Optimizing Ciphertext Management for Faster Fully Homomorphic Encryption Computation

1st Eduardo Chielle
New York University Abu Dhabi
0000-0002-1938-912X

2nd Oleg Mazonka
New York University Abu Dhabi
0000-0001-5131-9044

3rd Michail Maniatakos
New York University Abu Dhabi
0000-0001-6899-0651

***Abstract*—Fully Homomorphic Encryption (FHE) is the pinnacle of privacy-preserving outsourced computation as it enables meaningful computation to be performed in the encrypted domain without the need for decryption or back-and-forth communication between the client and service provider. Nevertheless, FHE is still orders of magnitude slower than unencrypted computation, which hinders its widespread adoption. In this work, we propose Furbo, a plug-and-play framework that can act as middleware between any FHE compiler and any FHE library. Our proposal employs smart ciphertext memory management and caching techniques to reduce data movement and computation, and can be applied to FHE applications without modifications to the underlying code. Experimental results using Microsoft SEAL as the base FHE library and focusing on privacy-preserving Machine Learning as a Service show up to 2x performance improvement in the fully-connected layers, and up to 24x improvement in the convolutional layers *without any code change*.**

***Index Terms*—fully homomorphic encryption, performance improvement, memory management**

I. Introduction

With the proliferation of cloud computing and the increasing amount of data generated by individuals and organizations, there is a growing need for privacy-preserving outsourced computation. Outsourcing computation to cloud providers can be cost-effective and provide flexibility. Furthermore, a client may want to use the service of a third-party to process its data. In many cases, sensitive data must be shared with third-party providers to perform computation, which creates a risk of data breaches and unauthorized access. Additionally, data privacy regulations and laws, such as GDPR and CCPA, impose strict requirements on how personal data is handled and stored.

While AES and RSA are commonly used to secure data at rest or in transit, decryption is still required for processing. One possible solution to the problem of data processing is to use a Trusted Execution Environment (TEE) which moves the root of trust to lower levels of the computation stack. However, the idea that the hardware is the ultimate root of trust has been challenged by several attacks [1]–[3]. Cryptographic solutions such as Multi-Party Computation (MPC) and Fully Homomorphic Encryption (FHE) remove the need for hardware trust. MPC requires continuous communication between parties, which may not be ideal when communication bandwidth is not available. FHE, on the other hand, works on a *fire-and-forget* manner; thus the client only needs to interact at the beginning, to send input data, and at the end of computation, to collect the output.

FHE is a special type of encryption that enables computation to be performed in the encrypted domain. Since its inception in 2009 [4], several FHE schemes have been proposed, namely BGV [5], BFV [6], CGGI [7], and CKKS [8]. Each scheme has its own strengths and weaknesses, and their suitability depends on the specific application requirements. While tremendous progress has been made in the last decade to improve the performance of FHE computation, it is still slower than unencrypted computation up to being impractical for certain applications.

There are several factors contributing to this lower performance. First, ciphertexts are polynomials having 4 to 65 thousands coefficients where each coefficient can be of several hundred bits. Hence, even handling ciphertexts in the program is an expensive part of computation. Second, the number of allowed operations on ciphertexts is limited to addition, subtraction, multiplication, and automorphism. This forces non-linear operations on ciphertexts to be typically approximated by high-order polynomials. Third, the multiplication of ciphertexts is not a trivial operation. For efficiency, it requires conversion of the underlying polynomials into an evaluation form using Number Theoretic Transformation (NTT). Finally, programs operating on ciphertext must run in a *data oblivious* manner (no branching on encrypted data). This also requires redesigning algorithms that makes computation even less efficient. Given all the above, any improvement in performance of FHE computation is highly desirable.

Contribution: In this work, we propose and develop Furbo, a framework that optimizes ciphertext management without user input, improving the performance of FHE without modifications to the source code. Furbo employs non-invasive techniques that reduce the computational complexity of both ciphertext handling and operations. Furbo can be used by any FHE library and is employed without any functional modification.

Without loss of generality, we perform experiments on private inference workloads. Using Microsoft SEAL as the underlying FHE library, experimental results show up to $2\times$ and $24\times$ performance improvement for fully-connected and convolutional layers, respectively.

