RESEARCH-ARTICLE

Exploiting Unstructured Sparsity in Fully Homomorphic Encrypted DNNs

AIDAN FERGUSON, University of Glasgow, Glasgow, Scotland, U.K.

PERRY GIBSON, University of Glasgow, Glasgow, Scotland, U.K.

LARA D'AGATA, University of Glasgow, Glasgow, Scotland, U.K.

PARKER MCLEOD, Advanced Micro Devices, Inc., Santa Clara, CA, United States

FERHAT YAMAN, Advanced Micro Devices, Inc., Santa Clara, CA, United States

AMITABH DAS, Advanced Micro Devices, Inc., Santa Clara, CA, United States

View all

Open Access Support provided by:

University of Glasgow

Advanced Micro Devices, Inc.

PDF Download
3721146.3721948.pdf
25 January 2026
Total Citations: 3
Total Downloads: 374

Published: 30 March 2025

Citation in BibTeX format

EuroMLSys '25: 5th Workshop on Machine Learning and Systems
March 30 - April 3, 2025
Rotterdam, Netherlands

Conference Sponsors:
SIGOPS

EuroMLSys '25: Proceedings of the 5th Workshop on Machine Learning and Systems (March 2025)
https://doi.org/10.1145/3721146.3721948
ISBN: 9798400715389

Exploiting Unstructured Sparsity in Fully Homomorphic Encrypted DNNs

Aidan Ferguson[1], Perry Gibson[1], Lara D'Agata[1], Parker McLeod[2],
Ferhat Yaman[2], Amitabh Das[2], Ian Colbert[2], José Cano[1]
University of Glasgow, UK[1] AMD[2]

Abstract

The deployment of deep neural networks (DNNs) in privacy-sensitive environments is constrained by computational overheads in fully homomorphic encryption (FHE). This paper explores unstructured sparsity in FHE matrix multiplication schemes as a means of reducing this burden while maintaining model accuracy requirements. We demonstrate that sparsity can be exploited in arbitrary matrix multiplication, providing runtime benefits compared to a baseline naïve algorithm at *all* sparsity levels. This is a notable departure from the plaintext domain, where there is a trade-off between sparsity and the overhead of the sparse multiplication algorithm. In addition, we propose three sparse multiplication schemes in FHE based on common plaintext sparse encodings. We demonstrate the performance gain is scheme-invariant; however, some sparse schemes vastly reduce the memory storage requirements of the encrypted matrix at high sparsity values. Our proposed sparse schemes yield an average performance gain of 2.5× at 50% unstructured sparsity, with our multi-threading scheme providing a 32.5× performance increase over the equivalent single-threaded sparse computation when utilizing 64 cores.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**.

Keywords: Fully Homomorphic Encryption, Neural Network Acceleration, Privacy-Preserving Machine Learning, Secure Computation, Sparse Matrix Multiplication.

ACM Reference Format:
Aidan Ferguson[1], Perry Gibson[1], Lara D'Agata[1], Parker McLeod[2],, Ferhat Yaman[2], Amitabh Das[2], Ian Colbert[2], José Cano[1]. 2025. Exploiting Unstructured Sparsity in Fully Homomorphic Encrypted DNNs. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3721146.3721948

Perry Gibson contributed to this work while at the University of Glasgow.

EuroMLSys '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1538-9/2025/03
https://doi.org/10.1145/3721146.3721948

1 Introduction

Deep neural networks (DNNs) have revolutionized the field of artificial intelligence (AI). These models are now integral to many real-world applications, from autonomous driving to healthcare diagnostics [16, 30]. However, utilizing DNNs in privacy-sensitive environments, such as healthcare, presents unique and challenging requirements to process sensitive data while maintaining confidentiality. Traditional encryption mechanisms ensure confidentiality in transit, leaving the underlying data exposed at inference time.

Fully homomorphic encryption (FHE) [14] has emerged as a powerful cryptographic technique that allows for computation on encrypted data. Despite its promise for privacy-preserving machine learning, FHE incurs a significant computational overhead due to the complexity of its encrypted computations, the growth of noise during operations, and the need for additional procedures to maintain correctness and accuracy for decryption. This overhead often rules out running real-time workloads such as DNN inference, a problem that is emphasized by the trend toward larger, more computationally expensive DNN models [24].

We target the most computationally intensive operation in DNN inference: matrix multiplication (`matmul`) [17–19, 33]. We observed up to $10^6\times$ higher runtime when multiplying square matrices in FHE compared to the unencrypted (plaintext) domain (see Figure 1). With our proposed sparse multiplication schemes, we show that utilizing unstructured sparsity in arbitrarily sized matrix operands can yield improved execution time in all cases relative to a naïve dense implementation; in contrast, plaintext sparsity often requires $\geq$ 70% to become advantageous due to overheads [13]. We also propose a multi-threading scheme that exhibits 32.5× performance gain over a single-threaded implementation when utilizing 64 threads on an AMD EPYC platform. Furthermore, the multi-threaded approach can be applied to arbitrary FHE schemes and implementations.

