AHEC: End-to-end Compiler Framework for Privacy-preserving Machine Learning Acceleration

Huili Chen
UC San Diego
San Diego, USA
huc044@ucsd.edu

Rosario Cammarota
Intel Lab
San Diego, USA
rosario.cammarota@intel.com

Felipe Valencia, Francesco Regazzoni
ALaRI
Lugano, Switzerland
valena@usi.ch, regazzoni@alari.ch

Farinaz Koushanfar
UC San Diego
San Diego, USA
farinaz@ucsd.edu

***Abstract*—Privacy-preserving machine learning (PPML) is driven by the emerging adoption of Machine Learning as a Service (MLaaS). In a typical MLaaS system, the end-user sends his personal data to the service provider and receives the corresponding prediction output. However, such interaction raises severe privacy concerns about both the user's proprietary data and the server's ML model. PPML integrates cryptographic primitives such as Multi-Party Computation (MPC) and/or Homomorphic Encryption (HE) into ML services to resolve the privacy issue. However, existing PPML solutions have not been widely deployed in practice since: (i) Privacy protection comes at the cost of additional computation and/or communication overhead; (ii) Adapting PPML to different front-end frameworks and back-end hardware incurs prohibitive engineering cost.**

We propose AHEC, the first *automated, end-to-end HE compiler* for efficient PPML inference. Leveraging the capability of Domain Specific Languages (DSLs), AHEC enables automated generation and optimization of HE kernels across diverse types of hardware platforms and ML frameworks. We perform extensive experiments to investigate the performance of AHEC from different abstraction levels: HE operations, HE-based ML kernels, and neural network layers. Empirical results corroborate that AHEC achieves superior runtime reduction compared to the state-of-the-art solutions built from static HE libraries.

I. Introduction

Machine Learning (ML) models are empowering the fourth industrial revolution due to their unprecedented performance. However, training highly accurate ML models is both time and resource consuming, thus is intractable for resource-constrained customers. Machine Learning as a Service (MLaaS) is a popular solution of ML deployment where the end-user sends his personal data to the ML service provider and receives the final output. The mechanism of MLaaS raises fundamental data privacy concerns since: the user's data is sensitive in the healthcare domain and the user has no control over the usage of his data delivered to the server. This suggests that the ML service provider is required to be a trusted entity, limiting MLaaS's applicability.

To address the privacy concerns in MLaaS, Privacy-Preserving Machine Learning (PPML) has been proposed as a promising solution. PPML techniques can be categorized into three types based on the underlying primitives: Homomorphic Encryption (HE), Yao's Garbled Circuit (GC), and hybrid protocols. Note that HE schemes is non-interactive and *computation-bounded*, while GC solutions requires multiple rounds of interaction and is *communication-bounded*. CryptoNets [1] demonstrates the first HE-based ML inference that performs computation on encrypted data. DeepSecure [2] is an innovative framework that provides a GC-optimized realization of diverse components used in MLaaS. Gazelle [3] presents a hybrid solution which combines HE and GC to achieve a good balance between computation and communication overhead.

In this paper, we focus on accelerating HE-based PPML scheme. We identify the following constraints of existing HE-based solutions: *(i)* The implementation is inflexible and bound to three factors: the ML framework (PyTorch, TensorFlow, MXNet, etc.), the HE library, and the hardware platform. This suggests that the engineering overhead of developing a PPML solution is proportional with respect to the number of front-end framework, the HE library, and the hardware back-ends. *(ii)* The generation of HE kernels is not automated nor optimized. In particular, current techniques such as CryptoNet [1] and nGraph-HE [4], [5] mainly run on CPUs because of their dependency on underlying HE libraries. Directly mapping HE computation to a new hardware requires manually writing kernels for the target platform and the specific HE library. This means that the direct mapping approach is not flexible nor scalable.

