Sparse Matrix Algorithms for Evolving Neural Networks

Carlos Ordonez

Department of Computer Science
University of Houston, USA

Abstract. There is significant research into sparse and dense matrix computations, with high-performance computing techniques, reducing time complexity and improving parallel speedup mostly with ample main memory, considering I/O on secondary storage as a less important aspect. On the other hand, there has been important work in the database, data mining and big data communities accelerating the computation of machine learning models on large data sets. However, massive neural networks and constantly changing data sets are pushing matrix computation demands further. We first present a survey on three key problems identifying research issues: maintaining a large data set updated under frequent matrix entry insertions and deletions, sparse matrix addition/multiplication and recomputing a deep neural network when a sparse data set changes frequently. We then propose a research agenda focusing on those three major problems solved with parallel I/O efficient algorithms storing and processing matrices with coordinate tuples (like a database relational table): matrix entry insertion/deletion, matrix addition/multiplication and assembling these algorithms into state of the art neural networks. We argue coordinate tuples complement and can potentially replace established main memory storage mechanisms, like dense arrays and compressed row/column formats. In summary, we believe database-inspired, parallel I/O efficient, algorithms tailored for sparse matrices can help updating, explaining and monitoring evolving neural networks on large dynamic data sets.

1 Introduction

1.1 Motivation

Computing and monitoring dynamic machine learning models plays a crucial role in addressing science and society problems involving evolving matrix data sets and evolving graphs, where matrix entries are inserted and deleted frequently. This challenge will persist as a significant concern in the future due to the versatility of sparse matrices, which can represent interactions or connections among numerous variables, objects, people, devices, and more [12,14]. The dynamic nature of these data structures, characterized by frequent updates and changes, necessitates the development of efficient and adaptive machine learning models that can accurately capture and respond to these shifts. As the complexity and

data scale of these problems continue to grow, the ability to compute and monitor dynamic models will remain essential for extracting meaningful insights and driving scientific advancements.

The Coordinate (COO) format [10] is generally not the preferred choice in most existing High-Performance Computing (HPC) and Artificial Intelligence (AI) libraries (e.g. Python NumPy, SciPy, PyTorch). This is largely because many libraries opt for dense formats to leverage CPU vectorized instructions for vector and matrix operations [24], while other libraries utilize compressed formats for sparse matrices, like Compressed Sparse Row (CSR [22]) and Compressed Sparse Column (CSC [19]) due to their efficiency in data transfer and access through the upper memory hierarchy, including RAM, CPU L1/L2 cache, and registers. The primary reasons for the limited adoption of COO format are its relatively slower performance and redundant coordinate information. However, as the landscape of HPC and AI continues to evolve, this mindset and inferiority may shift, potentially leading to a reevaluation of the COO format's role in AI, databases and HPC.

The gap between dense, compressed formats and coordinate (COO) format for sparse matrices may shrink as advancements in hardware technology continue to evolve. As CPUs increasingly feature more cores, allowing for greater parallelization, and main memory capacities expand, enabling larger datasets to be processed in-memory, the advantages of compressed formats may diminish. On the other hand, there is growing interest in optimizing GPUs for sparse matrices. Furthermore, the emergence of non-volatile memory technologies that increasingly approach the speeds of traditional memory will also help to reduce the performance differences between these formats. As a result, the benefits of using COO format, such as simpler algorithm implementation, easier code verification, straightforward transfer in and out and explainable intermediate results, may become more appealing, potentially making it a viable choice for dynamic applications.

1.2 Adapting Theory, Parallel Computing and Database Algorithms for Evolving Neural Networks

Insertion and deletion of a sparse matrix requires innovation that bridges the fields of algorithms [3], parallel computing [14], and database systems [16]. This is because traditional algorithms for dense matrices are not directly applicable to sparse matrices, and existing algorithms for sparse matrices are often limited to specific storage formats or operations. Furthermore, parallel algorithms for sparse matrix addition and multiplication, which are fundamental operations in many applications, require significant changes if the matrix is stored as a table containing coordinate tuples. While this storage format is slower in main memory compared to compressed formats, it offers broader algorithmic possibilities, easier verification of code correctness, and clear understanding of the algorithm. The time performance gap may shrink or become less important as CPUs and GPUs get faster and RAM grows larger.

