Does Fully Homomorphic Encryption Need Compute Acceleration?

Leo de Castro[1,*] Rashmi Agrawal[2,3,*,$] Rabia Yazicigil[2] Anantha Chandrakasan[1]
Vinod Vaikuntanathan[1] Chiraag Juvekar[3] Ajay Joshi[2]
[1]MIT, Cambridge, MA, USA; [2]Boston University, Boston MA, USA; [3]Analog Devices, Boston, MA USA
{ldec, anantha, vinodv}@mit.edu, {rashmi23, rty, joshi}@bu.edu, chiraag.juvekar@analog.com
[*]Equal Contribution [$]Work done during internship at Analog Devices

***Abstract*—The emergence of cloud-computing has raised important privacy questions about the data that users share with remote servers. While data in transit is protected using standard techniques like Transport Layer Security (TLS), most cloud providers have unrestricted plaintext access to user data at the endpoint. Fully Homomorphic Encryption (FHE) offers one solution to this problem by allowing for arbitrarily complex computations on encrypted data without ever needing to decrypt it. Unfortunately, all known implementations of FHE require the addition of *noise* during encryption which grows during computation. As a result, sustaining deep computations requires a periodic noise reduction step known as *bootstrapping*. The cost of the bootstrapping operation is one of the primary barriers to the wide-spread adoption of FHE.**

In this paper, we present an in-depth architectural analysis of the bootstrapping step in FHE. First, we observe that secure implementations of bootstrapping exhibit a low arithmetic intensity (< 1 Op/byte), require large caches (> 100MB) and as such, are heavily bound by the main memory bandwidth. Consequently, we demonstrate that existing workloads observe marginal performance gains from the design of bespoke high-throughput arithmetic units tailored to FHE. Secondly, we propose several cache-friendly algorithmic optimizations that improve the throughput in FHE bootstrapping by enabling up to $3.2\times$ higher arithmetic intensity and $4.6\times$ lower memory bandwidth. Our optimizations apply to a wide range of structurally similar computations such as private evaluation and training of machine learning models. Finally, we incorporate these optimizations into an architectural tool which, given a cache size, memory subsystem, the number of functional units and a desired security level, selects optimal cryptosystem parameters to maximize the bootstrapping throughput.

Our optimized bootstrapping implementation represents a best-case scenario for compute acceleration of FHE. We show that despite these optimizations, bootstrapping (as well as other applications with similar computational structure) continue to remain bottlenecked by main memory bandwidth. We thus conclude that secure FHE implementations need to look beyond accelerated compute for further performance improvements and to that end, we propose new research directions to address the underlying memory bottleneck. In summary, our answer to the titular question is: *yes, but only after addressing the memory bottleneck*!

I. Introduction

The rapid development of cloud-based systems has enabled reliable and affordable access to shared computing resources at scale. However, this shared access raises substantial privacy and security challenges. Therefore, new techniques are required to guarantee the confidentiality of sensitive user data when it is sent to the cloud for processing. Fully Homomorphic Encryption (FHE) [15], [29] enables cloud operators to perform complex computations on encrypted user data without ever needing to decrypt it. The result of such FHE-based computation is in an encrypted form and can only be decrypted by the data owner. An illustrative use case of how a data owner can outsource computation on private data to an untrusted third-party cloud platform is shown in Figure 1.

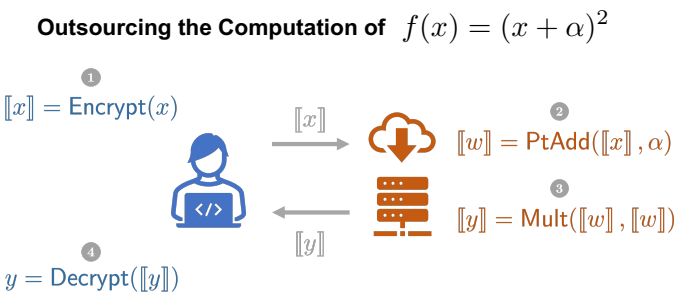

Fig. 1. Third-party cloud platform with outsourced FHE-based computing.

While FHE-based privacy-preserving computing is promising, performing large encrypted computations with FHE still remains several orders of magnitude slower than operating on unencrypted data, which makes broad adoption impractical. This slowdown is an inherent feature of all existing lattice-based FHE schemes. All of these schemes produce ciphertexts containing a noise term, which is necessary for security. Each subsequent homomorphic operation performed on the ciphertext increases its noise, until it grows beyond a critical level after which recovery of the computation output is impossible. Sustained FHE computation thus requires a periodic de-noising procedure, called *bootstrapping*, to keep the noise below a correctness threshold. Unfortunately, this bootstrapping step is expensive in terms of both compute and memory requirements and is often $> 100\times$ more expensive than primitive operations like addition and multiplication on encrypted data.

Real-world applications commonly attempt to amortize this bootstrapping cost across multiple homomorphic operations. Even when considering these application-specific optimizations, bootstrapping consumes more than 50% of the total compute and memory budget for end-to-end operations like machine learning training [22]. To make FHE-based comput-

ing practical, we need to consider a multi-layer approach to accelerate both the bootstrapping step as well as its primitive building blocks using a combination of algorithmic and hardware techniques.

In this work, we first perform a thorough compute and memory analysis of both simple and complex FHE primitives including the bootstrapping step, with an intent to determine the limits and potential opportunities for accelerating FHE. Our analysis reveals that all FHE operations exhibit low arithmetic intensity (< 1 Op/byte) and require working-set sizes of hundreds of MB for practical and secure parameters. In fact, we observe that most existing performance optimization techniques for FHE often *increase* memory bandwidth requirements. These include both linear and non-linear operation optimizations proposed by Han and Ki [20], Han, Hhan and Cheon [18], and Bossuat, Mouchet, Troncoso-Pastoriza and Hubaux [3]. Recent bootstrapping implementation on GPUs by Jung, Kim, Ahn, Cheon and Lee [22] is the first work to perform memory-centric optimizations for linear operations in bootstrapping. Even after these optimizations, their implementation continues to be bounded by main memory bandwidth and exhibits an arithmetic intensity of < 1 Op/byte. On the other side of the design spectrum, recent work by Samardzic et al. [30] presents an architecture for a high-throughput hardware accelerator for FHE. This work primarily focuses on smaller parameter sets where full ciphertexts fit in on-chip cache memory allowing them to bypass the memory bandwidth limitation. However, many natural applications such as SIMD boostrapping, deep-neural network inference (with complex activation functions) and machine learning require larger parameter sets that are not addressed in [30].

In this work, we focus on presenting our three key contributions, i.e., application benchmarking, new techniques to improve memory performance, and evaluation of these techniques on end-to-end applications. More specifically:

- We present detailed benchmarking of the compute and memory requirements of various FHE computations ranging from primitive operations to end-to-end applications such as machine-learning training. We show that all these benchmarks exhibit low arithmetic intensity and require large working-sets in on-chip memory. We observe that these working-sets do not fit in the last level caches of today's reticle-limited chips leading to bootstrapping and other applications being bottlenecked by memory accesses.
- We next present techniques to improve main memory bandwidth utilization by effectively managing the moderate last-level cache provided by currently available commercial hardware. For cache-pressured hardware ($<$ 20 MB LLC) we propose a domain-specific physical address mapping to enhance DRAM utilization. We then present hardware-independent algorithmic optimizations that reduce memory and compute requirements of FHE operations.
- We finally propose an optimized, memory-aware cryptosystem parameter set that maximizes the throughput in FHE bootstrapping and logistic regression training by enabling up to $3.2\times$ higher arithmetic intensity and $4.6\times$ lower memory bandwidth.

TABLE I
CKKS FHE PARAMETERS AND THEIR DESCRIPTION.

Parameter	Description
N	Number of coefficients in a polynomial in the ciphertext ring.
n	$N/2$, number of plaintext elements in a single ciphertext.
Q	Full modulus of a ciphertext coefficient.
q	Machine word sized prime modulus and a limb of Q.
Δ	Scaling factor of a CKKS plaintext.
P	Product of the additional limbs added for the raised modulus.
L	Maximum number of limbs in a ciphertext.
ℓ	Current number of limbs in a ciphertext.
dnum	Number of digits in the switching key.
α	$\lceil (L+1)/\mathsf{dnum} \rceil$. Number of limbs that comprise a single digit in the key-switching decomposition. This value is fixed throughout the computation.
β	$\lceil (\ell+1)/\alpha \rceil$. An ℓ-limb polynomial is split into this number of digits during base decomposition.

The techniques that we propose often compose with prior art and can be used as drop-ins to provide performance improvements in existing implementations without the need for new hardware. Our proposed bootstrapping parameter set represents an upper limit on the performance of FHE operations that can be attained through pure compute acceleration when paired with existing state-of-art memory subsystems. Even with this optimal parameter set, we observe that the bootstrapping step is still primarily memory bound. Thus:

> *Our key conceptual take-away is that to accelerate FHE, we need novel techniques to address the underlying memory bandwidth issues. Compute acceleration alone is unlikely to make a dent.*

Towards the goal of addressing memory bandwidth issues, we propose novel near-term algorithmic and architectural research directions.

II. Fully Homomorphic Encryption: The API

To set the stage, in this section we present the operations implemented by the Cheon-Kim-Kim-Song (CKKS) [11] FHE scheme. We organize these operations in the form of an API that can be used by any application developer to design privacy-preserving applications. Specifying the CKKS scheme requires several parameters, and we summarize our notation for these parameters in Table I. Though we focus on the CKKS scheme, the API is generic and can be used for the BGV [5] and B/FV [4], [14] schemes as well[1].

A. Homomorphic Encryption API

The basic plaintext data-type in CKKS is a vector of length n where each entry is chosen from $\mathbb{C}$, the field of complex numbers. All arithmetic operations on plaintexts are

[1]An exception is the Conjugate function, which the BGV and B/FV schemes do not support, since they do not encrypt complex numbers.

TABLE II
CKKS FULLY HOMOMORPHIC ENCRYPTION API.

Operation Name	Output	Implementation	Description
$\mathsf{PtAdd}(\llbracket \mathbf{x} \rrbracket, \mathbf{y})$	$\llbracket \mathbf{x}+\mathbf{y} \rrbracket$	$\llbracket \mathbf{x} \rrbracket + \mathbf{y}$	Adds a plaintext vector to an encrypted vector.
$\mathsf{Add}(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket)$	$\llbracket \mathbf{x}+\mathbf{y} \rrbracket$	$\llbracket \mathbf{x} \rrbracket + \llbracket \mathbf{y} \rrbracket$	Adds two encrypted vectors.
$\mathsf{PtMult}(\llbracket \mathbf{x} \rrbracket, \mathbf{y})$	$\llbracket \mathbf{x} \cdot \mathbf{y} \rrbracket$	Algorithm 1	Multiplies a plaintext vector and an encrypted vector.
$\mathsf{Mult}(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket)$	$\llbracket \mathbf{x} \cdot \mathbf{y} \rrbracket$	Algorithm 2	Multiplies two encrypted vectors.
$\mathsf{Rotate}(\llbracket \mathbf{x} \rrbracket, k)$	$\llbracket \phi_k(\mathbf{x}) \rrbracket$	Algorithm 3	Rotates a vector by k positions; see Section II-A for an illustration.
$\mathsf{Conjugate}(\llbracket \mathbf{x} \rrbracket)$	$\llbracket \bar{\mathbf{x}} \rrbracket$	Algorithm 3†	Outputs an encryption of the complex conjugate of the encrypted input vector.