There are three possible approaches to realize a HE-based PPML system: *(i)* Direct implementation of the HE library on the target hardware. This method incurs high engineering cost and is not flexible; *(ii)* Extending the Instruction Set Architecture (ISA) of the hardware to support HE operations. Such an approach has better flexibility than the first one, but does not automate/optimize HE kernel generation across different architectures; *(iii)* End-to-end compilation-based technique. The computation flow of the HE-based ML model inference is described in the domain-specific language (DSL). The program is then compiled and optimized given the hardware knowledge. The third approach features the highest degree of design automation and optimization.

This paper aims to develop an *end-to-end* framework to enable HE-based PPML inference, with various ML front-end frameworks and diverse hardware platforms. To this end, we adapt the *compiler-based* approach and present AHEC as a holistic solution. By introducing AHEC, we make the following explicit contributions:

- **Presenting the first end-to-end HE framework**: AHEC leverages *domain specific language* to describe the ML

workload, thus is compatible with arbitrary ML front-end framework and various hardware. Such an *agnostic* property facilitates the integration of AHEC within concurrent MLaaS paradigms with minimal engineering cost.

- **Enabling automated kernel generation and optimization**: AHEC deploys a DSL to describe the computation flow of the ML model, which can be lowered and optimized to the hardware using hardware abstraction layer automatically.
- **Leveraging Algorithm/Software/Hardware co-design**: To minimize the computation latency, we explore the intrinsic *data independence* within the HE-integrated ML workloads and optimize the required kernels while conforming to the hardware constraints.
- **Performing extensive performance evaluations and comparison**: We investigate AHEC's performance of various ML-related workloads. Empirical results show that AHEC outperforms the standard implementation by a large margin in terms of latency.

II. Preliminaries

A. Homomorphic Encryption

Homomorphic encryption is a promising solution to PPML since this primitive allows to perform computation (e.g., ML Inference), on the encrypted data without decrypting it [6]. The data used in HE evaluation is represented as element on a ring $\mathcal{R}$. Note that HE supports two types of computation, i.e., homomorphic addition (denoted by $\oplus$) and homomorphic multiplication (denoted by $\otimes$). These two properties are described in the following equations:

$$\mathbb{E}(x_1) \oplus \mathbb{E}(x_2) = \mathbb{E}(x_1 + x_2), \tag{1}$$

$$\mathbb{E}(x_1) \otimes \mathbb{E}(x_2) = \mathbb{E}(x_1 \times x_2). \tag{2}$$

In Eq. (1) and (2) above, $\mathbb{E}$ denotes the homomorphic encryption function. The two operators $+$ and $\times$ on the right-hand side are regular arithmetic operators. Note that HE primitives can be used to compute arbitrary *polynomial functions* given the properties of homomorphic addition and multiplication. Considering the PPML paradigm, HE-based protocols allow the user to send the encrypted data to the service provider and receives the corresponding encrypted output. In this way, we can ensure that the user's data, the final prediction output, and the server's ML model remain confidential during the computation task. *'Level'* is another key concept in the standard implementation of HE schemes [7]. Particularly, the two operands of HE addition and multiplication shall have the same level in the modulus switching chain. If their level values are not consistent, the operand with the higher level needs to be transformed into the target lower level.

We focus on the *levelled, approximate HE* scheme in this paper. A leveled HE scheme [8] supports HE evaluation within the pre-defined depth. Levelled homomorphic encryption (LHE) can be augmented with bootstrapping to achieve full homomorphic encryption (FHE) [9]. An Approximate HE scheme (e.g., CKKS [10]) allows HE computation with real-valued inputs while the decryption is only approximate. Since ML models are inherently tolerant to the perturbation of intermediate results (this property has been explored by model quantization and pruning), approximate HE is a suitable candidate for PPML.

B. Compilers for ML Workload

There has been a line of research on developing compilers to automate and optimize ML applications. TVM [11] is an end-to-end compiler stack for deep learning systems. It explores graph-level and operator-level optimization across various hardware back-ends. In particular, TVM introduces an innovative *tensor expression language* that can be used to build operators for the target ML workload. To explore optimal code in the large search space, TVM provisions program transformation primitives that generate different variants of the program with diverse optimizations. nGraph [12] is an open-source compiler stack from Intel that aims to accelerate the deployment of ML workloads developed with popular programming frameworks. nGraph optimizes the execution of the ML model by describing the task as a computation graph and performing various graph-level transformations.

