MAD: Memory-Aware Design Techniques for Accelerating Fully Homomorphic Encryption

Rashmi Agrawal
Boston University, Boston, USA
rashmi23@bu.edu

Leo de Castro
MIT, Cambridge, USA
ldec@mit.edu

Chiraag Juvekar
Analog Devices, Boston, USA
chiraag.juvekar@analog.com

Anantha Chandrakasan
MIT, Cambridge, USA
anantha@mit.edu

Vinod Vaikuntanathan
MIT, Cambridge, USA
vinodv@mit.edu

Ajay Joshi
Boston University, Boston, USA
joshi@bu.edu

ABSTRACT

Cloud computing has made it easier for individuals and companies to get access to large compute and memory resources. However, it has also raised privacy concerns about the data that users share with the remote cloud servers. Fully homomorphic encryption (FHE) offers a solution to this problem by enabling computations over *encrypted* data. Unfortunately, all known constructions of FHE require a *noise* term for security, and this noise grows during computation. To perform unlimited computations on the encrypted data, we need to perform a periodic noise reduction step known as *bootstrapping*. This bootstrapping operation is memory-bound as it requires several GBs of data. This leads to orders of magnitude increase in the time required for operating on encrypted data as compared to unencrypted data.

In this work, we first present an in-depth analysis of the bootstrapping operation in the CKKS FHE scheme. Similar to other existing works, we observe that CKKS bootstrapping exhibits a low arithmetic intensity (<1 Op/byte). We then propose memory-aware design (MAD) techniques to accelerate the bootstrapping operation of the CKKS FHE scheme. Our proposed MAD techniques are agnostic of the underlying compute platform and can be equally applied to GPUs, CPUs, FPGAs, and ASICs. Our MAD techniques make use of several caching optimizations that enable maximal data reuse and perform reordering of operations to reduce the amount of data that needs to be transferred to/from the main memory. In addition, our MAD techniques include several algorithmic optimizations that reduce the number of data access pattern switches and the expensive NTT operations. Applying our MAD optimizations for FHE improves bootstrapping arithmetic intensity by 3×. For Logistic Regression (LR) training, by leveraging our MAD optimizations, the existing GPU design can get up to 3.5× improvement in performance for the same on-chip memory size. Similarly, the existing ASIC designs can get up to 27× and 57× improvement in performance for LR training and ResNet-20 inference, respectively, while reducing the on-chip memory requirement by 16×, which proportionally reduces the cost of the solution.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614302

CCS CONCEPTS

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

KEYWORDS

Fully Homomorphic Encryption, CKKS Scheme, Memory Bottleneck Analysis, Cache Optimizations, SimFHE, Hardware Acceleration, Bootstrapping

ACM Reference Format:
Rashmi Agrawal, Leo de Castro, Chiraag Juvekar, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. MAD: Memory-Aware Design Techniques for Accelerating Fully Homomorphic Encryption. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3613424.3614302

1 INTRODUCTION

Cloud-based systems enable reliable and affordable access to shared computing resources at scale. However, these shared resources raise data security and privacy concerns [26]. We need techniques to guarantee the confidentiality of user data when it is shared with third-party cloud services. Fully homomorphic encryption (FHE) [14, 29] is one such technique that enables cloud operators to perform complex computations on encrypted user data without ever needing to decrypt it. While FHE provides impressive privacy gains, computing over encrypted data is multiple orders of magnitude slower than operating on unencrypted data [21].

At a high level, in all constructions of FHE, this large overhead is due to the presence of a noise term in the ciphertext. Each homomorphic operation performed on the ciphertext increases the noise in the ciphertext. If this noise grows beyond a critical level, the 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 [14]. This is true irrespective of the FHE scheme, i.e., Brakerski-Gentry-Vaikuntanathan (BGV) [6], Brakerski/Fan-Vercauteren (B/FV) [5, 13], or Cheon-Kim-Kim-Song (CKKS) [11].

In this work, we focus on the CKKS FHE scheme as it supports floating-point operations, enabling many practical privacy-preserving computing applications such as machine learning training and inference. The CKKS bootstrapping step is expensive in

terms of both compute and memory requirements, and is often >100× more expensive than primitive operations like addition and multiplication on encrypted data [1, 20]. In addition, in CKKS-based FHE applications, bootstrapping alone constitutes the majority of the runtime; even when heavily optimized, bootstrapping consumes ~80% of the time in machine learning applications [20, 25].

Given that this bootstrapping step is the main bottleneck of the CKKS FHE scheme, it is natural to ask if hardware could be employed to accelerate the bootstrapping operation. Numerous prior works have explored this question [20, 24, 25, 30, 31], and they have each empirically found that their compute-accelerated implementations are bottlenecked by the main-memory bandwidth. Some of the recent works such as ARK [24] and CraterLake [31] tried alleviating the memory bandwidth bottleneck by using a large 256 and 512 MB on-chip memory, respectively. However, to accommodate this large 512 MB memory on-chip, one needs to use an advanced technology node like the 7nm[12], which is prohibitively expensive [3, 23].

In this paper, we focus on developing cost-effective solutions for accelerating bootstrapping. In particular, we focus on developing solutions for chips that cannot afford to have more than a few dozen MBs of on-chip memory. To this end, we first measure the compute and memory bandwidth requirements of hardware-accelerated CKKS using a custom simulator called SimFHE. SimFHE simulates the execution of CKKS-based applications in detail, from primitive operations (basic building blocks of CKKS) to bootstrapping. For a given application, SimFHE tracks the number of compute operations and the data movement between DRAM and on-chip memory. SimFHE supports any CKKS parameter set and a variety of on-chip memory sizes. Our analysis using SimFHE reveals that, without a prohibitively large on-chip memory, all FHE operations exhibit low arithmetic intensity (<1 Op/byte) and are memory bounded.

To address the memory bottleneck resulting from the low arithmetic intensity, and in turn, improve performance, we propose several memory-aware design (MAD) techniques that reduce the memory requirements of CKKS-based application execution. Our MAD techniques, detailed in Section 3, include caching optimizations as well as algorithmic optimizations. They are agnostic of the underlying compute platform, and can be easily applied on top of existing GPU, CPU, FPGA, and ASIC solutions for FHE acceleration. Our caching optimizations reorder the operations to maximize data reuse in the bootstrapping algorithm. This reordering significantly reduces the number of DRAM accesses alleviating the memory bandwidth bottleneck. Our algorithmic optimizations reduce the number of times the bootstrapping algorithm oscillates between the *limb-wise* representation and *slot-wise* representation of a ciphertext, which greatly reduces the number of DRAM transfers. In summary, in this work, we make the following key contributions:

- We present caching optimizations that reduce the memory bandwidth requirements of CKKS bootstrapping. These optimizations enable maximal data reuse by reordering operations. Using our caching optimizations, we can accelerate CKKS-based applications running on compute platforms even with small on-chip memory (<32 MB).
- We present several hardware-agnostic algorithmic optimizations that reduce the number of data access pattern switches and the expensive NTT operations, by reducing the number of ModDown sub-operations in various operations such as Mult and Rotate. These optimizations reduce the number of DRAM accesses for CKKS bootstrapping.
- We develop SimFHE, a custom simulator that can be used to benchmark the compute and memory requirements of CKKS at different scales: from primitive operations to end-to-end applications such as machine-learning training. In the spirit of open science, we open source[1] SimFHE for the benefit of the research community at large.

Based on our benchmarking results of the MAD techniques using SimFHE, we propose an optimized, memory-aware CKKS parameter set that maximizes the throughput of CKKS bootstrapping for systems with limited on-chip memory. Applying our MAD optimizations for FHE improves bootstrapping arithmetic intensity by 3×. For Logistic Regression (LR) training, by leveraging our MAD optimizations, the existing GPU design can get up to 3.5× improvement in performance for the same on-chip memory size. Similarly, the existing ASIC designs can get up to 27× and 57× improvement in performance for LR training and ResNet-20 inference, respectively, while reducing the on-chip memory requirement by 16×, which proportionally reduces the cost of the solution.

2 BACKGROUND

In this section, we briefly review the CKKS scheme. In Table 1 we summarize the various parameters we use for describing the CKKS scheme.

2.1 CKKS Basics

The high-level operations of CKKS compute homomorphically over the plaintext space. 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. We denote the encryption of a length-n vector $\mathbf{x}$, i.e. the ciphertext, by $[\![\mathbf{x}]\!]$. All arithmetic operations on plaintexts are component-wise; the entries of the vector $\mathbf{x} + \mathbf{y}$ (resp. $\mathbf{x} \cdot \mathbf{y}$) are

[1]The code can be found at https://github.com/bu-icsg/SimFHE

Table 1: CKKS FHE Parameters and their description.

Param	Description
N	Number of coefficients in a polynomial in the ciphertext ring.
n	$N/2$, number of plaintext elements in a 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	$\lceil (L+1)/\text{dnum} \rceil$. 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.
fftIter	The number of iterations in the homomorphic evaluation of the DFT in bootstrapping.

the component-wise sums (resp. products) of the entries of $\mathbf{x}$ with the corresponding entries of $\mathbf{y}$. Table 2 gives a complete description of the core underlying API implemented by CKKS. We use a similar notation as [24].

