply alternates between row-major and column-major layouts, is also inefficient: its lack of compaction makes it 7× slower than Fhelipe due to extra bootstrapping (Sec. 8.1).

Second, LogReg demonstrates the need for backtracking. Fig. 9 shows backtracking in action. Initially, the second `input` (from Fig. 8) is given a row-major layout, $(m_{7:5}; m_{4:0}, k_{9:0})$, causing mismatched layouts at the multiply (Fig. 9 left). Backtracking resolves this by separately considering converting each of the two operands: the `replicate` (Fig. 9 top) and the `input` (Fig. 9 bottom). For each branch, backtracking first inserts a conversion at the operand; this is expensive because all layout bits mismatch, costing 2^{18}=262,144 rotate groups (Sec. 5.5). To reduce this cost, the conversion is then hoisted earlier in the computation: hoisting before `replicate` reduces the cost to 2^{10}=1,024, whereas absorbing the conversion into the `input` (by changing its layout) eliminates it completely. Finally, backtracking selects the cheaper option (eliminating the conversion). Without backtracking, the inserted conversions in would have increased runtime by about 10×.

6 AUTOMATIC BOOTSTRAP PLACEMENT

As discussed in Sec. 2.1, FHE ciphertexts have noise that grows during the computation. To avoid corruption, auxiliary noise management operations are required.

Rescale and mod-switch trim the noise of a ciphertext by reducing its coefficient bitwidth w, usually after each multiply. We associate each ciphertext with a *level* l: the number of noise trims it can go through before running out of bitwidth. All ciphertexts start at l=l_0, with l_0 typically ≈10.

Noise trimming is computationally cheap and prior work has found effective ways to apply it automatically; Fhelipe adopts EVA's waterline rescaling algorithm [25].

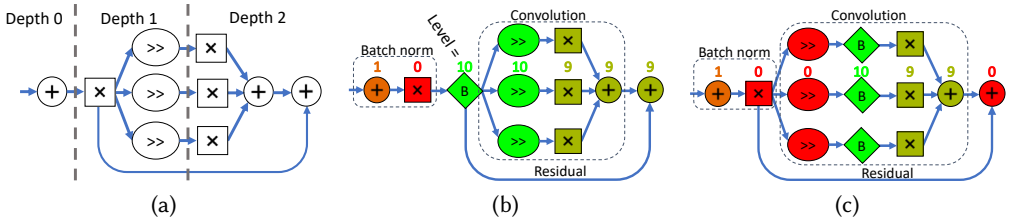

Fig. 10. Example of bootstrap placement. By partitioning the graph by depth (a), our dynamic programming approach (b) outperforms lazy bootstrapping (c), placing 3× fewer bootstraps (diamond *B* nodes).

Bootstrap resets a ciphertext back to the initial l_0, allowing for arbitrarily deep FHE programs. Bootstraps are very expensive, so minimizing them is crucial for performance. We propose a practical algorithm for placing bootstraps automatically.

6.1 Placing Bootstraps Is Challenging

Fig. 10 analyzes a simplified ResNet block with the input starting at l=1. As the graph is 2 multiplies deep (Fig. 10a), this forces a bootstrap. Then, it is best to bootstrap right after the batch normalization (Fig. 10b): this bootstraps only one ciphertext, and produces an output at a high level ($l_0 - 1$), which reduces the need for bootstraps in subsequent layers.

Systematically finding the above bootstrap placement is not easy (in fact, optimal placement is NP-hard [9]). As a comparison point, consider *lazy bootstrapping*: bootstrapping ciphertexts right before they run out of levels. While simple, this approach performs poorly (Fig. 10c). First, it inserts 3× more bootstraps, due to bootstrapping during the convolution, where computation is wide. Second, it produces an output at l=0, which forces an immediate bootstrap before the subsequent multiply. As a result, Fhelipe outperforms lazy bootstrapping by gmean 3.5× (Sec. 8.2).

6.2 Base Algorithm

We propose a novel heuristic that lets us place bootstraps using dynamic programming in close to linear time.

We define the *depth* of a node as the maximum number of noise trims (multiplies) on a path from an input to the node. Depth partitions the nodes of the graph, as shown in Fig. 10a. Depth and level are closely related: if there was no need for bootstraps, nodes at depth d would be at level $l = l_0 - d$.

Fhelipe's heuristic is to make bootstrapping decisions at *depth boundaries*: bootstrap either all edges crossing a boundary, or none of them. This avoids one of the main pitfalls we saw in Fig. 10c: lazy bootstrapping bootstraps all but one of the edges crossing the 1–2 depth boundary (the residual connection), producing an output at level 0; had it bootstrapped all of them, the output would have been at level l_0=10.