As hardware continues to evolve with advancements in CPUs and GPUs [8] featuring increased core counts, alongside novel storage technologies like Non-Volatile Memory (NVM) approaching main memory speeds, the processing capabilities for large-scale data sets will substantially improve. However, the exponential growth of data sets and the increasing prevalence of neural networks will create new challenges, particularly in observing and tuning neural network models as data changes. In this context, innovative parallel algorithms designed to leverage new hardware architectures and efficiently handle dynamic data will play a crucial role in addressing these challenges, enabling faster insights and more accurate predictions.

1.3 Potential Impact on Science and Society

Algorithms optimized for the Coordinate (COO) tuple format have the potential to significantly impact various scientific and societal problems in the future. Sparse matrices are used in several science applications involving large graphs or high-dimensional data for maintaining dynamic neural networks. Biology is one such field, where protein-protein interaction networks can be represented as large graphs with millions of vertices and edges, and dynamic neural networks can be used to predict new interactions or identify patterns in the data (e.g., the BioGRID database). Atmospheric sciences is another field, where climate models can generate large amounts of high-dimensional data that require efficient storage and analysis, and dynamic neural networks can be used to predict weather patterns or identify trends in the data (e.g., the Climate Data Online dataset). Additionally, geology and physics can also benefit from COO format sparse matrices, where seismic data and particle collision data can be represented as large graphs and high-dimensional data that require efficient analysis and updates, and dynamic neural networks can be used to predict earthquakes or identify patterns in particle collisions (e.g., the Incorporated Research Institutions for Seismology dataset and the CERN Open Data Portal).

The efficient representation and analysis of complex data structures is crucial for informed decision-making in several key areas of society, as illustrated in the following examples. In traffic management, for instance, sparse matrices can be used to represent large graphs of traffic flow and congestion, enabling dynamic neural networks to predict traffic patterns and optimize traffic light control (e.g., the Transportation Networks for Research dataset). Similarly, in epidemiology, COO format sparse matrices can be applied to disease transmission networks, allowing dynamic neural networks to predict the spread of diseases and identify key factors in disease transmission (e.g., the Centers for Disease Control and Prevention's (CDC) Influenza dataset). Furthermore, in economics, sparse matrices can be used to represent large graphs of economic transactions and relationships, enabling dynamic neural networks to identify patterns and trends in economic data (e.g., the US Census Bureau's economic datasets).

2 Survey: State of the Art

2.1 Updating a Large Sparse Matrix in Batches

Updating a large sparse matrix in secondary storage requires clever algorithms to minimize the number of I/O operations [18], using optimal space in main memory. Serial algorithms, such as the in-place update, involve reading the matrix (or a matrix block) from secondary storage, updating the elements in main memory, and writing the updated matrix back to secondary storage. This method is simple but may require multiple passes over the data, which can be time-consuming for large matrices. Another serial approach is the buffer-based update algorithm, which uses a buffer in main memory to accumulate updates before writing them to secondary storage. The buffer is flushed when it is full or when a certain change threshold is reached. On the other hand, in parallel computing, algorithms such as parallel in-place update and parallel buffer-based update can be used to speed up updates. These approaches involve dividing the matrix into smaller chunks and updating each chunk in parallel using multiple threads or processes. The log-structured merge (LSM) algorithm, commonly used in databases, can also be parallelized by dividing the log file into smaller chunks and merging each updated chunk with the main matrix in parallel.

From a theory perspective, I/O-efficient algorithms, are designed to minimize the number of I/O operations required to update the matrix. These algorithms can be applied to sparse matrix updates to reduce the number of I/O operations (read access). Cache-oblivious algorithms, which are designed to optimize cache performance, can also be used to improve the efficiency of sparse matrix updates.

In the context of database systems [16], indexed tables can be used to update the matrix, providing atomicity, isolation and consistency guarantees. On the hand, columnar database systems can provide efficient updates and exploration queries with sparse matrices. Finally, array database systems allow manipulating unlimited size arrays on secondary storage.

In conclusion, the three major approaches of serial and parallel I/O efficient algorithms, theoretical foundations, and database solutions must be combined to develop efficient algorithms for updating a large sparse matrix on secondary storage. By following theoretical foundations and combining the strengths of each approach, researchers can develop algorithms that minimize I/O operations, optimize cache performance, and provide atomicity and consistency guarantees.