† Through a clever encoding [11], the Conjugate operation implementation is identical to the Rotate implementation.

component-wise; the entries of the vector $\mathbf{x}+\mathbf{y}$ (resp. $\mathbf{x}\cdot\mathbf{y}$) are the component-wise sums (resp. products) of the entries of $\mathbf{x}$ with the corresponding entries of $\mathbf{y}$. We denote the encryption of a length-n vector $\mathbf{x}$ by $\llbracket \mathbf{x} \rrbracket$.

Table II gives a complete description of the API with the exception of the rotation operation, which we describe here. The Rotate operation takes in an encryption of a vector $\mathbf{x}$ of length n and an integer $0 \leq k < n$, and outputs an encryption of a rotation of the vector $\mathbf{x}$ by k positions. As an example, when $k = 1$, the rotation $\phi_1(\mathbf{x})$ is defined as follows.

$$\mathbf{x} = \begin{pmatrix} x_0 & x_1 & \dots & x_{n-2} & x_{n-1} \end{pmatrix}$$
$$\phi_1(\mathbf{x}) = \begin{pmatrix} x_{n-1} & x_0 & \dots & x_{n-3} & x_{n-2} \end{pmatrix}$$

The Rotate operation is necessary for computations that operate on data residing in different slots of the encrypted vectors.

B. Modular Arithmetic and the Residue Number System

Scalar Modular Arithmetic: Nearly all FHE operations reduce to scalar modular additions and scalar modular multiplications. Current CPU/GPU architectures do not implement modular arithmetic directly but emulate it via multiple arithmetic instructions, which significantly increases the amount of compute required for these operations. Therefore, optimizing modular arithmetic is critical to optimizing FHE computation.

To perform modular addition over operands that are already reduced, we use the standard approach of conditional subtraction if the addition overflows the modulus. For generic modular multiplications, we use the Barrett reduction technique [1]. When computing the sum of many scalars, we avoid performing a modular reduction until the end of the summation, as long as the unreduced sum fits in a machine word. As an optimization, we use Shoup's technique [31] for constant multiplication. That is, when computing $x \cdot y \pmod{p}$ where x and p are known in advance, we can precompute a value x_s such that $\mathsf{ModMulShoup}(x, y, x_s, p) = x \cdot y \pmod{p}$ is much faster than directly computing $x \cdot y \pmod{p}$.

Residue Number System (RNS): Often the scalars in homomorphic encryption schemes are very large, on the order of thousands of bits. To compute on such large numbers, we use the residue number system (also called the Chinese remainder representation) where we represent numbers modulo $Q = \prod_{i=1}^{\ell} q_i$, where each q_i is a prime number that fits in a standard machine word (less than 64 bits), as ℓ numbers modulo each of the q_i. We call the set $\mathcal{B} := \{q_1, \dots, q_\ell\}$ an *RNS basis*. We refer to each q_i as a *limb* of Q.

This allows us to operate over values in $\mathbb{Z}_Q$ without any native support for multi-precision arithmetic. Instead, we can represent $x \in \mathbb{Z}_Q$ as a length-ℓ vector of scalars $[x]_{\mathcal{B}} = (x_1, x_2, \dots, x_\ell)$, where $x_i \equiv x \pmod{q_i}$. We refer to each x_i as a *limb* of x. To add two values $x, y \in \mathbb{Z}_Q$, we have $x_i + y_i \equiv x + y \pmod{q_i}$. Similarly, we have $x_i \cdot y_i \equiv x \cdot y \pmod{q_i}$. This allows us to compute addition and multiplication over $\mathbb{Z}_Q$ while only operating over standard machine words. The size of this representation of an element of $\mathbb{Z}_Q$ is ℓ machine words.

C. CKKS Ciphertext Structure

In this section, we give the general structure of a ciphertext in the CKKS [11] homomorphic encryption scheme. A ciphertext is a pair of polynomials each of degree $N-1$. The coefficients of these ciphertexts are elements of $\mathbb{Z}_Q$, where Q has ℓ limbs. Thus, in total, the size of a ciphertext is $2N\ell$ machine words.

In CKKS, we are able to encrypt non-integer values, including complex numbers. The ciphertexts are "packed," which means they encrypt vectors in $\mathbb{C}^n$, where $n = N/2$, in a single ciphertext. For $\mathbf{m} \in \mathbb{C}^n$, we denote its encryption as $\llbracket \mathbf{m} \rrbracket = (\mathbf{a_m}, \mathbf{b_m})$ where $\mathbf{a_m}$ and $\mathbf{b_m}$ are the two polynomials that comprise the ciphertext. We omit the subscript $\mathbf{m}$ when there is no cause for confusion.

An example of ciphertext parameters that achieve a 128-bit security level is $N = 2^{17}$ and $\ell = 35$. With an 8-byte machine word, this gives a total ciphertext size of ~ 73.4 MB. Note that in today's reticle-limited systems, the largest last-level cache size is 40 MB [28]. Consequently, we won't be able to fit even a single ciphertext in the last-level cache, which indicates the need for multiple expensive DRAM accesses when operating on ciphertexts.

Polynomial Representation: In order to enable fast polynomial multiplication, we will have all polynomials represented by default as a series of N evaluations at fixed roots of unity. This allows polynomial multiplication to occur in $O(N)$ time. We refer to this representation as the *evaluation representation*. Certain subroutines, defined in section II-D, operate over the polynomial's *coefficient representation*, which is simply a vector of its coefficients. Addition of two polynomials and multiplication of a polynomial by a scalar are $O(N)$ in both the coefficient and the evaluation representation. Moving

between representations requires a number-theoretic transform (NTT) or inverse NTT, which is the finite field version of the fast Fourier transform (FFT) and takes $O(N \log N)$ time and $O(N)$ space for a degree-$(N-1)$ polynomial.

Encoding Plaintexts: CKKS supports non-integer messages, so all encoded messages must include a scaling factor Δ. The scaling factor is usually the size of one of the limbs of the ciphertext, which is slightly less than a machine word. When multiplying messages together, this scaling factor grows as well. The scaling factor must be shrunk down in order to avoid overflowing the ciphertext coefficient modulus. We discuss how this procedure works in Section II-D.

D. Implementing the API

To implement the homomorphic API described in Table II, we need some "helper" subroutines. We first describe these subroutines and then provide the implementations of the homomorphic API using the subroutines.

Handling a Growing Scaling Factor: As mentioned in section II-C, all encoded messages in CKKS must have a scaling factor Δ. In both the PtMult and Mult implementations, the multiplication of the encoded messages results in the product having a scaling factor of Δ^2. Before these operations can complete, we must shrink the scaling factor back down to Δ (or at least a value very close to Δ). If this operation is neglected, the scaling factor will eventually grow to overflow the ciphertext modulus, resulting in decryption failure.

To shrink the scaling factor, we divide the ciphertext by Δ (or a value that is close to Δ) and round the result to the nearest integer. This operation, called ModDown, keeps the scaling factor of the ciphertext roughly the same throughout the computation.[2] For a more formal description, we refer the reader to [10]. We sometimes refer to a ModDown instruction that occurs at the end of an operation as Rescale.

Handling a Changing Decryption Key: In both the Mult and Rotate implementations, there is an intermediate ciphertext with a decryption key that differs from the decryption key of the input ciphertexts. In order to change this new decryption key back to the original decryption key, we perform a KeySwitch operation. This operation takes in a switching key $\mathsf{ksk}_{\mathbf{s}\to\mathbf{s}'}$ and a ciphertext $[\![\mathbf{m}]\!]_{\mathbf{s}}$ that is decryptable under a secret key $\mathbf{s}$. The output of the KeySwitch operation is a ciphertext $[\![\mathbf{m}]\!]_{\mathbf{s}'}$ that encrypts the same message but is decryptable under a different key $\mathbf{s}'$.

Key Switching [6]*:* Since the KeySwitch operation differs between Mult and Rotate, we do not define it separately. Instead, we go a level deeper, and define the subroutines necessary to implement KeySwitch for each of these operations. In addition to the ModDown operation, we use the ModUp operation, which allows us to add primes to our RNS basis. We follow the structure of the switching key in the work of Han and Ki [20], where the switching key, parameterized by a length dnum, is a $2 \times \mathsf{dnum}$ matrix of polynomials.

$$\mathsf{ksk} = \begin{pmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{\mathsf{dnum}} \\ \mathbf{b}_1 & \mathbf{b}_2 & \dots & \mathbf{b}_{\mathsf{dnum}} \end{pmatrix} \tag{1}$$

The KeySwitch operation requires that a polynomial be split into dnum "digits," then multiplied with the switching key. We define the function Decomp that splits a polynomial into dnum digits as well as a KSKInnerProd operation to multiply the dnum digits by the switching key.

Before proceeding further, we refer the reader to Table III where all the subroutines described above are defined in more detail. The implementation of the API functions are given in Algorithms 1, 2 and 3. We also give a batched rotation algorithm HRotate in Algorithm 4, which computes many rotations on the same ciphertext faster than applying Rotate independently several times.

Algorithm 1 $\mathsf{PtMult}([\![\mathbf{m}]\!], \mathbf{m}') = [\![\mathbf{m} \cdot \mathbf{m}']\!]$

1: $(\mathbf{a}, \mathbf{b}) := [\![\mathbf{m}]\!]$
2: $(\mathbf{u}, \mathbf{v}) := (\mathbf{a} \cdot (\Delta \cdot \mathbf{m}'), \mathbf{b} \cdot (\Delta \cdot \mathbf{m}'))$
3: **return** $(\mathsf{ModDown}_{\mathcal{B},1}(\mathbf{u}), \mathsf{ModDown}_{\mathcal{B},1}(\mathbf{v}))$ ▷ Rescale

Algorithm 2 $\mathsf{Mult}([\![\mathbf{m}_1]\!]_{\mathbf{s}}, [\![\mathbf{m}_2]\!]_{\mathbf{s}}, \mathsf{ksk}_{\mathbf{s}^2\to\mathbf{s}}) = [\![\mathbf{m}_1 \cdot \mathbf{m}_2]\!]_{\mathbf{s}}$

1: $(\mathbf{a}_1, \mathbf{b}_1) := [\![\mathbf{m}_1]\!]_{\mathbf{s}}$
2: $(\mathbf{a}_2, \mathbf{b}_2) := [\![\mathbf{m}_2]\!]_{\mathbf{s}}$
3: $(\mathbf{a}_3, \mathbf{b}_3, \mathbf{c}_3) := (\mathbf{a}_1\mathbf{a}_2, \mathbf{a}_1\mathbf{b}_2 + \mathbf{a}_2\mathbf{b}_1, \mathbf{b}_1\mathbf{b}_2)$
4: $\overrightarrow{\mathbf{a}} := \mathsf{Decomp}_\beta(\mathbf{a}_3)$
5: $\hat{\mathbf{a}}[i] := \mathsf{ModUp}(\overrightarrow{\mathbf{a}}[i])$ for $1 \le i \le \beta$.
6: $(\hat{\mathbf{u}}, \hat{\mathbf{v}}) := \mathsf{KSKInnerProd}(\mathsf{ksk}_{\mathbf{s}^2\to\mathbf{s}}, \hat{\mathbf{a}})$
7: $(\mathbf{u}, \mathbf{v}) := (\mathsf{ModDown}(\hat{\mathbf{u}}), \mathsf{ModDown}(\hat{\mathbf{v}}))$
8: $(\mathbf{a}', \mathbf{b}') := (\mathbf{b}_3 + \mathbf{u}, \mathbf{c}_3 + \mathbf{v})$
9: **return** $(\mathsf{ModDown}_{\mathcal{B},1}(\mathbf{a}'), \mathsf{ModDown}_{\mathcal{B},1}(\mathbf{b}'))$ ▷ Rescale