C. Privacy-preserving Machine Learning

We categorize existing works on PPML into three types based on the underlying cryptography primitives.

▪ **HE-based.** CryptoNets [1] takes the first step to integrate HE primitive into ML inference. Since HE only supports addition and multiplication operations, CryptoNets *approximates* the non-linear activation function in the given neural network (NN) with a quadratic function. As a result, the accuracy of the estimated model is lower than the original one. nGraph-HE [5] extends nGraph with HE primitives by treating HE as an additional hardware target. CHET [13] is an optimizing compiler for homomorphic neural network inference. It introduces Homomorphic Instruction Set Architecture (HISA) as the intermediate representation to construct the data flow.

▪ **GC-based.** DeepSecure [2] is a scalable and provably secure framework based on Yao's GC protocol. To reduce the computation and communication overhead, DeepSecure proposes both data and neural network transformation as pre-processing steps as well as various GC optimization techniques.

▪ **Hybrid Protocols.** Gazelle [3] combines GC and Packed Additively Homomorphic Encryption (PAHE) for secure NN evaluation. The linear layers are computed via optimized PAHE kernels and the non-linear layers are processed using GC. Chemaleon is a hybrid framework that uses GC, Goldreich-Micali-Wigderson (GMW), and additive secret sharing protocol for secure two-party computation.

III. AHEC Framework

The system overview of AHEC is shown in Figure 1. AHEC is a *holistic, end-to-end* framework for PPML as a service with the goal of bridging the gap between diverse ML front-ends (e.g., TensorFlow, PyTorch, Caffe) and various hardware back-ends (e.g., CPUs, and GPUs, emerging NN accelerators). AHEC is the first framework that takes an *Algorithm/Software/Hardware co-design* approach to achieve

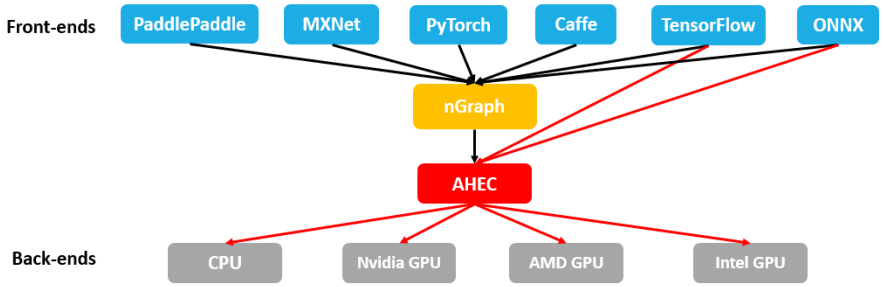


Fig. 1: Overview of AHEC framework. AHEC closes the gap between front-end programming frameworks and hardware back-ends. It uses a new DSL to integrate HE primitives and characterizes the target device with hardware abstraction layer.

automation and optimization of kernel generation for HE-based PPML inference across different hardware platforms.

A. HE-based PPML Protocol

Figure 2 illustrates the global flow of our HE-based secure evaluation protocol. To enable PPML service, AHEC consists of the following four steps:

(I) Key Generation. Given the pre-defined security level, AHEC first generates the public and secret keys that are used in later operations. The user and the MLaaS provider holds the secret key and the public key, respectively.

(II) Data Encoding and Encryption. After key generation, the user first encodes the raw data into a polynomial ring, which is the corresponding *plaintext*. The plaintext is then encrypted with the *public key* and the resulting *ciphertext* is sent to the ML service provider.