2.2 Fast Linear Algebra: Matrix Multiplication

Existing algorithms for fundamental matrix operators like matrix addition and matrix multiplication typically utilize dense and sparse compressed formats [17], which are optimized for modern multi-core CPU architectures [20]. Sparse formats, such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC), store non-zero elements in a contiguous block of memory, allowing for efficient memory access patterns and minimizing memory bandwidth usage [15,7]. This leads to significant performance improvements on modern CPUs, which are

designed to handle large amounts of data in parallel [13]. Moreover, the trend towards using Graphics Processing Units (GPUs) for matrix computations has further emphasized the importance of using dense matrix formats [7]. GPUs, with their massive parallelism and high-bandwidth memory, are particularly well-suited for dense matrix operations. Dense matrix formats can take full advantage of the GPU architecture, allowing for thousands of threads to perform math calculations simultaneously. GPUs have produced significant performance gains in many applications, including machine learning, scientific simulations, and data analytics.

In contrast, the Coordinate Format (COO) [10], which stores matrix elements as a table with tuples containing the row, column, and value, is generally considered slow, redundant, and space-inefficient. Why? Because the coordinate format requires more memory accesses and has a higher overhead due to the need to iterate over a list (sequence) of tuples. Additionally, the coordinate format is more memory-intensive, as it requires storing the row and column indices for each non-zero element, resulting in a slightly larger memory footprint (but still proportional to the number of non-zero entries, known as nnz).

As a result, dense and compressed matrix formats have become the defacto standard for high-performance matrix computations on both CPUs and GPUs. While the COO format may still be useful in certain niche applications, dense and compressed formats are generally the preferred choice for most use cases due to their superior performance and efficiency. Nevertheless, we believe the COO format may become more useful as hardware gets faster, sparse matrices change more frequently and they are read and written on secondary storage.

2.3 Computing Neural Networks with Sparse Input Matrices

The latest research on computing neural networks with sparse input matrices that have periodic changes has focused on developing efficient methods to handle these changes [23]. One approach is iterative linearization, which allows for sparse feature updates and quantifies the frequency of feature learning needed to achieve comparable performance. This method has shown remarkable performance on par with standard training methods, highlighting the importance of feature learning in neural networks.

Most current neural networks use dense matrices during forward and backward propagation. Dense matrices are also widely supported by popular deep learning frameworks and libraries. Additionally, dense matrices take advantage of optimized linear in multi-core CPUs and GPUs. However, some neural networks, such as those used in natural language processing and recommendation systems, often deal with large sparse matrices as input. For example, the Word2Vec model uses a sparse matrix to represent word embeddings, where each row corresponds to a word and each column corresponds to a feature. Similarly, the DeepWalk model uses a sparse matrix to represent graph embeddings, where each row corresponds to a node and each column corresponds to a feature. The GraphSAGE model also uses sparse matrices to represent graph data, where each row corresponds to a node and each column corresponds to a feature. On the other hand,

sparse matrices may appear in intermediate results during forward and backward propagation when neurons are dropped to avoid overfit or when gradients vanish. Overall, the development of efficient methods for updating neural networks with sparse input matrices is an active area of research, and new techniques are being explored to take advantage of the benefits of sparse matrices while minimizing their drawbacks.

2.4 Theoretical Models for Parallel Computation

The best theory computation models to study the time and space complexity of parallel matrix computations (to be discussed later) are the Parallel Random Access Machine (PRAM) model and the Bulk Synchronous Parallel (BSP) model, but extensions are needed for newer architectures with the massive parallelism of GPUs. Some alternative parallel model include the CGM (Coarse Grained Multi-computer; distributed, synchronous, each processor has limited memory), LogP model (Latency-overhead-gap-Processors, which may be more practical than PRAM, it assumes short messages among processors) and PP (Pipeline Parallelism; best for GPUs). The PRAM model is a widely used theoretical model for parallel computing that assumes a shared memory and multiple processors that can access and modify the memory simultaneously. It is suitable for studying parallel algorithms on multi-core CPUs. For GPU computing, the BSP model is more suitable as it takes into account the specific characteristics of GPU architectures, such as the bulk-synchronous execution model and the memory hierarchy. However, some researchers argue that a different model, such as the GPU-PRAM model, may be necessary to accurately capture the unique characteristics of GPU architectures, such as the SIMT (Single Instruction, Multiple Thread) execution model and the memory coalescing. It is necessary to use a different theoretical model for GPU compared to CPU because of the following reasons. Different memory hierarchies: GPU has a different memory hierarchy compared to CPU, with a larger register file and a smaller cache. Different execution models: GPU uses a bulk-synchronous execution model, whereas CPU uses a traditional one-task execution model. Different thread scheduling: GPU incorporate a more complex thread scheduling mechanism compared to CPU. Some models for GPU parallel computing worth mentioning include: GPU-PRAM: an extension of the PRAM model for GPU architectures; BSP-GPU: an extension of the BSP model for GPU architectures; SIMT model: a model specifically designed for SIMT architectures. In conclusion, while the PRAM and BSP models can be used to study parallel algorithms on multi-core CPUs and GPUs, a different model may be necessary to accurately capture the unique characteristics of GPU architectures.