Then, Fhelipe uses dynamic programming to find the best boundaries to bootstrap at. Let *(1)* $dp[i]$ be the minimum cost of computing all nodes up to depth i, *(2)* $b[i]$ the cost of bootstrapping all ciphertexts crossing boundary i, and *(3)* $c[i][j]$ the cost of computing all values at depth i at level j. We compute $dp[i]$ recursively by choosing the best option for the last bootstrapped boundary j:

$$dp[i] = \min_{1 \le l \le l_0} \left(dp[i-l] + b[i-l] + \sum_{j=0}^{j<l} c[i-j][j] \right).$$

$dp[i-l]$ guarantees that no node up to depth $i-l$ falls below level 0; bootstrapping the ciphertexts crossing boundary $i-l$ ($b[i-l]$), together with $l \le l_0$, guarantees the same for the nodes between depths $i-l$ and i.

For a program with maximum depth d, computing $dp[d]$ takes $O(d \cdot l_0)$ time; since l_0 is a small constant ($\approx$10), this is practically linear in the size of the program. We obtain $b[i]$ and $c[i][j]$ from CraterLake's per-operation costs [75]; other cost models can be used as well.

6.3 Additional Optimizations

Narrowing depth boundaries: Any node at depth i that does not trim noise can be treated as having depth i+1 without violating correctness. For instance, the rotates in Fig. 10a can be treated as having depth 2 instead of 1. This reduces $b[1]$, the cost of bootstrapping at the 1–2 depth boundary, from 3 ciphertexts to 1. Fhelipe uses this to reduce each $b[i]$ by running a simple min-cut algorithm.
Omitting shortcut bootstraps: Not all edges crossing a boundary need to be bootstrapped. This is common for *shortcut* edges that skip over multiple depths, like the residual connection in Fig. 10. Specifically, shortcuts that go from a higher- to a lower-level node do not need to be bootstrapped. Fhelipe exploits this by greedily omitting such shortcuts from $b[i]$. To implement this correctly, Fhelipe tracks the mapping from depth to level for each $dp[i]$; using persistent append-only lists [64], this increases runtime only by a factor of $\log d$.

7 METHODOLOGY

7.1 Benchmarks

We evaluate Fhelipe on a diverse set of challenging benchmarks. Table 3 summarizes their features.
Large FHE applications: Fhelipe seeks to compile large applications, which exacerbate programmability issues. But prior FHE compilers use simple benchmarks, like small kernels [23, 58, 79] with tens of scalar operations or shallow neural networks that require no bootstrapping [25, 26]. To test Fhelipe's capabilities against well-optimized baselines, we use three *large FHE programs* that have been manually developed by FHE experts. Each program is the state-of-the-art in its domain, and is beyond the reach of prior FHE compilers:
(1) ResNet-20 is one of the most complex neural networks to be ported to FHE. Our manual baseline is Lee et al.'s implementation [52], which uses state-of-the-art layouts. ResNet is a deep convolutional network with a non-linear structure, including skip connections that complicate data layouts. It approximates ReLU activation functions with high-degree polynomials, which achieve high accuracy but are much costlier than the low-degree approximations used by simpler FHE networks [13, 41]. As a result, ResNet-20 requires frequent bootstrapping. Each execution performs one inference using images from the CIFAR-10 dataset [48].
(2) RNN is an NLP benchmark that performs sentiment analysis using a Recurrent Neural Network [28]. Our manual baseline follows Podschwadt's algorithm [71], enhanced with the data layouts proposed by Samardzic et al. [75]. RNN processes a sequence of 200 word embeddings x_i, and incorporates each in its hidden state following $h_{i+1} = \sigma(W_{hh}h_i + W_{ih}x_i + b)$. $\sigma(\cdot)$ is a degree-3 approximation of $\tanh(\cdot)$, and x_i and h_i are both of dimension 128. The chain of $W_{hh}h_i$ matrix-vector multiplies has a similar structure to the workload in Sec. 2.2. We use the IMDB dataset [57].
(3) LogReg performs logistic regression to train a linear binary classifier. LogReg is one of the few FHE applications that *trains* an ML model, instead of performing inference. Our manual baseline is state-of-the-art HELR [37]. LogReg performs 32 iterations of Nesterov Accelerated Gradient Descent [73] with batch size 1024 and 197 features per sample; the sigmoid activation is approximated by a degree-7 polynomial. We use the MNIST dataset [49].

Table 3 shows that these applications perform tens to hundreds of millions of scalar operations (when unencrypted), and their multiplicative depth is in the hundreds, requiring frequent bootstraps.
Shallow neural network: Since prior compilers do not perform bootstrapping, we also use a shallow neural network that does not require it:

Table 3. Characteristics of our benchmarks: fixed-point scale s and total scalar operations; Fhelipe's multiplicative depth, compilation time, and lines of application code; lines of code in the manual implementation.