Algorithm 3 $\mathsf{Rotate}([\![\mathbf{m}]\!]_{\mathbf{s}}, k, \mathsf{ksk}_{\psi_k(\mathbf{s})\to\mathbf{s}}) = [\![\phi_k(\mathbf{m})]\!]_{\mathbf{s}}$

1: $(\mathbf{a}, \mathbf{b}) := [\![\mathbf{m}]\!]_{\mathbf{s}}$
2: $(\mathbf{a}_{\mathsf{rot}}, \mathbf{b}_{\mathsf{rot}}) := (\mathsf{Automorph}(\mathbf{a}, k), \mathsf{Automorph}(\mathbf{b}, k))$
3: $\overrightarrow{\mathbf{a}_{\mathsf{rot}}} := \mathsf{Decomp}_\beta(\mathbf{a}_{\mathsf{rot}})$ ▷ β digits.
4: $\hat{\mathbf{a}}[i] := \mathsf{ModUp}(\overrightarrow{\mathbf{a}_{\mathsf{rot}}}[i])$ for $1 \le i \le \beta$.
5: $(\hat{\mathbf{u}}, \hat{\mathbf{v}}) := \mathsf{KSKInnerProd}(\mathsf{ksk}_{\psi_k(\mathbf{s})\to\mathbf{s}}, \hat{\mathbf{a}})$
6: $(\mathbf{u}, \mathbf{v}) := (\mathsf{ModDown}(\hat{\mathbf{u}}), \mathsf{ModDown}(\hat{\mathbf{v}}))$
7: **return** $(\mathbf{u}, \mathbf{v} + \mathbf{b}_{\mathsf{rot}})$

Key Takeaway: The Shrinking Ciphertext Modulus A main observation coming out of our description of the homomorphic API is that the ciphertext modulus shrinks for each PtMult (algorithm 1) and Mult (algorithm 2) operation. This occurs in the ModDown operations at the end of these functions. If a ciphertext begins with L limbs, we can only compute a circuit with multiplicative depth $L-1$, since the ciphertext modulus shrinks by a number of limbs equal to the multiplicative depth of the circuit being homomorphically evaluated. This foreshadows the next section where we present an operation called *bootstrapping* [15] that increases the ciphertext modulus.

[2] A better name for this operation would be "divide and mod-down" because it reduces the scaling factor *as well as* the ciphertext modulus. In this paper, we stick to the standard ModDown terminology for consistency with the literature.

TABLE III
CKKS SUBROUTINES: *These subroutines enable the implementation of the CKKS API defined in Table II.*

Sub-routine Name	Output	Used-in	Description
$\mathsf{ModDown}_{\mathcal{B},d}([\mathbf{x}]_{\mathcal{B}})$	$[\mathbf{x}/P+e]_{\mathcal{B}'}$	PtMult Mult Rotate	This function takes in a polynomial $\mathbf{x}$ in the coefficient representation, where each coefficient is modulo $Q := \prod_{q\in\mathcal{B}} q$ and represented in the RNS basis $\mathcal{B} = \{q_1, \ldots, q_\ell\}$. Assume that $d < \ell$ and let $P := \prod_{i=\ell-d+1}^{\ell} q_i$ be the product of the last d limbs of $\mathcal{B}$. Let $\mathcal{B}' = \{q_1, \ldots, q_{\ell-d}\}$, and note that $Q/P = \prod_{q\in\mathcal{B}'} q$. The output of this function is a polynomial $[\mathbf{y}]_{\mathcal{B}'}$ where each coefficient of $\mathbf{y}$ equals the corresponding coefficient of $\mathbf{x}$ divided by P plus some small rounding error.
$\mathsf{ModUp}_{\mathcal{B},\mathcal{B}'}([\mathbf{x}]_{\mathcal{B}})$	$[\mathbf{x}]_{\mathcal{B}'}$	Mult Rotate	Takes a polynomial $\mathbf{x}$ where each coefficient is in the basis $\mathcal{B}$ and outputs the representation of $\mathbf{x}$ where each coefficient is in the basis $\mathcal{B}'$. $\mathcal{B}$ could be a subset or superset of $\mathcal{B}'$, or they could be unrelated. Note that this operation must also be performed in the coefficient representation.
$\mathsf{Decomp}_{\beta}(\mathbf{x})$	$\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\beta)}\}$	Mult Rotate	Takes in a polynomial $\mathbf{x}$ and a parameter dnum and splits $\mathbf{x}$ into dnum digits. If $\mathbf{x}$ has L limbs, each digit of $\mathbf{x}$ has roughly $\alpha := \lceil (L+1)/\mathsf{dnum} \rceil$ limbs.
$\mathsf{KSKInnerProd}(\mathsf{ksk}, \overrightarrow{\mathbf{x}})$	$(\mathbf{a}, \mathbf{b})$	Mult Rotate	Takes in a key-switching key ksk with the structure of eq. (1) and a vector of polynomials $\overrightarrow{\mathbf{x}}$ of length dnum. Let ksk_1 be the first row of ksk and let ksk_2 be the second row of ksk. The output of this operation is two polynomials $\mathbf{a} := \langle \mathsf{ksk}_1, \overrightarrow{\mathbf{x}} \rangle$ and $\mathbf{b} := \langle \mathsf{ksk}_2, \overrightarrow{\mathbf{x}} \rangle$.
$\mathsf{Automorph}(\mathbf{x}, k)$	$\psi_k(\mathbf{x})$	Rotate	Takes a vector $\mathbf{x}$ with N elements and an integer k and outputs a permutation $\psi_k(\cdot)$ of the elements. This permutation is an automorphism which is *not* simply a rotation; intuitively, the permutation ψ_k of an encoded message will result in the decoded value being permuted by the natural rotation ϕ_k.

Algorithm 4

$\mathsf{HRotate}(\llbracket \mathbf{m} \rrbracket_{\mathbf{s}}, \{k_i, \mathsf{ksk}_{\psi_{k_i}(\mathbf{s})\to\mathbf{s}}\}_{i=1}^{r}) = \{\llbracket \phi_{k_i}(\mathbf{m}) \rrbracket_{\mathbf{s}}\}_{i=1}^{r}$

$(\mathbf{a}, \mathbf{b}) := \llbracket \mathbf{m} \rrbracket_{\mathbf{s}}$
$\overrightarrow{\mathbf{a}} := \mathsf{Decomp}_{\beta}(\mathbf{a})$ ▷ β digits.
$\hat{\mathbf{a}}[j] := \mathsf{ModUp}(\overrightarrow{\mathbf{a}}[j])$ for $1 \le j \le \beta$.
for i from 1 to r **do**
  $\hat{\mathbf{a}}_{\mathsf{rot}} := \mathsf{Automorph}(\hat{\mathbf{a}}, k_i)$ for $1 \le j \le \beta$
  $(\hat{\mathbf{u}}, \hat{\mathbf{v}}) := \mathsf{KSKInnerProd}(\mathsf{ksk}_{\psi_{k_i}(\mathbf{s})\to\mathbf{s}}, \hat{\mathbf{a}}_{\mathsf{rot}})$
  $(\mathbf{u}, \mathbf{v}) := (\mathsf{ModDown}(\hat{\mathbf{u}}), \mathsf{ModDown}(\hat{\mathbf{v}}))$
  $\mathbf{b}_{\mathsf{rot}} := \mathsf{Automorph}(\mathbf{b}, k_i)$
  $\llbracket \phi_{k_i}(\mathbf{m}) \rrbracket_{\mathbf{s}} := (\mathbf{u}, \mathbf{v} + \mathbf{b}_{\mathsf{rot}})$
end for
return $\{\llbracket \phi_{k_i}(\mathbf{m}) \rrbracket_{\mathbf{s}}\}_{i=1}^{r}$

E. Concrete Costs

We present the hardware cost associated with various functions and subroutines in the FHE API in Table IV and Table V, and discuss the content of the tables briefly. To generate these performance numbers, we implement an architectural modeling tool that can perform an in-depth analysis given the number of functional units, cache size, and the memory subsystem parameters. In addition, our tool allows us to tune nearly all parameters of the algorithm, including N, dnum, and the maximum ciphertext modulus for a given security level.

Key Takeaway: Low Arithmetic Intensity. The key takeaway from the tables, in particular Table V, is that *the arithmetic intensity*, defined as the number of operations per byte transferred from DRAM, of all of the functions in the CKKS API is less than $<$ 1 Op/byte. This means that when the ciphertexts do not fit in memory, *any natural application (e.g. logistic regression training, neural network evaluation, bootstrapping, etc.) built using these functions will have performance bounded by the memory bandwidth and not the computation speed.*

Since our ciphertexts will remain too large to fit in the chip cache, much of this work will focus on improving the arithmetic intensity of CKKS bootstrapping. This translates to progressing further in the bootstrapping algorithm per memory transfer, which, in turn, translates to a faster bootstrapping implementation.

III. FULLY HOMOMORPHIC ENCRYPTION: APPLICATIONS

In this section, we describe how the FHE API from Section II can be leveraged to develop applications. As discussed in Section II-D, a CKKS ciphertext can only support computation up to a fixed multiplicative depth due to the shrinking ciphertext modulus. Once this depth is reached, a *bootstrapping* operation must be performed to grow the ciphertext modulus, which allows for computation to continue.

Many applications of interest have a deep circuit that requires bootstrapping multiple times: in general, machine learning training algorithms are good examples where deeper circuits for the training computation often lead to greater accuracy of the resulting model. In this section, we use *logistic regression training* over encrypted data as a running example to explain the process of FHE-based machine learning training. Logistic regression training contains both linear (e.g. inner-products) and non-linear (e.g. sigmoid) operations. The CKKS scheme naturally supports linear operations, while for non-linear operations we need to use a polynomial approximation (as in [19], [23]). The greater the degree of the polynomial, the greater the accuracy of the approximation, which further drives an increase in the circuit depth, in turn requiring bootstrapping.

TABLE IV

HARDWARE COST OF AUXILIARY SUBROUTINES: *These benchmarks were taken for* $\log(N) = 17$, $\ell = 35$, $\mathsf{dnum} = 3$. *The* ***Total Operations*** *column counts the number of modular additions and multiplications in the operations, (note that this count for the* Automorph *function is zero).* ***GOP*** *stands for Giga operations. The* ***Total DRAM Transfers*** *is the sum of* ***DRAM Limb Reads***, ***DRAM Limb Writes***, *and* ***DRAM Key Reads***, *the last of which counts the reads specifically for the switching keys. The* KSKInnerProd *operation has no limb writes because the limbs are immediately used in the next operation, the* ModDown. *The write is counted in the* ModDown *when the limbs are written out in to be read back in* slot-wise *format, as discussed in Section IV-A. The* ***Arithmetic Intensity*** *column defines the number of operations per byte transferred from DRAM.*

Sub-routine Name	Total Operations (in GOP)	Total Mults (in GOP)	Total DRAM Transfers (in GB)	DRAM Limb Reads (in GB)	DRAM Limb Writes (in GB)	DRAM Key Reads (in GB)	Arithmetic Intensity (in Op/byte)
ModDown	0.3000	0.1288	0.1877	0.1007	0.0870	0	**1.59**
ModUp	0.2847	0.1211	0.1510	0.0629	0.0881	0	**1.88**
Decomp	0.0092	0.0092	0.0734	0.0367	0.0367	0	**0.12**
KSKInnerProd	0.0629	0.0378	0.4530	0.1510	0	0.3020	**0.13**
Automorph	0	0	0.1468	0.0734	0.0734	0	**0**

TABLE V

HARDWARE COST OF FHE APIS: *These benchmarks were taken for* $\log(N) = 17$, $\ell = 35$, $\mathsf{dnum} = 3$. *The number of rotations computed in the* HRotate *benchmark is* 8. *See the caption of Table IV for a description of the columns.*

Operation Name	Total Operations (in GOP)	Total Mults (in GOP)	Total DRAM Transfers(in GB)	DRAM Limb Reads (in GB)	DRAM Limb Writes (in GB)	DRAM Key Reads (in GB)	Arithmetic Intensity (in Op/byte)
PtAdd	0.00459	0	0.1101	0.0734	0.0367	0	**0.04**
Add	0.0092	0	0.2202	0.1468	0.0734	0	**0.04**
PtMult	0.2747	0.1098	0.3282	0.1835	0.1447	0	**0.84**
Mult	1.8333	0.7826	1.9293	0.9070	0.7203	0.3020	**0.95**
Rotate	1.5310	0.6682	1.5645	0.6501	0.6124	0.3020	**0.98**
Conjugate	1.5310	0.6682	1.5645	0.6501	0.6124	0.3020	**0.98**
HRotate	6.2039	2.7363	8.1411	3.2632	2.4621	2.4159	**0.76**

For our running example, we use the logistic regression training application given in Han, Song, Cheon and Park [19] and depicted in fig. 2. The training process is an iterative process that repeatedly computes an inner product followed by a sigmoid function on a training data set and the model weights. The logistic regression update equation is as follows.

