PyTFHE: An End-to-End Compilation and Execution Framework for Fully Homomorphic Encryption Applications

Jiaao Ma
Dept. of Computer Science
Duke University
Durham, North Carolina, USA
jiaao.ma@duke.edu

Ceyu Xu
Dept. of Computer Science
Duke University
Durham, North Carolina, USA
ceyu.xu@duke.edu

Lisa Wu Wills
Dept. of Computer Science
Duke University
Durham, North Carolina, USA
lisa@cs.duke.edu

***Abstract*—Fully Homomorphic Encryption (FHE) is a powerful cryptographic scheme that enables computation on encrypted data, which allows clients to offload computation to an untrusted third party without compromising data privacy. However, FHE has not yet been widely adopted due to its enormous computational overhead. Further, cryptographic software development requires specialized expertise and presents a significant challenge when applying FHE to a broad range of applications.**

We present PyTFHE, a framework that tackles these difficulties by enabling highly productive FHE application development and orders of magnitude more efficient FHE application execution. PyTFHE is built on top of the TFHE (Fast Fully Homomorphic Encryption over the Torus) scheme, which is an FHE scheme that supports gate-level evaluation and arbitrary depth of boolean circuits. PyTFHE is designed in a TFHE-specific approach, allowing state-of-the-art optimizations for TFHE applications. Specifically, PyTFHE features *ChiselTorch*, the first compiler that allows easy generations of privacy-preserving deep neural network models with PyTorch-compatible APIs. PyTFHE is also the first FHE framework that employs a powerful backend enabling efficient execution of TFHE applications on distributed CPU systems and high-performance GPUs. We demonstrate the effectiveness of PyTFHE by benchmarking the framework using VIP-Bench and implementing privacy-preserving deep neural networks and evaluating their performance on various systems. We compare the performance of our generated TFHE program execution with three existing frameworks, Google Transpiler, Cingulata, and E3. We show that PyTFHE achieves one to two orders of magnitude performance advantage.

***Index Terms*—Fully homomorphic encryption, Compiler toolchain, Privacy-preserving computation, TFHE**

I. Introduction

As cloud computing becomes more popular, more sensitive data is being stored in the cloud, making it vulnerable to misuse. Encryption is a common method for protecting sensitive data and maintaining privacy. However, conventional encryption schemes require data to be decrypted before it can be used in computations, which can significantly limit the level of data security, allowing third-party entities, such as cloud providers, to collect sensitive data without client authorization. In contrast, fully homomorphic encryption (FHE) is a type of encryption scheme that allows computation to be performed directly on ciphertexts, without the need to decrypt them first. Figure 1 shows how the data privacy is perserved when using an FHE scheme in a cloud computing scenario. In this scheme, the ciphertexts can only be decrypted by the client after all the computations are performed in the cloud. Throughout the process, the cloud provider has no access to the plaintexts, protecting the privacy of the data.

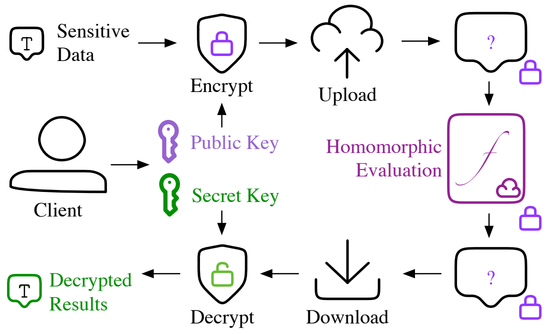

Fig. 1. Privacy-preserving Computation in a Cloud Computing Scenario

Although FHE is a viable method for offloading computations to the cloud while perserving data privacy, the computing industry has not yet embraced this scheme to a great extent. The performance of FHE is still far from being on par with traditional encryption techniques, significantly impacting the wide adoption of FHE. Moreover, developing FHE applications is challenging due to the complexity of the underlying FHE schemes and the lack of FHE-targeted toolchains.

In this work, we propose the PyTFHE framework to facilitate privacy-preserving computation offload on the server side. PyTFHE is a highly productive and flexible framework for developing complex FHE applications such as neural network models. The toolchain also generates high-performance FHE application binaries and in turn improves the runtime of

This work was supported in part by an National Science Foundation CAREER award CCF-2045973, and in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

FHE-enabled application execution by one to two orders of magnitude compared to existing frameworks.

This work makes the following contributions:

- An end-to-end toolchain to develop, compile, and execute TFHE (the specific FHE scheme we explore in this work) applications called PyTFHE. The framework takes in PyTorch models and compiles them into optimized TFHE binaries for highly efficient execution.
- A neural network to Verilog compiler called *Chisel-Torch* that guarantees correctness by utilizing pre-built Chisel [1] modules, is highly productive by providing a PyTorch-compatible API for developing TFHE-enabled neural network models, and is highly performant by employing model layer Verilog fusion and providing parameterizable data type selection.
- A highly performant distributed CPU backend for PyTFHE that supports the evaluation of TFHE applications on multiple server nodes and a state-of-the-art GPU backend for PyTFHE that performs 62$\times$ better than the current state-of-the-art TFHE GPU library and executor cuFHE [2].
- An in-depth comparison against state-of-the-art TFHE compilers/frameworks including High-Level-Synthesis-based Google Transpiler [3] and Domain-Specific-Language-based Cingulata [4] and E3 [5].

II. Background

A. Fully Homomorphic Encryption Schemes

An FHE scheme defines a set of functions that manipulate plaintext and encrypted data. The *key generation* function produces a pair of public key $\mathtt{pk}$ and secret key $\mathtt{sk}$ denoted as $\mathtt{(pk,\ sk)} \leftarrow \mathtt{KeyGen}$. The *encryption* function takes plaintext $\mathtt{m}$ as input and generates ciphertext $\mathtt{c}$ using the public key and a security parameter λ denoted as $\mathtt{c} \leftarrow \mathtt{Encrypt}_{pk,\lambda}\mathtt{(m)}$. The *decryption* function takes a ciphertext as input and generates plaintext using the secret key denoted as $\mathtt{m} \leftarrow \mathtt{Decrypt}_{sk}\mathtt{(c)}$. The *evaluation* function produces a ciphertext of the evaluation result on the input ciphertexts given a function to be evaluated and a public key denoted as $\mathtt{c'} \leftarrow \mathtt{Evaluate}_{f,pk}\mathtt{(c_1,c_2,...)}$. A correctly implemented FHE scheme guarantees that, given a function $\mathtt{f}$ and ciphertexts $\mathtt{c_1,c_2,...}$ that encrypt plaintext messages $\mathtt{m_1,m_2,...}$ under keypair $\mathtt{(pk,\ sk)}$, the function $\mathtt{Decrypt}_{sk}\mathtt{(Evaluate}_{f,pk}\mathtt{(c_1,c_2,...))}$ yields the same results as the plaintext operation $\mathtt{f(m_1,m_2,...)}$ without revealing any information about $\mathtt{m_1,m_2...}$ during homomorphic evaluation.

