NeuraChip: Accelerating GNN Computations with a Hash-based Decoupled Spatial Accelerator

Kaustubh Shivdikar[1] Nicolas Bohm Agostini[1] Malith Jayaweera[1] Gilbert Jonatan[2]
José L. Abellán[3] Ajay Joshi[4] John Kim[2] David Kaeli[1]

[1]Northeastern University [2]KAIST [3]Universidad de Murcia [4]Boston University
{shivdikar.k, bohmagostini.n, malithjayaweera.d, d.kaeli}@northeastern.edu
{gilbertjonatan, jjk12}@kaist.ac.kr, joshi@bu.edu, jlabellan@um.es

ABSTRACT

Graph Neural Networks (GNNs) are emerging as a formidable tool for processing non-euclidean data across various domains, ranging from social network analysis to bioinformatics. Despite their effectiveness, their adoption has not been pervasive because of scalability challenges associated with large-scale graph datasets, particularly when leveraging message passing. They exhibit irregular sparsity patterns, resulting in unbalanced compute resource utilization. Prior accelerators investigating Gustavson's technique adopted look-ahead buffers for prefetching data, aiming to prevent compute stalls. However, these solutions lead to inefficient use of the on-chip memory, leading to redundant data residing in cache.

To tackle these challenges, we introduce NeuraChip, a novel GNN spatial accelerator based on Gustavson's algorithm. NeuraChip decouples the multiplication and addition computations in sparse matrix multiplication. This separation allows for independent exploitation of their unique data dependencies, facilitating efficient resource allocation. We introduce a rolling eviction strategy to mitigate data idling in on-chip memory as well as address the prevalent issue of memory bloat in sparse graph computations. Furthermore, the compute resource load balancing is achieved through a dynamic reseeding hash-based mapping, ensuring uniform utilization of computing resources agnostic of sparsity patterns. Finally, we present NeuraSim, an open-source, cycle-accurate, multi-threaded, modular simulator for comprehensive performance analysis.

Overall, NeuraChip presents a significant improvement, yielding an average speedup of 22.1× over Intel's MKL, 17.1× over NVIDIA's cuSPARSE, 16.7× over AMD's hipSPARSE, and 1.5× over prior state-of-the-art SpGEMM accelerator and 1.3× over GNN accelerator. The source code for our open-sourced simulator and performance visualizer is publicly accessible on GitHub[1].

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; *Interconnection architectures*; • **Computing methodologies** → Neural networks; • **Theory of computation** → *Graph algorithms analysis*; • **Hardware** → Hardware accelerators.

KEYWORDS

Graph Neural Networks (GNN), Decoupled Computations, Spatial Accelerators, Sparse Matrix Multiplication (SpGEMM), On-chip Memory, Hardware-software co-design

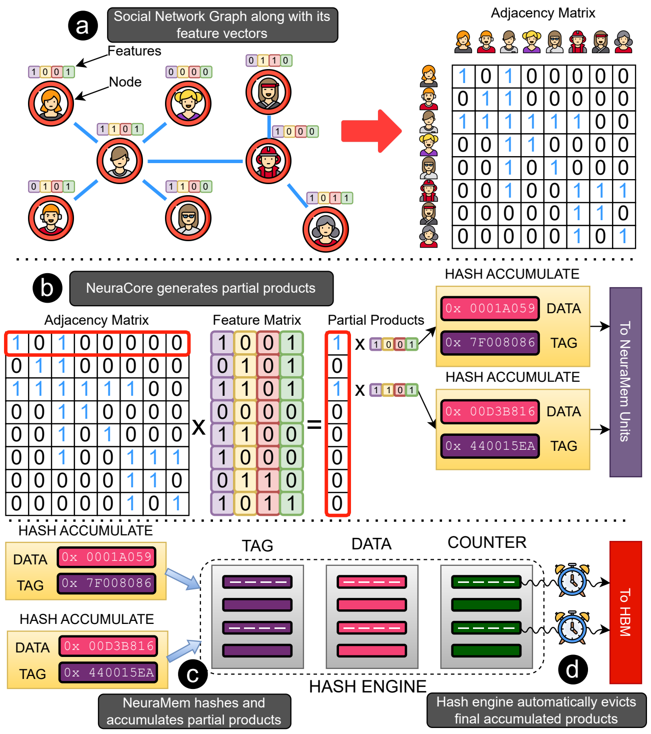


Figure 1: NeuraChip overview: (a) Aggregation phase of GCN, (b) NeuraCore generates partial products, (c) NeuraMem accumulates partial products, (d) writes back to HBM.

1 INTRODUCTION

Deep Neural Networks have proven to be powerful model for solving problems that rely on data with an underlying Euclidean or grid-like structure, such as computer vision, natural language processing, and audio vision. In contrast, Graph Neural Networks (GNNs) have emerged as a powerful tool for handling non-Euclidean data (e.g., social networks on the scale of billions [42]), achieving impressive performance across various domains such as social science, chemistry, and bioinformatics [49, 54]. However, the computational complexity of GNNs, especially when working with ultra-sparse,

[1]https://github.com/NeuraChip/neurachip

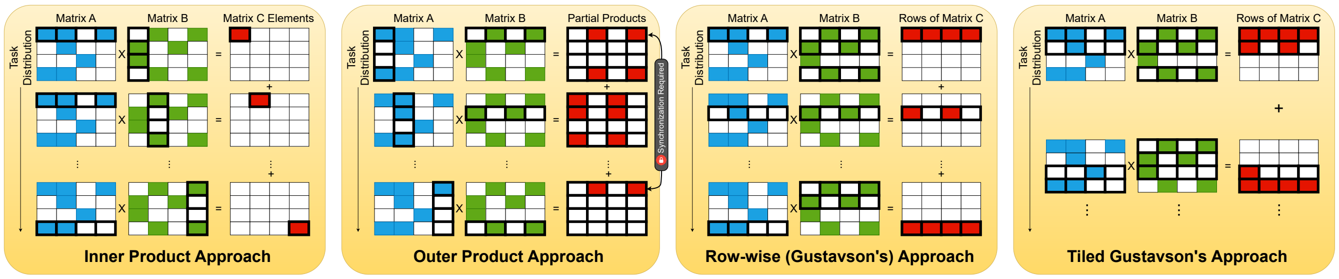


Figure 2: Matrix multiplication approaches, each showcasing varying degrees of data reuse for input and output matrices.

large-scale graph datasets, poses challenges due to architectural limitations of traditional hardware (i.e., CPUs / GPUs) [38]. Moreover, GNNs predominantly adopt a recursive neighborhood aggregation (i.e., message passing) methodology, in which each node aggregates the feature vectors of its neighboring nodes to derive its own updated feature vector. The scalability of message passing in GNNs, when applied to large graph structures, poses a significant bottleneck, especially as the size of the graphs surpasses the capacity of on-chip memory in today's CPUs and GPUs [19]. This leads to redundant and time-consuming memory transactions to fetch data from the main memory [5]. We identify three main bottlenecks causing redundant memory transactions and propose corresponding hardware/software solutions.

Diverse Data Dependence Patterns: The process of neighborhood aggregation in graphs can be split into a multiplication stage, followed by a merge/reduction stage. The multiplication stage creates partial products by multiplying the adjacency matrix of the input graph with the feature matrix. The reduction stage accumulates (merges) the partial products to update the node feature vectors. The multiplication stage's operands depend on data stored in the high-bandwidth memory (HBM). In contrast, the reduction stage's operands depend on data located within the on-chip memory. Utilizing a singular computational resource for both multiplication and accumulation operations proves suboptimal, as mapping multiplication operations on computing resources tends to compromise the efficiency of mapping the accumulation operations (due to varying data dependency patterns). To effectively address these distinct data dependencies during these stages, we present two dedicated components in our custom accelerator, tailored to specific data dependencies: NeuraCore, which executes multiplication tasks, and NeuraMem, which accumulates the intermediate partial products generated by NeuraCore (Figure 1).

Uneven Hardware Resource Utilization: The multiplication and accumulation stages are characterized by distinct architectural implications. The multiplication stage typically stalls due to data starvation (unavailability of input graph and feature matrix elements), whereas the accumulation stage suffers from uneven partial product distribution due to irregular sparsity patterns. To address these issues, we present two strategies, each catering to their respective problems. (a) *Multiplication mapping:* We implement a *tiled row-wise product approach* to partition the computation into distinct tasks, which are then dynamically allocated to NeuraCore, depending on its utilization. The row-wise product method, also known as Gustavson's algorithm, is a popular choice among recent accelerators such as Gamma [51], MatRaptor [46], and SPADA [23] as this approach has shown high-efficiency when targeting sparse matrix computations in the aggregation stages of GNNs. Developing dedicated components for multiplication enables mapping these operations to NeuraCore, independent of the accumulation stage, thus leveraging the locality of the input data. (b) *Accumulation Mapping:* We present a dynamic reseed hash-based mapping agnostic to sparsity patterns. This allows us to evenly distribute the partial products among the NeuraMem accumulation units.

Table 1: SpGEMM memory bloat analysis across various hyper-sparse graph datasets

Dataset	Node Count	Edge Count	Sparsity (%)	Bloat Percent
2cubes_sphere	101492	1647264	99.9840	205.87
ca-CondMat	23133	186936	99.9651	75.23
cit-Patents	3774768	16518948	99.9999	19.32
email-Enron	36692	367662	99.9727	68.90
filter3D	106437	2707179	99.9761	326.34
mario002	389874	2101242	99.9986	99.43
p2p-Gnutella31	62586	147892	99.9962	10.21
poisson3Da	13514	352762	99.8068	297.92
scircuit	170998	958936	99.9967	66.13
web-Google	916428	5105039	99.9994	104.27
amazon0312	400727	3200440	99.9980	97.21
cage12	130228	2032536	99.9880	127.23
cop20k_A	121192	2624331	99.9821	327.07
facebook	4039	60050	99.1519	2872.80
m133-b3	200200	800800	99.9980	26.93
offshore	259789	4242673	99.9937	205.45
patents_main	240547	560943	99.9990	14.18
roadNet-CA	1971281	5533214	99.9999	35.75
webbase-1M	1000005	3105536	99.9997	36.02
wiki-Vote	8297	103689	99.8494	148.09

Memory Bloat: Incorporating the row-wise product approach enhances input data locality but creates a large number of partial products [3]. Table 1 presents memory bloat for SpGEMM workload across various sparse graph datasets.

$$\text{Bloat Percent} = \frac{pp_{\text{interim}} - nnz_{\text{output}}}{nnz_{\text{output}}} \times 100 \tag{1}$$

We define bloat percent as shown in Equation 1, wherein pp_{interim} denotes the count of intermediate partial products and nnz_{output} signifies the count of non-zero elements in the resultant product matrix. Although tiling the computation partially mitigates this issue, it does not fully resolve it. Prior solutions like Gamma [51] have relied on large explicitly managed cache systems like FiberCache, which consumes up to 72% of the total chip's area. To address the memory bloat issue, we present a rolling eviction strategy, which automatically evicts a partial product from the on-chip memory once all contributing partial products have been fully accumulated. We enable this using an eviction counter integrated with the on-chip memory hashtables.