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{\mathsf{lr}}{n} \sum_{i=1}^{n} \sigma\left(\mathbf{z}_i^T \cdot \mathbf{w}\right) \cdot \mathbf{z}_i \tag{2}$$

The vector $\mathbf{w}$ is the weight vector, the values n and lr are scalars, and $\mathbf{z}_i$ represents the i^{th} vector of the training data set. The σ function is the sigmoid function.

To implement this iterative update, we split the update into two phases: a linear phase that contains the inner product[3] and a non-linear phase that contains the sigmoid function. We implement these phases separately with common building blocks shown in Table VI. The linear phase can be implemented with an InnerProduct routine that computes the inner product of two encrypted vectors. The non-linear phase is approximated with a polynomial, and the homomorphic evaluation of this polynomial can be implemented with PolyEval. The scalar products and summation can be implemented with the PtMult, Mult, and Add functions. After some number of iterations, the encrypted weights are passed through the Bootstrap routine. The exact placement of the Bootstrap operation in a circuit is application-dependent. In our running example, bootstrapping needs to be done every three iterations (see Figure 2).

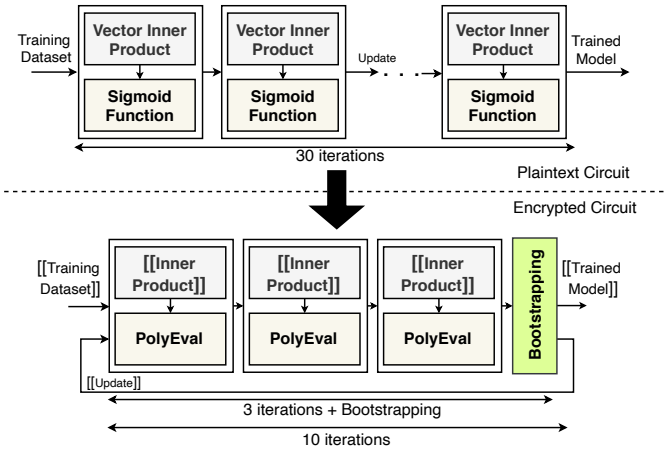

Fig. 2. Logistic regression training on encrypted data.

[3]In the real implementation of Equation (2), these inner products are batched into a matrix-vector product. We use the same algorithm as [19].

TABLE VI
HOMOMORPHIC ENCRYPTION APPLICATION BUILDING BLOCKS: *These building blocks are implemented using the API from Table II.*

Name	Output	Description
$\mathsf{InnerProduct}(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket)$	$\llbracket \langle \mathbf{x}, \mathbf{y} \rangle \rrbracket$	Computes the inner product of two encrypted vectors. This computation is the specific encrypted inner product algorithm from Han et al. [19].
$\mathsf{PolyEval}(\llbracket \mathbf{x} \rrbracket, p(\cdot))$	$\llbracket p(\mathbf{x}) \rrbracket$	This operation takes an encrypted vector $\mathbf{x}$ and a (univariate) polynomial p as input. The result is an encryption of the evaluation of p at $\mathbf{x}$, where each entry of $p(\mathbf{x})$ is the evaluation of p on the corresponding entry of $\mathbf{x}$.
$\mathsf{PtMatVecMult}(\mathbf{M}, \llbracket \mathbf{x} \rrbracket)$	$\llbracket \mathbf{M}\mathbf{x} \rrbracket$	This operation takes a plaintext matrix $\mathbf{M}$ and multiplies it by an encrypted vector $\mathbf{x}$. The result is an encryption of the vector $\mathbf{M}\mathbf{x}$. This is a major subroutine in Bootstrap.
$\mathsf{Bootstrap}(\llbracket \mathbf{x} \rrbracket)$	$\llbracket \mathbf{x} \rrbracket$	This operation takes in an encryption of a vector $\mathbf{x}$ and outputs an encryption of the same vector $\mathbf{x}$. This operation is necessary to be able to compute indefinitely on encrypted data. Far from being a null operation, this is nearly always the bottleneck operation when computing over encrypted data.

A. Bootstrapping

As discussed in Section II-D, the ciphertext modulus of CKKS shrinks with each multiplication. In order to compute indefinitely on a CKKS ciphertext, we must grow the ciphertext modulus without also growing the noise. This is not as simple as performing a ModUp function. The CKKS bootstrapping procedure [9] begins with this ModUp operation, which gives the new plaintext as $\Delta \cdot \mathbf{m} + \mathbf{k}q$ where q is the modulus for the input ciphertext and $\mathbf{k}$ is some polynomial with small integer coefficients. The primary goal of the bootstrapping operation is to homomorphically evaluate the modular reduction operation modulo q on this plaintext, returning the plaintext back to $\Delta \cdot \mathbf{m}$.

The CKKS bootstrapping algorithm follows a general structure that has remained relatively static in the literature [3], [8], [9], [18], [20] over the past few years. This structure has three main components: a linear operation, an approximation of the modular reduction function followed by another linear operation. The linear operations in bootstrapping require homomorphically evaluating the DFT on the encrypted data so that we perform modulus reduction on the *coefficient representation* of plaintext, rather than the *evaluation (or slot) representation*. The first of these DFT operations is called CoeffToSlot and the second is called SlotToCoeff. In between these two DFT operations is an approximation of the modular reduction function that consists of a polynomial evaluation followed by an exponentiation. For further details on polynomial evaluation and the exponentiation, we refer the readers to [3], [20].

To homomorphically evaluate the DFT, we use the observation that the DFT matrix can be factored into submatrices of smaller dimension. This turns the homomorphic DFT into a series of PtMatVecMult operations. However, there is a trade-off between the number of PtMatVecMult operations that must be computed and the size of the matrices in each PtMatVecMult instance. Each PtMatVecMult has a multiplicative depth of 1. The total dimension of the DFT is $n = N/2 = 2^{16}$ for our parameters. Options to evaluate this DFT include evaluating a single PtMatVecMult with an $n \times n$ input, which would require a very large number of rotations, or evaluating 16 PtMatVecMult instances in sequence with only two rotations per instance. The former corresponds to treating DFT as a matrix-vector multiplication without using the structure of the DFT matrix while the latter corresponds to running the $O(N \log N)$ algorithm for DFT.

We can interpolate between these two extremes to find the optimal depth vs. computation trade-off. Each sub-matrix in the factorization of the DFT matrix has a *radix* corresponding to the number of non-zero diagonals. The smaller the radix, fewer the rotations that must be computed during the PtMatVecMult instance. The rule is that the product of the radices of the PtMatVecMult iterations (in the DFT algorithm) must equal n. For example, for our parameter of $n = 2^{16}$, this gives the options of three PtMatVecMult iterations with radices of 2^5, 2^5, and 2^6, or five PtMatVecMult iterations with four iterations having a radix of 2^3 matrix and one iteration with a radix of 2^4. We call the number of iterations as fftIter. The homomorphic inverse DFT is computed in an analogous way.

Our approximation of the modular reduction function follows the literature, where we represent the modular reduction function modulo q with a sine function with period q, then approximate this sine function with a polynomial. We represent this polynomial with $\mathsf{sine}(\cdot)$, and we use the Chebyshev polynomial construction used in Han and Ki [20]. The degree of this polynomial is 63. We give a high-level pseudocode for the bootstrapping algorithm in Algorithm 5.

Algorithm 5 $\mathsf{Bootstrap}(\llbracket \mathbf{x} \rrbracket) = \llbracket \mathbf{x} \rrbracket$

$(\mathbf{a}, \mathbf{b}) := \llbracket \mathbf{x} \rrbracket$
$\llbracket \mathbf{t} \rrbracket := \mathsf{ModUp}(\mathbf{a}, \mathbf{b})$
for i from 1 to fftIter **do** ▷ CoeffToSlot phase.
    $\llbracket \mathbf{t} \rrbracket \leftarrow \mathsf{PtMatVecMult}(\mathbf{M}_i, \llbracket \mathbf{t} \rrbracket)$
end for
$\llbracket \mathbf{t} \rrbracket \leftarrow \mathsf{PolyEval}(\llbracket \mathbf{t} \rrbracket, \mathsf{sine}(\cdot))$
for i from 1 to fftIter **do** ▷ SlotToCoeff phase.
    $\llbracket \mathbf{t} \rrbracket \leftarrow \mathsf{PtMatVecMult}(\mathbf{M}_i, \llbracket \mathbf{t} \rrbracket)$
end for
return $\llbracket \mathbf{t} \rrbracket$

B. Concrete Costs

We give the concrete costs of the logistic regression and Bootstrap subroutines in Table VII and Table VIII respectively. As the table shows, the arithmetic intensity of the sub-routines is less than 1 Op/byte. As discussed in Section II-E, since our ciphertexts do not fit in cache, this means that the performance of all sub-routines is bounded by the main memory bandwidth. In Table VII, we give benchmarks for the logistic regression implementation based on our architecture modeling discussed in Section II-E. The parameters we use are from the work of Jung et al. [22], and these parameters were chosen to optimize their secure logistic regression application that leverages a GPU implementation of CKKS bootstrapping. We refer to the original work of Han et al. [19] for the full algorithm benchmarked in Table VII. We note that the logistic regression iteration is the "most expensive" of the three iterations that follow a Bootstrap, since the ciphertexts in this iteration are the largest. As the ciphertext shrinks due to the reduced ciphertext modulus, the computation becomes cheaper. However, the arithmetic intensity remains essentially the same, and the performance of each phase of the algorithm is bottle-necked by the memory bandwidth. Overall, roughly half of the total runtime is spent in bootstrapping.

Key Takeaway: Bootstrapping is often the bottle-neck operation in HE applications, especially applications that implement a deep circuit. For example, even when using a heavily-optimized GPU implementation of bootstrapping, nearly half of the time in HE logistic regression training is spent on bootstrapping [22] (table VII). This motivates the need to optimize the Bootstrap operation to efficiently support deep circuits. Furthermore, the building blocks of bootstrapping are the same as many other HE applications; there are essentially no subroutines that are unique to bootstrapping. Many of the optimizations we give in Section IV and Section V apply more generally to HE applications.

IV. CKKS Bootstrapping: Caching Optimizations

In this section and section V, we present our optimizations to the CKKS bootstrapping algorithm. These optimizations fall into two categories: those that rely on hardware assumptions and those that do not. Our first class of optimizations assume a lower bound on the amount of available cache size relative to the size of the ciphertext limbs while second class of optimizations are more general as they reduce the total operation count of CKKS bootstrapping as well as the total number of DRAM reads, regardless of the hardware architecture.