II. Preliminaries

Current FHE schemes define addition (and subtraction) and multiplication on finite fields [5], [6]. Plaintexts are usually encoded into polynomials over finite fields. In the BFV scheme, the polynomial basis is chosen to be the powers of a primitive element of the finite field. The plaintexts values are converted

979-8-3503-4859-0/DATE24/© 2024 EDAA

into a polynomial that evaluates to these values at specific points generated by the root of unity in the field defined by modulus t. This ensures that addition and multiplication of polynomials match the corresponding addition and multiplication of plaintexts.

The encoded plaintext can be encrypted by adding some random noise to the polynomial coefficients. The encrypted polynomial is then used to perform homomorphic computations on the encrypted data within modulus q, where $q \gg t$. Each BFV encryption ciphertext consists of two polynomials. Typical ciphertext sizes range from 1.5 to 64 MB [9]. When ciphertexts are added or subtracted, the operation is performed separately on each polynomial, whereas when ciphertexts are multiplied the polynomials get cross multiplications with a subsequent relinearization operation to bring the resulting three-polynomial ciphertext back into a two-polynomial ciphertext.

III. Methodology

Furbo is a framework that can be instantiated using any underlying FHE library (e.g. SEAL [9], HELib [10]) and can act as middleware between FHE compilers (e.g. Eva [11], Ramparts [12]) and FHE libraries. It can also be instantiated directly in the user code if the programmer uses directly the underlying libraries. At its core, Furbo includes two major components: an Automatic Ciphertext Reference Module (ACRM) and an Operation Optimization Module (OOM).

A. Automatic Ciphertext Reference Module

As ciphertexts are large objects and occupy big chunks of memory, a computer program consumes significant amount of computing resources even without actual computation, that is operations on ciphertexts. The program needs to create objects in the stack, heap or global memory; copy and move them around; and finally destroy them. For small objects like the native data types this object management is fast and efficient, but this is not the case for huge objects like ciphertexts.

To improve computational efficiency, the concept of abstract data type has been introduced. It simulates the behavior of an object by replacing the object with its reference. In simple words, instead of copying or moving objects, we can mark in a special register structure that the object is being copied or moved without actual copy or move. This extra work introduces some overhead; however, the overhead is negligible comparing to actually handling big objects. When the program is being compiled, the compiler generates the code for automatic creation, copy, and destruction of objects that temporarily appear in computational expressions. This often happens outside of the programmer control and can lead to degraded performance when objects are not small. In our proposal, we replace ciphertexts with their unique identifiers (`id`) that are just small numbers. Consider the following snippet:

```
c = a;
...
a = a+b;
```

In the first expression, the entire content of `a` is copied to `c`, effectively becoming two ciphertexts in memory with the same value. Note, the example above is not an example of poor coding. The compiler creates temporary variable in expressions and in passing arguments to functions. Thus, preventing this type of copies from happening has potential of improving performance and reducing memory consumption. To make it especially efficient for ciphertexts we introduce ACRM – a module managing handling and access to the underlying ciphertexts. ACRM includes a standard mechanism of reference counters to the native ciphertexts. So the program can safely use the ciphertext `ids` instead of actual ciphertexts by just replacing the ciphertext type declaration. With ACRM, both `a` and `c` refer to the same object, preventing allocation of memory for a new object. Only if one of the variables is modified later on, a new object must be created. ACRM counts the number of references in the program and updates the count as required. In the example above, the last statement would result in two actions by ACRM: 1) reduction of the counter of the old value of `a`, and 2) creation of a new `id` for the newly created ciphertext - the result of the operation. If the counter goes to zero, there are no references to the ciphertext in the program, and the value can be removed from ACRM.