3 Future Research

We defend the idea of storing sparse matrices in coordinate format, opposing established compressed formats like CSR and CSC. When a sparse matrix is stored

in coordinate format, represented as (i, j, v) tuples, it offers a unique advantage over compressed formats like Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC). Coordinate tuples enable a wide range of database systems and list-based algorithms to compute machine learning models. This flexibility is crucial, as it enables extending existing algorithms, exploiting the structural properties of the matrix and data-oriented parallel processing. Moreover, it provides a deeper understanding of the theoretical implications of sparse matrix computations. By working directly with the (i, j, v) tuples, research can gain insights into the underlying complexity of the algorithms and explore the relationships between the matrix elements. This understanding can lead to the development of more explainable, intuitive algorithms, but also efficient and scalable. The coordinate format uncompressed storage allows for a more direct analysis of computational and space complexity.

Algorithms for sparse matrices in coordinate tuple format are a relatively new area of research compared to the state of the art. This is because classical parallel I/O efficient algorithms have traditionally been designed with arrays in main memory as the preferred data structure, rather than sparse matrices stored in tuple format. As a result, adapting these algorithms to work with COO format sparse matrices has not received significant attention. However, there is a substantial body of existing work on theoretical algorithms for sparse matrices in main memory, as well as I/O efficient algorithms for other data structures, that can be combined and extended to develop new algorithms for sparse matrices in COO format. By building on these foundations, researchers can create novel algorithms that efficiently handle the unique characteristics of sparse matrices in COO format.

We envision future research should study these three major problems below, all of which assume that the input sparse matrix is maintained on secondary storage, intermediate matrices may be maintained on secondary storage and the output matrices (weights) may also need to be stored. Research is needed to address the following key challenges considering data-oriented parallel processing and reducing I/O cost:

1. Algorithms for inserting and deleting batches of matrix entries, which can efficiently handle the dynamic nature of sparse matrices.
2. Algorithms for fundamental matrix operations such as matrix addition and matrix multiplication, which can minimize the overhead of data transfer between secondary storage and main memory, preserving fast transfer in the upper memory hierarchy (registers, L1/L2 cache, RAM).
3. Incorporating the previous algorithms into evolving neural networks that are computed on sparse matrices, which are periodically updated with scattered changes, requiring efficient and scalable solutions to maintain model accuracy and performance.

3.1 Problem I: Insertion and Deletion of Batches of Matrix Entries

Data sets are not static: they continuously change. The Internet is pushing changes further and the rate of change may be unpredictable. Recent research

has explored various techniques for maintaining a sparse matrix up to date with batches of insertions and deletions of a few matrix entries, particularly within a time window. Under this window-based approach, parallel aspects such as $O(1)$ data structures, parallel insertion and deletion algorithms, and concurrent access control have been studied. The primary reason for this focus is that most High-Performance Computing (HPC) research assumes the sparse matrix can be stored with arrays maintained in main memory, limiting the applicability of these techniques to problems where the matrix is so large that it cannot fit in main memory. Notable contributions include research by Bender et al. on parallel data structures for dynamic sparse matrices [6], and studies by Liu et al. on efficient sparse matrix multiplication across diverse systems, including database systems [21]. However, a thorough review of research literature reveals a significant gap specifically addressing the challenges of maintaining sparse matrices with unpredictable batches of insertions and deletions, indicating a need for further research in this area.

Optimizing batches of insertions and deletions of sparse matrix entries is a more general problem than a sliding window approach because it can handle a wide range of dynamic scenarios, rather than just a fixed-size window of recent updates (insert new records, delete old records). In many domains, the rate and pattern of insertions and deletions can vary significantly over time, with different time scales, rhythms, and spikes in activity. By providing more flexibility and adaptability, optimizing batches of insertions and deletions can lead to better performance and efficiency in a wide range of applications, from social networks and financial markets to recommendation systems and more. For example, in a social network, the rate of new user sign-ups and friendships may be relatively steady, but there may be sudden spikes in activity around holidays or major events. In contrast, a financial market may experience rapid changes in trading activity during times of economic uncertainty. A sliding window approach may not be able to adapt to these varying patterns with spikes, whereas optimizing batches of insertions and deletions can handle these changes more effectively. Furthermore, optimizing batches of insertions and deletions requires collecting and processing more information than just the recent updates. In addition to the entry coordinates and values of the matrix entries, timestamps must also be collected and considered. This allows the optimization algorithm to take into account the temporal relationships between updates and make more informed decisions to process them in groups to reduce I/O cost.