This section focuses on the first set of optimizations. These caching optimizations do not affect the operation count of Bootstrap; instead, they reduce DRAM reads and writes to reduce the overall memory bandwidth requirement. Our optimizations demonstrate how best to utilize caches of various sizes relative to the size of the ciphertext limbs. We quantify the improvements of these optimizations in Section IV-E, where we give benchmarks for progressively larger cache sizes. Our baseline benchmark is the parameter set from the GPU bootstrapping implementation of Jung et al. [22]. The parameters are given in Table XI.

A. Caching $O(1)$ Limbs

This is the first in a series of optimizations that details how best to utilize a cache for various cache sizes relative to the ciphertext limbs. We begin by discussing how to utilize a cache that can store a constant number of limbs. Intuitively, this optimization computes as much as possible on a single limb before writing it back to the main memory. This often involves performing the operations of several higher-level functions on a single limb before beginning the same sequence of operations on the next limb. This technique was referred to by Jung et al. [22] as a "fusing" of operations, and we include all fusing operations listed in their work in our bootstrapping algorithm. In addition, we provide a novel data mapping technique to handle caching data with different *data access patterns*.

Data Access Patterns: Having a small-cache (about 1-3 MB) in any FHE compute system has a caveat that must be carefully addressed. Some operations in CKKS such as NTT and iNTT operate on data within the slots of the same limb, independent of the other limbs in the ciphertext. On the other hand, RNS basis change operations in ModUp and ModDown require interaction between a certain number of slots across various limbs. This requires having a few slots from multiple limbs in on-chip memory to reduce the number of accesses to main memory for a single operation. To account for this, we define two different types of data access patterns. For the functions where limbs can be operated upon independently, we define the data access pattern as *limb-wise* and for the functions where slots can be operated upon independently, we define the data access pattern as *slot-wise*. A summary of this is given in Table IX. We have also illustrated this by giving a high-level pseudo code of ModUp in Algorithm 6. From this algorithm, it is evident that the ModUp operation includes both *limb-wise* and *slot-wise* operations, requiring a memory mapping that is efficient for both access patterns. A naive memory mapping would result in low throughput for at least one of these access patterns. Therefore, we describe a novel memory mapping approach to handle these two access patterns.

Algorithm 6 $\mathsf{ModUp}_{\mathcal{B},\mathcal{B}\cup\mathcal{B}'}([\mathbf{x}]_{\mathcal{B}}) = [\mathbf{x}]_{\mathcal{B}\cup\mathcal{B}'}$

1: **for** i from 1 to $|\mathcal{B}|$ **do**
2: $\quad[\mathbf{x}]_i \leftarrow \mathsf{iNTT}([\mathbf{x}]_i)$ ▷ *limb-wise*
3: **end for**
4: **for** j from 1 to $|\mathcal{B}'|$ **do** ▷ Basis conversion.
5: $\quad[\mathbf{x}]_j \leftarrow \mathsf{NewLimb}_j([\mathbf{x}]_1, \ldots, [\mathbf{x}]_{|\mathcal{B}|})$ ▷ *slot-wise*
6: **end for**
7: **for** j from 1 to $|\mathcal{B}'|$ **do**
8: $\quad[\mathbf{x}]_j \leftarrow \mathsf{NTT}([\mathbf{x}]_j)$ ▷ *limb-wise*
9: **end for**
10: **return** $[\mathbf{x}]_{\mathcal{B}\cup\mathcal{B}'}$

TABLE VII

HARDWARE COST OF FHE APPLICATIONS: *These benchmarks were taken for* $\log(N) = 17$, $\ell = 35$, $\mathsf{dnum} = 3$. *See the caption of Table IV for a description of the columns. The number of features in the logistic regression is* $d = 256$. *The* InnerProduct *and* PolyEval *benchmarks are for the first iterations after a* Bootstrap. *The "Full LR Iteration" row is the first iteration of the training algorithm after a* Bootstrap. *The degree of the polynomial evaluated in* PolyEval *is* 3.

Sub-routine Name	Total Operations (in GOP)	Total Mults (in GOP)	Total DRAM Transfers(in GB)	DRAM Limb Reads (in GB)	DRAM Limb Writes (in GB)	DRAM Key Reads (in GB)	Arithmetic Intensity (in Op/byte)
InnerProduct	7.8558	3.3806	16.5413	7.2918	4.8455	4.4040	**0.47**
PolyEval	2.9314	1.2188	3.5484	1.7144	1.3118	0.5222	**0.83**
Full LR Iteration	92.4225	39.6322	195.052	86.7822	56.1387	52.131	**0.47**
Bootstrap	149.546	64.6859	207.982	109.91	65.2434	32.8288	**0.72**

TABLE VIII

HARDWARE COST OF BOOTSTRAPPING: *These benchmarks were taken for* $\log(N) = 17$, $\ell = 35$, $\mathsf{dnum} = 3$. *See the caption of Table IV for a description of the columns. These benchmarks represent the performance of the main sub-routines of bootstrapping. The degree of the polynomial in* PolyEval *is* 63.

Sub-routine Name	Total Operations (in GOP)	Total Mults (in GOP)	Total DRAM Transfers(in GB)	DRAM Limb Reads (in GB)	DRAM Limb Writes (in GB)	DRAM Key Reads (in GB)	Arithmetic Intensity (in Op/byte)
CoeffToSlot	58.486	25.8087	86.7424	46.8651	25.2875	14.5899	**0.67**
PolyEval	57.834	24.4496	65.643	33.0406	23.744	8.8584	**0.88**
SlotToCoeff	33.2265	14.4275	55.5001	30.004	16.1156	9.3806	**0.59**

TABLE IX

DATA DEPENDENCIES AND ACCESS PATTERN IN DIFFERENT FUNCTIONS

The NewLimb *function is used in both* ModUp *and* ModDown.

Operation	Interaction	Independent	Access pattern
NTT, iNTT	Intra-limb	Inter-limb	*limb-wise*
NewLimb	Inter-limb	Intra-limb	*slot-wise*

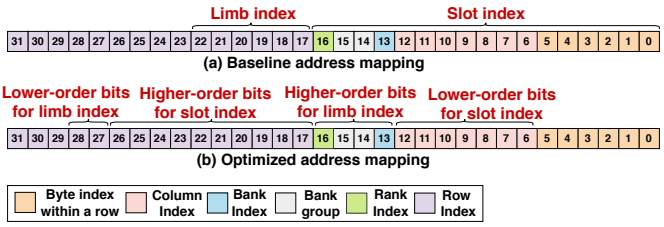

Fig. 3. DDR4 physical address mapping. *Baseline address mapping indexes all the slots* (2^{17}) *using the lower-order* 17 *bits and all limbs using the immediate next* 6 *bits. In optimized physical address mapping, slots are indexed using* 7 *bits from the column and* 10 *bits from the row, accounting for* 2^{17} *slots. The limbs are indexed using the* 4 *bits that index bank group, bank, and rank and* 2 *bits from the row index.*

Physical Address Mapping: When we re-purpose the last level cache to support both *limb-wise* and *slot-wise* access patterns, we observe that the physical address mapping of the data in main memory has a substantial impact on the time it takes to transfer data from the main memory. Figure 3 (a) shows a natural physical address mapping for a ciphertext. We call this the baseline address mapping. Through simulations in DRAMSim3 [25] we notice that for this baseline address mapping, the *limb-wise* accesses require 2.3 ms to read 35 limbs worth of data. However, we notice that the *slot-wise* access pattern requires 9.2 ms to transfer the same amount of data. This is significantly lower as with the peak theoretical bandwidth (i.e., 19.2 GB/s) for DDR4 the time required to read 35 limbs worth of data is 1.9 ms.

There are two reasons for this performance hit while doing *slot-wise* accesses. With $L = 35$, the size of the ciphertext is 36.7 MB whose limbs can be stored sequentially within a memory bank in one of the bank groups in main memory. Each limb of the ciphertext spans across multiple rows of the memory bank. Typically, each bank in main memory has a currently activated row whose contents are copied into a row buffer (acting as a cache) that can be accessed quickly. However, with *slot-wise* access pattern, every access is trying to read a different row, which takes longer because each row must be activated first. Moreover, with *slot-wise* accesses, we are unable to exploit the fact that bank accesses to different banks' groups require less time delay between accesses in comparison to the bank accesses within the same bank's group. Instead, we keep accessing data from the memory bank within the same bank group.

We propose an optimized physical address mapping as shown in Figure 3 (b). As shown in Table X, with this proposed address mapping, we observe that the *limb-wise* access requires a data transfer time of 2.5 ms, which is about 8% reduction in the times observed for the baseline *limb-wise* accesses. However, compared to the baseline *slot-wise* access pattern, our optimized *slot-wise* access pattern sees an increase in data transfer time by 76%, which is a significant improvement. We observe that the total data transfer time for baseline address mapping is about 2.4× higher than our optimized mapping. Our optimized physical

address mapping ensures that when performing *limb-wise* and *slot-wise* reads/writes, we exploit bank-level parallelism, and we focus on reducing the bank thrashing by not changing a bank's currently activated row frequently. Note that for a different DRAM type such as HBM2 or GDDR5/6, similar physical address mappings can be done to optimize the main memory bandwidth utilization.

TABLE X
DRAM TRANSFER TIMES WITH BASELINE AND OPTIMIZED MAPPING FOR DIFFERENT ACCESS PATTERNS: *Transfer times are computed for reading $L = 35$ limbs worth of data, which is 36.7 MB for our baseline parameter set.*

Mapping	*limb-wise* access	*slot-wise* access	Total Time
Baseline	2.3 ms	9.2 ms	11.5 ms
Optimized	2.5 ms	2.2 ms	4.7 ms

B. β-Limb Caching

The next optimization considers a cache size that is $O(\beta)$. Recall that β is the number of digits generated from a polynomial key switching. We refer to Han and Ki [20] for more details. For our parameters where $\beta \leq \mathsf{dnum} = 3$, this amounts to about 6 MB of cache. We need space for 3 limbs at all-times and 3 limbs worth of space to store intermediate results and other required constants. With this optimization, we can greatly reduce the number of accesses to main memory during key-switching.

Consider the HRotate function in Algorithm 4. There are β digits that are produced as the output of the ModUp operations. Naively, for each rotation we would read the limbs for each of the β digits, rotate them, then compute the inner product with the key-switching key. Since now we have space in the cache for β digits, we can instead pull in a single limb from each of the β outputs of ModUp, then compute the rotation and the inner product with the switching key limbs all at once. This allows us to read in the outputs of the ModUp function only once, regardless of the number of rotations computed.

C. α-Limb Caching

For this optimization, we assume that we have a relatively large LLC that can hold $O(\alpha)$ limbs. Recall that α is the number of limbs in a single digit after output by the Decomp function for key switching. We refer to Han and Ki [20] for more details. In practice, this optimization requires only slightly more than 2α limbs, using about 27 MB (2α + 3 MB) for $\alpha = \lceil L + 1/\mathsf{dnum} \rceil = 12$ as $L = 35$ and $\mathsf{dnum} = 3$.

Under this assumption, we observe a dramatic decrease in the number of accesses to the main memory. This is because all of the *slot-wise* basis conversion operations in ModUp (line 5 in algorithm 6) and ModDown operate over α limbs. If we can fit these α limbs in cache, then we can generate new limbs in their entirety within the cache. With each new limb in cache, we can perform the NTT on the limb, which completes the basis change operation, and write this limb out to memory. This lets us generate all new limbs in evaluation format without having to write them out in *slot-wise* format and then reading them back in *limb-wise* format.