Polynomial Rings and Ciphertexts: A CKKS ciphertext is a pair of elements in the polynomial ring $\mathcal{R}_Q := \mathbb{Z}_Q[x]/(x^N + 1)$. Each element of this ring is a polynomial with degree $N-1$ and coefficients in $\mathbb{Z}_Q$. For a message $\mathbf{m} \in \mathbb{C}^n$, we denote its encryption as $[\![\mathbf{m}]\!] = (\mathbf{a_m}, \mathbf{b_m})$ 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. To give a high-level intuition for the sizes of these parameters, the coefficient modulus Q considered in this work is typically between a few hundred and a few thousand bits. The polynomial modulus degree is a power of two and it is $N = 2^{17}$ in this work. These large sizes are required to maintain the security of the underlying Ring-Learning with Errors assumption. We refer the reader to [2, 28] for more information about the Ring-Learning with Errors problem and secure parameters.

Residue Number System: Often, the scalar values comprising a CKKS ciphertext are on the order of thousands of bits. To efficiently compute on such large numbers, we use the residue number system (RNS), also called the Chinese remainder representation. We select the coefficient modulus $Q = \prod_{i=1}^{\ell} q_i$, where each q_i is a prime number that fits in a standard machine word (less than 64 bits). We represent a scalar $x \in \mathbb{Z}_Q$ as ℓ integers modulo each of the q_i, making use of the isomorphism between $\mathbb{Z}_Q$ and the product group $\mathbb{Z}_{q_1} \otimes \ldots \otimes \mathbb{Z}_{q_\ell}$. We call the set $\mathcal{B} := \{q_1, \ldots, q_\ell\}$ an *RNS basis*, where each q_i is a *limb* of Q. Thus, we can operate over values in $\mathbb{Z}_Q$ without any native support for multi-precision arithmetic. A CKKS ciphertext consists of two ring elements, so the size of a CKKS ciphertext is $2N\ell$ machine words.

Changing the RNS Basis: The ModUp and ModDown operations (described later in this section) in CKKS require changing the RNS basis of a value $x \in \mathbb{Z}_Q$. Below we briefly explain the process for extending an RNS basis [10, 15] using Equation (1). This equation takes in an RNS representation of a value $x \in \mathbb{Z}_Q$ as a length-ℓ vector of scalars $[x]_{\mathcal{B}} = ([x]_{q_1}, [x]_{q_2}, \ldots, [x]_{q_\ell})$, where $[x]_{q_i} \equiv x \pmod{q_i}$. It outputs $x \mod p$, where p is a new modulus in an extended RNS basis.

$$[x]_p = \sum_{i=1}^{\ell} \left[[x]_{q_i} \cdot \tilde{Q}_i \right]_{q_i} \cdot Q_i^* \pmod{p} \quad (1)$$
$$= \mathsf{NewLimb}([x]_{\mathcal{B}}, p)$$

where $Q_i^* = Q/q_i$ and $\tilde{Q}_i = (Q_i^*)^{-1} \pmod{q_i}$. We refer to Equation (1) as NewLimb. To establish terminology, we note that this equation operates over all *limbs* $([x]_{q_1}, [x]_{q_2}, \ldots, [x]_{q_\ell})$ of a single *slot* or coefficient of the scalar $x \in \mathbb{Z}_Q$. This is in contrast with operations described below that operate over a fixed *limb* across all of the elements of some vector in $\mathbb{Z}_Q^N$.

Polynomial Representation: To enable fast polynomial multiplication, by default, we represent all polynomials 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*. This is in contrast to the *coefficient representation* of a polynomial, which is simply the vector of its coefficients. The RNS basis change operations must be performed over a polynomial's coefficient representation. The addition of two polynomials is $O(N)$ 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. The NTTs are always performed over the limb moduli; switching the representation of a full ring element requires ℓ NTTs over a machine-word-sized modulus.

Table 2: CKKS Primitive Operations.

Operation	Output	Rescale	KeySwitch	Description
PtAdd($[\![\mathbf{x}]\!], \mathbf{y}$)	$[\![\mathbf{x}+\mathbf{y}]\!]$	No	No	Adds a plaintext vector to an encrypted vector.
Add($[\![\mathbf{x}]\!], [\![\mathbf{y}]\!]$)	$[\![\mathbf{x}+\mathbf{y}]\!]$	No	No	Adds two encrypted vectors.
PtMult($[\![\mathbf{x}]\!], \mathbf{y}$)	$[\![\mathbf{x}\cdot\mathbf{y}]\!]$	Yes	No	Multiplies a plaintext vector and an encrypted vector.
Mult($[\![\mathbf{x}]\!], [\![\mathbf{y}]\!]$)	$[\![\mathbf{x}\cdot\mathbf{y}]\!]$	Yes	Yes	Multiplies two encrypted vectors.
Rotate($[\![\mathbf{x}]\!], k$)	$[\![\phi_k(\mathbf{x})]\!]$	No	Yes	Rotates a vector by k positions. Permutation of slots is denoted by Automorph.
Conjugate($[\![\mathbf{x}]\!]$)	$[\![\bar{\mathbf{x}}]\!]$	No	Yes	Computes the complex conjugate of $\mathbf{x}$.

Table 3: Data dependencies and access patterns.

Operation	Interaction	Independent	Access pattern
NTT, iNTT	Slots	Limbs	*limb-wise*
NewLimb	Limbs	Slots	*slot-wise*

Data access pattern: 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 NewLimb 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 account for this, we define two different types of data access patterns. For the functions that operate over all slots of a fixed limb (such as NTT and iNTT), we define the data access pattern as *limb-wise*. For the functions that operate over various limbs of a fixed slot (such as NewLimb in eq. (1)), we define the data access pattern as *slot-wise*. A summary of this is given in Table 3.

RNS Basis Change Operations: In CKKS, the source of the orientation change of the data access is in the RNS basis change operations; all other operations operate *limb-wise*. We now give the two primary RNS operations in CKKS. We abuse notation a bit to accommodate vectors over $\mathbb{Z}_Q^N$. For a vector $\mathbf{x} = (x_1, \ldots, x_N)$, we write $[\mathbf{x}]_q = ([x_1]_q, \ldots, [x_N]_q)$. Similarly, we write NewLimb$([\mathbf{x}]_{\mathcal{B}}, p)$ as NewLimb applied to each element of $\mathbf{x}$ in parallel.

The first operation is ModUp, which extends the RNS basis to include new primes. In Algorithm 1, we define the operation to

take the evaluation representation of a polynomial as input and to output this same representation. This highlights the change of orientation required for these operations.

The second operation is ModDown. This is a bit more complicated than simply reducing the RNS basis. Instead, ModDown can be viewed as a division operation where the denominator is the product of the dropped limbs. More specifically, if the input to ModDown is $[\mathbf{x}]_{\mathcal{B}\cup\mathcal{B}'}$, let $P = \prod_{p\in\mathcal{B}'} p$ be the product of the moduli of the dropped limbs. The output of ModDown is $[P^{-1}\cdot\mathbf{x}]_{\mathcal{B}}$. This is given in Algorithm 2 [10]. As with ModUp, this algorithm operates over polynomials where the inputs and outputs are in evaluation representation.

2.2 Memory Bottlenecks of CKKS

In this section, we explain the sources of the memory bottlenecks of CKKS. Broadly, these bottlenecks arise when adjacent operations require different orientations of the data.

Implications of a Small Cache: We begin with the observation that secure CKKS parameters result in large polynomials. An example of secure parameters that achieve a 128-bit security level is $N = 2^{17}$, $L = 35$, which gives a total ciphertext size of ~73.4 MB. Typical chips designed today have small on-chip memory (about 1-32 MB), meaning that the on-chip memory likely cannot even fit a single element of $\mathcal{R}_Q$, let alone a full ciphertext. Therefore, to switch the view of the ring element between *limb-wise* & *slot-wise* format, DRAM transfers are required. A high number of data access pattern switches will result in more frequent DRAM transfers. In the remainder of this section, we describe CKKS subroutines that require these transitions. These subroutines, called Rescale and KeySwitch, are denoted in Table 2. **Every "Yes" in Table 2 indicates that a memory-intensive operation is required to implement the primitive operation.**

Rescale**: Shrinking the Scaling Factor:** Since CKKS encrypts values in $\mathbb{C}^n$, the messages must be multiplied by a scaling factor Δ during encryption. This scaling factor is usually the size of one of the limbs of the ciphertext, which is slightly less than a machine word. 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). To shrink the scaling factor, we divide the ciphertext by one of its limbs (which are chosen to be close to Δ) and round the result to the nearest integer. This operation, called Rescale, keeps the scaling factor of the ciphertext roughly the same throughout the computation. Note that Rescale also divides the coefficient modulus itself, reducing the number of limbs in the modulus. The Rescale operation is a specialized implementation of ModDown. For an input basis $\{q_1,\ldots,q_\ell\}$, Rescale is equivalent to ModDown with $\mathcal{B} = \{q_1,\ldots,q_{\ell-1}\}$ and $\mathcal{B}' = \{q_\ell\}$. In Table 2, we denote the operations that require Rescale. For a more formal description, we refer the reader to [10]. Naively, for each Rescale we must perform an orientation switch of the $\mathcal{R}_Q$ element. However, in Section 3.2 we show how to combine this orientation switch with other operations.