The contributions of this paper are as follows: i) three unstructured sparse FHE matrix multiplication schemes which provide performance improvements over the naïve dense multiplication at all sparsity levels, and operate on arbitrarily sized matrices without structure requirements; and ii) a CPU-based multithreading scheme for sparse matrix multiplication and evaluation on an AMD EPYC CPU. We make our C++ implementation with Python bindings available at: https://github.com/aidan-ferguson/sparse-fhe-matmul.

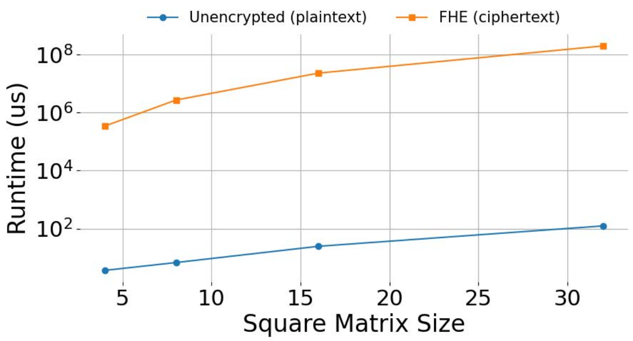

Figure 1. Execution time of square matrix multiplication in plaintext and FHE. Note that the Y-Axis is logarithmic.

2 Background and Related Work

Previous works that explore dense matrix multiplication within FHE often exploit special matrix structures such as square matrices [35] or square binary matrices [21]. These structures are restrictive for DNN inference, where non-square matrix operations are required.

Alternative approaches utilize mathematical structures such as hypercubes in the BGV scheme [23, 35], which is inappropriate for DNN inference as discussed in subsection 3.1. Existing naïve matrix multiplication schemes serve as our baseline for comparison with sparse implementations [11].

Existing attempts at applying sparsity to the FHE matrix multiplication problem often have restrictions. For example, one such restriction entails only applying sparsity to linear systems of form $Ax = b$, where A is a strictly diagonally dominant matrix [9]. Further work that promises unrestricted sparse matrix multiplication [8, 11] does not document how the application of sparsity affects runtime. Furthermore, these schemes only define dense-sparse matrix multiplication. In this work, we focus on exploiting sparsity in both operands, which has direct applications to accelerating DNN architectures with ReLU activation functions that often yield high unstructured activation sparsity [22]. Note that, while already common in convolution architectures (e.g., VGGNet [38], ResNet [20], etc.), ReLU activations are becoming more prominent in foundation LLMs [31].

Finally, open-source dense FHE matrix-multiplication implementations are scarce. TenSEAL [3], a Python wrapper for Microsoft SEAL [37], implements dense vector-matrix multiplication. This precludes applications that require full matrix-matrix multiplication. HEMat [25] is the most prominent open-source implementation of dense FHE `matmul` we could find, built using the HEAAN FHE library [39]. As such, we utilize HEMat in the process of profiling our results as an additional baseline showing performance without utilizing sparsity. To the best of our knowledge, no open-source FHE sparse `matmul` implementations exist; as such we have open-sourced our implementation.

3 Methodology

3.1 FHE Scheme

We utilize the Microsoft SEAL library [37] which implements three commonly used FHE schemes: Brakerski/Fan-Vercauteren (BFV) [6], Brakerski-Gentry-Vaikuntanathan (BGV) [7], and Cheon-Kim-Kim-Song (CKKS) [10].

CKKS supports approximate floating-point computations and is commonly used in the context of DNN inference [29, 32], where some approximation and reduced precision are acceptable. However, BFV/BGV work only with integer values and do not provide rescaling, which can lead to value growth and potential overflow of plaintext values if the multiplications are not managed properly. This is a key reason why BFV/BGV are less suited for deep computations compared to CKKS, which manages scale growth explicitly through rescaling. BFV/BGV also have limited multiplicative depth and they struggle with DNN computations unless bootstrapping [1] is applied frequently, making CKKS a better choice in this context. Furthermore, there are implementations of bootstrapping with the CKKS scheme that execute in more practical time frames than alternative schemes [2]. Additionally, in CKKS we can utilize polynomial approximations for common DNN activation functions such as ReLU without degrading accuracy [28].

Motivated by these reasons, we use CKKS as our FHE scheme. The polynomial modulus degree of the scheme is represented by p, where $p = 2^n$ for some $10 \geq n \geq 15$ and $n \in \mathbb{N}$ [1]. Larger values allow for more complex computations at the expense of slower homomorphic operations. We choose $p = 8192$ as the lowest polynomial modulus degree that accommodates our circuit depth. For the coefficient modulus, we choose a bit size vector of $\{50, 40, 40, 40, 40\}$ and an initial ciphertext scale of 2^{40}. These parameters provide enough precision and modulus switches to perform a matrix multiplication followed by an activation function, a single fully connected layer pass-through.