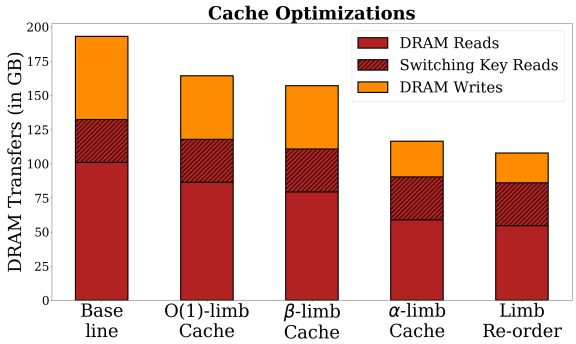

Fig. 4. DRAM transfers with various memory optimizations. As the cache size grows from left-to-right more optimizations become available. The impact is assessed cumulatively i.e. each successive optimization builds on top of the earlier ones. The order of the optimizations correspond to the order of the sections in Section IV.

Accumulator Caching: We briefly mention an optimization that is easily enabled by a large cache but is also available with smaller caches ($O(\beta)$ or even smaller). This optimization improves the memory bandwidth of the baby-step giant-step polynomial evaluation from Han and Ki [20]. A straightforward optimization is to cache the leaves (the baby-step) polynomials and reuse them to compute all of the giant-step limbs. However, if there is not enough space for the baby-step polynomials, we can still save DRAM reads by caching the partial sums of the giant step limb. When we read in a baby-step limb, we add this limb to all cached accumulators.

D. Re-Ordering Limb Computations

For the ModDown operation, the limbs that are being reduced need additional operations to be performed on them. The ModDown operations in key switching and bootstrapping drop α limbs. In this re-ordering optimization, we propose computing these α limbs first so that the additional operations can be performed immediately. This optimization is especially potent when these α limbs can be cached, since then there is no need to write out these limbs as they are being computed. Once we have the α limbs, we can begin the ModDown operation by computing the output of the basis conversion. Then, for each subsequent limb that is computed, this limb can be immediately combined with the basis conversion output, saving DRAM transfers.

E. Key Takeaway

The benefits of the optimizations in this section are presented in Figure 4. As the figure shows, growing the cache size reduces the DRAM transfers of the bootstrapping algorithm by employing the optimizations described in this section. Note that the number of compute operations in the bootstrapping algorithm remains fixed for all these benchmarks.

V. CKKS BOOTSTRAPPING: ALGORITHMIC OPTIMIZATIONS

In this section, we present our algorithmic optimizations to the CKKS bootstrapping algorithm. These optimizations

represent strict improvements to the CKKS bootstrapping algorithm and they do not depend on the cache size. However, as an added benefit of reducing the compute operation count, they also reduce the memory bandwidth, as displayed in Figure 5.

Our baseline for demonstrating the improvements of these optimizations is the memory-optimized algorithm from Section IV. Therefore, the left-most baseline bar in Figure 5 contains all of the memory optimizations described in Section IV. For the algorithm that includes all of our optimizations, we performed a parameter search to optimize the bootstrapping throughput for a 128-bit security level. We discuss our parameter search method further in Section VI. These parameters are given in Table XI, and all benchmarks in Figure 5 were taken using these same parameters.

A. *Combining* ModDown *and* Rescale *in* Mult

This optimization merges the two ModDown operations in lines 7 and 9 in Algorithm 2. To merge these ModDown operations, we must lift the addition step in line 8 above the first ModDown. We achieve this by modifying the double-hoisting method from Bossuat et al. [3], multiplying the two polynomials by P to efficiently lift the two polynomial to the modulus PQ. We denote the operation that multiplies by P modulo Q and then interprets the result modulo PQ as PModUp. By applying the PModUp function, we can move the addition above the first ModDown, making the two ModDown operations adjacent, which allows them to be combined. This new Mult algorithm, denoted as NewMult, is given in Algorithm 7, and the lines in blue denote the differences from Algorithm 2.

Faster Encrypted Inner Product: As a direct result of this optimization, we obtain a faster encrypted inner product. Consider the operation that computes $[\![\mathbf{z}]\!] = \sum_i \mathsf{Mult}([\![\mathbf{x}_i]\!], [\![\mathbf{y}_i]\!])$ where $\overrightarrow{[\![\mathbf{x}]\!]}$ and $\overrightarrow{[\![\mathbf{y}]\!]}$ are vectors of ciphertexts. Using the NewMult operation, we need to compute only one ModDown operation over the entire sum. This is because we can merge the additions in line 8 to sum all of the polynomials before any ModDown is computed.

Algorithm 7 $\mathsf{NewMult}([\![\mathbf{m}_1]\!]_{\mathbf{s}}, [\![\mathbf{m}_2]\!]_{\mathbf{s}}, \mathsf{ksk}) = [\![\mathbf{m}_1 \cdot \mathbf{m}_2]\!]_{\mathbf{s}}$

1: $(\mathbf{a}_1, \mathbf{b}_1) := [\![\mathbf{m}_1]\!]_{\mathbf{s}}$
2: $(\mathbf{a}_2, \mathbf{b}_2) := [\![\mathbf{m}_2]\!]_{\mathbf{s}}$
3: $(\mathbf{a}_3, \mathbf{b}_3, \mathbf{c}_3) := (\mathbf{a}_1\mathbf{a}_2, \mathbf{a}_1\mathbf{b}_2 + \mathbf{a}_2\mathbf{b}_1, \mathbf{b}_1\mathbf{b}_2)$
4: $\overrightarrow{\mathbf{a}} := \mathsf{Decomp}_{\mathsf{dnum}}(\mathbf{a}_3)$
5: $\hat{\mathbf{a}}_i := \mathsf{ModUp}(\overrightarrow{\mathbf{a}}[i])$ for $1 \le i \le \mathsf{dnum}$.
6: $(\hat{\mathbf{u}}, \hat{\mathbf{v}}) := \mathsf{KSKInnerProd}(\mathsf{ksk}_{\mathbf{s}^2 \to \mathbf{s}}, \hat{\mathbf{a}})$
7: $(\hat{\mathbf{b}}_3, \hat{\mathbf{c}}_3) := (\mathsf{PModUp}(\mathbf{b}_3), \mathsf{PModUp}(\mathbf{c}_3))$
8: **return** $(\mathsf{ModDown}(\hat{\mathbf{u}} + \hat{\mathbf{b}}_3), \mathsf{ModDown}(\hat{\mathbf{v}} + \hat{\mathbf{c}}_3))$

B. *Hoisting the* ModDown *in* PtMatVecMult

In section II-D, we discussed how r rotations on the same ciphertext can be computed more efficiently than simply applying the Rotate function r times. This function HRotate described in Algorithm 4 achieves an improved performance by identifying an expensive common subroutine in all of the Rotate operations: the ModUp routine.

Bossuat et al. [3] present an optimization that hoists the second *slot-wise* operation in the function: the ModDown routine. However, their technique is similar to the one in NewMult, where the message polynomial is lifted to the raised modulus via the inexpensive PModUp procedure. They call this optimization "double-hoisting." Our ModDown hoisting optimization is used in the context of a baby-step giant-step (BSGS) algorithm that implements PtMatVecMult. The trade-off in this algorithm is that a larger baby-step and a smaller giant step means more DRAM reads for the switching keys, while a smaller baby-step and a larger giant step means more DRAM reads for the ciphertexts, since the baby-step ciphertexts must be read in for each giant-step.

In Section V-C, we give a simple optimization to compress the size of the keys by a factor of 2. Using our architecture modeling tool, we determine that this optimization shifts the balance between the baby-step size and the giant-step size so significantly that the optimal number of giant steps is 1. This essentially collapses the baby-step giant-step structure into just a single step that computes all r iterations at once. Therefore, by removing the giant steps in the BSGS algorithm, the PtMatVecMult collapses into a single instance of HRotate that includes the PModUp double-hoisting optimization, which allows the PtMult to be absorbed into the inner loop. This algorithm is given in Algorithm 8, and the lines that differ from HRotate are in blue.

Algorithm 8 $\mathsf{PtMatVecMult}(\mathbf{M}, [\![\mathbf{x}]\!], \{k_i, \mathsf{ksk}_i\}_{i=1}^{r}) = [\![\mathbf{Mx}]\!]$

1: $(\mathbf{a}_{\mathbf{x}}, \mathbf{b}_{\mathbf{x}}) := [\![\mathbf{x}]\!]_{\mathbf{s}}$
2: $\overrightarrow{\mathbf{a}_{\mathbf{x}}} := \mathsf{Decomp}_{\beta}(\mathbf{a}_{\mathbf{x}})$ ▷ β digits.
3: $\hat{\mathbf{a}}_j := \mathsf{ModUp}(\overrightarrow{\mathbf{a}_{\mathbf{x}}}^{(i)})$ for $1 \le j \le \beta$.
4: $(\hat{\mathbf{a}}_{\mathbf{y}}, \hat{\mathbf{b}}_{\mathbf{y}}) \leftarrow 0, 0$ ▷ We will have $\mathbf{y} = \mathbf{Mx}$.
5: **for** i from 1 to r **do**
6: $\quad \hat{\mathbf{a}}_{\mathsf{rot}}^{(j)} := \mathsf{Automorph}(\hat{\mathbf{a}}_j, k_i)$ for $1 \le j \le \beta$
7: $\quad (\hat{\mathbf{u}}, \hat{\mathbf{v}}) := \mathsf{KSKInnerProd}(\mathsf{ksk}_i, \overrightarrow{\hat{\mathbf{a}}_{\mathsf{rot}}})$
8: $\quad \mathbf{b}_{\mathsf{rot}} := \mathsf{Automorph}(\mathbf{b}_{\mathbf{m}}, k_i)$
9: $\quad \hat{\mathbf{b}}_{\mathsf{rot}}, \hat{\mathbf{M}}_i \leftarrow \mathsf{PModUp}(\mathbf{b}_{\mathsf{rot}}), \mathsf{PModUp}(\Delta \cdot \mathbf{M}_i)$
10: $\quad$ ▷ $\mathbf{M}_i$ is the i^{th} non-zero diagonal of $\mathbf{M}$
11: $\quad (\hat{\mathbf{a}}_{\mathbf{y}}, \hat{\mathbf{b}}_{\mathbf{y}}) \mathrel{+}= \hat{\mathbf{M}}_i \cdot (\hat{\mathbf{u}}, \hat{\mathbf{v}} + \hat{\mathbf{b}}_{\mathsf{rot}})$ ▷ PtMult
12: **end for**
13: **return** $(\mathsf{ModDown}(\hat{\mathbf{a}}_{\mathbf{y}}), \mathsf{ModDown}(\hat{\mathbf{b}}_{\mathbf{y}}))$

Removing Giant-Steps Beyond Bootstrapping: This optimization is not a bootstrapping-only optimization. The hoisting optimizations that are described for PtMatVecMult for bootstrapping are more broadly applicable to the InnerProduct computation. When multiple InnerProduct operations needs to be performed in parallel, this hoisting optimization can be amortized across these parallel InnerProduct computations, which results in about 35% improvement in logistic regression training iterations for our running example.

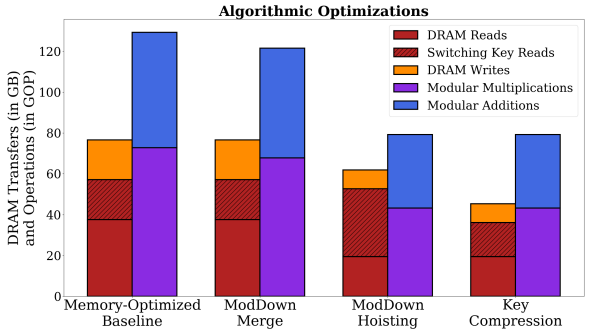

Fig. 5. This figure displays the algorithmic optimizations described in Section V. The impact is assessed cumulatively i.e. each successive optimization builds on top of the earlier ones. The baseline benchmark begins with all of the memory optimizations from Section IV. All benchmarks are taken with the **Best-case Parameters** from Table XI. GOP on y-axis stands for Giga operations.