(III) HE Evaluation of ML model. The server performs a series of HE operations on the ciphertext data received from the end user. In particular, *linear layers* including fully connected (FC) layers, convolutional (conv) layers, and mean pooling layers are implemented as the combination of HE addition and HE multiplication with the public evaluation keys. AHEC handles non-linear activation functions with two alternative approaches: (i) Approximating with polynomial functions (similar to CHET [13] and CryptoNets [1]); (ii) Delegating the computation to the end user. More specifically, the intermediate ciphertext is sent back from the server to the client, decrypted, used to perform non-linear computation in the plaintext domain. The updated ciphertext is then re-encrypted and sent to the server to proceed ML inference.

(IV) Output Decryption and Decoding. The server obtains the final encrypted result and sends it to the end user. The user decrypts the ciphertext using the *secret key* and decodes the plaintext to the desired (raw) output.

B. Domain Specific Language for HE

A domain specific language is a *specialized* computer programming language that is used to express statements in a particular problem space. Compared to general-purpose languages that are applicable in various domains, DSL generates program codes with superior performance in the domain it targets at. AHEC aims to automate and optimize *code generation* of HE kernels for the target hardware. As such, DSL is a suitable candidate for our objective.

AHEC extends the *Tile* DSL in PlaidML [14] to support efficient HE-based ML inference. In particular, AHEC describes HE operators (including HE addition, HE multiplication, and rescaling) using Tile language to fully leverage its optimization potential and hardware awareness. Let us first take a look at PlaidML framework. PlaidML [14] is a tensor compiler that leverages domain specific language and polyhedral optimization to accelerate deep learning workloads across different hardware. Tile language is amenable to generate efficient kernels since it has an intriguing feature called *'contraction'*. Particularly, Tile represents the index space of a tensor operation by specifying bounding polyhedron instead of using nested loops. Take matrix multiplication as an example, the formula of computing $C = AB$ is shown in Eq. (3).

$$C[i,j] = \sum_{k}(A[i,k] \cdot B[k,j]), \tag{3}$$

In Tile, the same operation is expressed as Eq. (4). Here, M is the first dimension of matrix A and N is the last dimension of matrix B. The *contraction operator* '$=+$' means that when multiple values are computed for the same output location, they are added up.

$$C[i,j:M,N] = +(A[i,k] * B[k,j]), \tag{4}$$

The main advantage of writing the computation as in Eq. (4) is that the *element-wise* multiplication can be done in *parallel*, thus reducing the runtime of the operation. We make a *key observation* that Tile DSL is particularly suitable for HE workloads since the operation between two ciphertexts and/or the operation between one ciphertext and one plaintext mainly consists of element-wise computation (e.g., multiplication, modulus arithmetic). As such, AHEC leverages the contraction property of Tile language to accelerate HE-based neural network inference.

C. Hardware-aware Optimization

AHEC is developed with *Algorithm/Software/Hardware co-design principle*. Besides the domain specific language that facilitates kernel generation (Sec. III-B), AHEC integrates the hardware knowledge into the design loop by explicitly modeling the target device. In particular, AHEC characterizes the target hardware with a set of constraints (e.g., number of threads, number of registers, memory architecture) that are used in the *cost model-based*, *hardware-aware* optimization. Using pre-defined fixed passes, AHEC's optimization is locally optimal and configuration-driven.

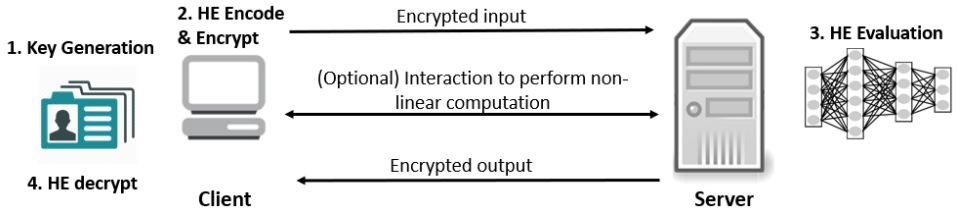


Fig. 2: Privacy-preserving inference protocol of AHEC framework. HE encryption and decryption is locally performed by the end user. AHEC targets to optimize the HE evaluation step performed by the service provider.