3.2 Matrix Chunking

The rotation operation is fundamental in CKKS, performing cyclic rotations of an encrypted vector of length $\frac{p}{2}$, where p is the polynomial modulus degree. Rotations are performed using computationally expensive key-switching operations [5], which modify the ciphertext and introduce noise. When many rotations are performed successively, this noise can accumulate instead of padding the ciphertext correctly, leading to errors in the data. This would cause the ciphertext to result in inaccurate values when decrypted.

For this reason, it is preferable to avoid rotations where possible. This can be achieved by implementing a chunking scheme for encoding matrix values into collections of ciphertexts. A chunk size parameter c controls how many

[1] An essential method for restoring ciphertext noise, allowing continuous computation of encrypted data without limitations.

values are encoded into each ciphertext, which can each encode a maximum of $\frac{p}{2}$ values. This places an upper bound on the number of rotations performed on each ciphertext to c, leading to a more accurate result at the expense of memory. For a non-sparse matrix of size $n \times m$, we must construct $\lceil \frac{n \cdot m}{c} \rceil$ ciphertexts. Furthermore, small chunk sizes facilitate our multi-threading approach by limiting the need for synchronization between threads.

3.3 FHE Matrix Multiplication Schemes

3.3.1 Dense Schemes. Dense schemes that do not exploit any sparsity serve as our baseline for comparison. We utilize two dense schemes: naïve and HEMat [25].

Naïve Dense. Our baseline implementation demonstrates a naïve approach to matrix multiplication, providing a lower performance bound for alternative methods. It does not utilize any sparsity in the matrix. This approach can be seen in Algorithm 1. For each result value we select the corresponding row and column vectors from the input matrices, masking the required index and accumulating the value in slot zero of a ciphertext which is then inserted into the encrypted result matrix.

HEMat. We also compare an open-source implementation of HEMat [26], which restricts our evaluation to matrices of dimensions $2^n \times 2^m$. Although our schemes support arbitrarily sized matrices, we conform to this requirement to evaluate against HEMat. HEMat employs the Number Theory Library (NTL) for distributing work across threads. Our multi-threading solution exhibits superior scaling to higher thread counts than HEMat, resulting in different behaviors as the number of threads increases.

3.3.2 Sparse Schemes. The encrypted nature of the matrix values means that we cannot perform conditional logic on it. Therefore, to skip the computation of the zero values, we must expose some information about the structure of the sparsity in the encrypted matrices. In such schemes, only information about the sparsity pattern is exposed with the matrix values remaining encrypted. We rationalize that this is acceptable for most privacy-sensitive DNN inference use cases; however, these sparse schemes may not be applicable in situations with very strict privacy requirements, such as DNNs that utilize one-hot encoding inputs, since the sparsity structure is enough information to reconstruct private DNN inputs.

We can circumvent this restriction in the situation where exposing the sparsity structure information is not acceptable. Some scenarios (e.g., cloud computing) may have knowledge of the weight sparsity structure and process the encrypted DNN input provided by the user without knowledge of the underlying sparse structure. As the model weights are already known in plaintext to the server and the user input is fully secure (including any matrix meta-data), this does not expose any additional information and is equivalent to performing a dense-sparse `matmul`. However, this approach does not fully exploit the underlying sparsity in both operands and could therefore yield increased inference time.

Algorithm 1 Algorithm for naive dense matrix multiplication in FHE.

```
Input: Encrypted list of chunks $A$ and $B$ representing respective matrices, and corresponding matrix information $LHS$ and $RHS$
Output: Encrypted matrix product, stored in list of chunks, $R$
for $row \leftarrow 0$ to $|result_{\text{rows}}| - 1$ do                ▷ Iterate over result rows
    for $col \leftarrow 0$ to $|result_{\text{cols}}| - 1$ do             ▷ Iterate over result columns
        for $k \leftarrow 0$ to $|LHS_{\text{cols}}| - 1$ do           ▷ Iterate over shared dimension
            ···                ▷ Load chunks, calculate value offsets within chunks
            $a \leftarrow$ fhe_rotate($A_{\text{chunk}}, A_{\text{offset}}$)   ▷ Load LHS val. into slot zero
            $b \leftarrow$ fhe_rotate($B_{\text{chunk}}, B_{\text{offset}}$)   ▷ Load RHS val. into slot zero
            $a \leftarrow a \times b$                               ▷ Multiply operands
            $a \leftarrow$ fhe_relin($a$)                                ▷ Re-linearize
            $a \leftarrow$ fhe_rescale($a$)                ▷ Switch modulus and rescale
            $a \leftarrow a \times$ zero_mask             ▷ Multiply with slot zero mask
            $a \leftarrow$ fhe_relin($a$)                                ▷ Re-linearize
            $a \leftarrow$ fhe_rescale($a$)                ▷ Switch modulus and rescale
            $a \leftarrow$ fhe_rotate($a, -R_{\text{offset}}$)  ▷ Rotate prod. to result offset slot
            $R_{\text{chunk}} \leftarrow$ fhe_add($R_{\text{chunk}}, a$)        ▷ Accum. into result chunk
        end for
    end for
end for
```

We introduce three schemes for sparse FHE matrix multiplication, adapted from three plaintext algorithms: a naïve sparse scheme, CSR [36], and ELLPACK [27].