The cryptographic *hardness* of a system describes the resistance of the system against attacks. *Learning with Error (LWE)* is a type of mathematical problem that serves as a basis for many FHE schemes to provide the required hardness guarantees. In the LWE problem, ”noisy” samples, also called *LWE samples*, are generated based on a secret key and deliberately added with random noise. Briefly speaking, the LWE problem involves creating these noisy samples and using them to recover the original secret key. One way for LWE to provide the required hardness is to make it computationally infeasible to recover the original key from the noisy samples. This implies that even having access to the encrypted data, an attacker can't decode them without the original secret key. However, while providing safety guarantees, the added noise would grow when homomorphic operations are performed and eventually lead to undesired decryption results.

To address this issue, many FHE schemes [6]–[8] provide a *bootstrapping* operation reducing the accumulated noise to allow further homomorphic operations. However, *bootstrapping* is a computationally expensive operation and often dominates the execution time of FHE applications.

B. The TFHE Scheme and the TFHE Library

The *TFHE (Fast Fully Homomorphic Encryption Over the Torus)* scheme (also known as the *CGGI* scheme) [7] is an FHE scheme that supports homomorphic binary gate evaluation. Particularly, it provides fast programmable bootstrapping which reduces the noise of a ciphertext while simultaneously performing an arbitrary lookup-table operation on the input. The TFHE scheme is implemented by several libraries [2], [9], [10]. The TFHE library [10] is a C++ implementation intended for CPUs and targets computation using binary gates (e.g., NAND). The basic building block in the TFHE library is a *bootstrapped-Gate*, which performs bootstrapping on a ciphertext after each gate evaluation to mitigate the noise accumulation problem.

C. Word-wise FHE Schemes

FHE schemes are primarily distinguished by the types of plaintext data and operations they support. Besides bit-wise FHE schemes such as TFHE, another type of FHE scheme is the word-wise FHE scheme, which supports homomorphic operations on a vector. The *Cheon, Kim, Kim and Son (CKKS)* scheme [6] is one of the most popular word-wise FHE schemes and has been studied in many prior works. CKKS supports encoding a vector of fixed-point numbers into a ciphertext and performing homomorphic element-wise addition, element-wise multiplication, and cyclic rotation on a granularity of an entire ciphertext. Being a Ring-variant LWE (RLWE)-based scheme, CKKS also requires adding noise into the ciphertexts. One highlight of CKKS is that the noise was treated as part of the error during homomorphic operations, which makes it efficient in approximated computations. However, like other word-wise FHE schemes, CKKS has no direct access to individual elements in a vector and operations must be performed on a ciphertext as a whole, which greatly limits its flexibility and makes operations such as look-up tables and multi-headed attention that require accessing individual elements in a vector more difficult compared to bit-wise schemes. While word-wise FHE schemes efficiently support linear operations over wide vectors of ciphertexts, they do not support non-linear operations, such as `ReLU` or `argmax` which are ubiquitous in AI and data analytics. Although bitwise schemes are less efficient per bit, they are much more flexible because boolean operations can be used to express arbitrary functions. Further, for real-world deep neural network applications, CKKS requires a large number of rotation keys to support cyclic rotations, which can take up to tens of gigabytes whereas the public key size of TFHE is a few megabytes. The rotation keys have to be transferred to the server for evaluation, which can be a bottleneck for the practicality of CKKS in real-world applications.

For this work, we choose to focus on providing a highly productive and performant development and execution compiler framework for neural network models using the TFHE scheme because of its great balance between flexibility and efficiency.

D. Threat Model and Security

We assume a semi-honest threat model where the adversary who performs the computation has access to the ciphertexts and is curious about the plaintexts but does not have the secret key. With that being said, the adversary still follows the FHE scheme and does not deviate from it. This model also applies to the case where a trusted server is compromised by malicious third parties that have access to the server's internal states.

λ denotes the desired security level of an encryption scheme. In this work, we choose λ = 128 bits, which is considered to be standard and secure for FHE applications. For the rest of the TFHE scheme parameters, we use the default parameter set as described in Section VIII of the TFHE paper [7].

III. Related Works

A. Privare-preserving Computing Solutions

To alleviate the privacy leakage concerns, many works in the past have been proposed based on different cryptographic primitives.

Hybrid HE-MPC (multi-party computation)-based frameworks such as Gazzelle [11] and Cheetah [12] combine two-party computation (2PC) and FHE to deliver fast deep learning inference. While these works claim to have shorter calculation times than pure FHE solutions, their performance is frequently constrained by communication/network bottlenecks between the two parties. Furthermore, they require clients to be engaged whenever the noise of ciphertexts reaches a critical point, which necessitates more computational resources on the client's part than pure-FHE-based solutions.

Pure FHE-based solutions have also been developed in recent years. FHE libraries such as Microsoft SEAL [13], HEAAN [6], OpenFHE [9], and Lattigo [14] target implementing the primitives defined by the schemes and are not end-to-end solutions for FHE application development.

To improve the programmability and provide higher abstractions of FHE libraries, FHE compilers such as EVA [15], [16] and SEALion [17] have been proposed to compile high-level programs to FHE primitives for the CKKS scheme. While these compilers simplify the development of RLWE-based FHE applications, their supported operations are also limited by the underlying FHE schemes which usually only support homomorphic polynomial operations. Complicated linear operations such as convolution require additional manual engineering to utilize the single-instruction-multiple-data (SIMD) parallelism of the underlying FHE schemes. For non-linear operations such as ReLU, existing works use either polynomial approximations which usually require excessive multiplication depth to achieve required accuracy [18], removing the non-linear operations from the model [19], or replacing the non-linear functions with linear functions such as replacing ReLU with x^2, which may hurt the accuracy of the model [20].

Besides compilers that support homomorphic polynomial operations, compilers for FHE schemes that support homomorphic evaluations on a gate-level [3]–[5], [21] have also been proposed to support more complex operations. These compilers are the most closely related to PyTFHE and we describe them in greater detail in the next subsection.

Hardware accelerators have also been proposed to bridge the performance gap between FHE and plaintext evaluation, most of which target accelerating CKKS-scheme-based programs with deep multiplicative depth. F1 [22] is the first application-specific integrated circuit accelerator (ASIC) targeting mainly small-scale FHE applications. To better support CKKS programs with deeper multiplicative depth, CraterLake [23], BTS [24], and ARK [25] have been proposed to provide more efficient support for bootstrapping operations that are necessary for deep neural network inference. MATCHA [26] is the first ASIC accelerator targeting the TFHE scheme.

B. Compilers and Frameworks that Aid the Development and Generation of TFHE Programs

Frameworks that are designed to program and execute homomorphic encryption applications over the TFHE scheme and other schemes that support boolean gates are the most closely related to our proposed PyTFHE framework. We describe three frameworks in detail (Google's Transpiler [3], Cingulata [4], and E3 [5]) and provide evaluations against these frameworks in Section V.