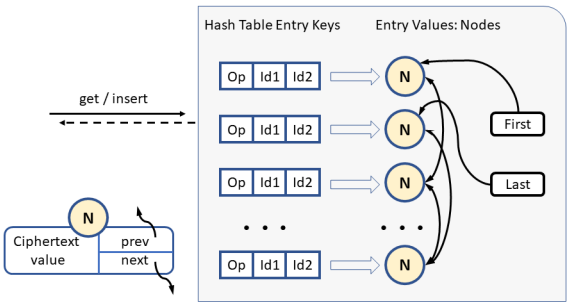


Fig. 1: OOM organization diagram. Hash table entries are the keys: the ciphertext operation `Op`, and two ciphertext `ids`; and the values, which are nodes of a linked list. OOM has member pointers to the First and the Last elements. The entry value (node) consists of a ciphertext value and the two list element pointers, bottom-left figure.

B. Operation Optimization Module

The purpose of ACRM is to make ciphertext handling fast including inside OOM. OOM serves as a mechanism to avoid repeated ciphertext operations by storing and retrieving the result values of ciphertext operations. Its organization is shown in Fig 1. OOM consists of a hash table, where entries are arranged in a doubly linked list. Also it has two pointers (*First* and *Last*) to the first and the last elements in this list. Each table entry has three elements: the ciphertext operation name, and two numbers representing either ciphertext `id` or a constant. There are six distinct possible operations appearing in the table entry: Three operation types (addition, subtraction, multiplication) vs two operand types (ciphertext-ciphertext, ciphertext-plaintext). Subtraction must differentiate between ciphertext-plaintext and plaintext-ciphertext operations as subtraction is not commutative. However, ciphertext-plaintext subtraction can be replaced with ciphertext-plaintext addition since the second

Algorithm 1 OOM::get

Input: Key ▷ operation and arguments: {Op, $\texttt{id}_1$, $\texttt{id}_2$}
Output: Ct or $null$ ▷ Ciphertext or nothing
Entry structure: { Key, Node }
Node structure: { value, prev, next }

1: $i \leftarrow$ find(Key) ▷ search in hash table
2: **if** $i = null$ **then return** $null$
3: $Ct \leftarrow i$.value ▷ result is found in OOM: set return value
4: **if** i.prev $\neq null$ **then** i.prev.next $\leftarrow i$.next
5: **else** First $\leftarrow i$.next
6: **if** i.next $\neq null$ **then** i.next.prev $\leftarrow i$.prev
7: First.next $\leftarrow i$
8: i.next $\leftarrow null$
9: i.prev $\leftarrow$ First
10: First $\leftarrow i$ ▷ push found entry to the back of the list
11: **return** Ct

operand is a plaintext and can easily be converted to its negation.

In particular, the ciphertext-plaintext operations are of special importance as they cannot be optimized during compilation time since the value of the plaintext, usually loaded through serialization, is not known. Thus, techniques such as Common Subexpression Elimination, Constant Folding, or Global Value Numbering cannot be applied. Nevertheless, plaintext values are available during computation; therefore, we can apply similar optimizations dynamically using our OOM. As an example, consider the following vector-matrix product between an encrypted vector and a plaintext matrix:

$$\begin{bmatrix} c_1 & c_2 \end{bmatrix} \cdot \begin{bmatrix} 4 & 1 & 4 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 4c_1 + 2c_2 & c_1 & 4c_1 & 2c_2 \end{bmatrix}$$

Operations $4 \cdot c_1$ and $2 \cdot c_2$ occur more than once. By storing and remembering these operations dynamically, the repeated computation can be avoided, as well as computing $1 \cdot c_1$, $0 \cdot c_1$, and $0 \cdot c_2$. Ciphertext c_1 can be returned when multiplied by 1 and a pre-computed encryption of zero when multiplied by zero. Zero-value weights are common in convolutional layers when the weights are quantized [13]. Note that to avoid leaking information about the model from the service provider to the client, a noisy encryption of zero must be added to the outputs.

The entries in the table consist of their keys (shown in Fig. 1) and values. Each value is an ACRM's `id` reference to the ciphertext equal to the result of the operation. On every ciphertext operation a specific function (operator) is called. Inside this function a triplet of ciphertext operation values (i.e. operation name and two operands `ids`) is created. At this point OOM checks if the operation result is available in the table, and if so, returns the result. Otherwise, the ciphertext operation is performed, the result is inserted into OOM, and finally the value of the result (`id`) is returned from the function. New entries are inserted into OOM during normal program execution. Insertion of a new entry into the table takes place if it does not exist in OOM and the ciphertext operation is performed. We restrict the infinite growth by introducing a size limit to the table.