Algorithm 1 $\mathsf{ModUp}_{\mathcal{B},\mathcal{B}\cup\mathcal{B}'}([\mathbf{x}]_{\mathcal{B}}) = [\mathbf{x}]_{\mathcal{B}\cup\mathcal{B}'}$

1: **for** $q_i \in \mathcal{B}$ **do** $[\mathbf{x}]_{q_i} \leftarrow \mathsf{iNTT}([\mathbf{x}]_{q_i})$ ▷ *limb-wise*
2: **for** $p_j \in \mathcal{B}'$ **do** $[\mathbf{x}]_{p_j} \leftarrow \mathsf{NewLimb}_j([\mathbf{x}]_{\mathcal{B}}, p_j)$ ▷ *slot-wise*
3: **for** $p_j \in \mathcal{B}'$ **do** $[\mathbf{x}]_{p_j} \leftarrow \mathsf{NTT}([\mathbf{x}]_{p_j})$ ▷ *limb-wise*
4: **return** $[\mathbf{x}]_{\mathcal{B}\cup\mathcal{B}'}$ ▷ No need to do NTT on the input limbs.

Algorithm 2 $\mathsf{ModDown}_{\mathcal{B}\cup\mathcal{B}'\to\mathcal{B}}([\mathbf{x}]_{\mathcal{B}\cup\mathcal{B}'}) = [P^{-1}\cdot\mathbf{x}]_{\mathcal{B}}$

1: **for** $q \in \mathcal{B}\cup\mathcal{B}'$ **do** $[\mathbf{x}]_q \leftarrow \mathsf{iNTT}([\mathbf{x}]_q)$ ▷ *limb-wise*
2: **for** $q_i \in \mathcal{B}$ **do**
3: $\quad[\hat{\mathbf{x}}]_{q_i} \leftarrow \mathsf{NewLimb}_j([\mathbf{x}]_{\mathcal{B}'}, q_i)$ ▷ *slot-wise*
4: $\quad[\mathbf{x}]_{q_i} \leftarrow P^{-1}\cdot([\mathbf{x}]_{q_i} - [\hat{\mathbf{x}}]_{q_i}) \pmod{q_i}$ ▷ P^{-1} is mod q_i.
5: **for** $q_i \in \mathcal{B}$ **do** $[\mathbf{x}]_{q_i} \leftarrow \mathsf{NTT}([\mathbf{x}]_{q_i})$ ▷ *limb-wise*
6: **return** $[\mathbf{x}]_{\mathcal{B}}$

KeySwitch**: Changing the 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. To change this new key back to the original key, we perform a KeySwitch operation [7]. 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 $\mathbf{s}'$.

While the use of the KeySwitch operation differs slightly between Mult and Rotate, both functions require the same basic operation. This operation (see Algorithm 3) takes in a polynomial $\mathbf{x}\in\mathcal{R}_Q$ and produces an encryption of $\mathbf{x}\cdot\mathbf{s}$ under some new secret key $\mathbf{s}'$. As an intermediate value, KeySwitch produces a tuple of the form $[\![P\cdot\mathbf{x}\cdot\mathbf{s}]\!]_{\mathbf{s}'}\in\mathcal{R}_{PQ}^2$, which is an "encryption" of $P\cdot\mathbf{x}\cdot\mathbf{s}$ under the secret key $\mathbf{s}'$ and over the modulus PQ. This intermediate value is very important, as we show in Section 3.2 how we can work directly with this value rather than immediately performing a ModDown operation.

We follow the structure of the switching key in the work of Han and Ki [19], where the switching key, parameterized by a length dnum, is a $2\times\mathsf{dnum}$ matrix of polynomials.

$$\mathsf{ksk} = \begin{pmatrix}\mathbf{a}_1 & \mathbf{a}_2 & \ldots & \mathbf{a}_{\mathsf{dnum}}\\ \mathbf{b}_1 & \mathbf{b}_2 & \ldots & \mathbf{b}_{\mathsf{dnum}}\end{pmatrix}\in\mathcal{R}_{PQ}^{2\times\mathsf{dnum}} \tag{2}$$

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 $\beta\le\mathsf{dnum}$ digits, where β is defined in Table 1. Note that on line 3 of Algorithm 3 the last columns of the ksk matrix are simply dropped when $\beta<\mathsf{dnum}$.

We conclude by noting that in KeySwitch we have to switch orientation at two places: the ModUp functions on line 2 and the ModDown functions on line 4. This makes KeySwitch the most expensive subroutine in the implementation of the CKKS API. The primary focus of our caching and algorithmic optimizations is to reduce the overhead of KeySwitch.

As discussed in Section 2.2, the ciphertext modulus of a CKKS ciphertext 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 a ModUp operation, which gives the new plaintext as

Algorithm 3 KeySwitch$(\mathbf{x}, \mathsf{ksk}_{\mathbf{s}\to\mathbf{s}'}) \to [\![\mathbf{x}\cdot\mathbf{s}]\!]_{\mathbf{s}'}$

$\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_\beta := \mathsf{Decomp}_\beta(\mathbf{x})$ ▷ Splits $\mathbf{x}$ into β digits.
for $1 \le i \le \beta$ **do** $\overrightarrow{\mathbf{x}}[i] := \mathsf{ModUp}(\hat{\mathbf{x}}_i)$ ▷ $\overrightarrow{\mathbf{x}} \in \mathcal{R}_{PQ}^{\beta}$
$(\hat{\mathbf{u}}, \hat{\mathbf{v}}) := \mathsf{ksk} \cdot \overrightarrow{\mathbf{x}}$ ▷ $(\hat{\mathbf{u}}, \hat{\mathbf{v}}) = [\![P\cdot\mathbf{x}\cdot\mathbf{s}]\!]_{\mathbf{s}'} \in \mathcal{R}_{PQ}^{2}$
$(\mathbf{u}, \mathbf{v}) := (\mathsf{ModDown}(\hat{\mathbf{u}}), \mathsf{ModDown}(\hat{\mathbf{v}}))$
return $(\mathbf{u}, \mathbf{v})$ ▷ $(\mathbf{u}, \mathbf{v}) = [\![\mathbf{x}\cdot\mathbf{s}]\!]_{\mathbf{s}'} \in \mathcal{R}_{Q}^{2}$

$\Delta\cdot\mathbf{m} + \mathbf{k}q$ 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 [4, 8, 9, 17, 19] over the past few years. This structure has three main components: a linear operation, a polynomial 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. These DFT operations consist of a number of plaintext matrix-vector products, which we denote by PtMatVecMult. The parameter fftIter determines the number of matrix-vector products required; higher the fftIter, lower the dimension of each matrix. In between these two DFT operations is an approximation of the modular reduction function that consists of a polynomial evaluation followed by an exponentiation. We give a high-level pseudocode for the bootstrapping algorithm in Algorithm 4. For further details on DFT, polynomial approximation and evaluation, we refer the readers to [4, 19].

2.3 Arithmetic Intensity Analysis

In Table 4, we present the results of our arithmetic intensity analysis for CKKS operations using SimFHE (details in Section 4.1). These values consider caches that are smaller than one element of the ring $\mathcal{R}_Q$ but large enough to hold a maximum 1 or 2 limbs. One immediate observation that we can make from this table is that all individual CKKS operations (defined in Table 2) have arithmetic intensities of <1 Op/byte and require large working sets in on-chip memory. This indicates that any implementation using CKKS operations as a black box will suffer from a memory bottleneck.

However, we note that improving arithmetic intensity does *not* necessarily equate to an improvement in performance. This is because some of our optimizations (see the Mult improvement in Section 3.2) such as ModDown merge and ModDown hoisting actually *decrease* the arithmetic intensity by reducing compute overhead more than the memory requirement, but these optimizations still improve the overall bootstrapping performance. This is because these two operations perform expensive NTT operations (having O($N \log N$) complexity). Thus, any reduction in the number of ModDown operations leads to a significant reduction in the compute, which improves the overall bootstrapping performance.

Algorithm 4 Bootstrap$([\![\mathbf{x}]\!]) = [\![\mathbf{x}]\!]$

$(\mathbf{a}, \mathbf{b}) := [\![\mathbf{x}]\!]$
$[\![\mathbf{t}]\!] := \mathsf{ModUp}(\mathbf{a}, \mathbf{b})$
for i from 1 to fftIter **do** ▷ CoeffToSlot phase.
  $[\![\mathbf{t}]\!] \leftarrow \mathsf{PtMatVecMult}(\mathbf{M}_i, [\![\mathbf{t}]\!])$
$[\![\mathbf{t}]\!] \leftarrow \mathsf{PolyEval}([\![\mathbf{t}]\!], \mathsf{mod}(\cdot))$ ▷ Approximate mod reduction.
for i from 1 to fftIter **do** ▷ SlotToCoeff phase.
  $[\![\mathbf{t}]\!] \leftarrow \mathsf{PtMatVecMult}(\mathbf{M}_i, [\![\mathbf{t}]\!])$
return $[\![\mathbf{t}]\!]$

3 MAD TECHNIQUES

In this section, we present our MAD techniques for accelerating the CKKS bootstrapping and applications in general. These techniques are designed for a variety of cache[2] sizes, ranging from sizes that can only cache a single limb to caches that can fit $O(\alpha)$ limbs (where α is defined in Table 1). We first describe our caching optimizations that reorder the operations to maximize the utilization of data in various-sized caches. Then we describe our main algorithmic optimizations that focus on reducing the number of times we must switch between the *limb-wise* and *slot-wise* orientations during various operations. We would like to note that the caching optimizations only reduce the number of accesses to the main memory while algorithmic optimizations reduce the orientation switch and the number of operations (expensive NTT operations) in turn reducing the main memory accesses.