Google's TFHE Transpiler [3] is a framework that bridges the gap between high-level abstractions and low-level gate operations by taking a C program and producing an equivalent TFHE-enabled C program. Transpiler is built on top of XLS [27], Google's open-source HLS (High-Level Synthesis) toolchain that converts C programs into hardware description languages. The hardware description language program is turned into an intermediate representation (IR) of gates (i.e., AND, OR, and NOT gates) and the gates are mapped statically to the API provided by the open-source TFHE library [10] via code generation to generate a TFHE-enabled C program. While Transpiler benefits from the HLS optimizations and the higher abstraction level for ease-of-use provided by XLS, it also has limitations such as being restricted to using C native data types and libraries. XLS makes generating sophisticated applications such as deep neural networks labor-intensive because neural network models need to be constructed from scratch in C. The execution backend of Transpiler recently added an experimental interpreter that parses the IR and exe-

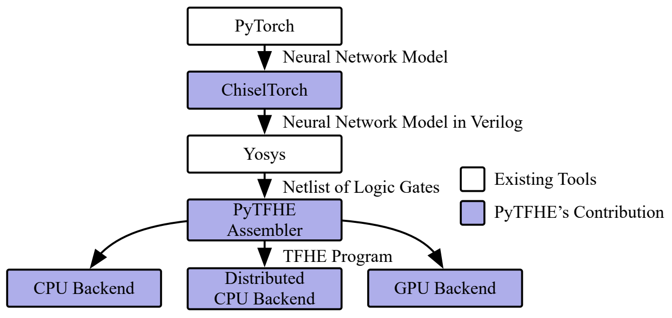

Fig. 2. Compilation and execution flow of PyTFHE.

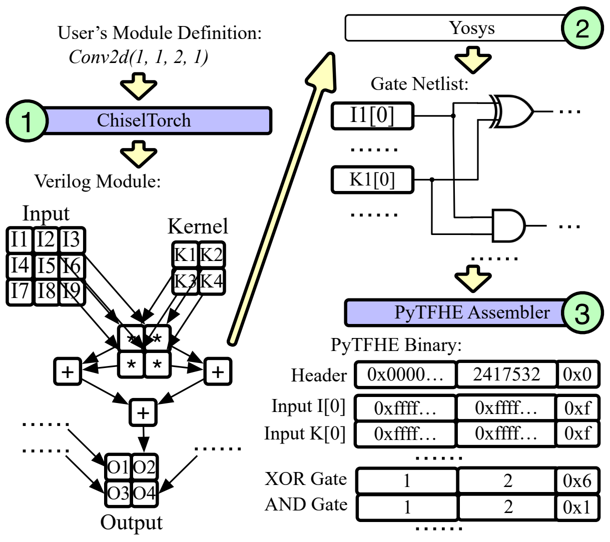

Fig. 3. An example compilation flow of PyTFHE for a Convolution layer.

cutes the gates in multi-threaded mode. While the interpreter provides a significant speedup over the code generator, it still lacks support for more sophisticated parallel execution, which limits its ability to support large-scale applications.

Cingulata [4] is a compiler toolchain that enables the development of FHE applications using a Domain-Specific Language (DSL). Cingulata supports two FHE schemes, the Brakerski-Fan-Vercauteren (BFV) scheme [28] and the TFHE scheme. Cingulata allows users to define encrypted integers by providing a customized class that has overloaded basic arithmetic operators. Despite Cingulata making integer arithmetic easier, real-world applications tend to use more than just basic integer operations. For example, it does not have built-in support for floating-point arithmetic, which limits its ability to implement any applications that utilize data types other than integers. In addition, applying TFHE to more complex applications such as neural network models requires significant programming effort to construct the models using Cingulata's DSL from scratch.

Similar to Cingulata, Encrypt-Everything-Everywhere (E3) [5] also supports the development of FHE applications using a DSL. E3 supports the TFHE scheme by providing a customized secure integer class and a set of arithmetic operations such as addition, multiplication, and comparison. However, besides limited support beyond integer arithmetics, it only supports bits and 8-bit integers as encrypted variables and hardcodes the gates for these types, which greatly limits its flexibility and programmability.

PyTFHE provides high productivity when converting complicated applications to TFHE programs using a PyTorch interface. In addition, PyTFHE is higher performance than these three compilers and frameworks because it employs an interpreter (i.e., runtime linking of the TFHE library for fast execution) as opposed to a code generator, and it provides a distributed CPU backend as well as a GPU backend. While Transpiler provides IR-level optimizations, it does not support more sophisticated backends for distributed execution and is unable to exploit the ample parallelism provided by TFHE programs. Both Cingulata and E3 do not provide any gate-level or boolean optimizations and have support for only single-threaded TFHE program execution.

IV. The PyTFHE Framework

A. PyTFHE Overview

PyTFHE is an end-to-end high-performance framework for compiling and executing TFHE programs. Figure 2 depicts the compilation and execution flow of PyTFHE in its entirety. PyTFHE implements a frontend called *ChiselTorch* that allows users to write neural network models in PyTorch and automatically generates the corresponding Verilog modules via a collection of pre-built Chisel [1] modules. The generated Verilog design is then synthesized into a netlist of gates with the help of an augmented open-source Yosys [29] synthesis suite. The synthesized netlist is converted into a compact PyTFHE binary format using the *PyTFHE Assembler*. PyTFHE also provides multiple backends, including a distributed CPU backend and a GPU backend, to run the compiled TFHE program binary efficiently. This framework provides high productivity by generating TFHE neural network models for inference as well as executing TFHE programs that are written using either supported PyTorch primitives or arbitrary hardware designs in Chisel without human intervention.

For the rest of this section, we use a running example of a simple 2D convolutional layer that takes in one input channel, outputs one convolutional feature, with a square kernel size of two, at a stride of one, `Conv2d(1,1,2,1)`, in a Convolutional Neural Network (shown in Figure 3) to illustrate how PyTFHE turns a model layer declared in PyTorch into a TFHE program ready for execution on a CPU or a GPU platform step-by-step.

B. Converting PyTorch Models into Verilog Designs

Step ① of the compilation flow takes in a model layer or a model definition in PyTorch and generates a Verilog module or modules using *ChiselTorch*. The *ChiselTorch* generator is designed specifically for implementing neural network models in the TFHE scheme. We focus on three desiderata of a

(a)
```
# PyTorch MNIST Model

mnist_model = nn.Sequential (
      nn.Conv2d (1, 1, 3, 1)
      nn.ReLU ( )
      nn.MaxPool2d (3,1)
      nn.Flatten ( )
      nn.Linear (576, 10)
)
```

(b)
```
# ChiselTorch MNIST Model

val mnist_model = new.Sequential (Seq (
          nn.Conv2d (1, 1, 3, 1)
          nn.ReLU ( )
          nn.MaxPool2d (3,1)
          nn.Flatten ( )
          nn.Linear (576, 10)
 ), dtype = Float(8,8)
```

Fig. 4. A MNIST [30] neural network model declared in PyTorch (a) and *ChiselTorch* (b). *ChiselTorch* also allows specific datatype selection for the neural network model. For example, `Float(8,8)` declares a bfloat16 data type with 8 bits for exponent and 8 bits for mantissa.