Organizing a doubly linked list needs additional metadata: two pointers to the previous and the next elements of the list. This is implemented by encapsulating the ciphertext values (actually, their `ids`) into a *node* structure. An important aspect of managing the entries in OOM is rearranging the entries in the table upon access of the elements. An insertion of a new entry is a simple OOM operation requiring push to the front of the linked list, optional removal of the last list element, and update First and Last pointers. This is performed every time a new ciphertext is computed.

On the other hand, if the ciphertext operation result is already present in the OOM, the computation is not needed; the retrieval simply moves the node to the beginning of the linked list. We note that moving nodes in the linked list has negligible performance overhead as it consists of simply reassigning a few reference pointers. The list represents the order of its elements from newest to oldest. Whenever an element is removed from OOM it must be the oldest, i.e. pointed by Last. Access to a particular ciphertext operation entry forces to update the entry as being the newest. OOM function `get`, presented in Algorithm 1, checks if the value is in the OOM (lines 1-3), i.e. has been previously computed. Then it moves the accessed entry to the front of the list. This is done by changing the pointers in the linked list so the access entry becomes the first element in the list, and its internal place in the list is extracted (lines 4-10). These updates are fast and do not involve any copies of neither elements of the linked list, entries, nor ciphertext values.

IV. Experimental results

In this section, we present experimental results for two common layers used in machine learning, dense and convolution, and for a complete convolutional neural network, Cryptonets [14]. We use a polynomial degree $N = 2^{13}$ as it is the smallest N that can run Cryptonets with a security level of 128 bits. Regarding the plaintext modulus t, we use the smallest t that enables batching and prevents overflows. It does not affect the performance of the dense and convolution experiments since the multiplicative depth of a layer is shallow. In addition, we use 8 bits of precision for inputs and weights in the dense and convolution experiments. In the case of Cryptonets, we use 6 bits since we observed that adding extra bits of precision does not increase accuracy.

We perform our experiments on a single thread Intel Xeon Silver 4214 with 448GB DDR4 running on Ubuntu 20.04 LTS. Our proposal is instantiated on top of SEAL. In each case study, we compare 1) a baseline implementation written using SEAL, 2) Furbo with ACRM-only enabled, and 3) Furbo with both ACRM and OOM enabled.

A. Results for Fully-Connected Layer

The fully-connected layer consists of a matrix multiplication followed by a nonlinear activation function. Since in CNNs the fully-connected layer is the last layer, it is common for its activation function to be dropped during privacy-preserving inference as it is monotonic and does not affect the arguments of the maxima. Therefore we evaluate its linear part. We follow the standard of multiplying an encrypted input matrix by a plaintext weight matrix [11], [14]–[18]. Also, it is common practice to pack several plaintexts into a ciphertext in FHE for performance. There are several packing techniques that

can be employed for matrix multiplication, some focusing on throughput [14] and others focusing on latency [15], [18]. We employ the former since it is used by Cryptonets.

Fig. 2 presents the experimental results of the matrix multiplication for several dimensions $d \in \{2^i \ \forall i \in \mathbb{Z} : i \in \{8, 10\}\}$. These dimensions have been chosen as they match commonly used datasets such as CIFAR-100 or ImageNet [19]. Our goal is to achieve the highest performance (i.e. lowest execution time). As discussed earlier, since the weight matrix is plaintext [11], [14], [15], [17], [18], there are only two types of operations: ciphertext-plaintext multiplication (`pmul`) and ciphertext-ciphertext addition (`cadd`). Thus, we define two instances of OOM, one only for ciphertext-plaintext operations, denoted as `pmul`, and another for both type of operations, denoted as `pmul,cadd`. We evaluate each instance of OOM with different table size limits $\in [2^0, 2^{21}]$. In addition, we also evaluate our ACRM without OOM (tag `ACRM-only`) and compare to the baseline.