Our complete accelerator design is named NeuraChip, a spatial accelerator featuring Coarse-Grained Reconfigurable Array (CGRA) on-chip interconnects. NeuraChip is versatile due to its ability to efficiently compute graphs with varying degrees of sparsity and sparsity patterns.

In this paper, we make the following contributions:

- **Heterogeneous Processing Approach:** We present NeuraChip, an innovative GNN spatial accelerator featuring a decoupled computation pipeline [43]. Segregating multiplication and accumulation operations into dedicated components, we optimize data reuse through strategic mapping.
- **Adaptive Hash-Based Compute Mapping:** Our approach introduces a flexible, dynamic reseeding hash-based compute mapping (DRHM) tailored for GNN workloads. DRHM benefits from the constant lookup times characteristic of hash functions, while also mapping tasks evenly across all computing resources by generating a new seed at predetermined intervals of computation. This approach achieves a uniform workload distribution independent of the sparsity patterns in graph workloads.
- **Mechanism for Rolling Evictions:** We propose a rolling eviction strategy to combat the issue of memory bloat. Enhanced on-chip hash tables enable the removal of partial products, effectively reducing memory congestion caused by their generation.

NeuraChip excels when compared to Intel MKL running on an Xeon CPU, surpassing it by a factor of 22.1×. Additionally, when compared against NVIDIA's H100 GPU using the CUSP library, NeuraChip achieves a performance boost of 13.3×. In comparison to the prior leading sparse matrix multiplication accelerator, Gamma, NeuraChip showcases an average performance improvement of 1.5×. Furthermore, NeuraChip outperforms the state-of-the-art GNN accelerator, FlowGNN [36], by an average factor of 1.3×.

2 BACKGROUND AND MOTIVATION

2.1 Fundamentals of GNN Workloads

Graph neural networks (GNNs) are capable of extracting important features such as structural motifs (i.e., arrangements of nodes, edges, and metadata) by learning not just the individual characteristics of each element (i.e., a node in the graph), but also the interconnections (i.e., the interrelationships between nodes) between elements.

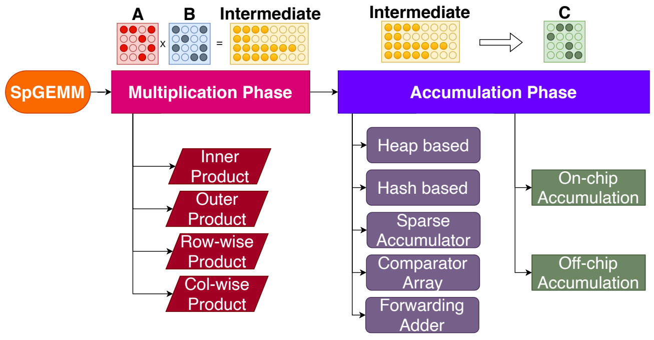


Figure 3: Multiplication and Accumulation phase techniques.

They utilize convolution operations to extract various features from the graph. The methodology employed is neighborhood aggregation, where the final feature vector for each vertex is computed by iteratively aggregating and transforming the input feature vectors of adjacent vertices. This process includes two steps, which are called the *aggregation and combination stages*. This process is carried out iteratively, and after k iterations through these stages, the resultant feature vector of the target vertex signifies the distinct structural data of the vertex's k-hop vicinity.

For instance, a Graph Convolutional Network (GCN) is one such GNN model. Equation 2 below computes the forward propagation for a single layer in a GCN.

$$X^{(l+1)} = \sigma(AX^{(l)}W^{(l)}) \tag{2}$$

where A represents the adjacency matrix of the graph, where each row lists the interconnections of a vertex to all other vertices in the graph. $X^{(l)}$ refers to the input feature vectors of every vertex in the l^{th} layer of matrix X. W contains the GNN's model parameters, which are obtained through model training. $\sigma()$ represents the non-linear activation functions like ReLU (Rectified Linear Unit).

2.2 Architectural Implications of GNNs

Aggregation Stage: The aggregation stage in GNN workloads is critical for capturing the structural information of graphs. It involves gathering and summarizing information from a node's neighbors, which can be a challenging task given the irregular data structures common in graph-based data. This is typically computed with sparse matrix multiplication kernels. Given the high level of sparsity in input graphs, typically above 99%, this stage is characterized by random access patterns in memory, which presents a challenge for traditional architectures that are more suited for linear data access. Additionally, the irregular sparsity patterns often lead to workload imbalance on computing resources [40], which can impact performance efficiency.

Combination Stage: The combination stage in GNNs involves the integration of node features with neighborhood information. This process is computationally intensive and typically comprises dense matrix multiplications, nonlinear activations, and dimensionality reduction operations [18]. Architecturally, this stage demands high memory bandwidth and efficient data reuse mechanisms to handle large matrices. It also necessitates a balance between compute utilization and memory access, as the combination of features from large graphs can lead to memory bottlenecks. While prior accelerators [23, 51, 53] often focus on sparse matrix multiplication tasks, they do not adequately address dense workload demands. Our NeuraChip accelerator model provides a more generalized solution, addressing the needs of both sparse graph computations and dense workloads. This approach positions NeuraChip as a versatile GNN accelerator, adept at handling both the aggregation and combination stages.

2.3 Sparse Matrix Mult: Algorithmic Overview

The Sparse General Matrix-Matrix Multiplication (SpGEMM) kernel execution is characterized by two main stages: the multiplication stage and the accumulation stage as visualized in Figure 3. The implementation variations in these stages lead to distinct SpGEMM algorithms. We describe the four approaches to execute the initial multiplication stage, as illustrated in Figure 2. These approaches vary in their memory access patterns and the level of parallelism they expose.

The inner product approach, incorporated in InnerSP [3] computes elements of the output matrix directly but is hindered by inefficient input reuse. Conversely, the outer product approach, utilized in OuterSPACE accelerator [31] is hampered by suboptimal output locality due to the creation of numerous batches of intermediate partial product matrices [53]. Our research adopts the row-wise multiplication approach (i.e., Gustavson's algorithm), selected for the extensive parallelism it provides. Notably, this approach efficiently avoids the memory bloat issue associated with handling numerous intermediate partial products [53].

The subsequent stage, known as the accumulation stage, merges the generated intermediate partial products. Various accumulation methods include heap-based [2], hash-based [28], sparse accumulator (SPA) based [13], comparator array based [53], and Forwarding Adder Network (FAN) based [27, 33], among others (illustrated in Figure 3). This stage can also be subdivided into on-chip and off-chip accumulation, based on the utilized memory hierarchy. NeuraChip merges partial products using on-chip accumulation to reduce redundant main memory data fetches. For sparse matrices with irregular non-zero distributions, the on-chip accumulation stage can result in uneven workload distribution, a factor that significantly impacts the overall performance and efficiency of SpGEMM operations.

2.4 Mapping Algorithms Design

Mapping algorithms play a crucial role in efficiently handling computational tasks, particularly in scenarios involving sparse data structures such as those found in Graph Neural Networks (GNNs). These algorithms are tasked with assigning tasks or data elements to computational nodes or memory locations. The key requirements for effective mapping algorithms include:

Consistency: The algorithm must consistently map the same index to the same node. This ensures correctness in data processing.

Low Computational Overhead: The lookup process should be relatively fast, with minimal computational and memory overheads. This efficiency facilitates cost-effective index matching, streamlining partial product reduction.

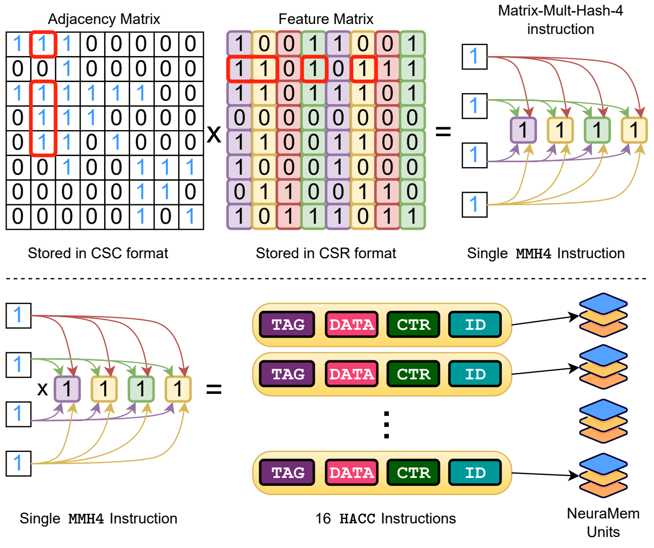


Figure 4: Implementation of tiled Gustavson's algorithm using NeuraCore for multiplication and NeuraMem for accumulation.

Sparsity Agnostic: Regardless of irregular sparsity patterns, the mapping algorithm should remain impartial to these variations. This ensures uniform performance across different data sets [7].

Given these requirements, hash-based mapping emerges as a viable solution [9, 37]. However, traditional hash-based methods such as Round Robin Hashing (or Ring Hashing) [47] and Prime Number Based Modular Hashing [6] have limitations [8]. Neither is fully insensitive to sparsity patterns; a specific set of indices might consistently map to the same node, leading to potential workload imbalance.

An alternative approach is random mapping, which ideally achieves sparsity-agnostic mapping by randomly distributing indices. However, to ensure consistency, this method requires maintaining a large lookup table, which is not practical due to memory constraints.

To address these challenges, we propose a novel approach: Dynamic Reseed Hash-Based Mapping (DRHM). This method is similar to prime modular hashing, but with a significant enhancement. After processing a predetermined set of computations, we reseed the hash function. The updated seed values are then stored in a compact lookup table. This dynamic reseeding ensures that the distribution of indices does not become predictable, effectively mimicking the sparsity-agnostic property of random mapping.