C. Compressing the Key with a PRNG

This optimization is not our own; rather, it is a folklore technique often used to reduce communication when sending ciphertexts or keys over a network (e.g. it is used in Kyber, a leading candidate public-key encryption scheme in the ongoing NIST post-quantum cryptography standardization [2]). However, to our knowledge, we are the first to use this optimization to reduce the memory bandwidth for hardware acceleration of homomorphic encryption as well as the first to analyze this optimization alongside the other optimizations listed in this section. As discussed in Section V-B, this optimization has subtle yet highly impactful effects on the other optimizations that we list, drastically changing the optimal parameters for CKKS bootstrapping.

This optimization is a natural result of the observation that half of the switching key consists of truly random polynomials. By replacing these truly random polynomials with pseudorandom polynomials generated via PRNG, we can avoid shipping the large random polynomials to and from DRAM, instead sending only the short PRNG key.

D. Key Takeaways

Figure 5 shows how various optimizations impact the operation count and the DRAM transfers for CKKS bootstrapping. Moving from left to right on the plot, arithmetic intensity starts to improve as each successive optimization is applied and enabling all our optimizations result in a cumulative $2.43\times$ improvement and a final arithmetic intensity value of 1.75. We now contextualize this compute and bandwidth optimization in the context of current computing platforms.

Datacenter CPUs: Consider an example of a top-of-line datacenter CPU such as the AMD EPYC 7763. This CPU supports a maximum of 128 parallel thread across 64 SMT cores running at a base clock frequency of 2.45 GHz. This configuration supports peak integer theoretical throughput of 2.5 TOp/s (Each operation here is a 64-bit Integer Fused Multiply Add in AVX256 mode). Each socket consists of 8 compute die (CCD) with a local 32 MiB L3 cache per die. The total L3 cache per socket comes out to 256 MiB. Additionally, the socket offers an 8-channel DDR4-3200 memory subsystem with an aggregate bandwidth of 204 GB/s.

At first glance the total L3 capacity appears to be more than sufficient for storing multiple ciphertexts in cache. However the die-to-die bandwidth is limited by the underlying interconnect (Infinity Fabric) to 51.2 GiB/s reads and 25.6 GiB/s writes. There are similar bandwidth limits at the L1-L2 and L2-L3 interfaces on each die. Thus, it is necessary to consider the compute available on each die in the context of the bandwidth available to that die.

Each CCD pairs 310 GOp/s with 51.2 GiB/s of memory bandwidth. This gives a theoretical INT64 FMA arithmetic intensity of ~ 6. On current hardware, 64-bit modular operations need to be emulated using multiple arithmetic operations as seen in section II-B. Compensating for this, we observe that the final arithmetic intensity of our bootstrapping procedure is similar to what can be supported by state-of-art CPUs. Note that the addition of modular arithmetic vector extension to existing vector engines would already result in the overall application being memory bottlenecked.

Datacenter GPUs: For GPU analysis we consider the NVIDIA A100 datacenter GPU. This GPU offers a peak 19.5 TOp/s 32-bit Integer FMA performance when clocked at 1.41 GHz. It has an on-chip 40 MB L2 last-level cache and uses an HBM2 DRAM interface supporting 1.55 TB/s of bandwidth. Note again that a single die cannot fit a complete ciphertext in memory. Applications with an INT32 FMA arithmetic intensity lower than ~ 12 will tend to be memory bottlenecked. In addition, 64-bit integer arithmetic is not natively supported on a datacenter GPU and must be emulated in assembly which has a significant overhead (up to 20 instruction for 64-bit Integer multiply). As such for GPU implementations, it is advisable to use an RNS representation with 32-bit limbs to avoid this overhead. Addition of native 32-bit modular multiplication to future GPU will further worsen the memory bottleneck.

While the above estimations are simplistic and do not take into account the intricacies of instruction scheduling, the underlying point remains that raw access to compute power is not what bottlenecks existing FHE implementations. Building new hardware that merely adds an order-of-magnitude to the compute capability is unlikely to give an order of magnitude performance improvements without addressing the memory side of the story.

VI. Evaluation

In this section, we compare our bootstrapping algorithm to prior art to demonstrate the improved throughput achieved by our optimizations. In addition, we show how our improved CKKS subroutines directly result in more efficient HE applications.

A. Maximizing Bootstrapping Throughput

Bootstrapping Throughput: Our metric to evaluate bootstrapping performance is based on the *bootstrapping throughput* metric of Han and Ki [20]. This metric attempts to capture

the effectiveness of a bootstrapping routine by improving with the number of slots the algorithm bootstraps (which is the number of plaintext slots n), the number of limbs ℓ in the resulting ciphertext (which translates to the number of compute levels supported by the ciphertext), and the bit-precision bp of the plaintext data. These factors are then divided by the runtime of the bootstrapping procedure, denoted as brt. This gives us the throughput metric in Equation (3).

$$\mathsf{throughput} = \frac{n \cdot \ell \cdot \mathsf{bp}}{\mathsf{brt}} \tag{3}$$

Optimal Bootstrapping Parameters: Given the throughput metric from Equation (3), we can select parameters to optimize it. We employ our architectural modeling tool to explore the parameter space of bootstrapping to maximize the throughput. As DRAM transfer times dominate in bootstrapping, our architectural model accounts for DRAM transfer time in the total runtime analysis, resulting in parameters that minimize DRAM transfers. The throughput-maximizing parameters for our fully-optimized bootstrapping algorithm (with all optimizations from Section IV and Section V) are given in Table XI.

TABLE XI
BOOTSTRAPPING PARAMETERS
The L parameter denotes the number of limbs in the ciphertext after the initial ModUp *procedure in* Bootstrap. *The* fftIter *parameter is the number of* PtMatVecMult *iterations in the* CoeffToSlot *and* SlotToCoeff *phases in* Bootstrap. *The radix values for these iterations are all balanced, with any values that need to be larger placed at the end. The λ value is the bit-security level.*

	L	dnum	fftIter	λ
Baseline	35	3	3	$< 128^{\dagger}$
Best-case	40	2	6	128

† The baseline set is based on [22] originally targeting $\lambda = 128$. Updated cryptanalysis in [3] reduces the security level for sparse keys. The parameters in this work include these updated recommendations.

B. Bootstrapping Performance Comparisons

We now compare the throughput of our most optimized bootstrapping algorithm to prior art. To compare our algorithm to prior works, we re-implemented each algorithm in our architecture model. We then took the parameters given in each of these works and ran the algorithm in our model with these parameters. This allowed us to measure the total operations as well as the DRAM transfer times for each of these algorithms.

From this analysis as well as our discussion in Section V-D, we know that all of these bootstrapping algorithms are bottlenecked by the memory bandwidth. Therefore, we used the memory bandwidth requirement of each of these algorithms as a proxy for the overall runtimes. The memory requirement was converted to DRAM transfer time based on the memory bandwidth of the NVIDIA Tesla V100 [27], which is 900 GB/s.

The results of this analysis is presented in Table XII. We now discuss each comparison in more detail. The parameter set selected from Jung et al. [22] is the same parameter set used as the baseline comparison in Section IV, which is the parameter set they give for their logistic regression implementation.

We selected the parameter set from Bossuat et al. [3] that maximized their throughput. Note that this parameter set maximized the throughput when the runtime was measured on a CPU. For our architecture model, we are considering the case where computation has been accelerated to the point where runtime is completely dominated by memory transfers.

The throughput computation for Samarzdic et al. [30] was computed slightly differently since this work gives the DRAM bandwidth of their algorithm. However, this work only gives benchmarks for unpacked CKKS bootstrapping (i.e., there is no slot packing and the ciphertext only holds one element). Rather than re-implementing their algorithm, we use the memory bandwidth usage they give for their unpacked CKKS bootstrapping, which is 721 MB. To compute the runtime, we also use the peak DRAM bandwidth provided by the authors for their architecture, which is 1 TB/s. Using these two numbers, we found their bootstrapping procedure runtime to be 0.721 milliseconds leading to the throughput number mentioned in Table XII.

TABLE XII
BOOTSTRAPPING COMPARISON
*This table measures the bootstrapping throughput. The **Throughput** column is computed using Equation (3) with the DRAM transfer time as a proxy for the runtime. The DRAM transfer time is measured in microseconds.*

Work	n	ℓ	bp	**DRAM Transfers (in GB)**	**Throughput**
Jung et al. [22]	2^{16}	20	19	193.09	116.07
Bossuat et al. [3]	2^{15}	16	19	75.30	119.05
Samarzdic et al. [30]	1	13	24	0.721	0.43
Our Best Throughput	2^{16}	19	19	45.33	469.68

C. Application Comparison

A faster bootstrapping algorithm directly results in faster HE applications. Continuing with our running example of logistic regression training, we give benchmarks of the logistic regression algorithm from Section III using our optimized bootstrapping routine and parameters. These benchmarks are given in Table XIII.

D. Key Takeaways

In this section, we demonstrated that our optimizations, which mostly focus on improving the arithmetic intensity of bootstrapping and other CKKS building blocks, result in a much higher memory throughput than prior art that mostly focused on optimizing the compute throughput. This shows that focusing on compute throughput overlooks a crucial bottle-neck in CKKS applications: the memory bandwidth. To improve the overall performance of many important CKKS applications such as bootstrapping and encrypted logistic regression training, the memory bandwidth must be directly optimized.

TABLE XIII
PERFORMANCE OF LOGISTIC REGRESSION TRAINING EXAMPLE
This table displays benchmarks of the logistic regression bootstrapping application using our optimized bootstrapping parameters. In parentheses next to each benchmark, we give the improvement over Table VII.

Sub-routine Name	Total Operations (in GOP)	Total DRAM Transfers (in GB)	Arithmetic Intensity (in Op/byte)
InnerProduct	6.8256(1.2×)	3.3261(4.9×)	2.05(4.4×)
PolyEval	2.2569(1.3×)	0.9745(3.6×)	1.97(2.4×)
Full LR Iteration	77.3846(1.2×)	41.0811(4.7×)	1.88(4×)
Bootstrap	79.2401(1.9×)	45.3341(4.6×)	1.75(2.43×)

VII. DISCUSSION

Despite our algorithmic and cache optimizations to CKKS FHE bootstrapping (see Sections IV and V), our analysis reveals that FHE bootstrapping continues to have low arithmetic intensity and is heavily bounded by main memory bandwidth. This issue is not specific to CKKS bootstrapping alone. For one, bootstrapping algorithms for other FHE schemes such as BGV [5] and B/FV [4], [14] have the same high-level structure and suffer from the same problem, although with different quantitative thresholds. Additionally, as discussed in Section III-B, many natural applications (e.g. logistic regression and secure neural network evaluation) have the same high-level structure as bootstrapping, namely, global linear operations followed by local non-linear operations, and consequently, they suffer from the main memory bottleneck as well.

Below, we discuss potential research avenues to solve this issue that is so central to the practicality of FHE.

Future Improvements to Bootstrapping: At a high-level, our optimizations can be viewed as improving the "thrashing" of various low-level operations in the bootstrapping algorithm (as well as other natural applications of FHE such as encrypted training of machine learning models). While future improvements may reduce thrashing in the baseline algorithms, the size of the ciphertexts and the size of the switching keys suggests that the overall arithmetic intensity is unlikely to drastically improve without a dramatic overhaul to FHE schemes.