The first observation is that Furbo with just ACRM enabled is always faster than baseline, approximately 20% for any matrix dimension. This is attributed to the reduction of ciphertext copies by nearly half. Regarding Furbo with OOM enabled, we observe additional performance improvement, which becomes more significant for larger matrices. When using larger matrices, there is already a high number of processor cache misses in the baseline approach. This reduces the relative performance degradation for large memory utilization. In addition, there still is the impossibility of using in-place ciphertext operations with OOM. This mainly affects the `pmul,cadd` setting because the number of `cadd` operations does not decrease and, thus, does not compensate for the extra memory allocation. On the other hand, the number of `pmul` operations reduces by nearly half for $d = 1024$, which results in a significant performance improvement (up to $2\times$).

B. Results for Convolutional Layer

We evaluate the performance of our methodology applied to the convolutional layer. Fig. 3 shows experimental results for different setups where we vary the number of output channels and filter size $(|c_o|, |f|) \in \{(8, 3), (32, 7)\}$. Usual number of output channels range from a few to hundreds and filter sizes range from 3 to 11 [19]. Our goal is to understand how our proposal scales for larger convolutional layers. We set the dimensions of the input matrix to the low end of common input image sizes (32×32) as it allows us to explore a larger number of output channels and filter sizes. For all cases, we select a stride equal to one. Similarly to the dense layer, the convolutional layer employs only two types of homomorphic operations `pmul` and `cadd`. Thus, we use the same settings for the data types in this experiment.

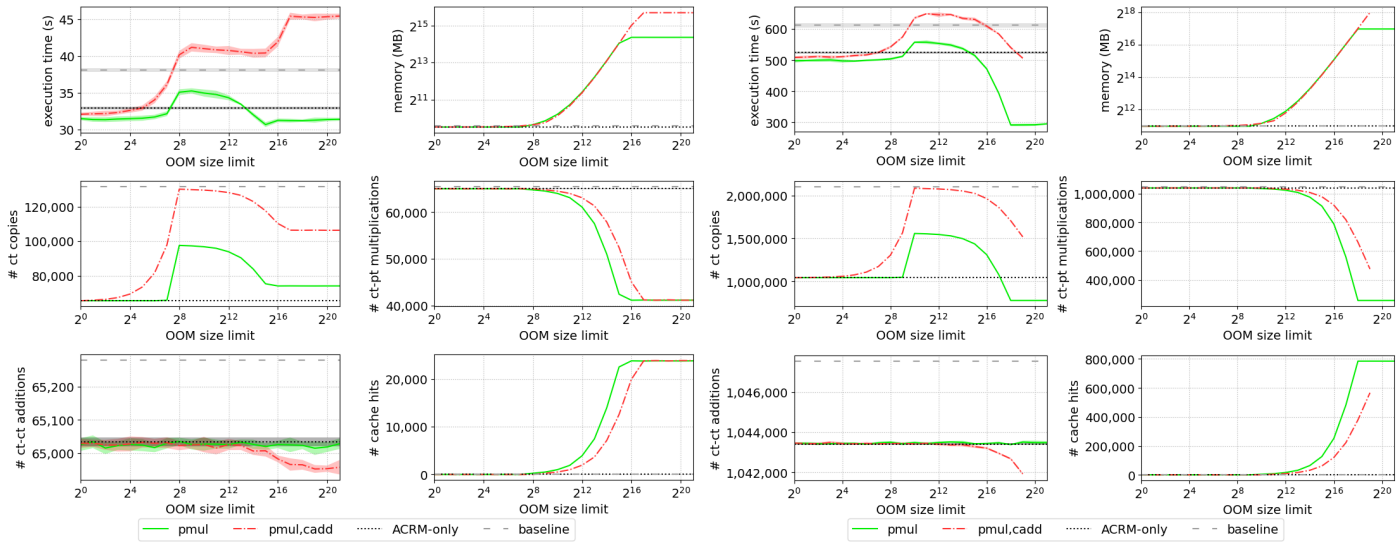
In Fig. 3, we observe that ACRM-only is similarly always faster than the baseline for the same reasons previously stated. Furthermore, we observe that OOM is extremely effective for the convolutional layer. We notice that the `pmul` setting is 2 to $3\times$ faster than the baseline, while the `pmul,cadd` version reduces the execution time by one to three orders of magnitude. This is explained by the fact that the number of both `pmul` and `cadd` operations reduces by several orders of magnitude when using OOM. Furthermore, the greater the number of output channels or the larger the filter size, the more prominent the reduction. In addition, the number of ciphertext copies is lower even when compared to ACRM-only. This is a consequence of the high re-utilization of results, which reduces the number of actual ciphertexts in memory.