Naïve Sparse. We evaluate a naïve sparse implementation as a simple counterpart to our naïve dense implementation. We encrypt input matrix $M \in \mathbb{R}^{m \times n}$, including zero values into ciphertexts according to the chunking parameters. At instantiation, a binary matrix B is constructed in parallel, where $B_{i,j} = [M_{i,j} = 0] \; \forall i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}$. Matrix B is exposed in plaintext during matrix multiplication, and we skip the computation of elements where the following condition holds: $B^{LHS}_{row,k} \vee B^{RHS}_{k,col}$.

Compressed Sparse Row (CSR). We evaluate the CSR format, which encodes non-zero values in the encrypted domain, storing the locations of these non-zero values in plaintext in row index and column index arrays. We choose not to implement Compressed Sparse Column (CSC), as it is equivalent to performing CSR on the transpose of the input matrix and both of them have equal number of operations for matrix multiplication. One possible optimization we do not explore in this work is transposing the RHS operand, which would then allow for efficient access of columns in the RHS operand.

ELLPACK. For input matrix $M \in \mathbb{R}^{m \times n}$, we encrypt j values per row where $j = \max(NZV(M_{i,0..n})) \; \forall i \in \{1, \ldots, m\}$ and $NZV(x)$ returns the number of non-zero values in a row. For each row, we encrypt the non-zero values and zero-pad as needed to reach length j. A parallel matrix C denotes the column numbers for the elements in the encrypted matrix.

3.4 Multi-Threading

We further accelerate computation by introducing multi-threading to both dense and sparse multiplication schemes. We allocate one thread per result value, utilizing all available threads in the CPU. Synchronization primitives are used to coordinate access to result chunks; small chunks allow for uncoordinated accesses, improving threading performance at the cost of a higher memory footprint. Formally, we delegate the calculation of $A_{rows} \times B_{cols}$ resultant matrix values by allocating the computation among a pool of n threads based on the index of the resultant value; $(row \times col) \mod n$. While computation occurs in parallel, threads must synchronize writing to result ciphertexts, as multiple result values may share a given ciphertext when $chunksize > 1$. In future work, we will look into using the GPU for acceleration, which has demonstrated many orders of magnitude increased performance in primitive homomorphic operations [40].

4 Evaluation

In this section, we discuss the insights gained by evaluating our proposed sparse methods against dense baselines on CPU hardware. We first describe the profiling procedure. Then we discuss the execution time and memory performance of the sparse schemes and how our approach scaled in a multi-threaded environment. After that, we look at scaling to larger matrices beyond trivial examples and investigate the accuracy of our methods. Finally, we show more experimental results with a discussion around notable and surprising observations.

4.1 Profiling Procedure

We repeat the following procedure three times for each sparsity level to capture performance variances. We record the execution time (excluding context setup, encryption, and decryption) of each FHE matrix multiplication scheme and the memory consumption of the ciphertexts involved.

1. Generate operand matrices of size $n \times n$ with $\lfloor s \cdot n \rfloor$ elements set to zero, where s denotes the desired sparsity level. Values are sampled from a Gaussian distribution $\mathcal{N}(0, 1)$, representing weights from a DNN initialized with this distribution [12, 34].
2. Perform dense matmul with Eigen3 [15], a C++ library for efficient linear algebra operations, to validate correctness for all following computations.
3. Perform all sparse matmul operations in plaintext to validate algorithmic correctness.
4. Perform naïve and HEMat matmul in FHE for baseline FHE performance.
5. Perform sparse matmul in FHE for all sparse implementations.
6. Verify that all plaintext and FHE result values are within $\epsilon = 10^{-3}$ of the ground-truth Eigen3 result.

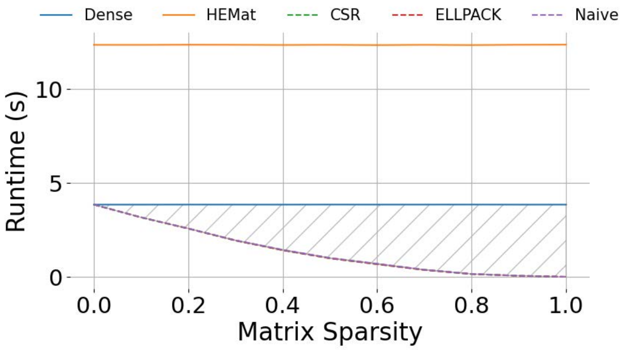

Figure 2. Runtime of dense and sparse schemes multiplying two 8×8 matrices. Shaded region denotes the runtime advantage of utilizing sparsity.