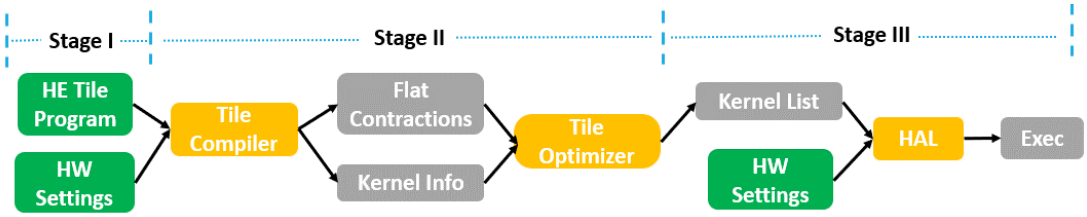


Fig. 3: Workflow of AHEC end-to-end HE compiler framework.

We deploy a diverse set of optimization techniques to reduce the latency of the HE-based PPML inference:

- **Vectorization.** AHEC rewrites loops such that the same operation is performed on multiple vector elements simultaneously while considering the memory constraints.
- **Tiling.** For each index, we use hill climb and a cost model to maximize the reuse while ensuring that the operands can fit into the cache and registers.
- **Data Layout Selection.** AHEC transforms ciphertext and plaintext data layouts into hardware-friendly forms. We leverage the domain-specific knowledge of HE and pre-specify the preferred data layout for each operator. Layout transformation is performed to match the
- **Threading.** AHEC unrolls inner loops into hardware threads given the knowledge of hardware constraints.

Leveraging Algorithm /Software /Hardware co-design, AHEC yields superior advantage over the existing PPML techniques that are oblivious of the underlying hardware back-ends.

D. AHEC Workflow

The overall workflow of AHEC is shown in Figure 3, consisting of three main stages. AHEC requires two types of inputs: the program description of the ML model inference written in Tile language; and the hardware constraints of the target device. The output of AHEC is the optimized executable (kernels) for the pertinent hardware. We detail each step of AHEC as follows:

(Stage I) Describing ML workload in Tile DSL. In the first stage, AHEC incorporates HE primitives into the inference data flow of the ML model and describes the program in Tile language. Note that *linear layers* (e.g., fully connected layer, convolutional layer, average pooling layer) and quadratic activation layers in the neural network consist of addition and multiplication operations that are inherently supported by HE primitives. As such, AHEC implements the homomorphic variant of these layers using HE addition and HE multiplication as the basic building blocks. The Tile code is sent to the *Tile compiler*, which flattens the contraction in the Tile program and generates the corresponding kernel description.

(Stage II) Kernel Optimization and Code Generation. In the second stage, AHEC deploys the *Tile optimizer* to rewrite the compiled kernel from Stage I and generate the functionality-equivalent kernels with better efficiency. In particular, the Tile optimizer *simulates* the execution of kernel on the target device with varying tile sizes, thus facilitating the automated conversion of the flat contractions into performant kernels. AHEC's optimizer generates kernel information objects that describes the input-output relation for each kernel using the semantic tree ('semtree') as the intermediate representation. The semantic trees of HE kernels are then transformed to produce equivalent output via more efficient code.

(Stage III) Runtime HAL-based Execution of Tile Code. In the third stage, the hardware abstraction layer (HAL) of the target device is used as an interface to generate machine code and schedule the execution of HE operations. AHEC explores the data layout of ciphertexts and plaintexts, tensorizes their representations, and pipelining the computation by tiling nested loops using the polyhedral optimization framework [15]. Note that our co-optimization approach also explores the memory hierarchy of the underlying hardware (Sec. III-C).

E. HE Evaluation Supports

AHEC is general and agnostic to the choice of HE scheme. We select CKKS scheme [10] as the HE primitive since it features fast HE arithmetic and supports fixed-point numbers as well as full RNS optimizations. AHEC identifies the key building blocks of HE evaluation in the CKKS scheme and provisions automated and optimized implementation of these operators for the target device using domain specific language and Algorithm/Software/Hardware co-design. We use the open-sourced HE library named SEAL [7] as the baseline implementation and explain how AHEC improves over SEAL.