C. Results for Private Inference

Cryptonets [14] employs three types of homomorphic operations: ciphertext-plaintext multiplication (`pmul`), ciphertext-ciphertext addition (`cadd`), and ciphertext squaring (`csq`). For this reason, we compare five settings: baseline, ACRM-only, and three variations of ACRM with OOM: `pmul` only, `pmul` and `cadd`, and all three types. Fig. 4 presents the experimental results. We observe that by simply employing our ACRM we improve performance by nearly 44% and reduce memory consumption by 25%. This is explained by the significant reduction in the number of ciphertext copies (from 1.41M to 233k).

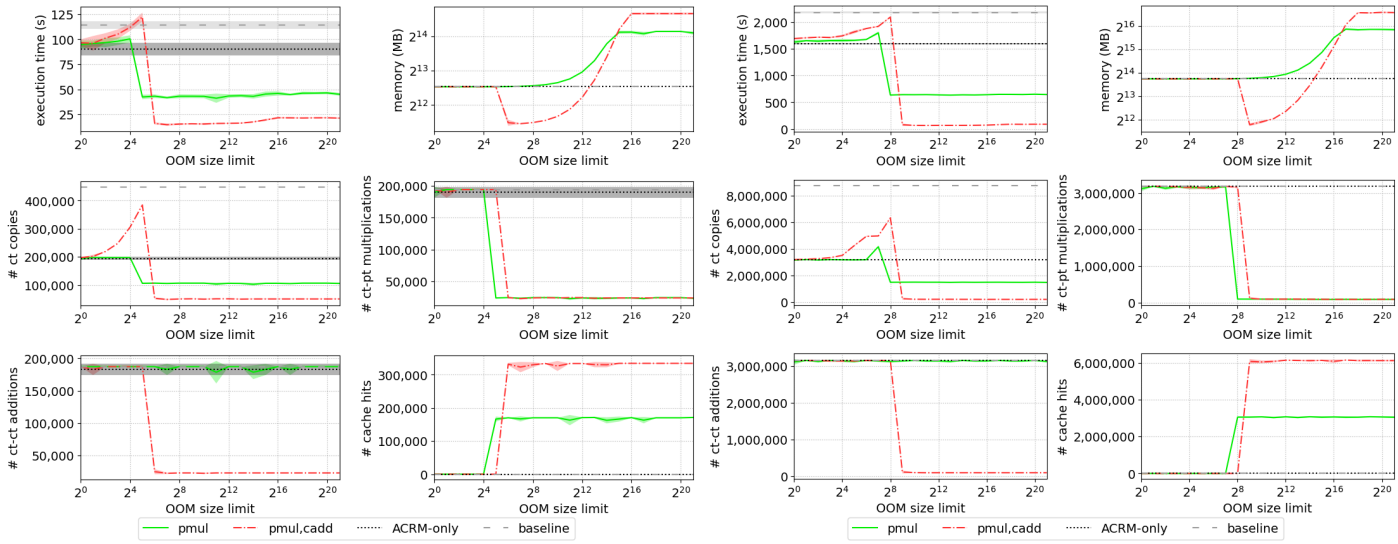
With respect to the OOM approaches, we notice that `pmul,cadd` and `pmul,cadd,csq` have similar results. This is due to the fact that the number of ciphertext squaring operations is relatively small (5,100); thus, few extra ciphertexts have to be recorded. In addition, results of ciphertext squaring are never re-utilized, making OOM ineffective with them. Similarly to the case of dense layer, we see that OOM with `cadd` operations is not helpful since the number of ciphertext additions reduces from 228k (baseline) or 224k (ACRM-only) to 218k, which is not enough to compensate for the delays caused by extra memory allocation. Nevertheless, using our ACRM in combination with OOM with ciphertext-plaintext multiplications reduces the number of `pmul` operations from 224k (baseline) or 211k (ACRM-only) to 133k. This improves performance by nearly 52% over the baseline (5.5% over ACRM-only approach).