TABLE I
ChiselTorch Supported Pre-Built Neural Network Primitives

Neural Network Layers	**Primitive Tensor Operations**
Conv1d/Conv2d	matmul, dot
BatchNorm1d/BatchNorm2d	==, !=, >, <, >=, <=
Linear	view, reshape, transpose, pad
ReLU	sum, prod
MaxPool1d/AvgPool1d	argmax, argmin
MaxPool2d/AvgPool2d	+, -, *, /
Flatten	max, min

compiler when we design the PyTFHE framework: correctness, productivity, and performance. To ensure correctness, we pre-build and validate model layers as Chisel modules that are commonly used in a neural network model such as convolution layers (e.g., `Conv1d, Conv2d`), pooling layers (e.g., `MaxPool1d, MaxPool2d`), non-linear activations (e.g., `ReLU`), normalization layers (e.g., `BatchNorm1d, BatchNorm2d`), and fully-connected layers (e.g., `Linear`).

To provide programmer productivity, ChiselTorch is built on top of the Chisel hardware description language framework [1] and provides API-level compatibility to the PyTorch deep learning library [31] as shown in Figure 4. Unlike existing TFHE frameworks [3]–[5], which require the users to write the entire neural network model from scratch, *ChiselTorch* allows the users to easily define a neural network model in a PyTorch-like API. In addition, we exploit Chisel's parameterizability and allow users to specify neural network models' input channel size, output channel size, kernel size, and stride size to name a few, similar to the PyTorch API. With the help of the provided primitive tensor operations such as reshape (`reshape`) and matmul (`matmul`), users may also implement their own neural network layers that are not yet available as pre-built modules in *ChiselTorch*. For example, the user may implement custom attention layers by using the reshape and matmul operations provided in our API. Table I shows a list of available neural network model layers as well as tensor operations supported by *ChiselTorch*.

PyTFHE is designed to maximize the performance of a generated TFHE program. Recall that a TFHE program is a DAG of gates (as described in Section II), and the execution time of a TFHE program measures how long it takes to traverse the DAG of gate evaluations. This means that the performance of a TFHE program is determined by the number of gates evaluated in a given implementation of TFHE primitives

	127 … 66	65 … 4	3 … 0
Header Inst	0	Total # of Gates	0
Input Inst	0x3FFF…	0x3FFF…	0xF
Gate Inst	INPUT 0 Gate Index	INPUT 1 Gate Index	Gate Type
Output Inst	0x3FFF…	OUTPUT Gate Index	0x3

Fig. 5. PyTFHE instruction encoding and binary format for fast TFHE program execution.

given the same amount of time per gate. Like other FHE schemes, TFHE requires all programs to be data-oblivious, meaning that the control flow and memory accesses of the program should not depend on the encrypted variables. This prohibits any conditional branching or recursion. The data-oblivious principle provides the opportunity to achieve the most efficient implementation by translating the computation DAGs into a purely fused combinatorial form. In contrast, programs implemented with sequential logic require additional gates for state retention and management, resulting in a higher number of gates being evaluated and are thus less efficient.

In PyTorch, users have the option of selecting a data type and precision to suit their needs. It is even more crucial to be able to parameterize the data type of the neural network when it comes to generating TFHE programs, since choosing a cheaper data type may result in a reduction in the number of gates by orders of magnitude. In *ChiselTorch*, data types are not limited to conventional byte or word alignment because TFHE programs operate at the gate level. *ChiselTorch* supports integer and fixed-point data types of arbitrary bit widths as well as floating-point data types with arbitrary bits of exponent and mantissa. For example, a `SInt(7)` in *ChiselTorch* declares a signed integer data type, while a `Float(5, 11)` data type declares a floating point number with 5 bits of exponent and 11 bits of mantissa (effectively a half-precision float). The flexibility of *ChiselTorch*'s data type selection allows finer-grained control of quantization tradeoffs (i.e., accuracy vs. performance), which subsequently results in higher TFHE program performance.

C. Compiling Verilog Designs into PyTFHE Binaries

Step ② uses an open-source synthesis tool suite Yosys to obtain a netlist of gates from the combinational logic in Verilog generated by *ChiselTorch*. The final step, Step ③, assembles a PyTFHE program binary from the gate netlist output using the *PyTFHE Assembler* as shown in Figure 3.

We create a specific binary encoding for TFHE instructions that allows representation of a large number of gates (2^{62} to be exact) and creates a sequential indexing scheme to "name" the gates using their assigned indices. This gate indexing scheme allows fast TFHE program DAG traversal and therefore fast TFHE program execution.

There are four types of TFHE instructions: *header*, *input*, *gate*, and *output* instructions; each instruction is 128 bits. The encoding of these instructions are shown in Figure 5. Each input or output gate has an assigned index of 62 bits. This allows the TFHE program to have up to a total number of

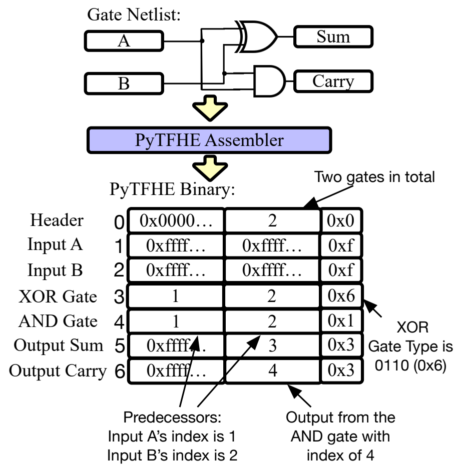
Gate Netlist:
A
B
Sum
Carry
PyTFHE Assembler
PyTFHE Binary:
Two gates in total
Header 0 0x0000… 2 0x0
Input A 1 0xffff… 0xffff… 0xf
Input B 2 0xffff… 0xffff… 0xf
XOR Gate 3 1 2 0x6
AND Gate 4 1 2 0x1
Output Sum 5 0xffff… 3 0x3
Output Carry 6 0xffff… 4 0x3
XOR Gate Type is 0110 (0x6)
Predecessors: Input A's index is 1 Input B's index is 2
Output from the AND gate with index of 4

Fig. 6. An example PyTFHE binary encoding of a half adder.

2^{62} gates. Each gate type is encoded using four bits because PyTFHE supports eleven different gates.

The first instruction in any PyTFHE binary is always a *header* instruction which encodes the total number of gates in the program. To encode the input signals of the TFHE program, we use *input* instructions. Input instructions reserve indices for the inputs. This sequential naming scheme allows fast traversal of the TFHE program which consists of a DAG of gates. All fields of any input instruction are set to all ones except for their assigned indices. To encode gates and their wiring connections, we use *gate* instructions. Each gate has two inputs and they can either be input signals or gates that generated these input signals (i.e., the current gate's predecessors in the DAG). Besides inputs, gate instructions also record gate types. For each output signal, we use an *output* instruction to record the gate that produced this output.