Modern hardware aspects must be considered. Future research should develop parallel algorithms for multi-core CPU or hybrid CPU/GPU, but excluding GPU-only (since the data set is assumed to be read from secondary storage), for efficient insertion/deletion of a few matrix entries (O(1) or O(log(n))) of a large sparse matrix with O(n) edges stored in Coordinate (COO) format. Two important I/O techniques include matrix tiling to improve data locality and identifying I/O patterns in sparse matrices to exploit buffers and group I/O operations. This research direction aims to leverage the strengths of both multi-core CPUs and hybrid CPU/GPU architectures to achieve significant performance

gains in a few updates of a sparse matrix. By focusing on COO format, which is well-suited for parallel processing, future research can explore novel algorithms that minimize the time complexity of insertion and deletion operations, making them suitable for large-scale applications. The development of such algorithms will require careful consideration of data structures, memory management, and synchronization techniques to ensure efficient and scalable performance.

Let X be the input data set, consisting of n column vectors with d dimensions (i.e. a $d \times n$ matrix). We envision two potential parallel algorithms and data structures to insert batches of matrix entries in COO format of a large matrix stored on secondary storage:

1. Batch-Insert-Delete Algorithm:
   We can assume there a single process updating X, to simplify algorithms and avoid concurrency mechanisms. However, concurrent updates should be eventually considered. This algorithm uses a combination of a buffer-based approach and a parallel merge-sort algorithm. The buffer is stored in main memory and it is used to accumulate incoming batches and deleted batches of matrix X entries. When the buffer is full, the algorithm sorts the buffer using a parallel merge-sort algorithm and then merges the sorted buffer with the existing matrix on secondary storage to propagate changes. This algorithm can be improved with subscript ranges to process the matrix in blocks to improve I/O locality.
   Data Structure: A combination of a buffer (in main memory) and a sparse matrix (on secondary storage) stored in coordinate tuple format. Time Complexity derivation: Buffer accumulation: $O(b)$, where b is the size of the buffer. Parallel merge-sort: $O(b \log(b)/p)$, where p is the number of machines (cores). Merge with existing matrix: $O(n_z + b)$, where n_z is the number of non-zero entries in the existing matrix.
   Parallel Speedup: A parallel merge-sort algorithm can potentially achieve a linear speedup of up to p, where p is the number of machines (cores).
2. Log-Structured-Merge (LSM) Algorithm:
   The LSM approach is commonly used in modern NoSQL database systems to handle high insertion rates. Therefore, new matrix update algorithms can use the log-structured merge (LSM) file approach to insert large batches of matrix entries, but with few or no deletions. That is, X is constantly growing, mainly n as data size grows, but also d can grow as more features are added. The algorithm can work with a combination of in-memory buffers and on-disk storage to accumulate and merge batches of matrix entries, similar to the more common algorithm introduced above.
   Data Structure: A combination of in-memory buffers and on-disk storage, with a sparse matrix stored in coordinate format.
   Time Complexity: Buffer accumulation: $O(b)$, where b is the size of the buffer. Parallel merge: $O(b \log(b)/p)$, where p is the number of machines (cores). Merge with existing matrix: $O(n_z + b)$, where n_z is the number of non-zero entries in the existing matrix. Parallel Speedup: a potential speedup of up to p, where p is the number of machines (cores).

There are important differences compared to algorithms used in query processing. Database algorithms typically focus on handling high insertion rates for transactions and demanding queries on normalized tables, whereas our envisioned algorithms focus on inserting batches of matrix entries in parallel, where there is no notion of normalized tables Database algorithms use different data structures, such as B-trees or hash tables on rows, whereas future algorithms will work on matrix sub-blocks, with coarser indexing In summary, our algorithms are designed to efficiently insert batches of matrix entries in parallel, taking into account the characteristics of secondary storage and multi-core CPUs. While database algorithms share similarities, new algorithms will be tailored to the specific requirements of sparse matrix insertion and deletion.