			Fhelipe			Manual	
Benchmark	s	Ops.	Depth	Compile	LoC	References	LoC
ResNet-20	45	120M	412	14.7s	100	[52, 76]	4,800
RNN	45	13M	802	1.6s	80	[71, 75]	—
LogReg	35	77M	224	27.3s	60	[36, 37]	600
LoLa-MNIST	35	1M	10	0.8s	50	[13, 61]	1,700
FFT	45	41M	59	18.5s	85	*new*	
TTM	45	34M	2	1.7s	7	*new*	
MTTKRP	45	34M	4	1.7s	9	*new*	

(4) LoLa-MNIST is a LeNet-style network from Low-Latency CryptoNets (LoLa-Dense) [13] that uses sophisticated layouts. It has unencrypted weights and uses the MNIST dataset [49].

Tensor kernels: Finally, we include three *tensor kernels* that would be hard to code manually in FHE. These kernels have no prior implementation, and showcase Fhelipe's generality:

(5) FFT computes the Fast Fourier Transform of a vector of 128K samples.

(6) TTM computes the third-order tensor-matrix product $A_{ijk} = \sum_l B_{ijl}C_{kl}$ [56]; all dimensions have size 64.

(7) MTTKRP computes the matricized tensor times Khatri-Rao product $A_{ij} = \sum_{kl} B_{ikl}C_{kj}D_{lj}$ [40]; all dimensions have size 64.

7.2 Compared Systems

We compare Fhelipe against carefully ported manual baselines and CHET+, an extension of the CHET compiler that incorporates EVA [25] and other improvements. CHET+ is representative of state-of-the-art FHE compilers; in Sec. 8.2, we also compare with other relevant prior work, including FHE-Booster [83] and HeLayers [6].

Fhelipe: We implement Fhelipe in 21,000 lines of Python and C++ code. Fhelipe compiles all applications in under a minute using a single CPU thread (Table 3). Compilation time scales linearly with program size.

Fhelipe automatically chooses the n and w CKKS parameters to meet a user-provided security level (128-bit security by default) [8]. It uses n=32K as it is the smallest n that allows for bootstrapping with 128-bit security. Then, benchmarks with bootstrapping (ResNet-20, RNN, LogReg, FFT) use the maximum w ciphertext modulus bitwidth allowed by the security level (w=1,552 for 128-bit security), whereas benchmarks without bootstrapping use the minimum w sufficient to cover their multiplicative depth. LoLa-MNIST uses n=16K to match the manual implementation.

Fhelipe leaves to the user the choice of the application fixed-point scale s. For the evaluation, applications with manual baselines use the scales selected by prior work (between 35 and 45 bits), whereas tensor kernels use 45-bit scales (Table 3).

Fhelipe uses Lattigo's state-of-the-art bootstrapping algorithm [2], which uses variable scales and consumes 742 bits of modulus per bootstrap. Fhelipe also uses multi-digit keyswitching (an important optimization [30, 47, 75]), which consumes 305 bits. This leaves $l_0 \approx 10$ levels of application computation for applications with bootstraps (the exact value depends on the application's scale).

Manual baselines: To perform a controlled comparison, we port manual baselines to a common framework (each baseline uses a different FHE library and bootstrap algorithm). We modify Fhelipe to expose a vector interface, disabling all layout passes and automatic bootstrapping. With our contributions removed, this compiler is essentially a reimplementation of EVA [25] that targets the same backends as Fhelipe.

Table 4. Performance on CraterLake for Fhelipe, Manual, and CHET+ baselines. All systems use EVA waterline rescaling.

	Execution time [ms] for			Speedup over	
	Fhelipe	Manual	CHET+	Manual	CHET+
ResNet-20	235.8	236.1	526.4	1.0×	2.2×
RNN	434.7	452.4	2,223	1.0×	5.1×
LogReg	141.5	1,741	4,592	12.3×	32.5×
LoLa-MNIST	0.3	0.9	90.1	3.2×	322.4×
ML gmean speedup				**2.5×**	**18.5×**
FFT	240.6	—	256.9	—	1.1×
TTM	1.1	—	8,105	—	7,643×
MTTKRP	1.5	—	8,106	—	5,569×
Tensor gmean speedup					**360.4×**

Table 5. Performance on CPU for Fhelipe, and speedups over Manual and CHET+. *t-out* denotes runs that timed out after 5h.