3.1 Caching Optimizations

Our MAD techniques can support a variety of cache sizes. Our baseline implementation, against which we compare our proposed work, is based on the GPU-accelerated CKKS bootstrapping work of Jung et al. [20]. This work is the closest to our setting since it is a small-cache acceleration of CKKS bootstrapping. We accurately implement the bootstrapping work of Jung et al. [20] in SimFHE using the same parameters as in [20], and we confirmed with SimFHE that these parameters maximize the bootstrapping throughput (see Table 5) of this baseline implementation. Note that we use these same parameters for all our caching optimizations.

We describe our caching optimizations by grouping them into three categories. These categories represent the size of the cache with respect to the parameters of the CKKS scheme. More specifically, the more ciphertext limbs a cache can store, the more optimizations it can leverage. The caching optimizations do not impact the number of operations in bootstrapping. Essentially, the number of compute operations remains constant, but we reduce the number number of DRAM transfers required for bootstrapping.

Caching $O(1)$ Limbs: This first optimization is for a cache that can store one limb of the ciphertext. The size of the cache considered for this optimization is 1 MB as the size of a ciphertext limb is ~1 MB. Intuitively, in this optimization, we compute as many sub-operations within an operation on a single limb before writing it back to the main memory. That way we avoid reading and writing a limb from and to the main memory multiple times. We include caching $O(1)$ optimization at several places in our bootstrapping

[2]Although we use on-chip memory and cache interchangeably, we imply on-chip memory as our optimizations are agnostic of the underlying compute platform.

Table 4: Total operations in giga-ops, total DRAM transfers in GB, and arithmetic intensity (AI) in ops/byte. Here $\log(N) = 17$, $\ell = 35$, dnum $= 3$. The cache size is smaller than a single element of the ring $\mathcal{R}_Q$ but large enough to hold a constant number of limbs. Rotate **and** Conjugate **have identical implementations.**

	PtAdd	Add	PtMult	Decomp	ModUp	KSKInnerProd	ModDown	Mult	Automorph	Rotate	Conjugate	Bootstrap
Operations	0.0046	0.0092	0.2747	0.0092	0.2847	0.0629	0.3000	1.8333	0	1.5310	1.5310	149.546
DRAM Transfers	0.1101	0.2202	0.3282	0.0734	0.1510	0.4530	0.1877	1.9293	0.1468	1.5645	1.5645	207.982
AI	0.04	0.04	0.84	0.12	1.88	0.13	1.59	0.95	0	0.98	0.98	0.72

implementation that compute more aggressively on a single limb as much as possible.

Out of the several operations that can leverage this optimization, below we present a concrete example of our $O(1)$ optimization for the Rotate operation. Say we want to Rotate a ciphertext having 35 limbs. As shown in Figure 1 (a), naïvely, we will begin with Automorph sub-operation on all 35 limbs, requiring 35 limb reads and writes as our cache can store only a single limb at a time. Then we will perform the Decomp sub-operation, which will again require 35 more limb reads and writes. We can keep progressing forward with the subsequent sub-operations, incurring 245 ciphertext limb reads and writes for a single Rotate operation.

We propose to eliminate these intermediate reads and writes by greedily performing as many sub-operations as possible on a single limb before writing it back to the main memory. Therefore, we read the first limb of the ciphertext and then perform Automorph, Decomp, and iNTT at once. The NewLimb sub-operation requires multiple slots across limbs (orientation switch), and so cannot be performed just using the current limb. So we write the current limb to the main memory (see Figure 1 (b)). We then perform the Automorph, Decomp, and iNTT sub-operations on the remaining limbs (one at a time) of the ciphertext. Our approach avoids 140 DRAM reads and writes per ciphertext, which equates to avoiding 124 MB of data transfer for a ciphertext. We would like to note that we perform the Rotate operation multiple times when executing any CKKS-based FHE application and we can apply our caching

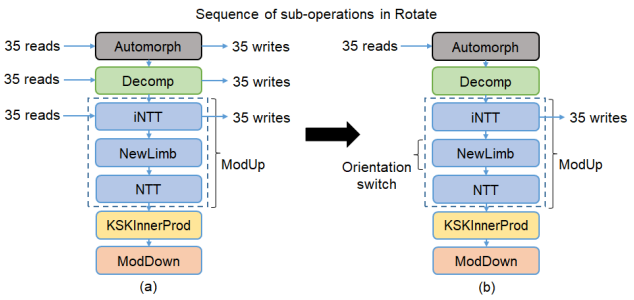

Figure 1: Conceptual view of caching $O(1)$ limbs optimization; (a) Naïve approach for Rotate **operation on a ciphertext with** 35 **limbs requires** 105 **DRAM reads and** 105 **DRAM writes. (b) Our proposed approach for** Rotate **operation requires** 35 **DRAM reads and** 35 **DRAM writes.**

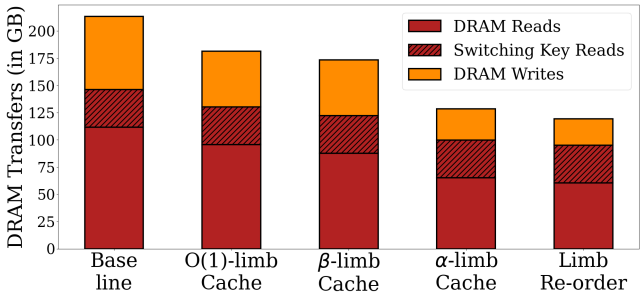

Figure 2: Cumulative impact of our caching optimizations when performing bootstrapping operation. Each successive optimization builds on top of the earlier ones. The baseline parameters are given in Table 5.

$O(1)$ limb optimization every time we perform the Rotate operation. In addition to the Rotate operation, the caching $O(1)$ limb optimization can also be applied to PtMult, Mult, and Conjugate.

Figure 2 summarizes the reductions in the DRAM transfers for the bootstrapping operation when incrementally applying the different caching optimizations that we have presented in this Section. With the $O(1)$-limb caching optimization, we observe a 15% reduction in overall DRAM transfers during a single bootstrapping operation. This reduction in data transfer includes a reduction in limb reads and writes by 14.3% and 23.7%, respectively.

Caching $O(\beta)$ Limbs: The next optimization considers a cache size that is $O(\beta)$, where β is the number of digits generated from a polynomial key switching operation. We refer readers to Han and Ki [19] for more details about the key-switching procedure. Note that Han and Ki [19] do not propose the optimization that we describe below. Our parameters where $\beta \leq$ dnum $= 3$ amount to 6 MB of cache. Below we explain the optimization idea using an example.

Consider the PtMatVecMult operation in bootstrapping, in which we need to perform many Rotate operations on a ciphertext. There are β digits that are produced as the output of the initial 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. Given that we have space in the cache for β digits, we can instead pull in a single limb from each of the β digits' output by the ModUp operation, then perform the Rotate operation on each limb and then calculate the inner product of limbs with the switching key limbs all at once. This allows us to

read the outputs of the ModUp function only once, regardless of the number of rotations. With this optimization, we can reduce the number of limb accesses from the main memory by at least 3× (as β equals 3) during key switching for the PtMatVecMult operation.

For a single bootstrapping operation, as we move from caching $O(1)$ limb optimization to caching $O(\beta)$-limbs optimization (see β-limb Cache in Figure 2), we have an additional 5 MB on-chip memory. Using this additional memory, compared to the $O(1)$-limb optimization we can reduce the number of DRAM transfers by ∼4.4% due to the 8.3% reduction in the number of ciphertext limb reads. In comparison to the baseline, with the $O(\beta)$-limbs optimization, we reduce the number of DRAM transfers by 22% due to the 21.4% reduction in the number of ciphertext limb reads. Note that the number of ciphertext limb writes remains the same here.

Caching $O(\alpha)$ Limbs: For this optimization, we assume that we have a large cache that can hold $O(\alpha)$ limbs, where α is the number of limbs in a single digit after the digit is output by the Decomp function for key switching. In practice, this optimization requires a cache size that is only slightly larger than the size of 2α limbs, which is about 27 MB (calculated using as 2α + 3 MB where $\alpha = \lceil L + 1/\text{dnum} \rceil = 12$ as $L = 35$ and $\text{dnum} = 3$). With this cache size assumption, we observe a significant decrease in the total number of ciphertext limb reads and writes from/to the main memory. This is because all of the *slot-wise* basis conversion operations in ModUp (line 2 in Algorithm 1) and ModDown operate over α limbs. If we can fit these α limbs in the cache, then we can generate new limbs in their entirety within the cache. With each new limb in the cache, we can perform the NTT on the limb, which completes the basis change operation and writes 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 back to main memory and then read them back from main memory in *limb-wise* format.