Let us consider three integers N, k, q (all larger than 1) where $N = 2^k$, t is a prime number, and a ring $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$. The *ciphertext space* and the *plaintext space* are denoted as $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ and $\mathcal{R}_t = \mathcal{R}/t\mathcal{R}$, respectively. Given two ciphertexts $\mathbf{c}_1 = (c_{0,1},\ c_{1,1})$ and $\mathbf{c}_2 = (c_{0,2},\ c_{1,2})$, AHEC supports the following three types of HE operations:

▪ **HE Addition.** HE addition is performed element-wise:

$$\begin{aligned}\mathbf{c}_{out} &= HE.Add\ (\mathbf{c}_1,\ \mathbf{c}_2)\\ &= (c_{0,1} + c_{0,2},\ c_{1,1} + c_{1,2})\end{aligned}$$

Note that each component $c_{i,j}$ is represented as a polynomial of degree N. AHEC's DSL vectorizes the above element-wise addition to reduce the runtime overhead.

▪ **HE Multiplication.** Homomorphic multiplication between two ciphertexts $\mathbf{c_1}$ and $\mathbf{c_2}$ are computed as follows:

$$\begin{aligned}\mathbf{c}_{out} &= HE.Mult\ (\mathbf{c}_1,\ \mathbf{c}_2)\\ &= (c_{0,1}c_{0,2},\ c_{0,1}c_{1,2} + c_{1,1}c_{0,2},\ c_{1,1}c_{1,2}).\end{aligned}$$

Similar to the analysis of HE addition, each component of $\mathbf{c_{out}}$ is computed element-wise between the corresponding polynomials, thus can be tiled by AHEC's optimization. Note that SEAL library implements the HE addition and multiplication kernels using nested for loops, thus incurring high computation latency. AHEC deploys various hardware-aware optimizations (Sec. III-C) to ensure minimize the runtime of HE-based inference.

▪ **Rescaling.** To support fixed-point data input, CKKS schemes need to scale the data with a proper factor. However, the scaling factor of the output ciphertext grows larger than the one of the input ciphertexts after HE multiplication. As such, *rescaling* is indispensable for CKKS schemes to control the precision of HE evaluation. To further reduce the latency of rescaling, AHEC implements Number Theoretic Transform (NTT) and its inverse transformation in the rescaling operation using the non-inplace version. This allows AHEC to tile the computationally expensive NTT operations.

IV. Evaluations

A. Experimental Setup

AHEC is a generic privacy-preserving inference framework that is *agnostic* to the *front-end* ML programming as well as the underlying hardware *back-ends*. To investigate the performance of AHEC, we test on two types of hardware platforms: CPU and GPU. Particularly, we use Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz as our CPU device and Nvidia Titan Xp with 11.91 GiB memory as the GPU platform. To emulate the MLaaS paradigm, we assume the data residing on the client's side is encrypted as ciphertext and the weight parameters residing on the service provider's side is encoded as plaintext. This assumption is consistent with the setting of CrypoNets [1]. All experiments are run with single core and single thread. To demonstrate the performance improvement of AHEC, we use SEAL library [7] as the baseline standard implementation. We choose the security parameters recommended by SEAL to ensure 128-bit security. The coefficient count (which is also the polynomial degree) is set to $N = 8192$. We repeat the measurement of each benchmark with different data for 1000 runs and report the average runtime for comparison.

B. Results

We investigate the performance of AHEC on various benchmarks. In particular, we characterize the workloads from three levels: HE operation-level, ML kernel-level, and neural network layers. We detail each type of benchmarks in the following sections.

1) HE Operation-Level: Figure 4 visualizes the relative speedup of AHEC-CPU compared to SEAL baseline on the HE-operator level. The three operators on the horizontal direction denotes HE addition between a ciphertext and a plaintext (Add_CP), HE addition between two ciphertexts (Add_CC), HE multiplication between a ciphertext and a plaintext ($Mult\_CP$). One can see from the comparison that AHEC demonstrates superior latency reduction on basic HE operators compared to SEAL baseline.