Execution time [s] for		Speedup over	
	Fhelipe	Manual	CHET+
ResNet-20	1,555	1.0×	7.3×
RNN	4,502	1.1×	*t-out*
LogReg	1,535	6.2×	*t-out*
LoLa-MNIST	19	2.1×	86.4×
ML gmean speedup		**1.9×**	**25.1×**
FFT	1,270	—	1.0×
TTM	18	—	*t-out*
MTTKRP	25	—	*t-out*

When the original baseline implementations use different bootstrapping algorithms that leave a different number of usable levels, we have to place bootstraps anew. We insert bootstrap manually at natural chokepoints, following the intuitions in the original papers. All of the manual reimplementations have better performance than reported in the original papers.

CHET+: CHET, a state-of-the-art tensor FHE compiler, cannot compile most of our benchmarks. First, CHET provides only a few coarse-grained kernels (e.g., matrix-vector multiply), and our benchmarks have operations that cannot be expressed with this limited interface (e.g., LogReg's gradient descent). Second, CHET does not support bootstrapping.

To compare Fhelipe with CHET's approach, we extend CHET to CHET+. CHET+ includes manual implementations of the additional kernels necessary for our applications, and automatic lazy bootstrapping at kernel boundaries (i.e., bootstrapping tensors between CHET's kernels right before they run out of noise budget).

We implement CHET+ by modifying Fhelipe: we group Fhelipe's fine-grained operations (e.g., `replicate`, *, `reduce`) into CHET kernels (e.g., matrix-vector multiply), and convert tensors to CHET's row-major layout after each kernel. We also disable optimizations that are unique to Fhelipe: repacking, replicating data inside of ciphertexts, and decomposing permutations into stages. This implements CHET's approach to layouts while allowing a controlled comparison. Like all other systems, CHET+ uses EVA waterline rescaling, so it supersedes the CHET+EVA combination in [25].

7.3 Platforms

We evaluate Fhelipe on CraterLake [75], a state-of-the-art FHE accelerator, and on a CPU. For the CPU results, we use the Lattigo state-of-the-art FHE library [2], and run experiments on a single core of a 3.5 GHz AMD Zen2 Threadripper PRO 3975WX CPU (Lattigo is single-threaded). For the CraterLake results, we target the CraterLake configuration in [75]. CraterLake uses double-prime rescaling, using two moduli per level, to support scales larger than its 28-bit wide datapath [45, 75]. We use CraterLake's backend compiler and simulator, which the authors shared with us.

8 EVALUATION

8.1 Fhelipe Achieves High Performance

Table 4 compares the performance of Fhelipe, state-of-the-art manual baselines, and CHET+ (Sec. 7) on CraterLake. On the four machine learning applications, Fhelipe outperforms the manual baselines by gmean 2.5× and CHET+ by 18.5×; on the three tensor kernels, Fhelipe outperforms CHET+ by up to 7,600×. Further, Fhelipe matches or outperforms the manual baselines *across all benchmarks.*

To help us analyze the performance of these different implementations, Fig. 11 shows the percentage of execution that each application and system spends on bootstrap and non-bootstrap computation. On the large applications (ResNet-20, LogReg, RNN), the well-optimized implementations (Fhelipe and manual) are dominated by bootstrapping, taking up 88%–99% of total runtime.

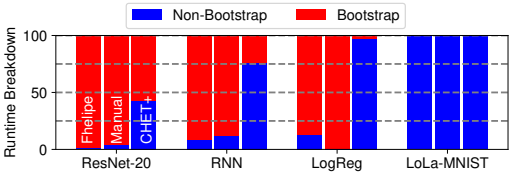

Fig. 11. Breakdown of execution time between bootstrap and non-bootstrap computation across benchmarks.

Although CHET+ spends similar time on bootstrapping as Fhelipe (1×–1.3× across all applications), CHET+ *performs much more non-bootstrap compute* due to poorly utilizing ciphertext slots and frequently permuting ciphertext elements to keep tensors in its restrictive row-major layouts. On applications without bootstrapping (LoLa-MNIST, TTM, MTTKRP) these layout inefficiencies translate directly into CHET+'s large end-to-end slowdowns.

ResNet-20: Fhelipe matches the performance of Lee et al.'s heavily-optimized manual ResNet-20 [52]. This implementation relies on complex layouts to replicate data within ciphertexts and fill gaps after strided convolutions (as in Fig. 6); an earlier version from the same authors [53] lacked these optimizations and performed 10× worse. Yet, these optimizations are naturally captured in Fhelipe's layout representation, and Fhelipe performs them automatically.

CHET+ incurs 58× more non-bootstrap compute because *(1)* it does not replicate data within ciphertexts, and *(2)* it needs to reshuffle data after each strided convolution. Further, CHET+'s naïve bootstrapping places 1.3× more bootstraps.