For a single bootstrapping operation, as we move from $O(\beta)$-limbs optimization to $O(\alpha)$-limbs optimization, we reduce the number of DRAM transfers by ∼26%, which includes a 26% and 44% reduction in the ciphertext limb reads and writes, respectively. In comparison to the baseline, we observe that with $O(\alpha)$-limbs optimization, DRAM transfers reduce by 44%, which includes a 42% and 57% reduction in the ciphertext limb reads and writes, respectively.

Re-ordering Limb Computations: The ModDown steps in the KeySwitch and bootstrapping drop α limbs. However, we need to perform additional operations on these α limbs before they get dropped in the ModDown step. In our re-ordering optimization, we propose computing these α limbs first so that the additional operations can be performed immediately. This re-ordering limb computation is feasible on top of $O(\alpha)$-limbs optimization as we have enough on-chip memory to cache these α limbs so as to avoid having to write and then read these limbs from main memory. Once we have the α limbs, we can begin the ModDown operation by computing the output of the NewLimb. Then, for each subsequent limb that is computed, we can immediately combine it with the NewLimb output, in turn avoiding more DRAM transfers.

For a single bootstrapping operation, as we move from $O(\alpha)$-limbs optimization to limb re-order optimization (see limb re-order in Figure 2), we observe that the DRAM transfers reduce by ∼7%, which includes a 7% and 16% reduction in the ciphertext limb reads and writes, respectively. In comparison to the baseline, we observe that with re-ordering limb optimization, DRAM data transfers reduce by 52%, which includes a 46% and 64% reduction in the ciphertext limb reads and writes, respectively.

After applying our caching optimizations, we observe that the bootstrapping arithmetic intensity changes from 0.72 to 1.25, which is an improvement by ∼1.7×. As the caching optimizations do not impact the switching key reads, they remain constant for all of the caching optimizations. However, they contribute to the total transfers from DRAM and thus impact the overall bootstrapping arithmetic intensity.

Algorithm 5 $\text{PModUp}([\mathbf{x}]_{\mathcal{B}}) \rightarrow [P \cdot \mathbf{x}]_{\mathcal{B} \cup \mathcal{B}'}$

1: $P \coloneqq \prod_{p \in \mathcal{B}'} p$ and $Q \coloneqq \prod_{q \in \mathcal{B}} q$
2: **for** q_i in $\mathcal{B}$ **do** $[\mathrm{y}]_{q_i} \leftarrow P \cdot [x]_{q_i} \mod q_i$
3: **for** $p_j \in \mathcal{B}'$ **do** $[\mathrm{y}]_{p_j} \leftarrow 0$
return $[\mathrm{y}]_{\mathcal{B} \cup \mathcal{B}'}$ ▷ $\mathbf{y} = P \cdot \mathbf{x} \mod PQ$

3.2 Algorithmic Optimizations

We now present the algorithmic optimizations for various operations in CKKS bootstrapping. These optimizations are all centered around the key idea of performing linear functions in the raised basis. Below we first describe this idea in detail, and then describe how we can apply this idea to several operations in CKKS bootstrapping.

Linear Functions in the Raised Basis: The KeySwitch operation (see Algorithm 3 in Section 2.2) changes the decryption key from $\mathbf{s}$ to $\mathbf{s}'$ by taking in an input polynomial $\mathbf{a}$ and producing a ciphertext $[\![\mathbf{a} \cdot \mathbf{s}]\!] \in \mathcal{R}_Q^2$. This ciphertext is produced by performing ModDown on the ciphertext $[\![P \cdot \mathbf{as}]\!] \in \mathcal{R}_{PQ}^2$. We note that an element $\mathbf{b} \in \mathcal{R}_Q$ can be efficiently converted into an element $P \cdot \mathbf{b} \in \mathcal{R}_{PQ}$. This is given in Algorithm 5, denoted by PModUp. This allows the canceling of the $\mathbf{a} \cdot \mathbf{s}$ term to occur over $\mathcal{R}_{PQ}$. If $\mathbf{m}$ is the message encrypted by the original ciphertext, this change leaves us with encryption of $P \cdot \mathbf{m}$, where the ciphertext is now in $\mathcal{R}_{PQ}^2$.

The key observation here is that this ciphertext remains additively homomorphic. This means that we can perform Add and PtMult in the raised basis before needing to perform a ModDown step. For any CKKS application where the inputs to a linear function are the immediate outputs of a KeySwitch operation, this optimization can be applied.

Merging ModDown in Mult: A conceptual view of the Merging ModDown in Mult optimization is presented in Figure 4. This optimization is a straightforward application of the above-introduced technique (PModUp operation). In the standard CKKS Mult implementation (see Figure 4 (a)), there is a KeySwitch followed by a Rescale operation (also known as ModDown operation) to reduce the plaintext scaling factor Δ. We can save ℓ multiplications per coefficient if we complete the KeySwitch operation in the raised basis by lifting Add above the first ModDown operation within KeySwitch (see Figure 4 (b)). Then we reduce the ciphertext polynomials by both P and the scaling factor Δ using a single ModDown operation (see Figure 4 (c)), in turn, reducing the number of orientation switches and the expensive NTT operations.

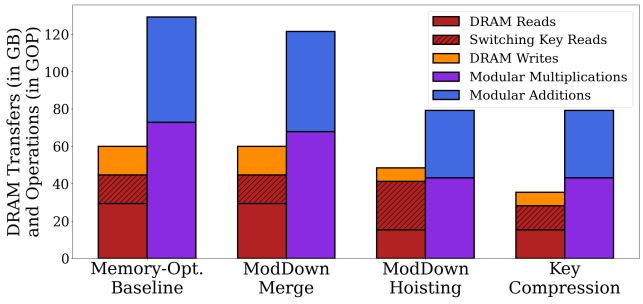

Figure 3: Cumulative impact of our algorithmic optimizations. Each successive optimization builds on top of the earlier ones. For the baseline case we apply all the memory optimizations from Section 3.1. For all designs we use the Best-case Parameters from Table 5.

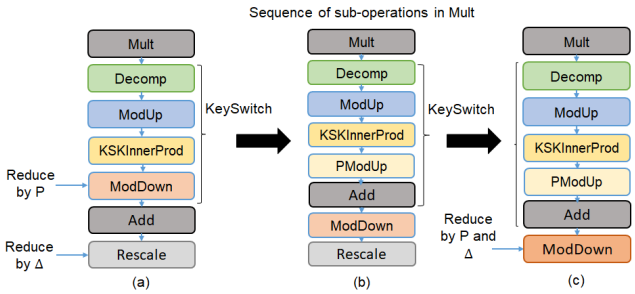

Figure 4: Conceptual view of merging ModDown **operation in** Mult**; (a) Standard sequence of sub-operations in** Mult **operation that requires us to perform two** ModDown **operations, (b) Modified sequence of operations in** Mult **that brings two** ModDown **operations adjacent by performing** Add **in raised basis, and (c) Optimized sequence of operations in** Mult **that performs single** ModDown**.**

We summarize the effects of various algorithmic optimizations in Figure 3, where the baseline includes the caching optimizations from the previous section. For a single bootstrapping operation, as we merge ModDown in Mult (see ModDown Merge in Figure 3), we observe that the overall compute reduces by 6%. However, the number of DRAM transfers remains the same as we have enough on-chip memory to perform the Rescale operation immediately following the ModDown operation on a limb without writing it back to the main memory.

Hoisting ModDown **in** PtMatVecMult**:** A conceptual view of Hoisting ModDown in PtMatVecMult optimization is presented in Figure 5. This optimization is a good example of how a linear function can be applied to many outputs of Rotate without performing a ModDown operation for each output. The PtMatVecMult function can be written as:

$$[\![\mathbf{y}]\!] \leftarrow \sum_i \text{PtMult}(\text{Rotate}([\![\mathbf{m}]\!], i, \text{ksk}), \mathbf{x}_i)$$

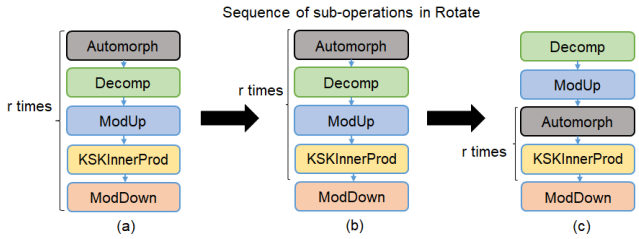

Figure 5: Conceptual view of hoisting ModDown **operation in** PtMatVecMult**; (a) Sequence of sub-operations performed** r **times in back-to-back** Rotate **operation within** PtMatVecMult**, (b) Modified sequence of sub-operations in** Rotate **where** ModDown **is hoisted to perform only single** ModDown **while performing many** Rotate **operations, and (c) Our** ModDown **hoisting optimization combined with an existing** ModUp **hoisting optimization to perform single** ModDown **and** ModUp **operations while performing back to back** Rotate **operations in** PtMatVecMult**.**

The naïve implementation of this operation is to complete the Rotate (where each Rotate contains a KeySwitch) on a ciphertext for each index i before multiplying it by the plaintext $\mathbf{x}_i$. We propose hoisting[3] the ModDown operation in KeySwitch whenever back-to-back Rotate operations are to be performed (as shown in Figure 5 (a)). Thus, using our optimization (as shown in Figure 5 (b)), we need to perform only two ModDown steps over this entire operation, which reduces the two elements in the ciphertext encrypting $\mathbf{y}$. Recall that every KeySwitch operation (see Algorithm 3) performs two ModDown operations.