4.2 Runtime performance

All evaluations are conducted on an AMD EPYC 7V13 64-core CPU. According to subsection 4.1, we profile while multiplying two square matrices with sizes $2^3 \times 2^3$, as this is the smallest square matrix size that allows us to allocate one thread per result element with 64 threads and satisfies HEMat's requirements discussed in subsection 3.1.

Figure 2 shows runtime performance when multiplying two 8×8 matrices on a single thread. As expected, both dense baseline algorithms that do not exploit any sparsity maintain a consistent runtime at varying sparsity levels. All sparse implementations display a runtime benefit over the dense schemes; moreover, we observe that the discrepancy between sparse schemes is negligible, with a standard deviation of runtime improvement of 0.039× at 30% sparsity. Furthermore, at this matrix size, our dense implementation runs 3.21× faster than HEMat's on average.

In the plaintext domain, the "break-even" point (sparsity value where sparse matmul becomes advantageous) has been observed to be 71% [13]. In our FHE implementation, it is at 0% sparsity. We hypothesize that, while we incur additional overhead processing the sparse structures, it is shadowed by the computational burden of homomorphic operations and the large memory requirements of encrypted ciphertexts relative to the overhead of sparse data structures.

4.3 Multi-Threading

Results from profiling performance on the ELLPACK sparse scheme can be found in Figure 3 and Figure 4. Our dense and sparse implementations observe similar gains with higher thread counts, with the "break-even" point remaining at 0% sparsity. Results comparing the runtime of differing schemes at a given thread count can be found in Figure 5 and Figure 6. Notably, our implementation scales to higher thread counts more effectively with HEMat's threading scheme, which reaches diminishing returns at 16 threads with a 3.46× performance gain compared to a single thread.

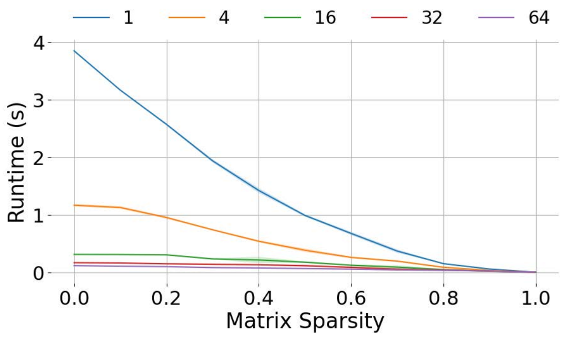

Figure 3. ELLPACK sparse scheme multi-threading `matmul` runtime. Multiplying 8×8 matrices with chunk size $c = 1$.

Our approach at 64 threads exhibits a 32.47× performance gain from one thread in Figure 3. We experience diminishing returns with 64 threads, providing a 1.39× gain compared to 32. Additionally, we show negligible memory overhead in Figure 4, only requiring synchronization primitives.

4.4 Scaling to Larger Matrices

Our analysis so far has been on 8×8 matrices. We now experiment with increasing matrix sizes and expect to scale to even higher dimensional matrices. Using 64 threads, we evaluate the performance of `matmul` at the following square matrix sizes: 8, 16, 32. The results can be found in Figure 7 and Figure 8, where we observe that our sparse schemes still exhibit a break-even point with the naïve dense implementation of 0% and outperform HEMat at ≥ 10% sparsity on matrix sizes of 32×32. However, our algorithms do not scale well to large matrix sizes relative to the HEMat implementation. We believe this is due to the poor $O(n^3)$ algorithmic complexity of our solution where HEMat exhibits $O(n)$ complexity; we leave scaling improvements for future work.

4.5 Correctness

We show the absolute mean error of each scheme at different sparsity levels in Table 1. While we ensure that all schemes have a per-value error of less than $\epsilon = 10^{-3}$, accuracy differs between schemes. Our implementations provide a 7.6×10^{-4} higher accuracy than the HEMat baseline, which may be advantageous in DNN models where precision is required. Furthermore, as sparsity trends to 100%, the sparse scheme results have no error as resultant zero values are effectively computed in plaintext.

4.6 Other Experiments

We demonstrate that our proposed sparse methods require less memory to store operand ciphertexts in Figure 4. This is demonstrated with the ELLPACK sparse scheme; however, similar memory savings are observed across all of our proposed sparse schemes.

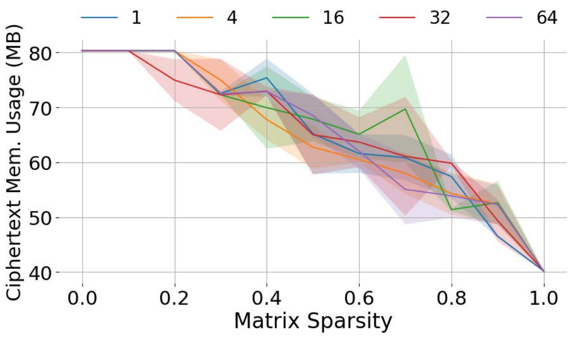

Figure 4. ELLPACK sparse scheme ciphertext memory usage. Multiplying 8×8 matrices with chunk size $c = 1$. The lines represent different thread count levels used for the computation with the shaded regions denoting 1σ deviation, highlighting the low memory overhead of our multi-threading scheme relative to the ciphertext size.