Dynamic Reseed Hash-Based Mapping strikes a balance between the ideal characteristics of random mapping and the practical limitations of traditional hash-based methods. By only storing seed values rather than the entire mapping of indices, it maintains a small memory footprint. Concurrently, it offers the sparsity-agnostic mapping necessary for handling diverse and irregular data sets efficiently. This method significantly enhances the performance of computational tasks, particularly in environments where data sparsity and distribution can vary widely.

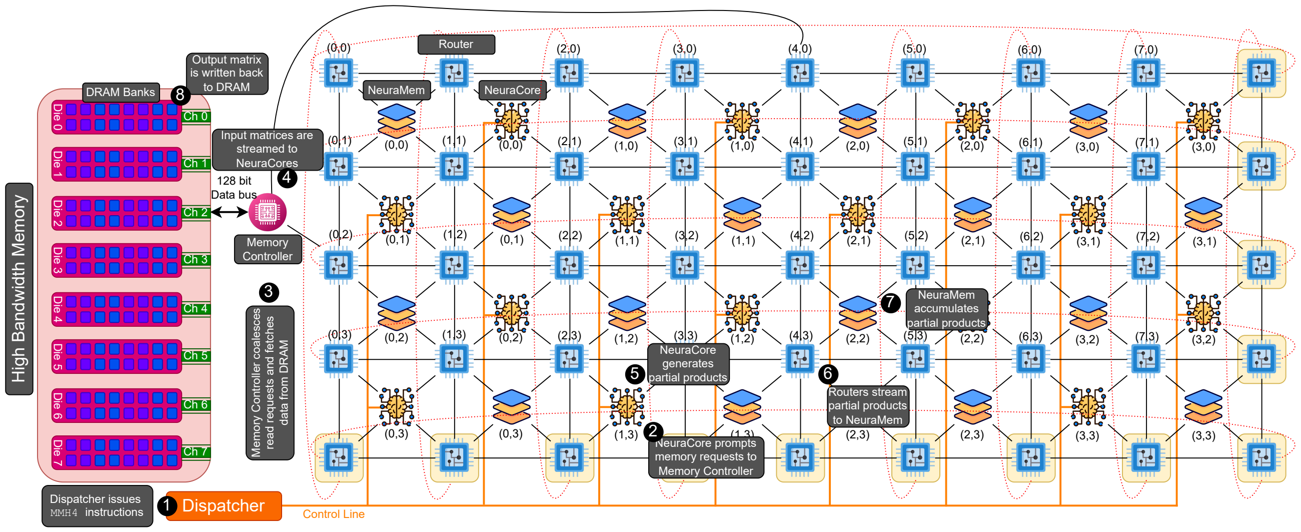

High Bandwidth Memory
DRAM Banks
Die 0
Die 1
Die 2
Die 3
Die 4
Die 5
Die 6
Die 7
Ch 0
Ch 1
Ch 2
Ch 3
Ch 4
Ch 5
Ch 6
Ch 7
8 Output matrix is written back to DRAM
4 Input matrices are streamed to NeuraCores
128 bit Data bus
Memory Controller
3 Memory Controller coalesces read requests and fetches data from DRAM
Router
NeuraMem
NeuraCore
5 NeuraCore generates partial products
2 NeuraCore prompts memory requests to Memory Controller
6 Routers stream partial products to NeuraMem
7 NeuraMem accumulates partial products
Dispatcher issues MMH4 instructions
1 Dispatcher
Control Line

Figure 5: NeuraChip Architecture: Tile 64 configuration with 16 NeuraCores and 16 NeuraMems per tile.

3 NEURACHIP ARCHITECTURE

NeuraChip is a decoupled spatial accelerator. Its two primary components include: i) the NeuraCore and ii) the NeuraMem. The NeuraCore is specifically tailored for multiplication tasks, whereas the NeuraMem focuses on accumulating data on-chip. They are arranged in an interleaved pattern and connected through a 2D torus network fabric, as shown in Figure 5. To facilitate efficient communication among these components, on-chip routers have been incorporated. NeuraCores and NeuraMems are organized into clusters known as tiles. The accelerator includes a total of eight tiles, each linked to a single Double Data Rate (DDR) channel. Each tile features a memory controller that interfaces with DRAM banks.

Buffers play a critical role in the functionality of the four major components of our accelerator. Both the NeuraCore and the NeuraMem are equipped with instruction buffers. Additionally, the on-chip routers incorporate packet buffers, and the memory controllers are fitted with buffers for managing both reading and writing operations.

The incorporation of these on-chip buffers enhances the accelerator's flexibility, allowing it to adapt to diverse sparsity patterns. In scenarios where irregular graph structures could lead to network congestion, these on-chip buffers prove beneficial. They ensure that the components consistently have instructions to execute, thus avoiding potential delays or bottlenecks in processing.

3.1 Tiled Gustavson's Multiplication Algorithm

GNNs typically employ two primary layers (phases) in their architecture: the neighborhood aggregation phase, which gathers information from a node's neighbors in the graph, and the combination phase, where a node's representation is updated by integrating its own features with those aggregated from its neighbors. This discussion focuses on the aggregation phase, which predominantly involves sparse matrix multiplications [16].

In this paper, we implement a modified version of Gustavson's matrix multiplication algorithm [15]. Gustavson's algorithm operates on a row-stationary approach, processing the output matrix one row at a time. Specifically, it traverses the adjacency matrix row by row, performing a linear combination of these rows as illustrated in Figure 4.

Gustavson's approach multiplies each element in a row of the adjacency matrix with all elements in the corresponding row in the feature matrix that has the same row index as the element's column index. Our adaptation enhances Gustavson's method by simultaneously processing multiple rows. We execute the multiplication of four rows at a time, aligning four elements from a column of the adjacency matrix with four elements from a row of the feature matrix. This is achieved using a specialized instruction, denoted as the `MMH4` instruction.

Our technique represents a fusion of Gustavson's algorithm and the outer-product method. Unlike the outer-product approach which finalizes the multiplication of an entire column with a row before moving to the next, our strategy concurrently processes four rows by employing the Gustavson method. The selection of the number 'four' for simultaneous row processing results from design space exploration specific to the NeuraChip accelerator.

To implement this modified Gustavson's approach, the adjacency matrix is stored in a compressed sparse column (CSC) format, and the feature matrix is stored in a compressed sparse row (CSR) format. However, this approach presents two primary challenges:

Unavoidable Index Matching: Employing Gustavson's algorithm and compressed matrix storage formats such as CSR and CSC inherently leads to the necessity of index matching [32, 46]. We address the index-matching overhead with a constant lookup hash function, facilitating the on-chip accumulation of partial products with a constant lookup time. The low overhead provided by our hash function is further optimized by adding a dedicated hash engine, as described in Section 3.4.

Memory Bloat Issue: The tiled Gustavson method can result in memory bloat, characterized by the generation of a large number of partial products. To tackle this issue, we have implemented a rolling eviction mechanism. This system accumulates partial products as they are generated and promptly evicts them once the reduction is complete, with further details provided in Section 3.4.

3.2 On-chip Dataflow

To illustrate the data flow within NeuraChip, we walk through an example of an SpGEMM kernel executed on the NeuraChip accelerator (see Figure 5). Step ❶ The process begins with the *Dispatcher* issuing **matrix_mult_hash_4** (MMH4) instructions to every *NeuraCore*. Step ❷ The *NeuraCores* trigger memory read requests that are routed to the memory controller. Step ❸ The *Memory Controller* coalesces requests for contiguous memory locations into a singular transaction and reorganizes memory transactions to enhance spatial locality. Step ❹ Input matrix data, fetched from DRAM, is streamed onto respective *NeuraCore* components. Step ❺ The *NeuraCores* compute the partial products, along with their corresponding rolling counters (further details in Section 3.3), subsequently generating the **hash_accumulate** (HACC) instructions. Step ❻ HACC instructions are streamed over on-chip routers into NeuraMem components, based on a hash-based mapping. Step ❼ The *NeuraMem* component employs another hash function to hash and accumulate these partial products onto their on-chip memory. Consecutive hashes of partial products with the same TAG are merged within NeuraMem, with each hash insertion decrementing the counter by 1. Step ❽ When the counter reaches zero; this triggers the eviction of the hashline, and the resultant data is written back to the High Bandwidth Memory (HBM).

3.3 NeuraCore

The NeuraCore is the primary compute engine in our accelerator. It computes the multiplication operation and generates corresponding partial products. It is a simple in-order core with support for matrix instructions. NeuraCore supports a special matrix instruction called `matrix_mult_hash_4` or simply MMH4.

Algorithm 1 presents the MMH4 instruction execution pseudocode where *opcode* represents the operation code, which specifies the MMH4 instruction to be executed by NeuraCore. $Base_{addr}$ denotes the base address used to offset the address of all other addresses involved in this instruction. A_data_{addr} refers to the memory address where the data of matrix A is located (matrix A is stored in CSC storage format). $B\_col\_ind_{addr}$ points to the memory address containing the column indices of matrix B (matrix B is stored in CSR storage format). B_data_{addr} indicates the memory address where data from matrix B is stored. $roll\_counter_{addr}$ denotes the memory address where the rolling eviction counter is located. The instruction layout for MMH4 is presented in Figure 7. Each MMH4 instruction has the capability to dispatch up to 16 HACC instructions (further elaborated in the NeuraMem section).