RNN: RNN performs a sequence of matrix-vector multiplies, similar to the example in Sec. 2.2. Both Fhelipe and manual achieve good performance by alternating between row-major and column-major layouts; CHET+ is 5.1× slower because it uses only row-major layouts.

LogReg: Fhelipe outperforms manual LogReg by 12.3× mainly due to performing 14× fewer bootstraps. First, thanks to compaction, Fhelipe needs to bootstrap only 1 ciphertext per tensor instead of 7. As we saw in Sec. 5.6, these compactions are cheap and add no overheads. Second, Fhelipe bootstraps only 1 tensor per iteration instead of 2 due to bootstrapping at depth boundaries that avoid shortcut bootstraps (Sec. 6.3).

CHET+ also removes gaps eagerly, and so performs only 6% more bootstraps than Fhelipe. However, CHET+ is 32.5× slower overall because its limited row-major layouts cause poor utilization, as little as $1/256$ of ciphertext slots; Fhelipe packs densely, as we saw in Fig. 8.

LoLa-MNIST: As LoLa-MNIST requires no bootstrapping, it stresses the efficiency of layouts more heavily than the large applications. Fhelipe outperforms manual by 3.2× due to manual missing opportunities for data packing and replication, and CHET+ by 322× due to CHET+ incurring an expensive layout conversion after the first strided convolution.

Note LoLa-MNIST uses only non-power-of-2 tensor dimensions. This shows that the padding overheads incurred by Fhelipe's bit-permutation layout (Sec. 5.1) are far outweighed by the efficiency gains from the additional data packing opportunities that this representation provides.

Tensor kernels: As the tensor kernels have no prior manual implementations, we compare Fhelipe only to CHET+. FFT operates on one-dimensional vectors of fixed size and its dataflow graph is a simple sequence of fixed-width stages. This stresses neither layouts, nor bootstrap placement, so CHET+ is only 1.1× slower than Fhelipe.

TTM and MTTKRP stress layouts the most, as they replicate and sum across multiple dimensions, and do not bootstrap (like LoLa-MNIST). CHET+ falters because it does not replicate data within ciphertexts, using as little as $1/512$ of ciphertext slots, and because it extensively shuffles data to keep tensors in row-major layout. As a result, CHET+ is 7,600× slower on TTM and 5,600× on MTTKRP.

Table 6. Fhelipe speedups over alternative bootstrap placement algorithms (Lazy and FHE-Booster); and over a version of Fhelipe that uses only row-major layouts.

Benchmark	vs. Lazy	vs. FHE-Booster	vs. Row-Major
ResNet-20	9.0×	1.4×	2.1×
RNN	1.1×	1.8×	5.8×
LogReg	5.9×	1.4×	22.5×
Lola-MNIST	—	—	32.1×
FFT	2.5×	1.5×	1.0×
TTM	—	—	706.9×
MTTKRP	—	—	520.3×
gmean	**3.5×**	**1.5×**	**22.8×**

Table 7. Prediction accuracy (with 95% confidence intervals) and number of error-free mantissa bits compared to unencrypted double-precision computation across all slots of the output (higher is better) for Fhelipe and manual baselines.

	Accuracy		Error-free mantissa bits	
Benchmark	Fhelipe	Manual	Fhelipe	Manual
ResNet-20	91%±1%	91%	13	13
RNN	78%±1%	78%	16	16
LogReg	97%±1%	97%	9	9

CPU performance: Table 5 compares Fhelipe to the manual and CHET+ baselines when running on a CPU. Again, Fhelipe matches or outperforms manual implementations (ML gmean 1.9×). We set a timeout of 5 hours; since CHET+ produces inefficient code, several CHET+ benchmarks exceed this timeout (note timeouts match the longest CHET+ runtimes in Table 4). Fhelipe outperforms CHET+ by gmean 25.1× on the ML benchmarks that complete, ResNet-20 and LoLa-MNIST.

While the general trends on CPU match CraterLake, speedups are somewhat different. This happens because CraterLake uses highly optimized functional units that change the relative costs of FHE operations. Specifically, higher-depth operations are cheaper in CraterLake, and some rotations are as well, due to the way Lattigo implements automorphisms [35, Sec.4.3]. These fully account for the differences in speedups, like CHET+'s higher slowdown on ResNet-20 (7.3× on CPU vs. 2.2× on CraterLake): its 58× higher non-bootstrap compute has a higher effect on the CPU.

8.2 Analysis of Fhelipe's Contributions and Comparison With Alternatives

Fhelipe's contributions are automatic data packing and bootstrap placement. Table 6 shows that both are crucial by comparing them to alternative techniques.