3.2 Problem II: Algorithms for Addition and Multiplication of Sparse Matrices

The development of parallel, I/O-efficient algorithms for sparse-sparse and sparse-dense matrix operators is crucial to process large matrices. Two fundamental matrix operators must be studied: addition and multiplication. For matrix addition, an algorithm can take advantage of the fact that the coordinate tuples allow efficient insertion and deletion of entries. By utilizing parallel processing on multi-core CPUs and hybrid CPU/GPU architectures, the algorithm can quickly identify and combine corresponding entries from the two input matrices, resulting in a new sparse matrix in COO format. The parallelization of this process can be achieved by dividing the matrices into smaller blocks and processing them concurrently. In contrast, matrix multiplication is a more complex operation that requires careful consideration of I/O patterns to get rows from the left matrix and columns from the right matrix, reading them as coordinate tuples. To avoid recomputing all multiplications, the algorithm can employ techniques such as caching intermediate results and reusing previously computed products. By leveraging the parallel processing capabilities of multi-core CPUs and hybrid CPU/GPU architectures, the algorithm can efficiently perform the necessary multiplications and accumulations to produce the resulting sparse matrix in coordinate format as well. Despite the differences between addition and multiplication, both algorithms share commonalities in their reliance on efficient I/O operations by block and parallel processing. In both cases, the use of coordinate tuples and partitioned storage enables efficient handling of sparse matrices.

HPC research on parallel algorithms for sparse matrix addition and multiplication, combining sparse and dense matrices, has explored various main memory formats, including dense arrays, COO, CSR, and CSC. Several studies have investigated the potential of these formats, such as the work by Buluç et al. on parallel sparse matrix-vector multiplication using compressed sparse blocks (CSB) [4]. Other notable contributions include the study by Kaya on parallel algorithms for computing sparse matrix permanents [3], and the work by Liu et al. on sparse matrix-matrix multiplication using the COO format [21]. The coordinate tuple format opens new possibilities on modern hardware with faster CPUs

and GPUs, offering opportunities for improved performance and efficiency. However, theoretical aspects, such as potential time complexity, I/O optimization, and parallel speedup, have not received sufficient attention, largely due to the efficiency of linear algebra libraries on modern CPUs and GPUs for dense matrices, as noted by Demmel et al. [11] and by Ballard et al. [5]. Parallel aspects of these algorithms have been explored in both multi-core and distributed memory settings. The impact of I/O patterns and matrix structure on performance has also been studied, highlighting the need for careful consideration of these factors in algorithm design.

Future research should focus on developing new parallel algorithms for sparse matrix addition and sparse matrix multiplication, specifically tailored for matrices stored with coordinate (COO) tuples. Let the two input matrices be A, B, compatible for matrix multiplication $A \cdot B$. Matrix addition is an easier and straightforward case since both matrices have equal dimensions. The primary goal should be to achieve ideal time complexity and parallel speedup for these matrix operators in three distinct scenarios: (1) when A, B are resident in main memory, (2) when A, B are read from secondary storage, (3) when A is large and it is read from secondary storage and B is in main memory (or vice-versa), but size(B) $\ll$ size(A). Scenario (3) where one matrix is in main memory and the other is on secondary storage can be reduced to Scenario (1) or (2) when both matrices are about the same size. For Scenarios (1) and (2) hashing, sorting can used to merge rows from A with columns from B, similar to a relational join operator. For Scenario (3) B can be converted to a dense representation, accessing B elements directly with subscripts i, j from A. NVM offers significantly faster access times, larger block sizes, and non-volatility, making it a default option for storing and processing large matrices. Therefore, this research direction is particularly relevant, given the increasing adoption of non-volatile memory (NVM) with fast PCI connection, as a replacement for traditional disks and SATA solid-state drives (SSDs). For GPUs A, B entries can be easily sorted by A column, B row and then aligned before transferring to GPU memory for parallel multiplication.

3.3 Problem III: Incorporating Sparse Matrix Algorithms into Evolving Neural Networks

The landscape of deep learning has undergone a significant shift, with Transformers and Graph Convolutional Networks (GCNs) [1] emerging as dominant architectures [2,9], leaving behind plain Deep Learning, Recurrent Neural Networks (RNNs), and Convolutional Neural Networks (CNNs). Current research is now focusing on incorporating sparse matrix algorithms into these two emerging types of neural networks. The first step involves utilizing algorithms that update the input sparse matrix with batches of insertions and deletions, allowing the network to efficiently adapt to changing data. In Transformers, these updated sparse matrices can be leveraged to compute self-attention mechanisms, where algorithms for sparse matrix addition and multiplication play a crucial role in combining and transforming input embeddings. Similarly, in GCNs, these algorithms

can be applied to update adjacency matrices and node features, enabling the network to learn from the evolving graph structure. By integrating sparse matrix algorithms, both Transformers and GCNs can benefit from improved efficiency and scalability, particularly when dealing with large and dynamic datasets.