In addition, when combined with the standard ModUp hoisting for the rotations [16, 22] (as shown in Figure 5 (c)), the number of RNS operations (either ModUp or ModDown) required by our PtMatVecMult is just three (one ModUp and two ModDown operations), *regardless* of the dimension of the matrix. More specifically, with this algorithmic optimization, the number of orientation switches reduces to 18 ($\text{fftIter} \times 3$, $\text{fftIter} = 6$ from Table 5) in our optimized PtMatVecMult as compared to 44 orientation switches required in the baseline algorithm [20]. The baseline algorithm uses the baby step giant step algorithm for FFT, wherein it performs an orientation switch for each baby step as well as each giant step, leading to a total of 44 orientation switches for their parameter set.

For a single bootstrapping, as we move from merge ModDown in Mult optimization to hoisting ModDown in PtMatVecMult optimization (see ModDown Hoisting in Figure 3), we observe that the overall compute reduces by 34% and the number of DRAM transfers for the ciphertext limbs reduce by 19%. We would like to note that the ModDown hoisting increases the number of DRAM access corresponding to additional switching key reads by 25%. This is because we use the ModDown hoisting optimization in the context of a baby step giant step algorithm that implements PtMatVecMult. The trade-off in this algorithm is that a larger baby step and a

[3] In FHE, if the same primitive operation is going to be performed r times successively, there may be an opportunity to perform one or more of the common sub-operations across the r instances upfront just once. This is referred to as hoisting.

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 as the baby step ciphertexts must be read in for each giant step. We chose the first option which, despite the increase in key reads, reduces the overall DRAM reads.

KeySwitch **Key Compression:** We briefly note one additional optimization unrelated to PModUp that is very simple but has a large impact on the number of DRAM transfers and also on the optimal CKKS scheme parameters. The first polynomial of all switching keys is a uniformly random ring element. Rather than transferring the entire polynomial (large ring element) from DRAM, we use a PRNG to generate this ring element. This allows us to only transfer the short PRNG key in place of the first switching key polynomial, which effectively reduces the KeySwitch key sizes by a factor of two. This is a folklore technique often used to reduce communication when sending ciphertexts or keys over a network. To the best of our knowledge, we are the first to use this optimization to reduce the memory bandwidth for hardware acceleration of FHE and the first to analyze this optimization alongside the other optimizations proposed in this section.

For a single bootstrapping operation, as we move from hoisting ModDown in PtMatVecMult optimization to key compression optimization (see Key Compression in Figure 3), we observe that the overall compute and the number of DRAM transfers for the ciphertext limbs remain unchanged. However, the number of DRAM transfers for the key reads reduce by 50%.

Broadly, as each algorithmic optimization gets applied, the number of compute operations and the DRAM transfers reduce. ModDown merge and ModDown hoisting optimizations reduce the compute more than they reduce the number of memory transfers, which reduces the overall arithmetic intensity of bootstrapping. However, as these two optimizations reduce the expensive NTT operations (having $O(NlogN)$ complexity), any reduction in the number of ModDown operations leads to a significant improvement in the compute performance, which improves the overall bootstrapping performance. We observe an overall 3× improvement in the bootstrapping arithmetic intensity with our final key compression optimization when compared to the baseline benchmark.

4 EVALUATION

In this section, we compare our work with prior art using the bootstrapping operation and two ML workloads.

4.1 SimFHE: Our Simulator

We begin by briefly describing SimFHE, our custom simulator that we use to benchmark the compute and memory requirements of a CKKS-based application. Any CKKS-based application consists of various complex operations, including the bootstrapping operation, that can be performed using the primitive operations listed in Table 2. SimFHE models these primitive operations by keeping track of both individual compute operations as well as the DRAM transfers for a given cache size. This enables us to benchmark the compute and memory requirements of individual complex operations as well as the CKKS-based application as a whole.

An execution of primitive operations and bootstrapping in SimFHE is parameterized by the CKKS scheme parameters, the number of functional units, the size of the on-chip memory, and the MAD optimizations to include in bootstrapping. SimFHE tracks compute at the modular arithmetic level, i.e., in terms of modular multiplications and additions. SimFHE tracks DRAM transfers based on the data size and the available cache size instead of directly tracking cache hits or misses through actual data reads. SimFHE implements our MAD optimizations in a modular fashion, allowing us to toggle between each optimization independently so as to isolate the benefit of each optimization. In addition, many of these optimizations are *memory-aware*, and so for a large enough on-chip memory, SimFHE will automatically deploy the applicable optimization.

SimFHE helps in optimal parameter selection by combining the simulation of algorithmic optimizations and hardware constraints. Given the on-chip memory size, SimFHE searches the CKKS parameter space using a brute-force approach to find the optimal parameters that maximize the bootstrapping throughput for a given system. The parameters computed by SimFHE include the high-level ring parameters such as the polynomial degree and coefficient modulus as well as the low-level, internal bootstrapping parameters such as the number of FFT iterations (fftIter) and the number of digits (dnum) in the KeySwitch operation. In prior works, the selection of parameters was quite opaque, and it was not clear how changing a specific CKKS algorithm parameter or system constraint such as on-chip memory size would affect the overall bootstrapping performance. With SimFHE, these questions can be immediately answered, which helps save significant design and development time. Moreover, the security level constraints limit the number of possible parameter sets, so the optimal parameter search takes only a few minutes.

Table 5: Baseline and Our Optimal Bootstrapping Parameters. Here L parameter = #limbs in the ciphertext after the initial ModUp **procedure in** Bootstrap**. The** fftIter **parameter is the** #PtMatVecMult **iterations in the** CoeffToSlot **and** SlotToCoeff **phases in** Bootstrap**.**

	n	q	L	dnum	fftIter
Baseline [20]	2^{16}	54	35	3	3
Ours	2^{16}	50	40	2	6

4.2 Bootstrapping Performance Comparison

To evaluate bootstrapping performance, we use the *bootstrapping throughput* metric given by Han and Ki [19]:

$$\text{throughput} = \frac{n \cdot \log Q_1 \cdot \text{bp}}{\text{brt}} \tag{3}$$

This metric attempts to capture the effectiveness of a bootstrapping routine by multiplying the number of slots the algorithm bootstraps (which is the number of plaintext slots n), the size of modulus $\log Q_1$ 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. The product is then divided by the runtime of the bootstrapping procedure, denoted as brt.

Optimal Bootstrapping Parameters: Using the throughput metric from Equation (3), we can select parameters that maximize the throughput. We employ SimFHE, with all our optimizations included, to explore the parameter space of bootstrapping to identify

Table 6: Bootstrapping comparison: $\log Q_1$ is the size of the modulus immediately after bootstrapping. The bit-precision achieved by all works is 19 except $F1$ which has a bit-precision of 24. All ASIC designs have a clock frequency of 1 GHz and their bootstrapping runtime is based on the simulation results. Throughput is computed using Equation (3). As different designs use a different number of plaintext slots (n) for bootstrapping, we compare bootstrapping throughput instead of bootstrapping runtime for a fair comparison.

Work	(N, $\log q$)	Modular Multiplier Count	On-chip Memory (MB)	Memory bandwidth	n	$\log Q_1$	Bootstrapping Runtime (in ms)	Throughput	Throughput (normalized wrt MAD)
GPU [20]	2^{17}, 54	-	6	900 GB/s	2^{16}	1080	328.7	409	0.1361×
MAD	2^{17}, 54	2250	32	900 GB/s	2^{16}	950	39.35	3006	
F1 [30]	2^{14}, 32	18432	64	1 TB/s	1	416	1.3	1.5	0.0005×
MAD	2^{17}, 54	18432	32	1 TB/s	2^{16}	950	40.6	2910	
BTS [25]	2^{17}, 50	8192	512	1 TB/s	2^{16}	1080	50.43	2667	1.7178×
MAD	2^{17}, 54	8192	32	1 TB/s	2^{16}	950	76.2	1552	
ARK [24]	2^{16}, 54	20480	512	1 TB/s	2^{15}	432	3.9	6896	2.1326×
MAD	2^{17}, 54	20480	32	1 TB/s	2^{16}	950	36.58	3234	
CraterLake [31]	2^{17}, 28	14336	256	2.4 TB/s	2^{16}	532	6.33	10465	4.6248×
MAD	2^{17}, 54	14336	32	2.4 TB/s	2^{16}	950	52.2	2263	

parameters that maximize the throughput when we have 32 MB on-chip memory (all our optimizations when applied together require a minimum 27 MB on-chip memory). As DRAM transfer times dominate in bootstrapping, our functional model in SimFHE 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 3 and Section 3.2) are given in Table 5. We would like to note that any bootstrapping implementation must specify the values for all parameters in Table 1. For brevity, we list the important parameters in Table 5. The unlisted parameters in Table 1 can be determined by the ones we list in Table 5. This is also true for all the other works listed in Table 6.