Figure 6 shows how a half adder netlist is encoded as a PyTFHE binary. The first instruction is a header instruction with the number of gates set to two since there are a total of two gates in this netlist. The second and third instructions are input instructions used to reserve the two indices for input signals `A` (index 1) and `B` (index 2). We then encode the two gates, `XOR` (index 3) and `AND` (index 4), in the half adder. The XOR gate instruction encodes its two inputs `A` and `B` with index 1 and 2 respectively. XOR's gate type is encoded as 0110. Outputs `Sum` and `Carry` are encoded with the gates that generated them, index 3 and 4 respectively. The assembled PyTFHE binary is then ready for backend execution.

D. Distributed CPU Backend

Fully Homomorphic Encryption provides the ability to perform computations directly on encrypted data. Though it is highly desirable to perform privacy-preserving computations such as FHE, the computation cost of executing FHE programs is exorbitantly high, making it critical to have a high-performance PyTFHE backend to execute and run TFHE programs efficiently. The PyTFHE backend consists of the open-source TFHE library [10], a scheduler, and an executor. As described in Section III-B, there are several existing TFHE backends [9], [10] that enable the execution of TFHE programs on a single CPU core. However, none of them are able to exploit the abundant parallelism in TFHE programs by executing them in a distributed manner.

Algorithm 1 TFHE Program DAG Traversal Algorithm

```
Require: $T = (V, E)$ being a valid TFHE program directed acyclic graph
  with $k$ cycles.

  for $i = 1, 2, \dots, k$ do
      $I \leftarrow \{v \in V | v$ is input node$\}$                ▷ input set
      $O \leftarrow \{v \in V | v$ is output node$\}$              ▷ output set
      $G \leftarrow V - I - O$                                     ▷ gate set
      $ready = I$
      $finished = I$
      while $V - finished \neq \varnothing$ do
                                                  ▷ until all nodes are ready
          $C \leftarrow \{v \in V | \forall u \rightarrow v, u \in ready\}$
                                  ▷ find all nodes that are ready to compute
          Compute $(C - finished)$
                                  ▷ submit compute job to distributed system
          $ready \leftarrow ready \cup C$
          $finished \leftarrow finished \cup C$
      end while
  end for
```

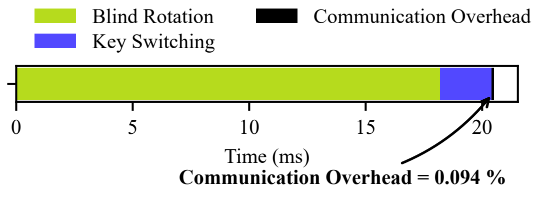

Fig. 7. Profiling of a TFHE gate evaluation on a single core CPU.

In our PyTFHE framework, we use a breadth-first search (BFS)-based algorithm (shown in Algorithm 1) to traverse the DAG of a TFHE program and to exploit the parallelism in the DAG. The algorithm starts traversing from the input nodes where the values are already known. When a node is visited, the traversing algorithm is able to guarantee that the parents of the node are already computed, so that the current node being visited can be computed. After all the nodes have been traversed and computed, the algorithm will return the value of the output nodes.

To implement the PyTFHE backend, we wrap the TFHE library [10] into a python library with pybind11 [32] so that TFHE basic operations such as keypair generation, ciphertext encryption and decryption, and bootstrapped gate computations (e.g., bootstrapped-NAND, bootstrapped-XOR) are exposed to the user. We then use the Ray [33] framework to implement the distributed execution of TFHE programs in PyTFHE. Ray is a distributed execution framework that provides a unified API for distributed execution of Python functions and actors. At the beginning of the execution, the PyTFHE framework will create a Ray cluster with multiple cores and multiple nodes whose numbers are specified by the user. Then, the PyTFHE framework will create a Ray actor on each processor core in the cluster. After that, the public key

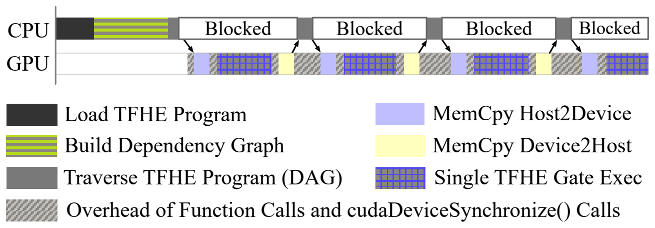

Fig. 8. Execution time breakdown of the GPU backend using cuFHE.

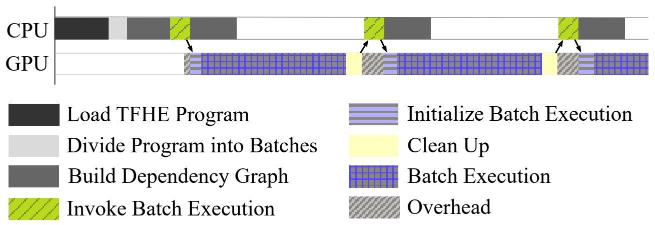

Fig. 9. Execution time breakdown of the GPU backend using PyTFHE.

will be broadcasted to all the cores in the cluster. When all the actors are launched and have received the public key required for computing TFHE gates, PyTFHE will use Algorithm 1 to traverse the DAG of the TFHE program and the computation jobs are submitted to the Ray actors.

In Figure 7, we show the profiling results of the execution of a single TFHE gate. We choose to submit each gate as a separate Ray task to the distributed system because the TFHE programs are mainly bottlenecked by computation rather than communication. When a gate is computed, both the input and output ciphertexts require communication. Since a piece of ciphertext in the TFHE context is only 2.46 KB in size, the communication overhead created by submitting each gate as a separate Ray task contributes to only 0.094% of the total runtime, which is negligible compared to the computation cost of the gate.

The distributed execution of TFHE programs significantly reduces the execution time of TFHE programs. More evaluation results can be found in Section V.

E. GPU Backend

GPU is a powerful computing platform that can be used to accelerate many applications. It is especially well-suited for executing TFHE programs because of its large amount of SIMD pipelines and the massive parallelisms provided by its SIMT execution model. TFHE programs can utilize the parallelism provided by GPUs from two perspectives. First, each bootstrapped-Gate (or *gate* for simplicity) evaluation consists of highly vectorizable operations such as Number Theoretic Transform (NTT) and key-switching operations while GPU's micro-architecture is designed and optimized for vector operations. Second, real-world TFHE programs usually consist of an enormous number of gate evaluations. These gate evaluations can be executed in parallel as long as there's no dependency between them. GPUs usually have a plethora of streaming multiprocessors (SMs) that are capable of executing multiple TFHE gates in parallel.