Despite the extensive research on sparse matrices in HPC and parallel computing, there is scarce work on utilizing sparse matrices within Transformers and Graph Convolutional Networks (GCNs) neural networks, where the input data set undergoes periodic batches of insertions and deletions. The vast majority of existing research on sparse matrices has focused on optimizing their use in traditional HPC applications, such as linear algebra operations and scientific simulations. However, the potential benefits of sparse matrices in neural networks, particularly in the context of dynamic and evolving data, remain largely unexplored. As a result, there is a significant gap in the literature regarding the application of sparse matrices in Transformers and GCNs, where the input data is constantly changing due to insertions and deletions. This lack of research presents an opportunity for innovative work that could lead to significant advancements in the efficiency and scalability of these neural networks.

Adapting the Coordinate (COO) format to optimize I/O can yield substantial algorithm efficiency benefits, particularly in scenarios involving frequent matrix updates to recompute neural networks faster. By harnessing the COO format, matrix updates can be executed more rapidly, and neural network models can be recomputed with increased efficiency, thereby avoiding the need for costly recomputations from scratch. Moreover, incremental algorithms can process updates in small batches, facilitating a faster update cycle characterized by a short lag of a few seconds between the insertion or deletion of matrix batches and the subsequent update of the neural network. This database-oriented approach enables more responsive and adaptive modeling, making it well-suited for applications where data is constantly evolving and timely insights are required.

4 Deployment

Since it would be a long-time effort to develop prototypes to solve the three major problems introduced above, here we provide some guidelines for programming and experimental evaluation. To deploy the algorithms and data structures discussed above, Python is a better language than C++ or Java for faster prototype development of sparse matrix algorithms due to its ease of use, flexibility, and extensive community support. Python's syntax and nature make it an ideal language for rapid prototyping and development, allowing developers to focus on the theoretical aspects of the algorithms rather than the implementation details. Additionally, Python's vast array of libraries and frameworks provide a wealth of pre-built functionality, enabling developers to build upon existing work and accelerate their development process. However, low-level languages like C++ will still be needed for specific sparse matrix bottlenecks, where performance is critical and optimization is required. C++'s ability to provide direct memory

access and fine-grained control over hardware resources make it an ideal choice for optimizing performance-critical components.

The following popular Python libraries can be used to develop and test the theory of matrix algorithms and neural networks, listed in order of maturity and popularity:

1. NumPy: A mature and widely-used library for efficient numerical computation, including linear algebra.
2. SciPy: A comprehensive library for scientific computing, providing functions for linear algebra, optimization, and more.
3. PyTorch: A popular deep learning framework that provides a dynamic computation graph, tensors and automatic differentiation.
4. scikit-learn: A widely-used library for machine learning, providing tools for data pre-processing, feature selection, and model evaluation.
5. Pandas: A library for data manipulation and analysis, providing data structures and functions for working with structured data.
6. TensorFlow: similar to PyTorch, which was more popular a few years ago.

To experimentally validate time complexity and parallel speedup theory results, the following steps can be followed:

1. Program the algorithms using the chosen Py libraries and frameworks, with a focus on simplicity and readability.
2. Use synthetic data to test the algorithms under various conditions, such as different matrix sizes and numbers of threads.
3. Measure the execution time and parallel speedup of the algorithms using profiling tools and benchmarking techniques.
4. Compare the results to the theoretical predictions and analyze any discrepancies, using statistical methods and data visualization techniques to identify trends and patterns.
5. Iterate on the design and implementation of the algorithms based on the results of the experiments, refining the implementation and testing new hypotheses as needed.

5 Conclusions

We presented a vision paper, with a tentative research agenda. Therefore, neither theory results nor experiments were provided. From a research and code development perspective, the coordinate tuple format holds potential benefits that can streamline the development process and facilitate innovation, bridging database systems and HPC. Specifically, this format can be processed using established I/O-efficient algorithms, which can accelerate computation and reduce overhead. The format's simplicity also makes it easier to analyze space and time complexity, enabling researchers to better understand the computational resources required and optimize their code accordingly. Furthermore, the coordinate format straightforward structure simplifies the process of writing correct

code, reducing the likelihood of errors and bugs. Additionally, the explicit representation of coordinates provides query capabilities and storage transparency, making it easier to track and interpret results, and ultimately facilitating the development of more reliable and efficient algorithms.