Table 1. Mean absolute error between decrypted FHE matrices and plaintext ground truth results for multiplying 8×8 matrices with one thread and chunk size $c = 1$. The lowest error scheme for each sparsity level is highlighted in **bold**. Our proposed sparse schemes exhibit higher accuracy as sparsity increases, a desirable attribute as we want encrypted computations to approximate plaintext inference closely. As we store sparsity information in plaintext, we compute zero values accurately and avoid noisy homomorphic operations.

Sparsity	Dense	HEMat	CSR	ELLPACK	Naïve
0.0	**5.98E-09**	1.34E-03	6.03E-09	6.97E-09	6.66E-09
0.1	5.63E-09	1.44E-03	5.74E-09	6.01E-09	**5.14E-09**
0.2	5.47E-09	1.35E-03	**5.15E-09**	5.77E-09	6.44E-09
0.3	6.03E-09	1.27E-03	6.29E-09	**4.59E-09**	5.11E-09
0.4	4.84E-09	6.81E-04	3.45E-09	**3.21E-09**	3.38E-09
0.5	5.75E-09	7.64E-04	**3.24E-09**	4.36E-09	3.55E-09
0.6	5.06E-09	5.02E-04	2.72E-09	**2.24E-09**	2.80E-09
0.7	4.73E-09	3.96E-04	1.38E-09	**1.32E-09**	1.77E-09
0.8	4.84E-09	1.96E-04	**5.44E-10**	5.92E-10	1.31E-09
0.9	4.61E-09	8.80E-05	**2.85E-10**	2.91E-10	1.28E-09
1.0	4.40E-09	3.15E-06	**0.00E+00**	0.00E+00	8.90E-10

The runtime of our proposed sparse schemes as we utilize multi-threading on an AMD EPYC 7V13 64-core CPU can be seen in Figure 5 and Figure 6. Our sparse schemes (denoted by dashed lines) are compared to our naïve implementation and HEMat baselines. Notably, our proposed schemes scale better in performance with more threads than HEMat, as such we remove the HEMat baseline from Figure 6 due to the difference between our methods and HEMat being too large for meaningful visualization. At 32 threads (figure excluded for conciseness), our multi-threading scheme exhibits over a 25× increase in speed. However, at thread counts beyond

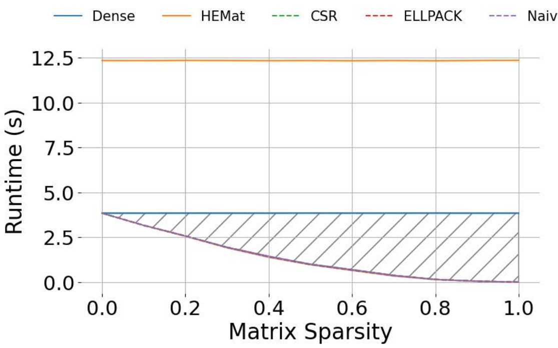

Figure 5. Relative runtime between multiplication schemes when multiplying 8 × 8 matrices with one thread.

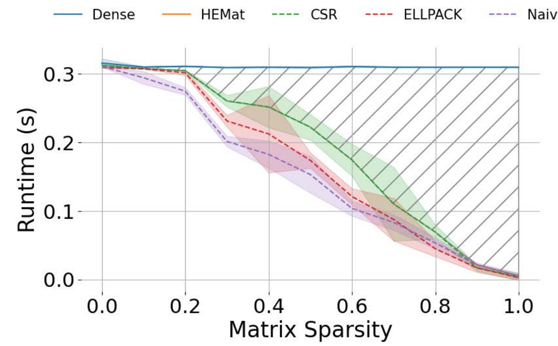

Figure 6. Relative runtime between multiplication schemes when multiplying 8 × 8 matrices with 16 threads. HEMat is excluded due to high runtime.

this, we reach diminishing returns, with a 32.5× performance increase at 64 threads. Furthermore, our sparse schemes maintain a performance gain over the naïve baseline at all sparsity levels as the thread count varies.

While we have demonstrated sparse matrix multiplication schemes in FHE that show improvements compared to baseline naïve performance, state-of-the-art solutions such as HEMat exhibit superior algorithmic complexity. These solutions scale better with respect to matrix size, a drawback that our approach exhibits with a $O(n^3)$ complexity. However, in many real-world situations, our approach outperforms these baselines as demonstrated in Figure 7 and Figure 8.

In summary, for small matrix sizes and high thread counts our solution outperforms HEMat, taking advantage of our multi-threading scheme. However, for larger matrix sizes, such as 32 × 32, HEMat is superior. The "break-even" sparsity threshold at which our proposed schemes are faster, increases. At our maximum evaluated thread count of 64 threads, this begins to occur at matrix sizes of 32 × 32 and larger. However, for low thread counts this will happen at lower matrix sizes.