While CPU backends have been widely adopted to execute the TFHE programs [3]–[5], [34], we are the first toolchain that provides a GPU backend to offer significant speedup for TFHE program execution. Our GPU backend is built on top of the CUDA kernels that evaluate `BootstrappedGate`, the gate primitive from the cuFHE library [2] and uses batch scheduling based on CUDA Graphs [35] to achieve high performance.

The TFHE libraries designed for CPU platforms [9], [10] usually provide APIs at the granularity of single logic gate evaluation. Following a similar approach, cuFHE also provides function calls for individual gate evaluation. However, while such gate-level API design provides an intuitive interface for TFHE program developers, such design is not favorable to achieving efficient GPU execution and fully utilizing GPU's parallelism.

Figure 8 depicts the execution time breakdown when executing four TFHE gates using the cuFHE library. During each cuFHE gate evaluation, the input ciphertext is first copied from the CPU (host) memory to the GPU (device) memory, then a GPU kernel is launched to perform the gate evaluation. After the evaluation completes, the result is copied back to the host memory regardless of whether the ciphertext can be reused in later gate evaluations. During the execution of a GPU kernel, the CPU thread is blocked and cannot perform any other computation.

In order to utilize more SMs simultaneously, cuFHE allows ciphertexts to be batched and evaluated in a vectorized fashion. However, this type of batching does not allow interdependent ciphertexts or mixed types of gate evaluations to be batched. Real-world TFHE programs, however, usually contain lots of interdependent operations with different types of gates, limiting the size of each cuFHE batch. In addition, the CPU thread must wait until all gate evaluations within a batch have finished before processing the next batch, which further lowers the utilization of the GPU resources, making cuFHE inefficient at executing TFHE programs.

Inspired by the recent advances in the ML community to use kernel fusion to reduce data transfer and work submission overheads [36], [37], we utilize CUDA Graphs, an NVIDIA kernel launcher, to define a series of operations before the actual execution and fuse them as a single operation rather than a sequence of individually-launched operations.With this approach, the batch size is now primarily determined by the available GPU memory. CUDA Graphs allow interdependencies between gates to be defined within a batch, enabling GPUs to perform more computations at once. Each batch is a sub-directed-acyclic-graph (sub-DAG) of the entire TFHE program that contains up to around hundreds of thousands of nodes in our GPU execution backend. Unlike the API design flavor in the cuFHE framework [2] where the CPU thread must wait for each API call to finish, we made an essential modification to allow the current batch execution on the GPU and the next batch construction on the CPU to proceed in parallel, further reducing dependency management overheads. The workflow

TABLE II
CONFIGURATION OF THE DISTRIBUTED CPU BENCHMARKING PLATFORM

Host Processor	2 Intel Xeon Gold 5215 2.5GHz (20C/40T)
Memory	4 32GB 2933 MHz DDR4
Network Interface	Intel Corporation I350 Gigabit NIC

for PyTFHE GPU execution backend is shown in Figure 9.

V. EVALUATION

A. Evaluation of PyTFHE Backends on VIP-Bench and Neural Networks

VIP-Bench [38] is a benchmark suite designed for privacy-enhanced computation framework evaluation. A wide range of 18 benchmarks is provided in VIP Bench, including linear arithmetic functions such as *Dot-Product*, iterative approximation algorithms such as *Eulers's Approximation*, and real-world applications such as *MNIST Network* and *Roberts-Cross Edge Detection*. To evaluate the performance of PyTFHE backends, we implemented the benchmarks in the Chisel Hardware Description Language [1] with the exception that the MNIST network benchmark is implemented with *ChiselTorch*. We also implemented two larger MNIST-CNN benchmarks, with two and three convolutional kernels respectively. We refer to the three MNIST-CNN benchmarks as *MNIST_S* (the one included in VIP-Bench), *MNIST_M*, and *MNIST_L* based on their size.

To further evaluate the performance over more complicated neural network structures, we additionally implemented *self-attention* layers, a key component in the BERT (Bidirectional Encoder Representations from Transformers) models [39]. Self-attention enables BERT to effectively capture long-range dependencies between tokens in the input context, resulting in state-of-the-art performance on a wide range of natural language processing tasks. The implementation of self-attention layers also demonstrates the flexibility of ChiselTorch to support non-native complicated neural network structures with the provided primitives. We implemented two versions of self-attention layers. Based on the input sequence length and hidden size, we refer to the self-attention layers as *Attention_S* (for a hidden dimension of 32) and *Attention_L* (for a hidden dimension of 64). The benchmarks are compiled using the PyTFHE flow described in Section IV. The generated TFHE programs are evaluated using the high-performance distributed CPU backend implemented with Ray as described in Section IV-D. We use a medium-sized server for our experiments (Table II).

The evaluation results of PyTFHE's distributed CPU backend are shown in Figure 10. To see how well our distributed CPU backend scales, we first run multiple independent single-threaded dummy TFHE programs with no dependencies between the gates to saturate the node's CPU usage and measure the maximum throughput of processing TFHE gates. The throughput obtained from independent single-threaded TFHE programs indicates the ideal throughput of the CPU server platform. The results show that PyTFHE's distributed CPU backend scales nearly perfectly for large-scale benchmarks

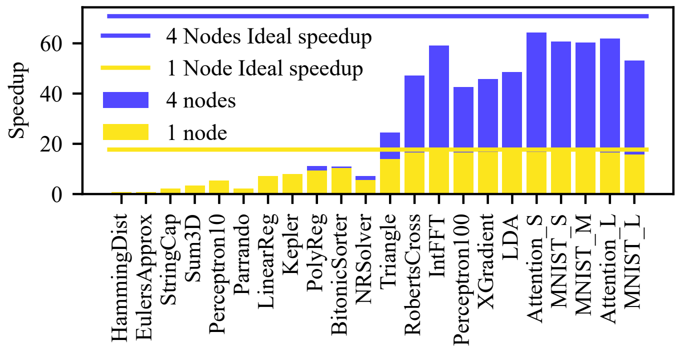

Fig. 10. PyTFHE Distributed CPU vs. Single-Threaded CPU on VIP Bench The benchmark results are normalized to the single-threaded CPU backend. The benchmarks are sorted by the number of gates in the TFHE program in ascending order.

TABLE III
CONFIGURATION OF THE GPU BENCHMARKING PLATFORM

Host Processor	16C/24T Intel Core i9-12900K 4.90 GHz
Memory	4 32GB 3600 MHz DDR4
GPU	NVIDIA RTX A5000 24G, 4090 24G

such as the three MNIST networks. It achieved a speedup of 17.4 (the ideal speedup is 18) compared to the single-threaded CPU backend on a single node and a speedup of 60.5 (the ideal speedup is 72) on a cluster with four nodes.

For smaller benchmarks such as *Hamming Distance* and *Euler's Constant Approximation*, however, the benefit of a distributed system is not as obvious. Two reasons can be identified. First, the runtime of smaller benchmarks is dominated by the overhead of thread creation, data transfer, and synchronization of the distributed system. Second, some of the small benchmarks such as *Newton-Raphson Solver (NRSolver)* have a mostly serial workflow and are not easily parallelizable. As such, it is difficult for these mostly serial benchmarks to fully utilize the parallelism of the distributed system.