Algorithm 1 MMH4 instruction execution

```
1: for $i = 0$ to 3 do
2:     for $j = 0$ to 3 do
3:         $TAG \leftarrow \text{Mem}[(Base_{addr} + B_{col\_ind\_addr} + j)]$
4:         $DATA \leftarrow \text{Mem}[(Base_{addr} + A_{data\_addr} + i)]$
5:             $\times \text{Mem}[(Base_{addr} + B_{data\_addr} + j)]$
6:         $CTR \leftarrow \text{Mem}[(Base_{addr} + roll\_counter + i * 4 + j)]$
7:         Dispatch HACC$(TAG, DATA, COUNTER)$
8:     end for
9: end for
```

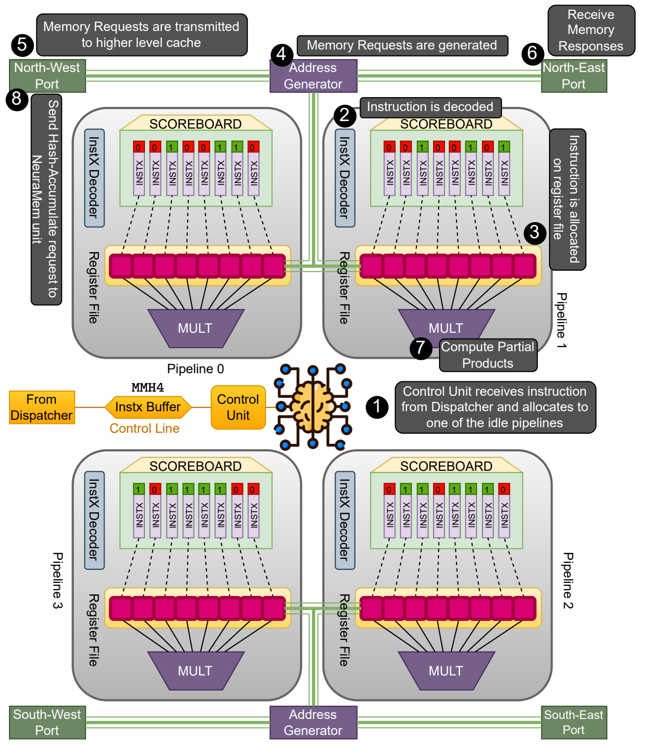
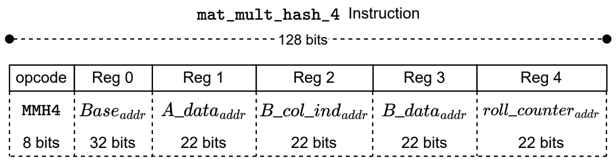


Figure 6: NeuraCore's quad-pipeline layout.

The operational sequence within NeuraCore is shown in Figure 6, and can be broken down into the following steps: Step ❶: The operation starts with the dispatcher transmitting a MMH4 instruction to NeuraCore, allocating the instruction to one of the available pipelines. Pipelines are allocated using a round-robin scheme. Step ❷: The MMH4 instruction is decoded by the on-chip decoder. Step ❸: Following decoding, NeuraCore maps instruction variables to the register file, utilizing dynamic register allocation. Step ❹: Post register allocation, the NeuraCore's internal address generator constructs memory requests to fetch elements from the input matrices. Step ❺: An adaptive routing algorithm [1] selects the best port to dispatch the memory request, which is then forwarded to a higher-level cache. Step ❻: Upon completing the memory request, a response is received at one of the NeuraCore's four ports. This response is then routed toward its respective pipeline. Step ❼: As soon as all memory responses corresponding to a particular instruction are received, the instruction is deemed ready for execution by the scoreboard. Subsequently, the multiplication pipeline calculates the partial product and generates up to 16 HACC instructions. Step ❽: Lastly, the HACC instructions are relayed to NeuraMem units using the most suitable port, as determined by the on-chip hash-based mapping function.



`mat_mult_hash_4` Instruction

128 bits

opcode	Reg 0	Reg 1	Reg 2	Reg 3	Reg 4
MMH4	$Base_{addr}$	A_data_{addr}	$B\_col\_ind_{addr}$	B_data_{addr}	$roll\_counter_{addr}$
8 bits	32 bits	22 bits	22 bits	22 bits	22 bits

Figure 7: MMH4 instruction bit layout.

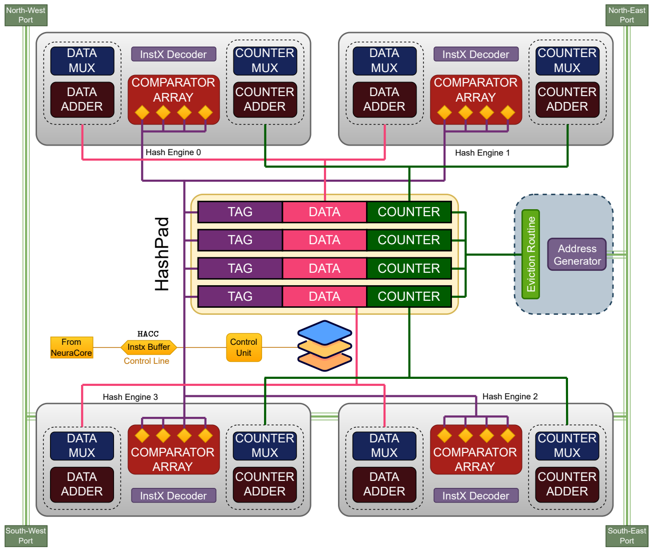


Figure 8: NeuraMem's quad-hash-engine block diagram.

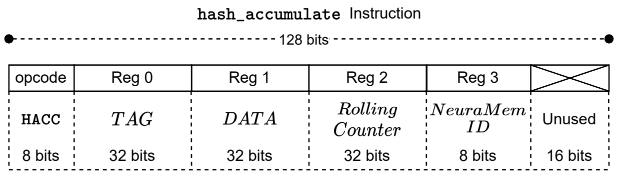
3.4 NeuraMem

NeuraMem is a crucial component of the NeuraChip accelerator. While NeuraCore units generate partial products, NeuraMem units handle the on-chip accumulation of these partial products. The central component of NeuraMem units is the Hash-Engine. The layout of various components within NeuraMem is as shown in Figure 8.

HashPad: The Hash-Engine operates on what we refer to as "hash-lines" Figure 8. A hash-line comprises a single TAG, DATA, and COUNTER entry. The collective TAG array, DATA array, and COUNTER array, essentially the whole set of hash-lines, form what is known as the HashPad, as shown in Figure 8.

HACC instruction: NeuraMem supports a special instruction for partial product accumulation called `hash_accumulate`, or simply HACC instruction. The bit layout of HACC instruction is illustrated in Figure 9. Algorithm 2 presents a pseudocode of the HACC instruction, providing clearer insight into its functionality.

Hash-Engine workflow: Figure 10 shows a typical sequence of events during the execution of a HACC instruction by the Hash-Engine (illustrated using pseudocode in Algorithm 2). The process starts in step ❶, where the Hash-Engine receives a HACC instruction from the NeuraCore units. This instruction's TAG is simultaneously compared with all the TAGs currently present on the HashPad (step ❷). The multiplexers select the hash-line with the matching TAG in step ❸. The corresponding hash-line's DATA gets accumulated with the HACC instruction's data. Simultaneously, the counter for that hash-line is decremented by one (step ❹). The accumulated


`hash_accumulate` Instruction

128 bits

opcode	Reg 0	Reg 1	Reg 2	Reg 3	
HACC	*TAG*	*DATA*	*Rolling Counter*	*NeuraMem ID*	Unused
8 bits	32 bits	32 bits	32 bits	8 bits	16 bits

Figure 9: HACC instruction bit layout.

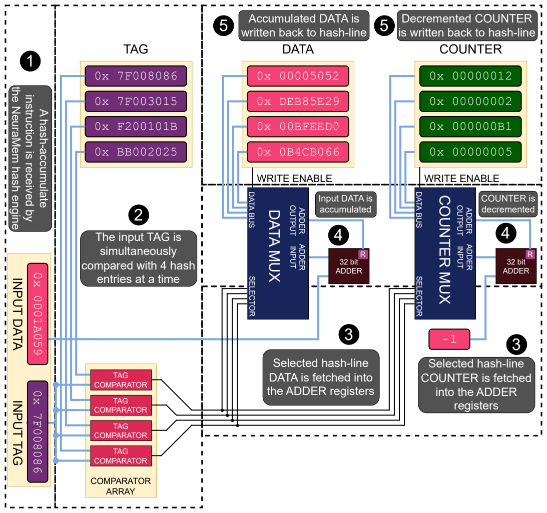

A hash-accumulate instruction is received by the NeuraMem hash engine
INPUT DATA
0x 0001A059
INPUT TAG
0x 7F008086
TAG
0x 7F008086
0x 7F003015
0x F200101B
0x BB002025
The input TAG is simultaneously compared with 4 hash entries at a time
TAG COMPARATOR
COMPARATOR ARRAY
Accumulated DATA is written back to hash-line
DATA
0x 00005052
0x DEB85E29
0x 00BFEED0
0x 0B4CB066
WRITE ENABLE
Input DATA is accumulated
DATA MUX
32 bit ADDER
Selected hash-line DATA is fetched into the ADDER registers
Decremented COUNTER is written back to hash-line
COUNTER
0x 00000012
0x 00000002
0x 000000B1
0x 00000005
COUNTER is decremented
COUNTER MUX
-1
Selected hash-line COUNTER is fetched into the ADDER registers

Figure 10: NeuraMem Hash-Engine accumulates partial products using the HACC instruction.

data and the updated counter are then written back to the HashPad in step ❺. If the TAG from the instruction does not match any of the TAGs in the HashPad in step ❷, the Hash-Engine creates a new entry for the hash instruction and stores its content in a new hash-line.

Rolling Evictions: The Hash-Engine monitors the completion of partial product accumulation (via the COUNTER, as seen in Figure 8. Once the COUNTER reaches zero, indicating that all partial products for a particular TAG have been accumulated, the Hash-Engine automatically evicts the corresponding hash-line, and the accumulated result is written back to the main memory (HBM). This ensures that the hashed partial product spends the minimal possible number of cycles in the HashPad, addressing the memory bloat issue.

3.5 Dynamically Reseeding Hash-based Mapping

The performance benefits provided by the NeuraChip accelerator are primarily due to our sparsity-agnostic mapping algorithm, named Dynamically Reseeding Hash-based Mapping (DRHM). DRHM

Algorithm 2 HACC Instruction Execution

```
 1: $index \leftarrow$ Hash($TAG$)
 2: if $tag\_array[index]$ == EMPTY then
 3:     $data\_array[index] \leftarrow DATA$
 4:     $counter\_array[index] \leftarrow COUNTER$
 5: else if $tag\_array[index]$ == $TAG$ then
 6:     $data\_array[index]$ += $DATA$
 7:     $counter\_array[index]$ −= 1
 8:     if $counter\_array[index]$ == 0 then
 9:         Hash Line Eviction Routine
10:     end if
11: else
12:     Hash Collision Routine
13: end if
```

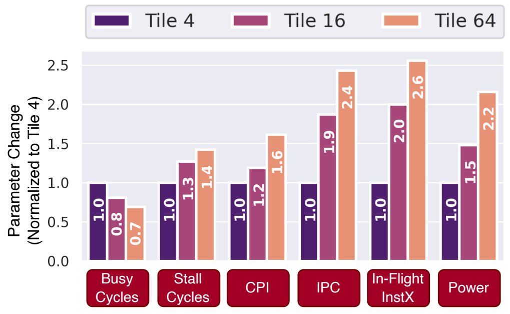


Figure 11: Architectural impact of GCN model varying tile configuration on Cora dataset.

is designed to eliminate computational patterns, promoting an even distribution of workload across all computational resources. Traditional hash-based mappings often lead to concentrated areas of high activity, known as hot spots, especially when the hash function is optimized for a specific sparsity pattern but encounters a different one. An ideal solution would involve uniformly distributing computational tasks across resources. One such method is random mapping, where tasks are allocated to random resources. However, maintaining consistency in random mapping requires extensive record-keeping (a large lookup table), which is impractical.