We break down the execution time per layer in Table I to understand how each layer is affected by our method. There is a total of seven layers divided into four types: convolution followed by addition, square (activation function), mean pool, and dot product followed by addition. We select the best performing settings with table size limits large enough to reduce the number of operations, and thus, avoid converging the setting to ACRM-only approach. Regarding execution times, we do not observe significant improvement for the square layer. The square operation is very costly computationally and there are relatively few of them. Nevertheless, all other layers observe a substantial speed-up for exactly the opposite reason, i.e., there are many relatively fast operations. Results in general are comparable between ACRM-only and ACRM with OOM settings, with the only noticeable difference in the second convolutional layer, which is faster in the `pmul` setting due to a higher number of output channels. The second convolutional layer has ten output channels, while the first layer has five. This corroborates to the results observed in Section IV-B, where a larger number of output channels gets higher performance improvement from using OOM with `pmul` operations. Since



(a) *dimension* $= 256$ (b) *dimension* $= 1024$

Fig. 2: Experimental results for the dense layer. Shaded area indicates standard deviation. `ACRM-only` uses our ACRM without OOM, while `pmul` uses OOM for ciphertext-plaintext multiplications and `pmul,cadd` uses OOM for both ciphertext-plaintext multiplications and ciphertext-ciphertext additions.



(a) $|c_o| = 8$, $|f| = 3$ (b) $|c_o| = 32$, $|f| = 7$

Fig. 3: Experimental results for the convolutional layer. Shaded area indicates standard deviation. `ACRM-only` uses our ACRM without OOM, while `pmul` uses OOM for ciphertext-plaintext multiplications and `pmul,cadd` uses OOM for both ciphertext-plaintext multiplications and ciphertext-ciphertext additions.

larger CNN models tend to have convolutional layers with more output channels [20], we expect higher performance gains when using OOM with `pmul` operations in those architectures.

V. Related work

Proposed FHE compilers have targeted machine learning applications, aiming to manipulate input data through encoding techniques (e.g., CHW) to reduce inference time [16], [17]. These packing techniques, along with our work, can be used together as complementary approaches. Modern FHE compilers also incorporate standard compiler techniques like common subexpression elimination to optimize computations by identifying and merging common structures within a directed acyclic graph (DAG) [11]. However, compiler optimizations are limited to the information available during compilation, lacking knowledge of ciphertext values as they are encrypted and plaintext values, commonly serialized during computation.

When considering ciphertext-ciphertext additions, both approaches have pros and cons. The compiler performs optimizations during compilation without increasing memory consumption, but it does not optimize additions involving ciphertexts known to be zero due to multiplication by a plaintext zero. Conversely, our approach leverages ciphertext management

TABLE I: Execution time breakdown per layer for different settings.

Setting	OOM Size Limit	ConvAdd (s)	Square (s)	Meanpool (s)	ConvAdd (s)	Meanpool (s)	Square (s)	DotAdd (s)	Total (s)
baseline	0	102.63	85.68	4.29	78.60	0.92	3.49	2.12	277.79
ACRM-only	0	58.60	83.62	0.68	45.17	0.31	3.41	1.36	193.15
pmul	2^{14}	59.12	84.45	0.69	33.66	0.32	3.44	1.47	183.15
pmul, cadd	2^{15}	67.28	84.79	0.95	41.27	0.36	3.45	1.57	199.67
pmul, cadd, csq	2^{15}	67.35	83.71	0.90	40.96	0.35	3.42	1.57	198.26