Bootstrapping Performance: To estimate the runtime for bootstrapping when using our MAD techniques, we first use SimFHE to determine the total number of operations and the total number of DRAM transfers when we have 32 MB on-chip memory. For each comparison in Table 6, we estimate the compute latency by using the total number of operations, an operating frequency of 1 GHz (same as the ASIC designs in related works), and by accounting for the number of operations that can be done in parallel (using the modular multiplier count listed in Table 6). For each comparison, we determine the memory access latency for MAD using the memory bandwidth of the corresponding related work. For the related works, the bootstrapping runtime shown in Table 6 is the bootstrapping runtime presented by the authors in their respective papers. As the authors in [20, 24, 25] mentioned that their parameters are similar to [4], which has a bit precision of 19 bits, we assume these works also have a bit precision of 19 bits.

In Table 6 we compare the throughput of the bootstrapping algorithm with all our caching and algorithmic optimizations to the prior art. For GPU implementation, we use the parameter set used by Jung et al. [20] for bootstrapping and logistic regression. F1 primarily focuses on smaller parameter sets ($N = 2^{14}$) where full ciphertexts fit in on-chip cache memory, allowing them to bypass the memory bandwidth limitation. F1 has the lowest bootstrapping throughput because it only implements "unpacked" CKKS bootstrapping, where the ciphertext only holds one element. For CraterLake [31], we use the bootstrapping parameter set from their paper that gives 128-bit security. We select the parameter set for BTS-2 from BTS [25] as it provides the highest bootstrapping throughput for the BTS design. ARK [24] uses a single parameter set for its evaluation and we use that in our evaluation. We carefully modeled all these designs in SimFHE and added our caching and algorithmic optimizations to these designs. Applying our MAD techniques to these designs improves their bootstrapping throughput.

Compared to the original GPU implementation [20] and F1 design, adding our MAD optimizations provides ~7× and ~2000× higher bootstrapping throughput, respectively. This is because GPU and F1 designs are memory-bound, indicative of the fact that increasing the cache size in these designs can help improve the bootstrapping throughput. In contrast, after adding our MAD optimizations on the original BTS, ARK, and CraterLake designs, the bootstrapping throughput reduces by ~3×. This is because after applying our MAD optimizations these three designs become compute-bound, and cannot take advantage of the large on-chip memory (>256 MB) that is available in the original designs. In fact, any increase in the on-chip memory beyond 32 MB does not improve the bootstrapping throughput. So we need to increase the compute throughput by 2× in BTS, 1.05× in ARK, and 3.5× in CraterLake to generate a balanced design. As discussed in Section 1, the ASIC designs can be prohibitively expensive. So applying our MAD techniques on top of the original ASIC designs can help reduce their cost as they would need smaller chips due to the smaller 32 MB on-chip memory instead of the larger 256 MB and 512 MB on-chip memory. In Section 4.4, we discuss the performance vs. area/cost

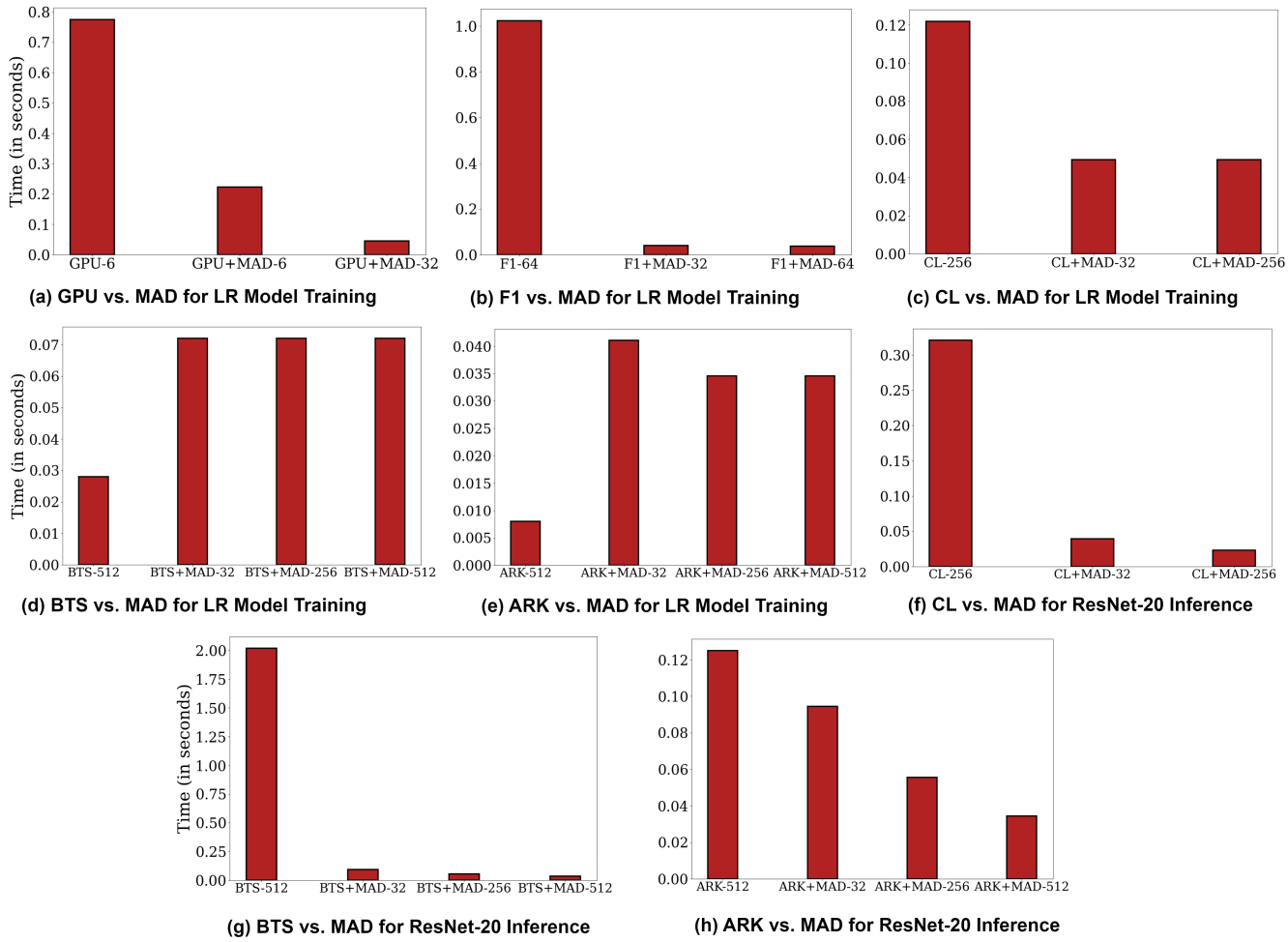

Figure 6: LR model training and ResNet-20 inference performance comparison. For the first bar in each sub-figure, we use the following notation: name of the original design-cache size. For example, GPU-6 refers to the original GPU design with 6 MB cache. For the remaining bars, we use the following notation: name of the original design+MAD-cache size. For example, GPU+MAD-6 refers to the original GPU design with our MAD optimizations using a 6 MB cache.

trade-offs associated with adding our MAD optimizations on top of the existing designs.

4.3 ML Application Performance Comparison

Here we compare the performance of encrypted logistic regression (LR) model training and ResNet-20 inference using our MAD optimizations (see Figure 6). In each of the sub-figure, the first bar reports the time reported in the respective paper of GPU, F1, CraterLake (CL), ARK, and BTS, and the remaining bars correspond to the time when we apply MAD techniques on top of the original designs. For generating the data for the remaining bars, we carefully modeled each one of the original designs in SimFHE and then added our MAD optimizations on top.

For LR training, we implement the HELR algorithm [18] in SimFHE using all our optimizations and the parameters in Table 5. With our optimal parameter set (see Table 5), we need to perform bootstrapping after every three training iterations. The ResNet-20 inference is based on the ResNet-20 CNN design by Lee et al. [27]. We perform inference on encrypted images (one at a time) from the CIFAR-10 dataset. We adopt the same approach as that for determining bootstrapping runtime (described in Section 4.2) to generate the LR model training time and ResNet-20 inference time.

LR Training: When compared to the GPU implementation, the GPU+MAD-6 and GPU+MAD-32 need 3.5× and 17× less training time, respectively, (see Figure 6 (a)). Note that with 6 MB cache size limitation, we can only apply β-limb caching optimization for GPU+MAD-6. When compared to the F1 ASIC design, F1+MAD-32 and F1+MAD-64, we can reduce the training time by ∼25× and ∼27×, respectively, (see Figure 6 (b)). The original F1 design is memory-bound, but we do not observe a significant difference in its performance between 32 MB to 64 MB on-chip memory. This is

because we observe only a 10% difference in total data transfers for the two cache sizes.