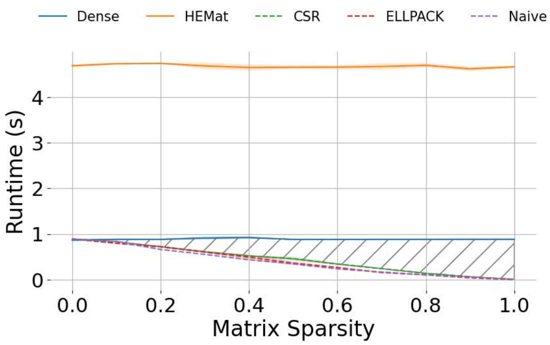

Figure 7. Relative runtime between multiplication schemes when multiplying 16 × 16 matrices with 64 threads.

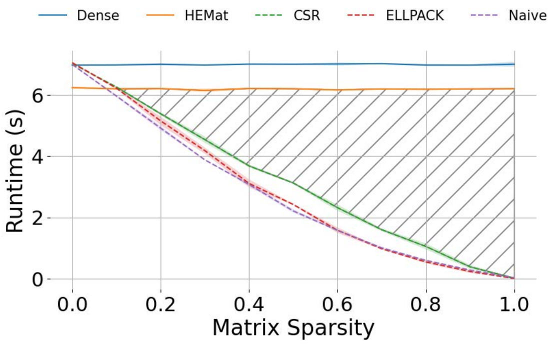

Figure 8. Relative runtime between multiplication schemes when multiplying 32 × 32 matrices with 64 threads. HEMat's superior algorithmic complexity has moved the "break-even" point to ≈ 0.1 sparsity for this matrix size and thread count.

5 Conclusion

We have proposed sparse FHE matrix multiplication schemes within the context of DNN inference. We demonstrated a performance increase of 2.5× over a dense baseline implementation when operating at 50% sparsity. Similarly, we show how our parallelism scheme can significantly increase performance with an observed 32.47× improvement over dense baselines on an AMD EPYC 7V13 64-core CPU.

In future work, we aim to i) explore reducing the memory overhead of sparse matrix multiplication in GPU-accelerated FHE, ii) extend the use of sparsity beyond the $O(n^3)$ algorithmic approach and analyze end-to-end sparse inference GPU-accelerated FHE and iii) demonstrate the utilization of sparsity with larger matrices, such as those found in common small language models (SLMs) such as Pythia-1B [4] that contain projection layers of size 4096 × 1024.

Acknowledgments

This work was supported in part by Advanced Micro Devices, Inc. under the AMD AI & HPC Cluster Program.

References

[1] 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.

[2] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. 2024. General functional bootstrapping using CKKS. *Cryptology ePrint Archive* (2024).

[3] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. 2021. TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption. arXiv:2104.03152 [cs.CR]

[4] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*. PMLR, 2397–2430.

[5] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. *Lecture notes in computer science* (Jan 2021), 587–617. https://doi.org/10.1007/978-3-030-77870-5_21

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory* 6, 3 (Jul 2014), 1–36. https://doi.org/10.1145/2633600

[7] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. *Advances in Cryptology – CRYPTO 2011* (2011), 505–524. https://doi.org/10.1007/978-3-642-22792-9_29

[8] Chaochao Chen, Jun Zhou, Li Wang, Xibin Wu, Wenjing Fang, Jin Tan, Lei Wang, Alex X Liu, Hao Wang, and Cheng Shong Hong. 2021. When Homomorphic Encryption Marries Secret Sharing: Secure Large-Scale Sparse Logistic Regression and Applications in Risk Control. (Aug 2021). https://doi.org/10.1145/3447548.3467210

[9] Xiaofeng Chen, Xinyi Huang, Jin Li, Jianfeng Ma, Wenjing Lou, and Duncan S. Wong. 2015. New Algorithms for Secure Outsourcing of Large-Scale Systems of Linear Equations. *IEEE Transactions on Information Forensics and Security* 10, 1 (Jan 2015), 69–78. https://doi.org/10.1109/tifs.2014.2363765

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. *Advances in Cryptology – ASIACRYPT 2017* (2017), 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[11] Jamie Cui, Chaochao Chen, Lingjuan Lyu, Carl Yang, and Li Wang. 2021. *Exploiting Data Sparsity in Secure Cross-Platform Social Recommendation.* https://www.cs.emory.edu/~jyang71/files/s3rec.pdf

[12] Gianni Franchi, Andrei Bursuc, Emanuel Aldea, Séverine Dubuisson, and Isabelle Bloch. [n. d.]. *TRADI: Tracking deep neural network weight distributions.* https://www.ecva.net/papers/eccv_2020/papers_ECCV/papers/123620103.pdf

[13] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 17, 14 pages.