Bootstrap placement: Table 6 shows the speedup of Fhelipe with our DP-based bootstrap placement algorithm over Fhelipe using two alternative algorithms: *Lazy* is lazy bootstrapping, i.e., bootstrapping when ciphertexts run out of noise budget; and *FHE-Booster* is FHE-Booster's heuristic-based algorithm [82, 83]. Table 6 shows speedups only for benchmarks with bootstrapping.

Lazy Boot incurs gmean 3.5× slowdown. ResNet-20, LogReg, and FFT are 2.5×–9.0× slower due to the issues shown in Fig. 10c: Lazy Boot bootstraps in the middle of kernels where computation is wide (e.g., convolutions in ResNet-20), and doesn't account for shortcuts. Lazy Boot works well only on RNN, because RNN's simple structure causes Lazy Boot to bootstrap at regular intervals that, at 10 usable levels, happen to coincide with RNN's natural chokepoints.

Note that Lazy Boot differs from CHET+'s lazy bootstrapping algorithm in that Lazy Boot considers bootstrapping after every Fhelipe operation, whereas CHET+ considers bootstrapping only at the boundaries of CHET kernels. This fine-grained approach is necessary to support Fhelipe's general tensor interface, but creates more opportunities for mistakes. This is why Lazy Boot has higher bootstrapping overheads than CHET+.

FHE-Booster incurs gmean 1.5× slowdown. While better than Lazy overall, it is consistently worse than our DP-based algorithm, and shows pathological behavior on RNN, where it places many more bootstraps than Lazy. This happens because FHE-Booster places bootstraps greedily, trying to cover the maximum amount of paths per bootstrap. This approach fails to find RNN's regular chokepoints. Moreover, FHE-Booster relies on path enumeration, which has exponential runtime and fails to complete in some cases [82]; our DP-based algorithm does not have this problem.

Layouts: To study the importance of using flexible layouts independently of CHET+, Table 6 compares with *Row-Major*, a version of Fhelipe that restricts all tensors to row-major layouts. Row-Major differs from CHET+ in that it keeps all other Fhelipe features (bootstrap placement, decomposed permutations, replicating in ciphertexts) and in that it converts to row-major after every Fhelipe operation, whereas CHET+ converts only after each of its coarse-grained kernels.

Row-Major incurs gmean 22.8× slowdown. This is due to layout conversions, e.g., to remove gaps after each `sum` and `stride`. The slowdowns of Row-Major and CHET+ are highly correlated across applications. However, Row-Major outperforms CHET+ on TTM and MTTKRP due to using Fhelipe's efficient algorithm for decomposing layout conversions into stages (Sec. 5.5).

We've so far focused our layout comparisons to CHET, but HECO and HeLayers also select layouts automatically. HECO [79] uses only column-major layouts (and lacks other Fhelipe features, like our decomposed layout conversions), so it would suffer large overheads on these benchmarks.

HeLayers [6] supports a wider range of layouts than CHET or HECO. Since implementing its approach within Fhelipe would be hard, we compare using a single representative benchmark, ResNet-20. We manually apply HeLayers' layouts to ResNet-20, following the exact implementation of convolutions and striding in [6] and selecting the layout that maximizes performance. We found its performance to be 8.3× worse than Fhelipe's (on CraterLake).

This slowdown is due to gaps in HeLayers' layouts leading to 2×–16× more bootstraps per tensor. HeLayers' layouts can be viewed as a subset of Fhelipe's, where each dimension has a *tiling* specifying its slot-selecting bits. However, HeLayers *forces the same tiling to be used through the entire computation*. So gaps introduced by, for example, summing across input channels (Sec. 5.6) cannot be immediately filled, as the vacated slot-selecting bits are coupled to the summed dimension.

Fhelipe uses a fundamentally different approach from HeLayers: rather than fixing one layout and using profiling to find the best single layout, Fhelipe assigns different layouts freely throughout the computation, and implements each tensor operation for its specific layout. This lets Fhelipe simultaneously *(1)* keep data packed throughout by using compaction, and *(2)* avoid expensive layout conversions that add work and depth. In ResNet-20, the first effect is crucial, as keeping data packed drastically reduces bootstrapping. This highlights the advantage of our analytical approach.

8.3 Fhelipe Preserves Accuracy

The unencrypted versions of ResNet-20, RNN, and LogReg achieve accuracies of 91%, 78%, and 97%, which are typical for the datasets they use. Table 7 shows that the Fhelipe and manual versions match the accuracy of their unencrypted counterparts. We run enough samples to achieve 95-th percentile confidence intervals below ±1%[38]. This result is expected, since Fhelipe uses the same scales as the manual versions. Nonetheless, Fhelipe versions have a different computation graph, and this shows that our transformations are correct and do not impact accuracy.