We also assessed the performance of our GPU backend using the above benchmarks. We use cuFHE [2], the state-of-the-art GPU TFHE library implementation, as the GPU baseline for comparison. The hardware configuration of the GPU platform is listed in Table III. Both the NVIDIA RTX A5000 GPU and the NVIDIA RTX 4090 GPU are evaluated. PyTFHE's GPU backend achieves up to 61.5× better performance compared to the baseline implemented with cuFHE library shown in Figure 11.

We used NVIDIA Nsight Compute profiler to analyze benchmarks that had modest speedups such as *Parrando*, *Eulers's Approximation*, and *NRSolder*. We observed that the runtime of these benchmarks are dominated by the serial portion of the code and are not easily parallelizable, preventing them from fully exploiting GPU's SMs and the ample parallel execution SMs provide.

B. Comparing PyTFHE with Google's Transpiler

In this subsection, we present the evaluations of Google's Transpiler and PyTFHE. We first describe the experimental

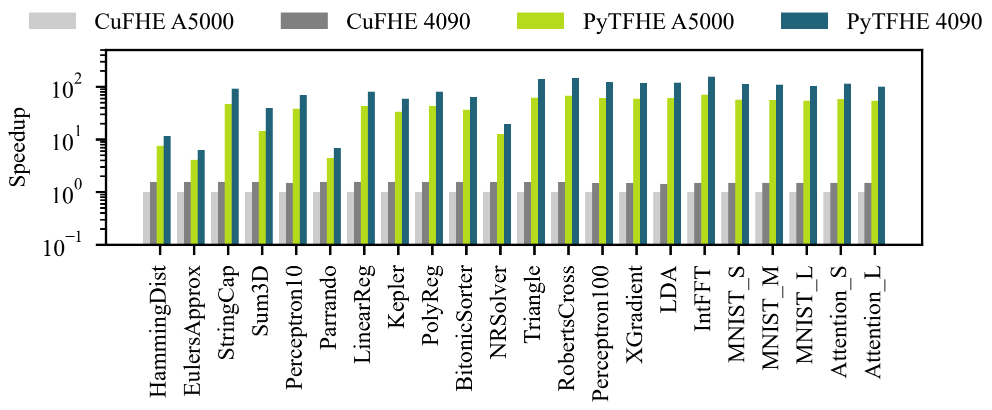

Fig. 11. PyTFHE GPU vs. cuFHE on VIP-Bench and neural networks

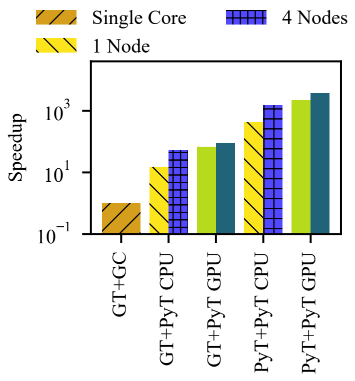

Fig. 12. Transpiler vs. PyTFHE on MNIST.

setup and then present the results.

Experimental Methodology Both Transpiler and PyTFHE are modular and generate intermediate program files, which makes it possible to study the performance impact of different framework components. We use different combinations of the frontends and backends in our experiments to assess 1) how well the PyTFHE framework can generate optimized binaries compared to the Transpiler's XLS frontend, and 2) how well the PyTFHE distributed CPU and GPU backends perform compared to the Transpiler code generation backend. For all the experiments in this subsection, the baseline is the Google Transpiler (frontend and backend).

For the first experiment, we built *MNIST_S* using Transpiler. `GT+GC` denotes using Google Transpiler XLS frontend and Google Transpiler code generation backend to generate the TFHE program. The program is then run on a single-core CPU. This experiment generates our baseline performance. For the second experiment, we tested the frontend from Google Transpiler and our PyTFHE execution backend. We built the same *MNIST_S* model and used Google Transpiler to compile and optimize the model to get the most optimized XLS IR. We then converted the XLS IR to PyTFHE binary format that preserves the same dataflow structure. The converted PyTFHE binary is evaluated using PyTFHE backend on both distributed CPU and GPU backends (denoted as `GT+PyT CPU` and `GT+PyT GPU`).For the third experiment, we use the *MNIST_S* model generated by our *ChiselTorch* frontend and execute with the PyTFHE backend. We performed the evaluation on both distributed CPU and GPU platforms (denoted as `PyT+PyT CPU` and `PyT+PyT GPU`).

Experimental Results Transpiler comparison results are shown in Figure 12. The end-to-end solution of Google Transpiler took days to evaluate the *MNIST_S* model, which makes it impractical for real-world FHE deep learning inference. For the same IR generated and optimized by Transpiler, our distributed CPU backend achieves 52× better performance on four nodes. Our GPU execution backend achieves 69×–89× better performance on NVIDIA RTX A5000 and 4090 GPUs respectively. The speedup of *MNIST_S* inference improves

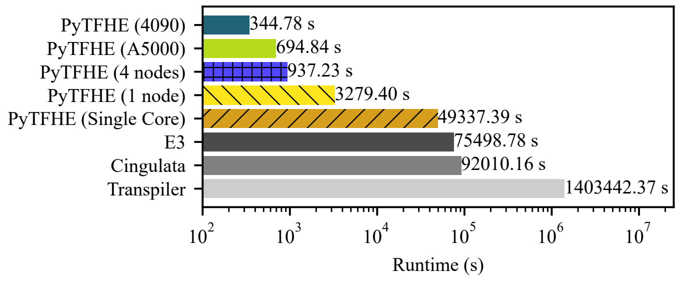

Fig. 13. PyTFHE vs Existing TFHE Frameworks Runtime

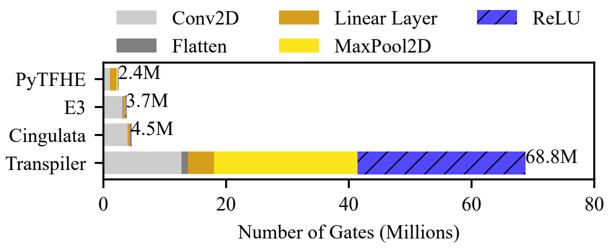

Fig. 14. Gate Distribution of MNIST Network

even further when we use the TFHE program generated by *ChiselTorch* as shown in Table IV. Our distributed CPU backend and GPU backends achieve speedups ranging from 28 to 3369 compared to Google Transpiler.

These results show that our distributed CPU and GPU backends can achieve a significant speedup by utilizing the massive parallelism of multiple CPUs and high-performance GPUs. It also shows that our *ChiselTorch* generates more optimized TFHE programs for *MNIST_S* than Transpiler. Together, PyTFHE allows FHE application developers to build more optimized DNN applications with significantly less effort and achieves much better TFHE program execution performance.