[14] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09* (2009). https://doi.org/10.1145/1536414.1536440

[15] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[16] Surbhi Gupta, Manoj K. Gupta, Mohammad Shabaz, and Ashutosh Sharma. 2022. Deep learning techniques for cancer classification using microarray gene expression data. *Frontiers in Physiology* 13 (Sep 2022). https://doi.org/10.3389/fphys.2022.952709

[17] Jude Haris, Perry Gibson, José Cano, Nicolas Bohm Agostini, and David Kaeli. 2021. SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators for Edge Inference. In *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.

[18] Jude Haris, Perry Gibson, José Cano, Nicolas Bohm Agostini, and David Kaeli. 2023. SECDA-TFLite: A Toolkit for Efficient Development of FPGA-based DNN Accelerators for Edge Inference. In *Journal of Parallel and Distributed Computing*.

[19] Jude Haris, Rappy Saha, Wenhao Hu, and José Cano. 2024. Designing Efficient LLM Accelerators for Edge Devices. In *Workshop on New Approaches for Addressing the Computing Requirements of LLMs and GNNs (ARC-LG) at ISCA*.

[20] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

[21] Ryo Hiromasa, Masayuki Abe, and Tatsuaki Okamoto. 2016. Packing messages and optimizing bootstrapping in GSW-FHE. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences* 99, 1 (2016), 73–82.

[22] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures. *arXiv:1607.03250 [cs]* (Jul 2016). https://arxiv.org/abs/1607.03250

[23] Hai Huang and Haoran Zong. 2022. Secure matrix multiplication based on fully homomorphic encryption. *The Journal of Supercomputing* (Oct 2022). https://doi.org/10.1007/s11227-022-04850-4

[24] Jiachen Jiang, Yiqi Zhong, and Jinxin Zhou. 2023. *The Efficiency Spectrum of Large Language Models: An Algorithmic Survey.* https://arxiv.org/pdf/2312.00678

[25] Xiaoqian Jiang, Miran Kim, Kristin E Lauter, and Yongsoo Song. 2018. Secure Outsourced Matrix Computation and Application to Neural Networks. (Oct 2018). https://doi.org/10.1145/3243734.3243837

[26] K-miran. 2018. GitHub - K-miran/HEMat: Homomorphic matrix computation. https://github.com/K-miran/HEMat

[27] D Kincaid, T Oppe, and D Young. 1989. ITPACKV 2D user's guide. *OSTI OAI (U.S. Department of Energy Office of Scientific and Technical Information)* (May 1989). https://doi.org/10.2172/7093021

[28] Junghyun Lee, Eunsang Lee, Joon-Woo Lee, Yongjune Kim, Young-Sik Kim, and Jong-Seon No. 2023. Precise Approximation of Convolutional Neural Networks for Homomorphically Encrypted Data. *IEEE Access* 11 (06 2023), 62062 – 62076. https://doi.org/10.1109/ACCESS.2023.3287564

[29] Joon-Woo Lee, Hyungchul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2022. Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network. *IEEE Access* 10 (2022), 30039–30054. https://doi.org/10.1109/ACCESS.2022.3159694

[30] ŞS Mağara, C Yıldırım, F Yaman, B Dilekoğlu, FR Tutaş, E Öztürk, K Kaya, Ö Taştan, and E Savaş. 2021. ML with HE: Privacy Preserving Machine Learning Inferences for Genome Studies. *arXiv e-prints* (2021), arXiv–2110.

[31] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo Del, Mundo Oncel, Tuzel Golnoosh, Samei Mohammad, Rastegari Mehrdad, and Farajtabar Apple. 2023. *ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models.* https://arxiv.org/pdf/2310.04564

[32] George Onoufriou, Marc Hanheide, and Georgios Leontidis. 2022. EDLaaS: Fully Homomorphic Encryption over Neural Network Graphs for Vision and Private Strawberry Yield Forecasting. *Sensors* 22, 21 (2022). https://doi.org/10.3390/s22218124

[33] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. https://arxiv.org/abs/2211.05102

[34] PyTorch. [n. d.]. torch.nn.init — PyTorch 2.2 documentation. https://pytorch.org/docs/stable/nn.init.html

[35] Deevashwer Rathee, Pradeep Kumar Mishra, and Masaya Yasuda. 2018. Faster PCA and Linear Regression through Hypercubes in HElib. *2018 Workshop on Privacy in the Electronic Society* (Jan 2018). https://doi.org/10.1145/3267323.3268952

[36] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems Second Edition Yousef Saad.* https://www-users.cse.umn.edu/~saad/IterMethBook_2ndEd.pdf

[37] SEAL 2023. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[38] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

[39] snucrypto. 2023. HEAAN. https://github.com/snucrypto/HEAAN. https://github.com/snucrypto/HEAAN

[40] Hao Yang, Shiyu Shen, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. 2024. Phantom: A CUDA-Accelerated Word-Wise Homomorphic Encryption Library. *IEEE Transactions on Dependable and Secure Computing* (2024), 1–12. https://doi.org/10.1109/TDSC.2024.3363900