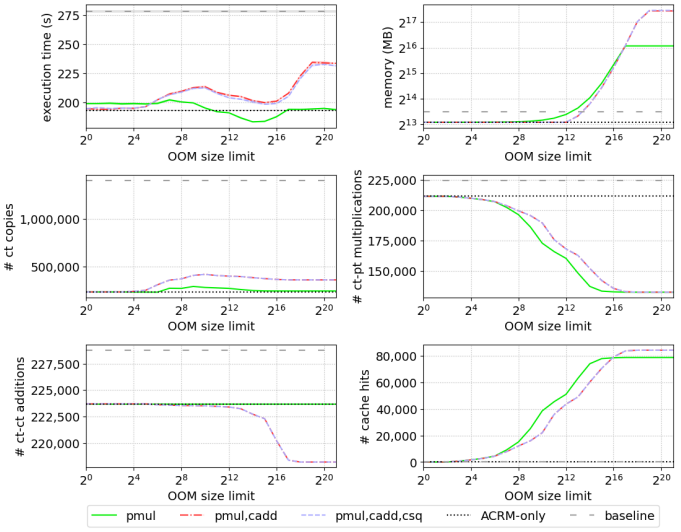


Fig. 4: Results for the privacy-preserving CNN inference. Shaded area indicates the standard deviation. We present results for five settings: 1) `baseline`: state of the art, 2) `ACRM-only`: ACRM without OOM, and ACRM with OOM for 3) `pmul`: ciphertext-plaintext multiplications (`pmul`) only, 4) `pmul,cadd`: `pmul` and ciphertext-ciphertext additions (`cadd`), and 5) `pmul,cadd,csq`: `pmul`, `cadd`, and ciphertext squaring operations (`csq`).

and dynamic determination of repeated plaintext values, offering optimization potential beyond what can be inferred from the DAG. Additionally, our Automatic Ciphertext Reference Module could be integrated with FHE compilers that employ classical optimizations and tailored encoding techniques. Furthermore, the Operation Optimization Module employed by our approach could be beneficial for a profiler accompanying the compiler, allowing dynamic data analysis and generation of truly optimized FHE applications. The combination of our ciphertext management techniques, FHE compilers, classical optimizations, and tailored encoding techniques holds promise for achieving comprehensive optimization in FHE applications.

VI. Conclusion

This work introduces a plug-and-play algorithmic methodology that improves the performance of FHE applications. By employing smart ciphertext memory management and caching techniques, the proposed approach reduces data movement and computation, resulting in significant performance enhancements in the use case of private inference using FHE (1.5x) and generally 24x improvement for convolutional layers. The proposed method can be deployed using any FHE library without the need of any input from the user.

Resources

Furbo source files as well as benchmarks are available at https://github.com/momalab/furbo.

References

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," *preprint arXiv:1801.01203*, 2018.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *preprint arXiv:1801.01207*, 2018.

[3] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 54–72.

[4] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM symposium on Theory of computing*, 2009, pp. 169–178.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[6] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[7] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *ASIACRYPT*, 2016.

[8] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT*, 2017.

[9] "Microsoft SEAL (release 3.7)," https://github.com/Microsoft/SEAL, Sep. 2021, microsoft Research, Redmond, WA.

[10] S. Halevi and V. Shoup, "Design and implementation of helib: a homomorphic encryption library," Cryptology ePrint Archive, Paper 2020/1481, 2020, https://eprint.iacr.org/2020/1481.

[11] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *PLDI*, 2020, p. 546–561.

[12] D. Archer, J. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "Ramparts: A programmer-friendly system for building homomorphic encryption applications," in *WAHC*, Nov. 2019.

[13] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *CVPR*, June 2018.

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

[15] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *Proceedings of the 36th International Conference on Machine Learning*, vol. 97, 09–15 Jun 2019, pp. 812–821.

[16] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An optimizing compiler for fully-homomorphic neural-network inferencing," in *PPLDI*, 2019.

[17] Q. Lou and L. Jiang, "Hemet: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture," in *International Conference on Machine Learning*, vol. 139, Jul 2021, pp. 7102–7110.

[18] E. Sarkar, E. Chielle, G. Gursoy, L. Chen, M. Gerstein, and M. Maniatakos, "Privacy-preserving cancer type prediction with homomorphic encryption," in *Nature Scientific Reports*, vol. 13 (1661), 30 Jan 2023.

[19] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, may 2017. [Online]. Available: https://doi.org/10.1145/3065386