In one extreme, we could be in the best-case-scenario for FHE bootstrapping. In this world that we call "FHE-mania", all of bootstrapping can be done in cache without any DRAM reads or writes beyond the initial input and the final output. This world would call for true hardware acceleration of bootstrapping and would make our DRAM optimizations useless. On the other hand, we could be living in a world where the best possible bootstrapping algorithms remain bounded by the memory bandwidth. In this world that we call "thrashy-land", our optimizations remain crucial to achieving the highest throughput for bootstrapping. While it may be possible to optimize our way out of thrashy-land, as long as the RNS representation remains the dominant format of FHE data, our α-limb and β-limb caching optimizations will remain relevant.

A realistic possibility is a world that is somewhere in between FHE-mania and Thrashy-land. For example, it turns out that bootstrapping in GSW-like FHE schemes [13], [16] incurs slower noise growth and consequently smaller parameters N and Q; however, it does not support packed bootstrapping as in BGV, B/FV and CKKS FHE schemes, a feature that is fundamentally important for efficiency. Can we achieve the best of both worlds? We believe there is exciting research to be done here (see [26] for a preliminary attempt); our analysis provides a compelling reason to pursue this line of research.

Increase Main Memory Bandwidth: There are two approaches to increasing the main memory bandwidth. First, we can use multiple DDRx channels, effectively using parallelism to increase the main memory bandwidth. We could also use alternate main memory technologies like HBM2/HBM2e [21] that provide several times higher bandwidth than DDRx technology. The second approach involves improving the physical interconnect between the compute cores and the memory by using silicon-photonic link technologies [32]. Judicious use of silicon-photonic technology can help improve the main memory bandwidth, and has the additional benefit of reducing the energy consumption for memory accesses.

Improve Main Memory Bandwidth Utilization: Here, there are two complementary approaches. The first is to attempt a cleverer mapping of the data to physical memory to take advantage of spatial locality in cache lines such that we reduce the number of memory accesses required per compute operation. To complement this, we can improve FHE-based computing algorithms such that we perform more operations per byte of data that is fetched from main memory, i.e., improve temporal locality. The second approach is algorithmic: namely, improve FHE bootstrapping algorithms (as discussed above) so that we reduce the size of the key-switching parameter, the main culprit for low arithmetic intensity, or eliminate it altogether. These two complementary approaches may result in an increase in the arithmetic intensity, effectively reducing the time required for bootstrapping and FHE as a whole.

Use In-Memory/Near-Memory Computing: Two potential architecture-level approaches include performing the operations in FHE APIs within main memory i.e., in-memory computing, and having a custom die very close to main memory for performing operations in FHE APIs, i.e., near-memory computing. In the in-memory computing approach, we can eliminate a large number of expensive main memory accesses by performing matrix-vector multiplication operations in the main memory itself [12]. In contrast, in case of near-memory computing, we perform all the FHE compute operations in a custom accelerator that is placed close to the main memory. Here, we cannot eliminate the memory accesses, but the cost of a memory access is lower than that of accessing a traditional memory.

Use Wafer-Scale Systems: A radical technology-level solution is to design large-scale distributed accelerators such as Cerebras style wafer-scale accelerators [7] that have 40 GB of high-performance on-wafer memory. Tesla's Dojo accelerator [33] also fits in this category wherein a large wafer is

diced into 354 chip nodes, which provides high bandwidth and compute performance. Effectively, we can have large SRAM arrays i.e. large caches on the same wafer as the compute blocks, thus limiting all communication to on-chip wafer communication and avoiding expensive main memory accesses after the initial loads.

VIII. Related Work

Algorithmic optimizations for CPUs: The key bottleneck in the FHE bootstrapping process is the large *homomorphic matrix-vector multiplication* required to convert ciphertexts from coefficient to evaluation representation and back. This requires many key-switching operations, which require accessing large number of switching keys from the DRAM, adding both to the computational cost and to data access latency. Initial implementations of bootstrapping in software (for example, the HEAAN library [11]) did try to reduce the number of rotations required in this linear transformation step by using baby-step giant-step (BSGS) algorithm, originally invented by Halevi and Shoup [17]. Using this algorithm, one can reduce the number of rotations to $O(\sqrt{N})$ while still requiring only $O(N)$ scalar multiplications. The HEAAN library also optimizes the operational cost of approximating the modular reduction step by evaluating the sine function using a Taylor approximation. With these techniques, the HEAAN library takes about eight minutes to bootstrap 128 slots within a ciphertext of degree 2^{16} on a CPU.

Chen, Chillotti and Song [8] proposed a level collapsing technique along with BSGS for the linear transformation step to improve the number of rotations. They also replaced the Taylor approximation with a more accurate Chebyshev approximation to evaluate a scaled-sine function instead. For the same parameter set as the HEAAN library, they observe a $3\times$ speedup. More recently, Han and Ki [20] proposed a hybrid key-switching approach to efficiently manage the amount of noise added through the key-switching operation. They evaluated a scaled, shifted cosine function instead of the scaled-sine function in modular reduction to reduce the number of non-scalar multiplications by half. Their optimizations led to an additional $3\times$ speedup. Bossuat et al. [3] further lower the operational complexity of the linear transformations by optimizing rotations through double-hoisting the hybrid key-switching approach. Double-hoisting the key-switch operation reduces the number of basis conversion operations significantly, which are expensive in terms of accessing the main memory. They also carefully manage the scale factors for non-linear transformations for error-less polynomial evaluation. Their implementation in Lattigo library [24] shows a further speedup of $1.5\times$ on a CPU.

Algorithmic optimizations for GPUs: All the above mentioned optimizations heavily focused on lowering the operation complexity of bootstrapping, which led to a minor reduction in the main memory accesses as well. Recently, Jung et al. [22] presented the first ever GPU implementation of CKKS bootstrapping. Their analysis, even though limited to GPUs, rightly points out the main-memory-bounded nature of the bootstrapping operation. Thus, their optimizations, such as inter- and intra-kernel fusion, are all focused on improving the memory bandwidth utilization rather than accelerating the compute itself. Their bootstrapping implementation is so far the fastest requiring only 328.25 ms (total time) for bootstrapping all the slots of a ciphertext of degree $N = 2^{16}$. As discussed in Section IV and V, our techniques are composable with all these prior works and consequently, result in $3.2\times$ higher arithmetic intensity and $4.6\times$ reduction in main memory accesses.

Hardware Accelerators for HE: Samardzic et al. [30] recently presented the architecture of a programmable hardware accelerator for FHE operations. Their analysis also shows the fact that the FHE operations are memory bottlenecked. However, they implement a massively parallel compute block (having 4096 modular multiplications) in their accelerator. From their performance analysis, it is evident that the compute block is underutilized due to the memory bottleneck.

IX. Conclusion

In this paper, we undertook a thorough architecture-level analysis of the compute and memory requirements for fully homomorphic encryption to identify the limits and opportunities for hardware acceleration. Our analysis shows that the bootstrapping step is the critical performance bottleneck in FHE-based computing, and it has low arithmetic intensity and is heavily constrained by today's main memory systems. We argue that to accelerate FHE-based computing, the research community should focus on improving the arithmetic intensity of FHE-based computing and leverage novel memory system architectures. We proposed several architecture-independent and cache-friendly optimizations that improve arithmetic intensity by about $2.43\times$. We also propose custom physical address mapping for *limb-wise* and *slot-wise* operations to enhance the main memory bandwidth utilization. To further mitigate the impact of memory bandwidth on FHE-based computing, we suggest directions for the research community to explore novel techniques to either increase main memory bandwidth or improve its utilization, use in-memory/near-memory computing, and/or use wafer-scale systems with large on-chip memory.

References

[1] P. Barrett, "Implementing the Rivest-Shamir-Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323. 3

[2] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - kyber: A cca-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 353–367. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00032 12

[3] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Advances in Cryptology – EUROCRYPT 2021*, A. Canteaut and F.-X. Standaert, Eds. Cham: Springer International Publishing, 2021, pp. 587–617. 2, 7, 11, 13, 15

[4] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 868–886. 2, 14

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ITCS '12*, 2012. 2, 14

[6] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, R. Ostrovsky, Ed. IEEE Computer Society, 2011, pp. 97–106. [Online]. Available: https://doi.org/10.1109/FOCS.2011.124

[7] "Cerebras wse-2," Online: https://cerebras.net/, April 2021, cerebras. 14

[8] H. Chen, I. Chillotti, and Y. Song, "Improved bootstrapping for approximate homomorphic encryption," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 34–54. 7, 15

[9] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 360–384. 7

[10] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson Jr., Eds. Cham: Springer International Publishing, 2019, pp. 347–368. 4

[11] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437. 2, 3, 15

[12] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 27–39. [Online]. Available: https://doi.org/10.1109/ISCA.2016.13 14

[13] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, E. Oswald and M. Fischlin, Eds., vol. 9056. Springer, 2015, pp. 617–640. [Online]. Available: https://doi.org/10.1007/978-3-662-46800-5_24 14

[14] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://ia.cr/2012/144. 2, 14

[15] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009. 1, 4

[16] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Canetti and J. A. Garay, Eds., vol. 8042. Springer, 2013, pp. 75–92. [Online]. Available: https://doi.org/10.1007/978-3-642-40041-4_5 14

[17] S. Halevi and V. Shoup, "Faster homomorphic linear transformations in helib," in *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, 2018, pp. 93–120. [Online]. Available: https://doi.org/10.1007/978-3-319-96884-1_4 15

[18] K. Han, M. Hhan, and J. H. Cheon, "Improved homomorphic discrete fourier transforms and fhe bootstrapping," *IEEE Access*, vol. 7, pp. 57 361–57 370, 2019. 2, 7

[19] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 9466–9471, Jul. 2019. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5000 5, 6, 7, 8

[20] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Topics in Cryptology – CT-RSA 2020*, S. Jarecki, Ed. Cham: Springer International Publishing, 2020, pp. 364–390. 2, 4, 7, 10, 12, 15

[21] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "HBM (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, 2017, pp. 1–4. 14

[22] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 4, p. 114–148, Aug. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9062 1, 2, 8, 13, 15

[23] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon, "Logistic regression model training based on the approximate homomorphic encryption," *BMC Medical Genomics*, vol. 11, 2018. 5

[24] "Lattigo v2.2.0," Online: http://github.com/ldsec/lattigo, April 2021, ePFL-LDS. 15

[25] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020. 9

[26] D. Micciancio and J. Sorrell, "Ring packing and amortized FHEW bootstrapping," in *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, ser. LIPIcs, I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, Eds., vol. 107. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 100:1–100:14. [Online]. Available: https://doi.org/10.4230/LIPIcs.ICALP.2018.100 14

[27] NVIDIA, "NVIDIA Tesla V100 GPU Architecture," 2017. 13

[28] NVIDIA, "NVIDIA A100 Datasheet," 2021, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf. 3

[29] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of Secure Computation, Academia Press*, pp. 169–179, 1978. 1

[30] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 238–252. [Online]. Available: https://doi.org/10.1145/3466752.3480070 2, 13, 15

[31] V. Shoup *et al.*, "NTL: A library for doing number theory," 2001, https://libntl.org/. 3

[32] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avizienis, S. Lin, B. R. Moss, R. Kumar, F. Pavanello, A. H. Atabaki, H. M. Cook, A. J. Ou, J. C. Leu, Y.-H. Chen, K. Asanović, R. J. Ram, M. Popović, and V. M. Stojanović, "Single-chip microprocessor that communicates directly using light," *Nature*, vol. 528, no. 7583, pp. 534–538, 2015. [Online]. Available: https://doi.org/10.1038/nature16454 14

[33] "Tesla dojo d1," Online: https://www.datacenterdynamics.com/en/news/tesla-details-dojo-supercomputer-reveals-dojo-d1-chip-and-training-tile-module/, Aug. 2021, tesla. 14