C. Comparing PyTFHE with Cingulata and E3

As introduced in Section III, Cingulata and E3 are two DSL-based frameworks that support TFHE scheme. We built the same *MNIST_S* model for both Cingulata and E3 and

TABLE IV
SPEEDUP OF PYTFHE OVER E3, CINGULATA, AND TRANSPILER

	E3	Cingulata	Transpiler
PyTFHE Single Core	1.5	1.8	28.4
PyTFHE 1 Node	23.0	28.1	427.9
PyTFHE 4 Nodes	80.6	98.2	1497.4
PyTFHE A5000 GPU	108.7	132.4	2019.8
PyTFHE 4090 GPU	218.9	266.86	4070.54

compared their TFHE program execution runtime. The results are shown in Figure 13 and Table IV[1]. For completeness, we also include the runtime for Transpiler and single-core PyTFHE.

We further analyze the number of gates generated by each framework in Figure 14. Our PyTFHE *ChiselTorch* frontend generates TFHE programs with the least number of gates among these four frameworks, which is 65.3% of the number of gates generated by Cingulata and 53.6% of the number of gates generated by E3. Amongst all the frameworks we've studied, Google Transpiler generates a significantly larger program. We believe the reason for Transpiler's poor performance is due to its converting a C/C++ program that is in total ordering to a TFHE program that evaluates gates in partial ordering. This ordering mismatch may prevent Transpiler from detecting parallelism and performing optimizations, and hence results in subpar performance. For example, all other frameworks can detect that the *Flatten* layer in a neural network model performs a tensor *reshape* function and can be optimized into wiring connections while Transpiler still emitted gates for the *Flatten* layer.

VI. CONCLUSION

In conclusion, PyTFHE is a framework that provides state-of-the-art performance and an easy-to-use interface for implementing applications in the TFHE scheme. We demonstrate the effectiveness of PyTFHE by benchmarking it on the VIP-Bench benchmark suite. We also implemented privacy-preserving MNIST image classification applications and self-attention layers and evaluated their performance on multiple backends of PyTFHE. By comparing the performance of PyTFHE with other end-to-end TFHE-supported frameworks, we show that PyTFHE achieves orders of magnitude better performance.

REFERENCES

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.

[2] G. S. Cetin, W. Dai, B. Opanchuk, and Kitsu, "cuFHE: CUDA-accelerated Fully Homomorphic Encryption Library," 2018. [Online]. Available: https://github.com/vernamlab/cuFHE

[3] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak *et al.*, "A general purpose transpiler for fully homomorphic encryption," *arXiv preprint arXiv:2106.07893*, 2021.

[1]The runtime of Cingulata, E3, and Transpiler are estimated using the gate count divided by the average throughput of the TFHE library running on a single CPU core.

[4] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, ser. SCC '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 13–19. [Online]. Available: https://doi.org/10.1145/2732516.2732520

[5] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands," Cryptology ePrint Archive, Paper 2018/1013, 2018, https://eprint.iacr.org/2018/1013. [Online]. Available: https://eprint.iacr.org/2018/1013

[6] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International conference on the theory and application of cryptology and information security*. Springer, 2017, pp. 409–437.

[7] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[8] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 617–640.

[9] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library," Cryptology ePrint Archive, Paper 2022/915, 2022, https://eprint.iacr.org/2022/915. [Online]. Available: https://eprint.iacr.org/2022/915

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption library," August 2016, https://tfhe.github.io/tfhe/.

[11] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1651–1669.

[12] B. Reagen, W. Choi, Y. Ko, V. T. Lee, G. Wei, H. S. Lee, and D. Brooks, "Cheetah: Optimizations and methods for privacypreserving inference via homomorphic encryption," *CoRR*, vol. abs/2006.00505, 2020. [Online]. Available: https://arxiv.org/abs/2006.00505

[13] "Microsoft SEAL (release 4.0)," https://github.com/Microsoft/SEAL, Mar. 2022, microsoft Research, Redmond, WA.

[14] C. Mouchet, J.-P. Bossuat, J. Troncoso-Pastoriza, and J. Hubaux, "Lattigo: A multiparty homomorphic encryption library in go," in *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2020.

[15] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 546–561.

[16] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi, "Eva improved: Compiler and extension library for ckks," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 43–55. [Online]. Available: https://doi.org/10.1145/3474366.3486929

[17] T. van Elsloo, G. Patrini, and H. Ivey-Law, "Sealion: A framework for neural network inference on encrypted data," *arXiv preprint arXiv:1904.12840*, 2019.

[18] J. Lee, E. Lee, J.-W. Lee, Y. Kim, Y.-S. Kim, and J.-S. No, "Precise approximation of convolutional neural networks for homomorphically encrypted data," *arXiv preprint arXiv:2105.10879*, 2021.

[19] D. Comi, "Herbert: a privacy-preserving natural language processing solution for text classification," 2021.

[20] A. Al Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "Privft: Private and fast text classification with homomorphic encryption," *IEEE Access*, vol. 8, pp. 226 544–226 556, 2020.

[21] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "Ramparts: A programmer-friendly system for building homomorphic encryption applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019, pp. 57–68.

[22] A. Feldmann, N. Samardzic, A. Krastev, S. Devadas, R. Dreslinski, K. Eldefrawy, N. Genise, C. Peikert, and D. Sanchez, "F1: A fast and

programmable accelerator for fully homomorphic encryption (extended version)," *arXiv preprint arXiv:2109.05371*, 2021.

[23] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data." in *ISCA*, 2022, pp. 173–187.

[24] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 711–725.

[25] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1237–1254.

[26] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," *arXiv preprint arXiv:2202.08814*, 2022.

[27] Google, "Xls: Accelerated hw synthesis," 2020. [Online]. Available: https://github.com/google/xls

[28] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Paper 2012/144, 2012, https://eprint.iacr.org/2012/144. [Online]. Available: https://eprint.iacr.org/2012/144

[29] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," 2013.

[30] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[32] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11–seamless operability between c++ 11 and python," *URL: https://github. com/pybind/pybind11*, 2017.

[33] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," 2017. [Online]. Available: https://arxiv.org/abs/1712.05889

[34] C. Gouert and N. G. Tsoutsos, "Romeo: Conversion and evaluation of hdl designs in the encrypted domain," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[35] "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs, 2022, [Accessed 12-Dec-2022].

[36] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 711–732. [Online]. Available: https://proceedings.mlsys.org/paper/2021/file/c9e1074f5b3f9fc8ea15d152add07294-Paper.pdf

[37] X. Wang, Y. Wei, Y. Xiong, G. Huang, X. Qian, Y. Ding, M. Wang, and L. Li, "Lightseq2: Accelerated training for transformer-based models on gpus," 2021. [Online]. Available: https://arxiv.org/abs/2110.05722

[38] L. Biernacki, M. Z. Demissie, K. B. Workneh, G. B. Namomsa, P. Gebremedhin, F. A. Andargie, B. Reagen, and T. Austin, "Vip-bench: A benchmark suite for evaluating privacy-enhanced computation frameworks," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021, pp. 139–149.

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.