We introduce a hybrid approach, Dynamically Reseeding Hash-based Mapping (DRHM), which combines the advantages of consistent lookup times in hashing, a distribution akin to random mapping, and minimal overhead similar to small lookup tables. This method significantly reduces the occurrence of hot spots in the allocation of computational resources.

DRHM utilizes a flexible mapping that adjusts based on a 'seed' parameter, denoted as γ. This parameter is specifically designed to alter the mapping, and consequently, the hash function dynamically. After each row of the input sparse matrix is computed, γ is initialized with a random number. DRHM offers two implementation approaches: one using the k upper bits of the TAG, and the other utilizing the k lower bits of the TAG. The lower-bit and upper-bit hashing equations that accommodate γ seed are presented in Equations 3 and 4.

$$H_l(\text{TAG}_{32}, \gamma) = ((\text{TAG}_{32} \ll k) \gg k) \cdot \gamma \mod N \tag{3}$$
$$H_h(\text{TAG}_{32}, \gamma) = ((\text{TAG}_{32} \gg k) \ll k) \cdot \gamma \mod N \tag{4}$$

where TAG represents the unique identifier for each row of the input graph. The term γ acts as a 'seed' to introduce randomness in the mapping. N signifies the total number of available output hash spaces. The operations "$\ll k$" and "$\gg k$" refer to bitwise left and right shifts by k positions, respectively. The modulus operation mod ensures that the result of the hash function falls within the predefined range of the hash table. These equations assume that the bit-shift operations conform to standard behavior where bits shifted beyond the boundary of the number's bit-width are discarded.

In our experiments, we assessed both upper k-bit address hashing and lower k-bit address hashing. We found that the lower k-bit address hashing method had a lower incidence of hash collisions, due to the higher variability in the lower bits of the address. Consequently, in all the work presented here, we employ the lower k-bit address hashing technique (Equation 3). Our compute mapping efficiancy using DRHM approach is evaluated in Section 4.

Table 2: Individual Component Configuration

Component	Elements	Tile-4	Tile-16	Tile-64
NeuraCore	Pipeline Registers	4	8	16
	Pipelines	2	4	8
	Multipliers	2	4	8
	Addr. Generators	1	2	2
	Ports	4	4	4
NeuraMem	Comparators	1	4	8
	Hash-Engines	2	4	8
	Hashlines	4096	2048	2048
	Accumulators	128	256	512
	Ports	4	4	4

4 DESIGN SPACE EXPLORATION OF NEURACHIP

The flexibility of our NeuraSim simulator enables us to evaluate multiple NeuraChip configurations. We have two primary design goals: i) optimizing resource utilization across the accelerator to enhance speedup and ii) striking a balance between performance, chip area, and power consumption to make sure the advantages outweigh the costs [4, 45].

Tile Size Variation: We define a tile as a modular unit that can be configured with varying amounts of computational and memory resources. Each tile is connected to a dedicated HBM memory channel, with NeuraChip hosting a total of eight tiles to match the eight available memory channels. We introduce three distinct configurations of NeuraChip, named Tile-4, Tile-16, and Tile-64, derived from experimenting with various workloads. The detailed configurations of NeuraCore and NeuraMem components are provided in Table 2, while the overall accelerator configurations for these tile sizes are listed in Table 3. We focus on six key parameters to assess the architectural impact of these configurations, as shown in Figure 11. Key observations include:

- **Register File Size**: Expanding the register file size allows more MMH4 instructions to be in-flight and increases the number of read memory instructions that can be issued to HBM. Beyond 8 registers per pipeline (1024 bits per pipeline), we noticed that the DRAM channels are unable to keep up with the high memory demands. This bottleneck is evident in the rise in the cycles per instruction (CPI) and the number of stall cycles [26, 39], as shown in Figure 11.
- **HashPad Size**: Choosing between smaller HashPads with a larger number of NeuraMems versus larger HashPads with fewer NeuraMems, the former proves advantageous for handling extremely sparse matrices. This configuration benefits from high accumulation throughput as the number of accumulators increases with the number of NeuraMems. This can be seen in the larger number of in-flight HBM memory instructions in Figure 11.
- **Component Counts**: With 32, 128, and 512 NeuraCores and NeuraMems in Tile-4, Tile-16, and Tile-64, respectively, while more components enhance peak compute throughput, the configuration is bound by a peak DRAM bandwidth of 128 GB/s. Additionally, workloads do not require a 12 MB on-chip memory HashPad (of tile-64 configuration).

Table 3: NeuraChip Configuration

Parameter	**Tile-4***	**Tile-16***	**Tile-64***
Tile Count	8	8	8
NeuraCores per tile	1	4	16
Total NeuraCores	8	32	128
NeuraMems per tile	1	4	16
Total NeuraMems	8	32	128
Memory Controller Count	8	8	8
Routers per tile	4	8	32
Total Routers	32	64	256
Total Pipelines	32	128	512
Pipeline Register File (bits)	512	1024	2048
Total Hash-Engines	16	128	1024
Hash-Engine comparators	2	4	8
Total TAG comparators	32	512	8192
Total HashPad Size (MB)	0.75	3	12
Max frequency (*GHz*)	1	1	1

Hash-based Mapping Algorithm Variations: We tested four hash-based mapping schemes. The first, a ring-based mapping (see Figure 12), follows round-robin resource allocation, though encounters hot spots in workload distribution. The second, a modular hash-based mapping, uses prime numbers for workload mapping, proposed in previous studies [14, 30, 44, 52]. DRHM, shown in Figure 12, addresses hot spots in modular and ring-based mappings by reseeding the hash function after each row of computations. Lastly, we evaluate a random mapping that maintains a lookup table for each entry. All four techniques are compared in Figure 13 for varying sparsity patterns.

Variations in MMH and HACC Instructions: NeuraChip introduces MMH and HACC instructions (bit layout of these instructions is illustrated in Figure 7 and Figure 9), supporting its decoupled architecture [41]. We analyze the cycle count of various MMH instruction tile sizes, presented in a CPI histogram in Figure 14. MMH4 emerges as the top choice, balancing temporal locality benefits and cycle count.

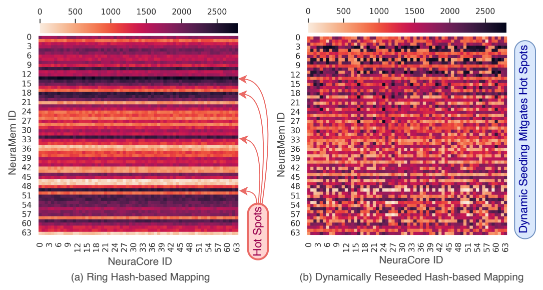


Figure 12: Compute mapping heat map, where the X-axis represents multiplications mapped to NeuraCores and Y-axis represents accumulations mapped to NeuraMem.

We compare the HACC instruction's efficiency using two eviction schemes: barrier-based eviction (HACC-BE) and our rolling eviction approach (HACC-RE). The latter's superiority in reducing average cycle completion is seen in Figure 15.

5 EVALUATION

5.1 Experimental Setup

To evaluate the benefits of NeuraChip, we perform benchmarking across two distinct categories of workloads. The first category involves examining NeuraChip's efficiency in handling sparse matrix multiplication tasks. This evaluation uses a standard array of sparse matrices obtained from the Stanford SNAP sparse matrix collection [21]. Our evaluation includes a comparison with some of the latest state-of-the-art sparse matrix accelerators [51, 53] and off-the-shelf mainstream hardware platforms. NeuraChip is benchmarked against the Intel MKL library [48] with an Intel Xeon E5-2630 CPU. We also compare against cuSPARSE [29] and CUSP [11] NVIDIA libraries, as run on a Hopper architecture H100 GPU, and we also consider for comparison an AMD's MI100 GPU using the hipSPARSE library with a rocSPARSE backend [34]. For accelerator comparisons, we compare NeuraChip against OuterSPACE [31] SpArch [53], and Gamma [51]. Additionally, as to the second category of workloads, our evaluation targets a Graph Convolutional Network (GCN) [20] layer using various datasets, allowing us to compare NeuraChip against existing Graph Neural Network (GNN) accelerators EnGN [25], GROW [17], HyGCN [50], and FlowGNN [36].

5.2 Simulator Framework

In this study, we present NeuraSim, a cycle-accurate, multi-threaded, modular simulation engine inspired by the Structural Simulation Toolkit (SST) [35]. NeuraSim's modular framework allows for flexible integration of new architectural features, without the need for an entire overhaul of the simulation engine. Developed using POSIX threads (pthreads), NeuraSim facilitates parallel simulation. Its dispatcher unit recognizes independent tasks and concurrently executes them on different threads. Additionally, NeuraSim employs MongoDB for backend data storage. NeuraSim also incorporates HBM2 memory simulation, integrating with DRAMsim3 [22], a cycle-accurate and validated DRAM simulator.

Regarding simulation efficiency, NeuraSim achieves 112 Kilocycles per second (KCPS), 48 KCPS, and 11 KCPS on average for the Tile-4, Tile-16, and Tile-64 configurations, respectively. NeuraSim is open-source and faithfully simulates the extended NeuraChip ISA. The NeuraSim source code is accessible on our GitHub repository.

5.3 Comparative Analysis with Sparse Matrix Accelerators

In Figure 16, the performance of the NeuraChip in sparse matrix multiplication tasks is compared against various off-the-shelf high-end CPU and GPU platforms, as well as against state-of-the-art SpGEMM accelerators.