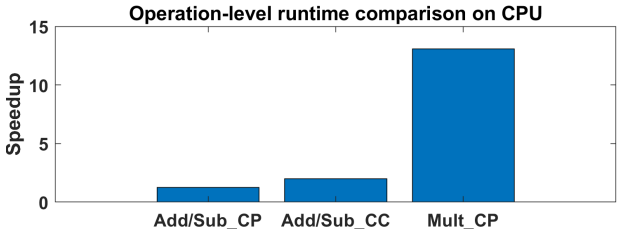
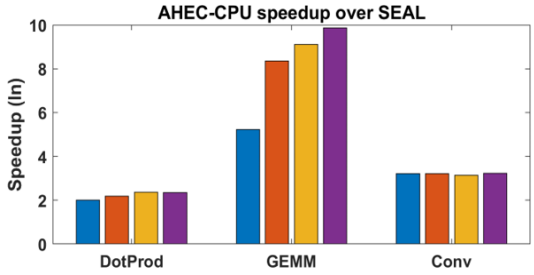


Fig. 4: HE Operation-level runtime comparison on CPU.

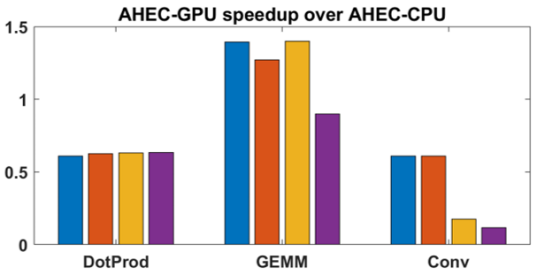
2) ML Kernel-level: Figure 5 shows the kernel-level runtime performance of AHEC. We evaluated three types of kernels that are typical in ML applications: dot product (DotProd), general matrix multiplication (GEMM), and convolution (Conv). To corroborate the *scalability* of AHEC, we vary the dimensionality of the inputs and measure the corresponding runtime. More specifically, we test four vector dimensions for DotProd: 32, 64, 128, 256. For GEMM, we evaluate four input sizes: [32,32] with [32,16], [64,64] with [64,16], [128,128] with [128,16], and [256,256] with [256,16]. As for Conv, we fix the filter size to $[3, 3]$ and set the stride to 1. No input padding is used here. The input to Conv is a 2D matrix with size [28,28], [32,32], [64,64], or [256,256]. The above specified four input sizes are denoted with different colors respectively in Figure 5.

Figure 5a illustrates the relative speedup of AHEC running on CPU compared to the SEAL baseline. We take the natural logarithm (ln) of the speedup for better visualization. It can be seen that AHEC greatly outperforms SEAL in terms of runtime overhead. This is because AHEC leverages the *Algorithm/Software/Hardware co-design* principle to optimize the execution of the Tile kernels on the pertinent device. The Tile DSL enables polyhedral optimization that fully explores data independence and achieves computation parallelism. Furthermore, we can see that AHEC scales approximately linearly on DotProd and GEMM kernels, while its scalability degrades on Conv operations with large inputs.

To study the impact of *hardware architecture* on AHEC's performance, we run the same benchmarks on both CPU and GPU platform. Figure 5b shows the relative speedup of AHEC-GPU with respect to AHEC-CPU on HE kernels with different data sizes. One can see that the architecture of GPUs is not uniformly better than CPU for HE operations. We emphasize that, as an end-to-end framework, AHEC succeeds in providing a portable HE solution on GPUs without prohibitive manual redesign efforts. It will be interesting as the future work to study



(a)



(b)

Fig. 5: Runtime comparison between AHEC-CPU/GPU and SEAL with different input sizes (denoted by different colors).

how to determine the optimal hardware component for running a specific HE kernel in a heterogeneous computing system.