In the case of the CraterLake design, with both CL+MAD-32 and CL+MAD-256, we can reduce the training time by 2.5× (see Figure 6 (c)). The similar improvement in performance for both cases is due to the fact that applying MAD optimizations makes the CraterLake design compute bound. In the case of the BTS design, when we apply our MAD optimizations, the resulting BTS+MAD-512 design becomes compute bound and we observe an increase in the training time by 2× (see Figure 6 (d)). This compute-boundedness can be evidenced by the fact that decreasing the on-chip memory of the BTS+MAD-512 design to 256 MB and 32 MB does not reduce the training time (see Figure 6 (d)). In the case of the ARK design, similar to BTS, applying our MAD optimizations makes it compute bound. So for ARK+MAD-512, we see a 4× increase in training time (see Figure 6 (e)). When we decrease on-chip memory of ARK+MAD-512 design to 32 MB, we see an increase in training time. This is because the smaller on-chip memory leads to a 20% increase in the number of DRAM transfers.

ResNet-20 Inference: In the case of the CraterLake design, with CL+MAD-32 and CL+MAD-256, we can reduce the inference time by 8× and 13×, respectively, (see Figure 6 (f)). The improvement in performance for both on-chip memory sizes is due to the fact that applying MAD optimizations makes the CraterLake design compute bound by reducing the data transfers more than the compute. In the case of the BTS design, with BTS+MAD-32, BTS+MAD-256, and BTS+MAD-512, we can reduce the inference time by 21×, 36×, and 57×, respectively, (see Figure 6 (g)). Similarly, in the case of the ARK design, with ARK+MAD-32, ARK+MAD-256, and ARK+MAD-512, we observe a reduction in the inference time by 1.3×, 2.2× and 3.6×, respectively, (see Figure 6 (e)). For both ARK and BTS designs, we observe a reduction in the total amount of data transfers from the main memory by 2.7× as we move from 32 MB to 512 MB on-chip memory.

We would like to note that the increase (c.f. decrease) in bootstrapping throughput does not necessarily equate to a decrease (c.f. increase) in performance. This is because we compute the bootstrapping throughput for a fully-packed bootstrapping, while for the applications, we utilize bootstrapping implementation with fewer ciphertext slots as described in HELR [18] and ResNet-20 inference [27].

4.4 Performance vs. Area/Cost Tradeoffs

The primary strategy of prior works to mitigate the CKKS memory bottleneck is to have a large on-chip memory. For example, CraterLake uses 256 MB, and BTS and ARK use 512 MB on-chip memory. A large on-chip memory results in a large chip area, which equates to an expensive solution. Our analysis in the previous section reveals that in some cases applying MAD techniques on top of the existing ASIC designs can improve performance as well as reduce the required size of the on-chip memory, which reduces the chip area and in turn the cost. Essentially, we end up in a win-win situation. In other cases, applying MAD techniques with small on-chip memory leads to performance loss. In these cases, we need to understand the tradeoff between performance and area/cost, and then depending on the performance and area/cost specifications, choose which MAD optimizations to apply.

5 RELATED WORK

Recently, Jung et al. [20] 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.

Samardzic et al. [30] presented the first architecture of a programmable 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 bootstrapping and machine learning require larger parameter sets that are not addressed in [30]. A follow up ASIC design from Samardzic et al. [31] addresses the issue of supporting large parameters and implements packed bootstrapping. From their performance analysis, it is evident that the compute block is underutilized ($\sim$ 40% utilization during bootstrapping) due to memory bottleneck.

Recent ASIC proposals by Kim et al. [24, 25] implement fully packed bootstrapping using 7nm technology. Their analysis also shows that the FHE operations are memory bottlenecked, which they mitigate by employing a large register file (>22MB) and a massive amount of on-chip memory (512MB). Notably, ARK [24] includes algorithmic optimizations to significantly reduce the memory bandwidth of CKKS bootstrapping by reducing the number of switching keys required in the homomorphic FFT evaluation. This optimization crucially relies on a large on-chip memory (at least a few hundred MB), but this improvement seems to completely address the memory bottleneck issue; to our knowledge, it is the only balanced hardware acceleration of CKKS bootstrapping.

6 CONCLUSION

In this paper, we performed a thorough architecture-level analysis of the compute and memory requirements for the CKKS FHE scheme to identify the limits and opportunities for hardware acceleration of CKKS bootstrapping. Our analysis shows that the bootstrapping step, the critical performance bottleneck in FHE-based computing, has low arithmetic intensity and is heavily constrained by the main memory bandwidth. We proposed several memory-aware design optimizations that are agnostic of the underlying hardware. Our optimizations improve bootstrapping throughput and at the same time reduce the on-chip memory requirements. Applying our MAD optimizations for FHE improves bootstrapping arithmetic intensity by 3×. For Logistic Regression training, by leveraging our MAD optimizations, the existing GPU design can get up to 3.5× improvement in performance for the same on-chip memory size. Similarly, the existing ASIC designs can get up to 27× and 57× improvement in performance for Logistic Regression training and ResNet-20 inference, respectively, while reducing the on-chip memory requirement by 16×, which proportionally reduces the cost of the solution.

ACKNOWLEDGMENTS

This research was supported in part by DARPA under Agreement No. HR00112020023, NSF CNS-2154149, a grant from the MIT-IBM Watson AI, and a Thornton Family Faculty Research Innovation

Fellowship from MIT. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA. This research was also partially funded by the NSF CNS CSE 2312276 grant.

REFERENCES

[1] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 882–895.

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

[3] VA Bespalov, NA Dyuzhev, and V Yu Kireev. 2022. Possibilities and Limitations of CMOS Technology for the Production of Various Microelectronic Systems and Devices. *Nanobiotechnology Reports* 17, 1 (2022), 24–38.

[4] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In *Advances in Cryptology – EUROCRYPT 2021*. Springer International Publishing, Cham, 587–617.

[5] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 868–886.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS '12*.

[7] Zvika Brakerski and Vinod Vaikuntanathan. 2011. 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*, Rafail Ostrovsky (Ed.). IEEE Computer Society, 97–106. https://doi.org/10.1109/FOCS.2011.12

[8] Hao Chen, Ilaria Chillotti, and Yongsoo Song. 2019. Improved Bootstrapping for Approximate Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2019*, Yuval Ishai and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 34–54.

[9] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 360–384.

[10] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A Full RNS Variant of Approximate Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2018*, Carlos Cid and Michael J. Jacobson Jr. (Eds.). Springer International Publishing, Cham, 347–368.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[12] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115.

[13] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://ia.cr/2012/144.

[14] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford university.

[15] Shai Halevi, Yuriy Polyakov, and Victor Shoup. 2019. An Improved RNS Variant of the BFV Homomorphic Encryption Scheme. In *Topics in Cryptology – CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, Proceedings*, Mitsuru Matsui (Ed.). Springer Verlag, Germany, 83–105. https://doi.org/10.1007/978-3-030-12612-4_5

[16] Shai Halevi and Victor Shoup. 2017. Presentation at the Homomorphic Encryption Standardization Workshop.

[17] Kyoohyung Han, Minki Hhan, and Jung Hee Cheon. 2019. Improved Homomorphic Discrete Fourier Transforms and FHE Bootstrapping. *IEEE Access* 7 (2019), 57361–57370. https://doi.org/10.1109/ACCESS.2019.2913850

[18] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic Regression on Homomorphic Encrypted Data at Scale. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 9466–9471. https://doi.org/10.1609/aaai.v33i01.33019466

[19] Kyoohyung Han and Dohyeong Ki. 2020. Better Bootstrapping for Approximate Homomorphic Encryption. In *Topics in Cryptology – CT-RSA 2020*, Stanislaw Jarecki (Ed.). Springer International Publishing, Cham, 364–390.

[20] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 4 (Aug. 2021), 114–148. https://doi.org/10.46586/tches.v2021.i4.114-148

[21] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Keewoo Lee, Namhoon Kim, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. 2020. HEAAN Demystified: Accelerating Fully Homomorphic Encryption Through Architecture-centric Analysis and Optimization. *arXiv preprint arXiv:2003.04510* (2020).

[22] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*. 1651–1669.

[23] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. 2017. Moonwalk: Nre optimization in asic clouds. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 511–526.

[24] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. *arXiv preprint arXiv:2205.00922* (2022).

[25] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2021. BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption. *arXiv preprint arXiv:2112.15479* (2021).

[26] SO Kuyoro, F Ibikunle, and O Awodele. 2011. Cloud computing security issues and challenges. *International Journal of Computer Networks (IJCN)* 3, 5 (2011), 247–255.

[27] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. 2022. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access* 10 (2022), 30039–30054.

[28] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010*, Henri Gilbert (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23.

[29] R L Rivest, L Adleman, and M L Dertouzos. 1978. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation, Academia Press* (1978), 169–179.

[30] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3466752.3480070

[31] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 173–187.