As we can see, NeuraChip outperforms the CPU and GPU computing platforms in all cases. The average performance improvements are 22.2× over the CPU, a 17.1× and 13.3× average speedup

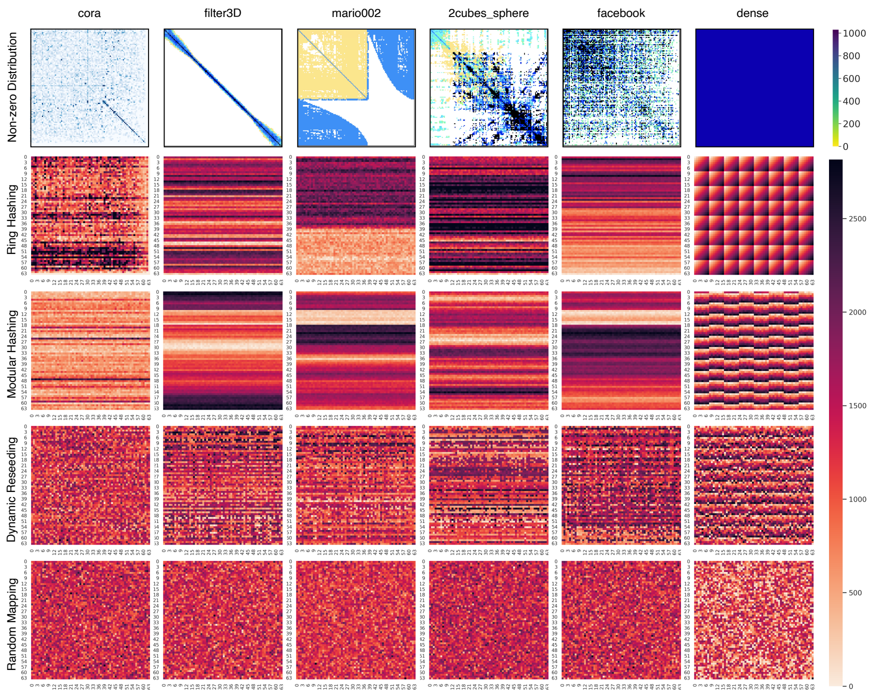


Figure 13: Computation mapping heat maps for four distinct hash-based mapping methods, evaluated across five sparse matrices and one dense matrix multiplication. The dynamic reseeding mapping technique is insensitive to sparsity patterns and effectively addresses hot spots in dense matrix computations.

over the NVIDIA Hopper GPU using the cuSPARSE and CUSP libraries, respectively, and 16.7× average speedup over the AMD's MI100 GPU using hipSPARSE.

Further, the performance of NeuraChip is evaluated against two outer-product-based sparse matrix accelerators: OuterSPACE [31] and SpArch [53]. While OuterSPACE leverages input data reuse, it encounters excessive generation of partial products (the memory bloat issue), leading to degraded performance. SpArch addresses this with on-chip merger trees; however, these trees require large comparator arrays, occupying about 60% of the chip area. NeuraChip counters the memory bloat through an on-chip cache organization with rolling counters, effectively managing the eviction of accumulating partial products and alleviating the bloat issue. In comparison, NeuraChip surpasses OuterSPACE and SpArch by factors of 6.6× and 2.4×, respectively.

Additionally, the performance of NeuraChip is compared with a row-wise product-based SpGEMM accelerator, Gamma [51], which is based on Gustavson's algorithm. Gamma employs a resource-intensive storage mechanism, FiberCache, to prefetch data, aiming to reduce data fetch latency and prevent compute stalls. However, this approach results in data remaining idle in the caches prior to being accessed by the processing elements. NeuraChip, in contrast, optimizes on-chip storage through a rolling-eviction strategy, enabling automatic eviction of partial products after the reduce operation is complete. Against Gamma, NeuraChip demonstrates a performance superiority of 1.5× average speedup.

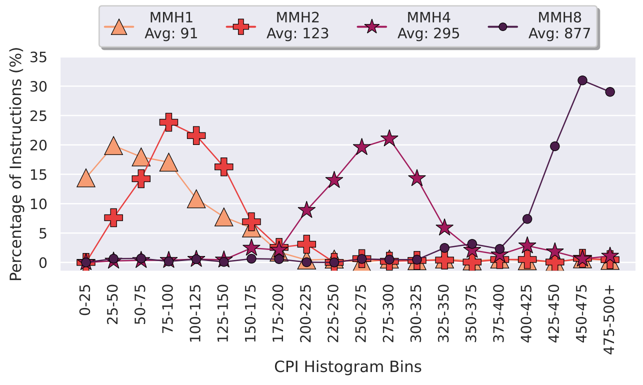


Figure 14: Cycles Per Instruction (CPI) histogram plot for four MMH instructions with varying tile sizes.

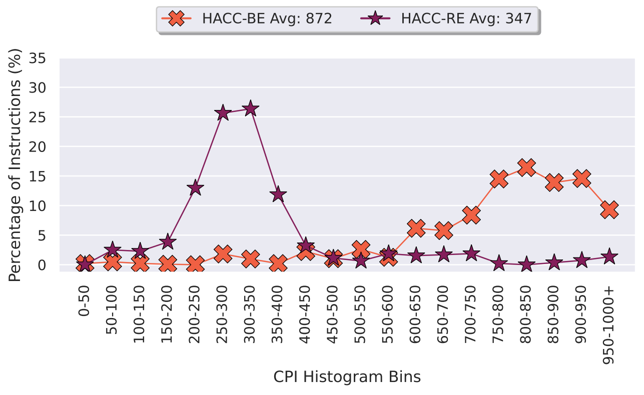


Figure 15: CPI histogram for HACC instructions: barrier-based evictions HACC-BE and rolling evictions HACC-RE.

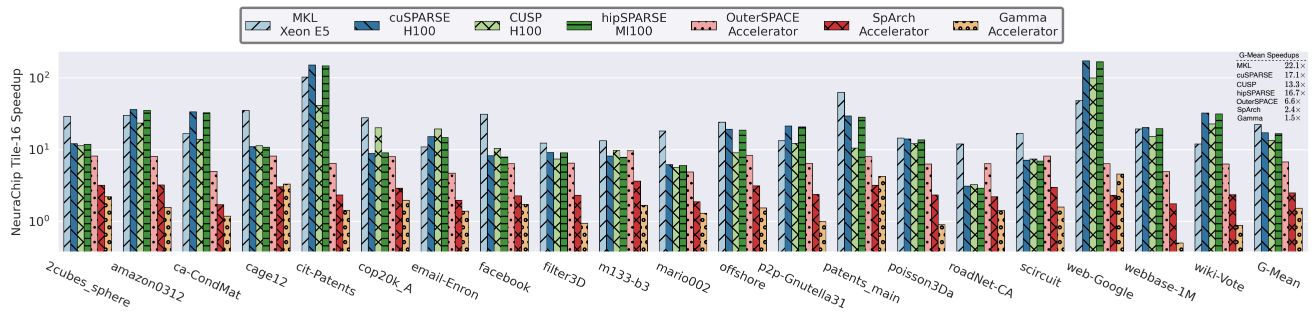


Figure 16: Speedup comparison of NeuraChip Tile-16 for SpGEMM against CPUs, GPUs, and SpGEMM accelerators.

5.4 Comparative Analysis of GNN Accelerators

In Figure 17, we compare the GNN performance of NeuraChip against various state-of-the-art GNN accelerators. The NeuraChip configuration used for GNN assessment differs from that used to compare to SpGEMM accelerators in Table 3. Specifically, for the Tile-16 configuration in the GNN accelerator analysis, an architecture comprising 8 tiles is used. Each tile includes a 16 × 16 grid of NeuraCores, with each core featuring a quad-pipeline design. We have significantly reduced the number of TAG comparators and port buffers, while retaining the hashpad sizes. This particular configuration is capable of delivering a peak performance of 8192 GFLOPs, with an average power consumption of 4.3W.

First, we consider EnGN, a hash-based GNN accelerator [25], and GROW [17]. EnGN employs a unique ring-based edge reducer to efficiently map vertex IDs. However, it encounters challenges in achieving a uniform distribution of computational tasks among its processing elements. In comparison, NeuraChip demonstrates superior performance, outperforming EnGN by 29% on average. This improvement is primarily attributed to the dynamic reseed hashing function within NeuraChip, which ensures balanced task distribution across its computational resources, namely NeuraCore and NeuraMem, thus minimizing processing delays.

GROW utilizes a row-wise multiplication method, incorporating hardware and software co-design elements. A notable aspect of GROW's software strategy is its reliance on graph partitioning, which significantly increases the computational overhead for GNN processing. From a hardware perspective, GROW is equipped with vector processors and employs streaming buffers for handling input and output matrix data. Despite these features, GROW encounters issues similar to those seen in Gamma's prefetcher system, where data idling results in suboptimal usage of on-chip memory resources. Comparative performance metrics indicate that NeuraChip surpasses GROW's performance by an average of 58%.

Next, we evaluate our accelerator compared to HyGCN, a hybrid Graph Neural Network (GNN) accelerator, which has specialized engines for aggregation and combination phases [50]. The primary advantage of HyGCN's architecture is its ability to pipeline computations, which is particularly beneficial for GNN layers that typically alternate between aggregation and combination phases. However, a significant limitation arises when the compute duration for one phase substantially exceeds the other, leading to a pipeline stall due to the uneven execution duration of each pipeline stage.

Table 4: NeuraChip Power and Area Breakdown for SpGEMM workloads

Unit	Area (mm^2) Tile-4	Area (mm^2) Tile-16	Area (mm^2) Tile-64	Average Power (W) Tile-4	Average Power (W) Tile-16	Average Power (W) Tile-64
NeuraCore	0.28	2.74	9.36	1.05	1.86	5.76
NeuraMem	1.22	5.10	18.64	6.85	7.36	11.19
Router	0.49	1.98	6.88	2.15	4.88	4.43
Memory Controller	0.38	0.38	0.38	1.41	1.96	2.84
Total	2.37	10.2	35.26	11.46	16.06	24.22

Instead, NeuraChip incorporates distinct components specifically for multiplication and accumulation operations, utilized in both the aggregation and combination phases. This design choice renders NeuraChip impervious to the inefficiencies caused by varying computational times between aggregation and combination phases. On average, NeuraChip outperforms HyGCN's performance by 69%.