References

1. Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Comput. Surv.*, 54(9):191:1–191:38, 2022.
2. Charu C. Aggarwal. *Neural Networks and Deep Learning - A Textbook*. Springer, 2023.
3. A. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison/Wesley, Redwood City,California, 2nd edition, 1983.
4. Ariful Azad and Aydin Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pages 688–697. IEEE Computer Society, 2017.
5. Grey Ballard, James Demmel, and Andrew Gearhart. Brief announcement: communication bounds for heterogeneous architectures. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *SPAA: Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures*, pages 257–258. ACM, 2011.
6. Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. In Phillip B. Gibbons and Christian Scheideler, editors, *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*, pages 61–70. ACM, 2007.
7. Benjamin Brock, Aydin Buluç, and Katherine A. Yelick. Rdma-based algorithms for sparse matrix multiplication on gpus. In *Proceedings of the 38th ACM International Conference on Supercomputing, ICS 2024, Kyoto, Japan, June 4-7, 2024*, pages 225–235. ACM, 2024.
8. Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. *IEEE Access*, 8:225134–225180, 2020.
9. Chaoqi Chen, Yushuang Wu, Qiyuan Dai, Hong-Yu Zhou, Mutian Xu, Sibei Yang, Xiaoguang Han, and Yizhou Yu. A survey on graph neural networks and graph transformers in computer vision: A task-oriented perspective. *IEEE Trans. Pattern Anal. Mach. Intell.*, 46(12):10297–10318, 2024.
10. Hoang-Vu Dang and Bertil Schmidt. The sliced COO format for sparse matrix-vector multiplication on CUDA-enabled GPUs. In *Proceedings of the International Conference on Computational Science, ICCS*, volume 9 of *Procedia Computer Science*, pages 57–66. Elsevier, 2012.
11. James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.*, 34(1), 2012.
12. J.W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1st edition, 1997.
13. Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. *Parallel Comput.*, 78:33–46, 2018.

14. J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vost. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.
15. Sana Ezouaoui, Olfa Hamdi-Larbi, and Zaher Mahjoub. Towards efficient algorithms for compressed sparse-sparse matrix product. In *2017 International Conference on High Performance Computing & Simulation, HPCS 2017, Genoa, Italy, July 17-21, 2017*, pages 651–658. IEEE, 2017.
16. H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2nd edition, 2008.
17. Hua Huang and Edmond Chow. Exploring the design space of distributed parallel sparse matrix-multiple vector multiplication. *IEEE Trans. Parallel Distributed Syst.*, 35(11):1977–1988, 2024.
18. Myung-Hwan Jang, Yun-Yong Ko, Hyuck-Moo Gwon, Ikhyeon Jo, Yongjun Park, and Sang-Wook Kim. SAGE: A storage-based approach for scalable and efficient sparse generalized matrix-matrix multiplication. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management, CIKM*, pages 923–933. ACM, 2023.
19. Shakya Jayakody and Jun Wang. EMBARK: memory bounded architectural improvement in CSR-CSC sparse matrix multiplication. In *9th IEEE International Conference on Collaboration and Internet Computing, CIC 2023, Atlanta, GA, USA, November 1-4, 2023*, pages 8–17. IEEE, 2023.
20. Byeongho Kim, Jongwook Chung, Eojin Lee, Wonkyung Jung, Sunjung Lee, Jaewan Choi, Jaehyun Park, Minbok Wi, Sukhan Lee, and Jung Ho Ahn. Mvid: Sparse matrix-vector multiplication in mobile DRAM for accelerating recurrent neural networks. *IEEE Trans. Computers*, 69(7):955–967, 2020.
21. Weifeng Liu and Brian Vinter. CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. In Laxmi N. Bhuyan, Fred Chong, and Vivek Sarkar, editors, *Proceedings of ACM on International Conference on Supercomputing (ICS)*, pages 339–350. ACM, 2015.
22. Yuta Nagahara, Jiale Yan, Kazushi Kawamura, Masato Motomura, and Thiem Van Chu. Efficient COO to CSR conversion for accelerating sparse matrix processing on FPGA. In *IEEE International Conference on Consumer Electronics, ICCE 2024, Las Vegas, NV, USA, January 6-8, 2024*, pages 1–2. IEEE, 2024.
23. Mohammadreza Soltaniyeh, Richard P. Martin, and Santosh Nagarakatte. An accelerator for sparse convolutional neural networks leveraging systolic general matrix-matrix multiplication. *ACM Trans. Archit. Code Optim.*, 19(3):42:1–42:26, 2022.
24. Hong Zhang, Richard Tran Mills, Karl Rupp, and Barry F. Smith. Vectorized parallel sparse matrix-vector multiplication in petsc using AVX-512. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*, pages 55:1–55:10. ACM, 2018.