Table 7 also shows the number of error-free mantissa bits measured against the unencrypted computation with 64-bit floating-point values. This conveys the maximum absolute error of the output. (e.g., 13 error-free mantissa bits means absolute error $< 2^{-13}$ across all output slots). Fhelipe matches the manual versions on this finer-grained metric as well.

9 CONCLUSION

FHE is enticing but very hard to program. We have presented Fhelipe, a novel FHE compiler that is the first to address FHE's key remaining programmability challenge, automating data layouts to use FHE's huge vector ciphertexts well. Fhelipe also automatically manages noise end-to-end by placing bootstraps efficiently. We show that these contributions enable programs written in a simple tensor language to match or outperform hand-optimized FHE circuits, widely outperform prior compiler techniques, and dramatically simplify programming.

ACKNOWLEDGMENTS

We thank Axel Feldmann, Courtney Golden, Fares Elsabbagh, Hyun Ryong (Ryan) Lee, Joel Emer, Nithya Attaluri, Quan Nguyen, Shabnam Sheikha, Victor Ying, Xingran (Maggie) Du, Yifan Yang, and our anonymous reviewers for their helpful feedback on this paper. This research was funded in part by a Wistron research grant, and by NSF grants 2115587 and 1955270. Aleksandar Krastev was supported in part by an MIT EECS graduate fellowship.

REFERENCES

[1] 2020. HEAAN software library. https://github.com/snucrypto/HEAAN.
[2] 2020. Lattigo. https://github.com/ldsec/lattigo.
[3] 2020. Microsoft SEAL HE library. https://github.com/microsoft/SEAL.
[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for Large-Scale machine learning. In *OSDI-12*.
[5] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *HPCA-29*.
[6] E Aharoni, A Adir, M Baruch, N Drucker, G Ezov, A Farkash, L Greenberg, R Masalha, G Moshkowich, D Murik, et al. 2023. HElayers: A tile tensors framework for large neural networks on encrypted data. PoPETs (2023). *23rd Privacy Enhancing Technologies Symposium*.
[7] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. 2022. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 53–63.
[8] 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)*.
[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*.
[10] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*.
[11] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*.
[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 *ICML*.
[14] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: a compilation chain for privacy preserving applications. In *3rd International Workshop on Security in Cloud Computing*.
[15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2020. Cingulata. https://github.com/CEA-LIST/Cingulata.
[16] Huili Chen, Rosario Cammarota, Felipe Valencia, Francesco Regazzoni, and Farinaz Koushanfar. 2020. AHEC: End-to-end Compiler Framework for Privacy-preserving Machine Learning Acceleration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218508
[17] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: a vectorizer generator for SIMD and beyond. In *ASPLOS-XXVI*.
[18] 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)*.
[19] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. 2024. DaCapo: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption. In *33rd USENIX Security Symposium (USENIX Security 24)*.

[20] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. 2018. E3: A Framework for Compiling C++ Programs with Encrypted Operands. *Cryptology ePrint Archive* (2018).
[21] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. In *ASIACRYPT*.
[22] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2020. CONCRETE: Concrete operates on ciphertexts rapidly by extending TfhE. In *WAHC 2020-8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*.
[23] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption. In *PLDI*.
[24] Eric Crockett, Chris Peikert, and Chad Sharp. 2018. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1020–1037.
[25] 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 *PLDI*.
[26] 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 *PLDI*.
[27] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *EUROCRYPT*.
[28] Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science* 14, 2 (1990).
[29] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* (2012).
[30] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Annual Cryptology Conference*. Springer, 850–867.
[31] 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 *International conference on machine learning*.
[32] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce J. Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson. 2021. A general purpose transpiler for fully homomorphic encryption. *arXiv preprint arXiv:2106.07893* (2021).
[33] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. 2022. Sok: New insights into fully homomorphic encryption libraries via standardized benchmarks. *Cryptology ePrint Archive* (2022).
[34] Shai Halevi and Victor Shoup. 2014. Algorithms in helib. In *Annual Cryptology Conference*.
[35] Shai Halevi and Victor Shoup. 2018. Faster homomorphic linear transformations in HElib. In *Annual International Cryptology Conference*.
[36] Kyoohyung Han et al. 2019. HELR: Homomorphic Logistic Regression on Encrypted Data. https://github.com/KyoohyungHan/HELR.
[37] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2018. Efficient Logistic Regression on Large Encrypted Data. Cryptology ePrint Archive, Report 2018/662.
[38] James A Hanley and Abby Lippman-Hand. 1983. If nothing goes wrong, is everything all right?: interpreting zero numerators. *Jama* 249, 13 (1983), 1743–1745.
[39] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020).
[40] Koby Hayashi, Grey Ballard, Yujie Jiang, and Michael J Tobia. 2018. Shared-memory parallelization of MTTKRP for dense tensors. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
[41] Nandan Kumar Jha, Zahra Ghodsi, Siddharth Garg, and Brandon Reagen. 2021. DeepReDuce: Relu reduction for fast private inference. In *International Conference on Machine Learning*.
[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* (2021), 114–148.
[43] 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).