Our final comparison is with FlowGNN [36], a reconfigurable dataflow GNN accelerator comprising Node Transformation Units (NTs) and Message Passing Units (MPs). FlowGNN uses queues for real-time task buffering and relies on dynamic pull-based mapping for task distribution to NTs and MPs. In contrast, NeuraChip adopts a push-based mapping strategy for multiplication tasks and a hash-based approach for accumulation. The Dispatcher in NeuraChip assigns MMH4 instructions to NeuraCores, optimizing input data temporal locality (reuse in NeuraCore register files). The dynamic reseeding hash-based mapping, as detailed in Section 3.5, ensures uniform workload distribution regardless of sparsity patterns. Consequently, NeuraChip achieves an average speedup of 30% over GCN workloads tested on the FlowGNN architecture.

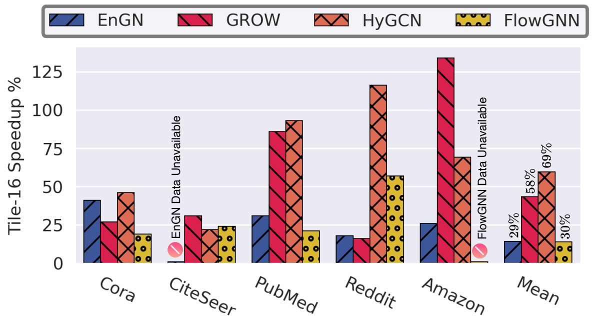


Figure 17: Tile-16 speedup over previous GNN accelerators for GCN workloads on various graph datasets.

Table 5: Performance comparison of state-of-the-art SpGEMM accelerators across various NeuraChip system configurations.

Architectural Parameters	Xeon E5	NVIDIA H100	AMD MI100	Outer SPACE	SpArch	Gamma	**NeuraChip Tile-4**	**NeuraChip Tile-16**	**NeuraChip Tile-64**
Compute Units	8 Cores AVX2	7296 FP64	7680 FP64	256 PEs	2×8 Mults 16×16 Merger	32 PEs Radix-64	2×4 NeuraCores	2×16 NeuraCores	2×64 NeuraCores
Frequency (GHz)	2.9	1.6	1.5	1.5	1	1	1	1	1
Peak Performance	186 GFLOPs	26 TFLOPs	11.5 TFLOPs	384 GFLOPs	32 GFLOPs	32 GFLOPs	8 GFLOPs	32 GFLOPs	128 GFLOPs
SpGEMM Perf.$^{\Phi}$ (GOP/s)	1.12	1.86	1.48	2.9	10.4	16.5	5.15	24.75	30.69 93.17^{α}
On-chip Memory	15 MB$^{\tau}$	50 MB†	8 MB†	4 MB	15 MB*	3 MB*	0.75 MB$^{\delta}$	3 MB$^{\delta}$	12 MB$^{\delta}$
Off-chip Memory	DDR4 136GB/s	HBM 2TB/s	HBM 1.2TB/s	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s
Technology (nm)	32	4	7	32	40	45	7	7	7
Area (mm^2)	356	814	750	86.74	28.49	30.6‡	2.37	10.2	35.26
Power (W)	85$^{\diamond}$	300$^{\diamond}$	300$^{\diamond}$	24	9.26	❖	11.46	16.06	24.22
Energy Efficiency (GOPS/W)	◆	◆	◆	0.120	1.123	❖	0.449	1.541	1.266
Area Efficiency (GOPS/mm^2)	◆	◆	◆	0.034	0.365	0.539	2.171	2.426	0.870
Tile-16 Speedup	22.1×	13.3×	16.7×	6.6×	2.4×	1.5×	4.8×	1×	0.807×

$\diamond$Max thermal dissipation power from datasheet ❖Gamma lacks a power performance model † L2 cache size $^{\tau}$ L3 cache size $^{\delta}$HashPad Size *FiberCache Size ◆Power and area metrics sourced from vendor datasheets; derived metrics excluded. $^{\alpha}$ Simulated with dual stacked HBM offering peak bandwidth of 256 GB/s. ‡Gamma synthesizes accelerator using 45 nm and 40 nm processes, resulting in computing areas of 30.6 mm^2 and 20.44 mm^2, respectively. *Represents column fetchers, row prefetchers, and partial matrix fetchers and writers. $^{\Phi}$Computed on common set of matrices as shown in Table 1.

5.5 Power Consumption and Area Analysis

We assess our accelerator's area and power overheads by implementing its design in Register Transfer Level (RTL). Using Cadence Genus Synthesis Solutions, we synthesize these RTL components targeting an ASAP7 technology library [10], allowing us to determine the area and power consumption for each proposed microarchitectural element. The synthesized chip area requirements for NeuraChip amount to 2.37mm^2, 10.2mm^2, and 35.26mm^2 for the Tile-4, Tile-16, and Tile-64 configurations, respectively. The breakdown of NeuraChip's area and power is shown in Table 4. The majority of the area requirement for NeuraChip is allocated to the NeuraMem unit, as it includes the tag comparator array and the hash-pad (on-chip storage).

6 RELATED WORK

SpGEMM Accelerators: The InnerSP [3] accelerator uses the inner-product method for matrix multiplication. This method offers advantages, eliminating the need for on-chip memory for accumulation. However, it suffers from limited input data reuse of both matrices, leading to performance issues when the sparsity patterns do not align with their task mapping algorithm. MatRaptor [46] employs a row-wise multiplication strategy and a round-robin greedy algorithm for allocating input rows to processing elements (PEs). Although this approach enhances input data reuse, it struggles with irregular sparsity patterns. The simplistic round-robin distribution may result in computational hot spots (as elaborated in Section 4). SIGMA [33] offers an SpGEMM accelerator equipped with adaptable interconnects. Utilizing a smart global controller, SIGMA dynamically assigns each non-zero pair to PEs via a Benes network. Despite its efficiency in general SpGEMM tasks, SIGMA is less effective with large sparse matrix computations due to the substantial overhead introduced by its bitmap compression format.

GNN Accelerators: LISA [24] performs GNN computations on Coarse-Grained Reconfigurable Arrays (CGRAs). LISA generates a dataflow graph and utilizes a simulated annealing method for mapping. I-GCN [12] aims to enhance data locality through an *islandization* strategy, clustering densely connected nodes to reduce off-chip memory accesses. However, both the simulated annealing and graph clustering methods introduce considerable computational overheads.

7 CONCLUSION

NeuraChip demonstrates the potential advantages that sparse matrix multiplication workloads can gain from a decoupled architectural design. We have presented an open-source, cycle-accurate simulator called NeuraSim, used to demonstrate the effectiveness of our design. GNN workloads acceleration is achieved through a blend of high-level optimizations and microarchitectural features. We synthesized our design in RTL, giving us power and area requirements for various NeuraChip Tile sizes. NeuraChip outperforms state-of-the-art SpGEMM accelerator by a factor of 1.5× and prior GNN accelerators by 1.46× on average. In future work, we plan to explore the fabrication of NeuraChip to fully demonstrate the benefits of our approach.

A APPENDIX

A.1 Artifact Evaluation

NeuraChip is a GNN accelerator that is built using the NeuraSim simulator. NeuraSim is a cycle-accurate simulator that contains six modules as follows:

(1) **NeuraSim Engine**: The cycle accurate simulator engine built in C++.
(2) **NeuraCompiler**: A pythonic compiler that takes graphs as inputs and generates a SpGEMM workload binary to be executed by the NeuraSim Engine. The NeuraCompiler uses an extended x86 ISA.
(3) **Mongo**: The Mongo module is a part of NeuraSim Engine that is coupled with an instance of MongoDB server. Throughout the simulation, NeuraSim pushes performance metrics to the MongoDB server.
(4) **NeuraViz**: A pythonic module that generates all plots to visualize the data. NeuraViz is coupled with MongoDB and fetches simulation metrics from the MongoDB server.
(5) **Dashboard**: For ease of use, NeuraChip is hosted on our servers and accessible to users via a WebGUI named Dashboard, available at https://neurachip.us.
(6) **DRAMSim3**: NeuraChip utilizes the memory simulator provided by Li et al., called DRAMSim3 [22] for computing HBM memory request latencies.

This appendix describes where to access our code artifact and how to reproduce part of our experiments and results. We use a Git repository and host an instance of our simulator, which is available through a public WebGUI, to schedule simulations of different configurations and display results. Additionally, we provide instructions on how to natively build NeuraSim in the project's README.

A.2 Artifact Checklist

- Benchmarks: SpGEMM workload over a common set of matrices mentioned in Table 1.
- Runtime environment: Tested on Ubuntu 22.04, though it should be reproducible when run on any operating system as long as Docker is utilized. Simulations using the hosted WebGUI only require a web browser.
- Hardware: To compile and run the simulation experiments, we recommend a machine with 32 GB of DRAM memory, 50 GB of disk storage, and 8 cores.
- Execution: We provide a web interface that allows the execution of different experiments. Docker instructions are also provided so users can build the environment themselves.
- Metrics: SpGEMM performance in GFLOP/s, Execution time, NeuraCore utilization, NeuraMem utilization, Histogram of number of cycles spent by instructions in Register File, NeuraRouter Utilization, Average In-Flight Memory Transactions Per Cycle (Memory Pressure).
- Output: Experiments deployed on the WebGUI generate simulation metrics, which are used to generate the graphs/plots.
- Disk space required: Around 50GB, which includes installing necessary dependencies, building NeuraSim, and running one simulation.
- How long does it take to prepare the workflow?: The compilation/installation using Docker takes around 30 minutes on a 32-cores machine with 64GB DDR4 memory.
- Experiment completion time: 30 minutes.
- Publicly available: Yes, https://github.com/NeuraChip/neurachip.
- Archived: Yes, https://doi.org/10.5281/zenodo.10896280.

A.3 How to access?

The easiest way to access NeuraChip is via the WebGUI; however, compiling it from source is also possible. Here are the simplified steps to run simulations using the WebUI:

(1) Visit https://neurachip.us
(2) Click on "Launch a new simulation".
(3) On the *New NeuraSim Simulation* page, click on "Launch" and wait for 20 minutes.
(4) Once all the plots are generated, a "Results" button will pop up at the bottom of the page. Click on the "Results" button to view many simulation performance metrics on the *NeuraViz Results* page.