3) Neural Network Layers: Table I shows the topology of the neural network we evaluate in our experiment. Compared to CryptoNets [1], AHEC achieves 750x, 3.9x, 6350x, 1.2x, and 400x runtime speedup on each of these five layers, respectively. Note that AHEC also automatically performs the *rescaling* operation following each layer outlined in the table to control the output precision.

TABLE I: Architecture of the evaluated neural network.

Layer	Description
Conv1	Kernel size 5x5, stride is (2, 2), and number of output channels is 5
Square1	Element-wise squaring of the input tensor
Pooling	Weighted sum layer with kernel size 3x3
Square2	Element-wise squaring of the input tensor
FC (Output)	Weighted sum that generates 10 outputs for each class

C. Discussion

We discuss the *architectural impact* of the hardware on AHEC's performance here. Comparing the relative runtime of AHEC on CPU and GPU shown in Figure 5b, We can see that these two architectures are suitable at different tasks. GPUs are believed to be more powerful than CPUs due to their capability of processing massive streams of data in parallel with multiple cores, while an individual GPU core runs slower than a CPU core. In our experiments, we evaluate the runtime using *single-core* and *single-thread*. As such, AHEC runs faster on GPU particularly on GEMM kernels, since in this case the amount of data being processed in parallel is much larger than the ones in DotProd and Conv kernels. It is worth noticing that the hardware back-end is desired to feature sufficient memory and support 64-bit unsigned integer data type to fully explore the optimization capability of AHEC for realizing the CKKS scheme. AHEC can be extended with efficient *bootstrapping* techniques to achieve fully homomorphic encryption. This will allow AHEC to execute inference on deep neural networks with many layers.

V. Conclusion

We present AHEC as the first end-to-end homomorphic encryption compiler framework for privacy-preserving machine learning. AHEC is *agnostic* to both the front-end machine learning programming frameworks as well as various hardware back-ends, provisioning a holistic HE solution for different application scenarios. The core of AHEC is the *Algorithm/Software/Hardware co-design* principle. More specifically, we leverage Tile *domain specific language* to describe the computation flow of the ML model and incorporates the hardware constraints as an abstraction layer while automatically generating optimized kernels. We perform extensive experiments to investigate the performance of AHEC on HE operators and common ML kernels and compare it with the standard HE implementation. Empirical results show that AHEC reduces the inference latency by a large margin compared to the baseline implementation and features superior *scalability* when the data size becomes large.

References

[1] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International Conference on Machine Learning*, 2016, pp. 201–210.

[2] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: Scalable provably-secure deep learning," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 2.

[3] 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.

[4] F. Boemer, Y. Lao, and C. Wierzynski, "ngraph-he: A graph compiler for deep learning on homomorphically encrypted data," *arXiv preprint arXiv:1810.10121*, 2018.

[5] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "ngraph-he2: A high-throughput framework for neural network inference on encrypted data," *arXiv preprint arXiv:1908.04172*, 2019.

[6] C. Gentry *et al.*, "Fully homomorphic encryption using ideal lattices." in *Stoc*, vol. 9, no. 2009, 2009, pp. 169–178.

[7] Microsoft, "Microsoft seal library." https://github.com/microsoft/SEAL, 2019.

[8] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand, "A guide to fully homomorphic encryption." *IACR Cryptology ePrint Archive*, vol. 2015, p. 1192, 2015.

[9] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *International Workshop on Public Key Cryptography*. Springer, 2012, pp. 1–16.

[10] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *International Conference on Selected Areas in Cryptography*. Springer, 2018, pp. 347–368.

[11] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: end-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, pp. 1–15, 2018.

[12] I. AI, "ngraph - open source c++ library, compiler and runtime for deep learning ." https://github.com/NervanaSystems/ngraph, 2019.

[13] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019, pp. 142–156.

[14] PlaidML, "Plaidml: A platform for making deep learning work everywhere." https://github.com/plaidml/plaidml, 2019.

[15] "Polyhedral compilation," https://polyhedral.info, accessed: 2019-09-16.