[44] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security 18*.
[45] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *ISCA-50*.
[46] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *MICRO-55*.
[47] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *ISCA-49*.
[48] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report. University of Toronto.
[49] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998).
[50] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing homomorphic evaluation circuits by program synthesis and term rewriting. In *PLDI*.
[51] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2021. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. Cryptology ePrint Archive, Paper 2021/1688.
[52] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. In *International Conference on Machine Learning (ICML)*.
[53] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2021. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *arXiv preprint arXiv:2106.07229* (2021).
[54] Yongwoo Lee, Seonyoung Cheon, Dongkwan Kim, Dongyoon Lee, and Hanjun Kim. 2023. ELASM: Error-Latency-Aware Scale Management for Fully Homomorphic Encryption. In *32nd USENIX Security Symposium (USENIX Security 23)*.
[55] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. 2022. HECATE: Performance-Aware Scale Optimization for Homomorphic Encryption Compiler. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
[56] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
[57] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*.
[58] Raghav Malik, Kabir Sheth, and Milind Kulkarni. 2023. Coyote: A Compiler for Vectorizing Encrypted Arithmetic Circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 118–133.
[59] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (1987), 122–126.
[60] Charith Mendis and Saman Amarasinghe. 2018. goSLP: globally optimized superword level parallelism framework. In *OOPSLA*.
[61] Microsoft. 2019. CryptoNets. https://github.com/microsoft/CryptoNets.
[62] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference system for neural networks. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*. 27–30.
[63] 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)*.
[64] Chris Okasaki. 1995. Purely functional random-access lists. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*. 86–95.
[65] 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 2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*.
[66] Jaiyoung Park, Donghwan Kim, Jongmin Kim, Sangpyo Kim, Wonkyung Jung, Jung Hee Cheon, and Jung Ho Ahn. 2023. Toward Practical Privacy-Preserving Convolutional Neural Networks Exploiting Fully Homomorphic Encryption. *arXiv preprint arXiv:2310.16530* (2023).

[67] Jaiyoung Park, Michael Jaemin Kim, Wonkyung Jung, and Jung Ho Ahn. 2022. AESPA: Accuracy preserving low-degree polynomial activation for fast private inference. *arXiv preprint arXiv:2201.06699* (2022).
[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32.
[69] Chris Peikert. 2016. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science* 10, 4 (2016).
[70] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle inventor: data movement synthesis for GPU kernels. In *ASPLOS-XXIV*.
[71] Robert Podschwadt and Daniel Takabi. 2020. Classification of Encrypted Word Embeddings using Recurrent Neural Networks. In *PrivateNLP@ WSDM*.
[72] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*.
[73] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).
[74] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54*.
[75] 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 *ISCA-49*.
[76] SNU-CCL. 2022. FHE-MP-CNN. https://github.com/snu-ccl/FHE-MP-CNN.
[77] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. 2019. SEALion: A Framework for Neural Network Inference on Encrypted Data. *arXiv preprint arXiv:1904.12840* (2019).
[78] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *ASPLOS-XXVI*.
[79] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. {HECO}: Fully Homomorphic Encryption Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4715–4732.
[80] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. 2021. SoK: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1092–1108.
[81] Alexander Viand and Hossein Shafagh. 2018. Marble: Making fully homomorphic encryption accessible to all. In *Proceedings of the 6th workshop on encrypted computing & applied homomorphic cryptography*. 49–60.
[82] Tommy White. 2023. *Scheduling General Purpose Encrypted Computation on Multicore Platform.* Master's thesis. University of Delaware.
[83] Tommy White, Charles Gouert, Chengmo Yang, and Nektarios Georgios Tsoutsos. 2023. FHE-Booster: Accelerating Fully Homomorphic Execution with Fine-tuned Bootstrapping Scheduling. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*.
[84] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *ASPLOS-XXV*.
[85] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical Homomorphic Encryption Accelerator. In *HPCA-29*.
[86] Zama. 2022. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. https://github.com/zama-ai/tfhe-rs.
[87] Zama. 2023. Concrete FHE compiler. https://github.com/zama-ai/concrete.
[88] Zama. 2023. TFHE-rs Benchmarks. https://docs.zama.ai/tfhe-rs/getting-started/benchmarks. *Internet Archive*, https://web.archive.org/web/20230601080350/https://docs.zama.ai/tfhe-rs/getting-started/benchmarks.
[89] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: Enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *ASPLOS-XXVII*.