To compile the NauraSim locally on a platform with Docker:

(1) Clone the repository from GitHub
`git clone https://github.com/NeuraChip/neurachip.git`
(2) Enter the directory
`cd neurachip`
(3) Build Docker image from docker file
`./build-docker.sh`
(4) Start the docker container
`./start-docker.sh`
(5) Run MongoDB server within Docker container
`sudo -u mongodb mongod –config /etc/mongod.conf –fork`
(6) Enter NeuraSim directory
`cd /home/ktb/git/NeuraChip/NeuraSim`
(7) Compile NeuraSim with compile script
`./compile-run.sh`
(8) Run the Simulator
`./chippy.bin`
(9) After execution, the NeuraSim will print to the terminal the total number of cycles for a NeuraChip to execute the workload, duration of simulation, and number of simulated instructions per second. Other metrics are stored in the simulation database.

A.3.1 Hardware dependencies.
: We recommend a minimum of 64 GB of DRAM memory and 50 GB of storage space.

A.3.2 Software dependencies.
In our execution environment, all software dependencies necessary for operation are fully satisfied. Additionally, when compiling NeuraSim simulator from scratch, we utilize Docker to manage and meet all the required dependencies.

A.3.3 Datasets.
Datasets for our simulations are derived from Table 1. Using the default configurations, we perform simulations of SpGEMM computations on NeuraChip configured with 16 NeuraCores and 16 NeuraMems. These simulations utilize the Cora workload under a setup referred to as the Tile-16 configuration.

A.4 Execution

We grant evaluators remote access to our simulator's web interface. Below are the steps to interact with the NeuraSim simulator available on our website.

(1) Accessing the NeuraSim Simulator WeGUI:
    - Open your web browser and navigate to the URL provided for the NeuraSim simulator website: https://neurachip.us.

(2) Navigating to Configuration Selection:
    - Upon loading the website, you will be directed to the main page of the NeuraSim simulator.
    - Look for a button labeled *Launch a new Simulation*.

(3) Selecting Configurations:
    - Once on the Configuration Selection page, you will see a list of selectable presets that configure NeuraChip parameters.
    - Select the configurations to evaluate.
    - If you wish to save the results to view later, change the "Experiment Name" to something unique, and after the results have been generated, bookmark the webpage in your browser. To preserve anonymity, do not use personal/recognizable experiment names.

(4) Initiating the Simulation:
    - After selecting the desired configurations, locate the *Launch* button or option to initiate the simulation process.
    - Click on the button to start the simulation with the chosen configurations.

(5) Monitoring Simulation Progress:
    - While the simulation is running, you may observe progress indicators or status updates to track the simulation's progress.
    - Depending on the complexity of the simulation and the chosen configurations, this process may take some time.

(6) Viewing Simulation Results:
    - Once the simulation completes, the website should display the *Results* button.
    - Look for a section labeled *Simulation Results*, where you can observe the outcome of the simulation.
    - If you have given your experiment a unique *Experiment Name*, you can bookmark this link and re-visit it at a later time. Experiments with unique names are preserved on the database and not deleted even if the browser window is closed.

(7) Analyzing the Results:
    - Simulation metrics, along with their description, are provided on the web page.

ACKNOWLEDGMENTS

This research was supported in part by the Institute for Experiential AI and the NSF IUCRC Center for Hardware and Embedded Systems Security and Trust (CHEST), NSF CNS 2312275, NSF CNS 2312276, Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd., NRF-2023R1A2C2004229, and IITP No.RS-2023-00228255. Additionally, we acknowledge the financial assistance from grant RYC2021-031966-I funded by MCIN/AEI/10.13039/501100011033 and the "European Union NextGenerationEU/PRTR.", and grant PID2022-136315OB-I00 funded by MCIN/AEI/10.13039/501100011033/ and by "ERDF A way of making Europe", EU.

REFERENCES

[1] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi, and Davide Patti. 2008. Implementation and analysis of a new selection strategy for adaptive routing in networks-on-chip. *IEEE transactions on computers* 57, 6 (2008), 809–820.

[2] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2016. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 38, 6 (2016), C624–C651.

[3] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. Innersp: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 116–128.

[4] Nilanjan Banerjee, Praveen Vellanki, and Karam S Chatha. 2004. A power and performance model for network-on-chip architectures. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 2. IEEE, 1250–1255.

[5] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A Mojumder, Kihoon Jung, José L Abellán, Yash Ukidave, Ajay Joshi, John Kim, et al. 2021. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–23.

[6] Rohit K Bhullar, Lokesh Pawar, Vijay Kumar, et al. 2016. A novel prime numbers based hashing technique for minimizing collisions. In *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*. IEEE, 522–527.

[7] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. 2008. Strong accumulators from collision-resistant hashing. In *Information Security: 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings 11*. Springer, 471–486.

[8] Zhiruo Cao, Zheng Wang, and Ellen Zegura. 2000. Performance of hashing-based schemes for internet load balancing. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, Vol. 1. IEEE, 332–341.

[9] Lianhua Chi and Xingquan Zhu. 2017. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (Csur)* 50, 1 (2017), 1–36.

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

[11] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/ Version 0.5.0.

[12] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. 2021. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *MICRO-54: 54th annual IEEE/ACM international symposium on microarchitecture*. 1051–1063.

[13] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.

[14] Xiangyang Gou, Chenxingyu Zhao, Tong Yang, Lei Zou, Yang Zhou, Yibo Yan, Xiaoming Li, and Bin Cui. 2018. Single hash: Use one hash function to build faster hash based data structures. In *2018 IEEE international conference on big data and smart computing (BigComp)*. IEEE, 278–285.

[15] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (sep 1978), 250–269. https://doi.org/10.1145/355791.355796

[16] William L Hamilton. 2020. *Graph representation learning*. Morgan & Claypool Publishers.

[17] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo Rhu. 2023. GROW: A Row-Stationary Sparse-Dense GEMM Accelerator for Memory-Efficient Graph Convolutional Neural Networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 42–55.

[18] Malith Jayaweera, Kaustubh Shivdikar, Yanzhi Wang, and David Kaeli. 2021. JAXED: Reverse Engineering DNN Architectures Leveraging JIT GEMM Libraries. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. 189–202. https://doi.org/10.1109/SEED51797.2021.00030

[19] Gilbert Jonatan, Haeyoon Cho, Hyojun Son, Xiangyu Wu, Neal Livesay, Evelio Mora, Kaustubh Shivdikar, José L Abellán, Ajay Joshi, David Kaeli, et al. 2024. Scalability Limitations of Processing-in-Memory using Real System Evaluations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 8, 1 (2024), 1–28.

[20] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[21] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[22] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[23] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 747–761. https://doi.org/10.1145/3575693.3575706

[24] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. 2022. Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 444–459.

[25] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. 2020. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Trans. Comput.* 70, 9 (2020), 1511–1525.

[26] Neal Livesay, Gilbert Jonatan, Evelio Mora, Kaustubh Shivdikar, Rashmi Agrawal, Ajay Joshi, José L. Abellán, John Kim, and David Kaeli. 2023. Accelerating Finite Field Arithmetic for Homomorphic Encryption on GPUs. *IEEE Micro* 43, 5 (2023), 55–63. https://doi.org/10.1109/MM.2023.3253052

[27] Francisco Muñoz Martínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. STIFT: A Spatio-Temporal Integrated Folding Tree for Efficient Reductions in Flexible DNN Accelerators. *J. Emerg. Technol. Comput. Syst.* 19, 4, Article 32 (sep 2023), 20 pages. https://doi.org/10.1145/3531011

[28] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, Piscataway, NJ, 101–110.

[29] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

[30] Wing WY Ng, Yueming Lv, Daniel S Yeung, and Patrick PK Chan. 2015. Two-phase mapping hashing. *Neurocomputing* 151 (2015), 1423–1429.

[31] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[32] Hongwu Peng, Xi Xie, Kaustubh Shivdikar, MD Hasan, Jiahui Zhao, Shaoyi Huang, Omer Khan, David Kaeli, and Caiwen Ding. 2023. Maxk-gnn: Towards theoretical speed limits for accelerating graph neural networks training. *arXiv preprint arXiv:2312.08656* (2023).

[33] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, 58–70.

[34] ROCmSoftwarePlatform. [n. d.]. Rocmsoftwareplatform/hipSPARSE: Rocm Sparse Marshalling Library. https://github.com/ROCmSoftwarePlatform/hipSPARSE

[35] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, et al. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (2011), 37–42.

[36] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2023. FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1099–1112.

[37] Kaustubh Shivdikar. 2021. *SMASH: Sparse matrix atomic scratchpad hashing*. Ph. D. Dissertation. Northeastern University.

[38] Kaustubh Shivdikar. 2023. Enabling Accelerators for Graph Computing. *arXiv preprint arXiv:2312.10561* (2023).

[39] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L Abellán, John Kim, et al. 2023. Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 670–684.

[40] Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L Abellán, John Kim, and David Kaeli. 2022. Accelerating polynomial multiplication for homomorphic encryption on GPUs. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 61–72.

[41] Kaustubh Shivdikar, Kaushal Paneri, and David Kaeli. 2018. Speeding up dnns using hpl based fine-grained tiling for distributed multi-gpu training.

[42] Aaron Smith. 2020. What people like and dislike about Facebook. https://www.pewresearch.org/fact-tank/2014/02/03/what-people-like-dislike-about-facebook/

[43] James E Smith. 1982. Decoupled access/execute computer architectures. *ACM SIGARCH Computer Architecture News* 10, 3 (1982), 112–119.

[44] Fenglong Song, Zhiyong Liu, Dongrui Fan, Junchao Zhang, Lei Yu, Nan Yuan, and Wei Lin. 2009. Design of new hash mapping functions. In *2009 Ninth IEEE International Conference on Computer and Information Technology*, Vol. 1. IEEE, 45–50.

[45] Mohit Srinivasan, Ahan Kak, Kaustubh Shivdikar, and Chirag Warty. 2016. Dynamic power allocation using Stackelberg game in a wireless sensor network. In *2016 IEEE Aerospace Conference*. IEEE, 1–10.

[46] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[47] Toyofumi Takenaka, Satosi Kato, and Hidetosi Okamoto. 2004. Adaptive load balancing content address hashing routing for reverse proxy servers. In *2004 IEEE International Conference on Communications (IEEE Cat. No. 04CH37577)*, Vol. 3. IEEE, 1522–1526.

[48] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. 2014. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures* (2014), 167–188.

[49] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[50] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 15–29.

[51] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

[52] Haowen Zhang, Yuandong Chan, Kaichao Fan, Bertil Schmidt, and Weiguo Liu. 2018. Fast and efficient short read mapping based on a succinct hash index. *BMC bioinformatics* 19 (2018), 1–14.

[53] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[54] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI open* 1 (2020), 57–81.