Understanding and Optimizing Memory Access Behaviors via Hardware/Software Co-design

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Alif Ahmed

December 2023

Abstract

Due to the ever-increasing gap between the speed of processing elements and the speed at which memory systems can feed them with data, current computing systems are often bottlenecked by limited memory bandwidth. To alleviate the memory bandwidth bottleneck, algorithms and data structures for data-intensive applications must be designed to leverage hardware features in the memory hierarchy, and in some cases, a software-hardware co-design approach is beneficial. To that extent, first, we propose Hopscotch, which is a micro-benchmark suite for memory performance characterization. It aims to fill the gap left by existing memory benchmarks by providing kernels covering a wide range of spatio-temporal locality spectrum. Additionally, the Hopscotch suite contains tools for memory-access pattern visualization and empirically deriving the Roofline model, thereby helping users to isolate performance bottlenecks and reverse-engineer memory characteristics. Next, we optimize two data-intensive applications by leveraging the memory-centric hardware features in traditional general-purpose architecture. The BigMap approach optimizes the memory access behavior of a popular fuzzer, AFL, to enable large bitmaps for accurate coverage tracking. In BigMap, we propose a two-level hashing scheme that consolidates scattered random accesses, vastly improving the spatial locality of the bitmap operations. In our next work, GraphTango, we minimize cache line accesses on streaming graphs by proposing a hybrid storage format and a cache-friendly hashing scheme. GraphTango provides excellent update/analytics throughput regardless of a graph's degree distribution, unlike prior approaches. My last two works focus on hardware-software co-optimization for reducing data movement. In Pulley, we provide an in-memory sorting accelerator that can leverage the subarray-level parallelism for a distributed LSB-first radix sorting. This approach avoids the single-point merging bottleneck of the prior near-data and in-memory sorting accelerators. Finally, we propose a vault-level PIM architecture for accelerating the inference tasks on temporal graph neural networks. We incorporate a feature-based partitioning scheme to minimize inter-vault communication and improve the workload balance, resulting in significant throughput gain and latency reduction over other approaches.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Alif Ahmed

This dissertation has been read and approved by the Examining Committee:

Kevin Skadron, Adviser

Ashish Venkat, Committee Chair

Felix Lin

Jundong Li

Adwait Jog

Accepted for the School of Engineering and Applied Science:

Jennifer L. West, Dean, School of Engineering and Applied Science

December 2023

Acknowledgements

I am grateful to many people who have contributed to shaping this dissertation. This dissertation would not have been possible without the influence, advice, and support of many colleagues, friends, and family.

First, I would like to thank my advisor, Prof. Kevin Skadron for his constant and genuine support and guidance during my academic career. He always encouraged me to pursue projects that piqued my interest without constraining me to projects strictly adhering to funding requirements. I will always gratefully remember the time he went through great lengths to reopen the admission portal of UVA so that I can pursue my Ph.D. study alongside my then-wife. Whenever I was stuck in a problem or stressed over any issues, personal or work related, he never showed impatience and always guided me towards the right direction. He was my guardian angle during the demanding years of my Ph.D. life and I will always remember him fondly no matter where I end up in my life. It has been a great honor and privilege to work with him.

Second, I would like to extend my heartfelt gratitude towards my collaborators and co-authors, including Ashish Venkat, Felix Lin, Jundong Li, Jack Davidson, Farzana Ahmed Siddique, Marzieh Lenjani, Jason D. Hiser, Anh Nguyen-Tuong, Farimah Farahmandi, Yuanwen Huang, Yangdi Lyu, Subodha Charles, Jonathan Cruz, and many others. Without their knowledge and inputs, none of my work would have be possible. I would also like to express my gratitude to the many friends and colleagues within and outside our research group, including Lingxi Wu, Rasool Sharifi, Yiqing Yang, Wole Jaiyeoba, Tommy Tracy, Sergui Mosanu, Akhil Sekhar, Tauhid Ahmed, Taufiq Ahmed, Moniruzzaman Liton, Zakaria Mehrab, Kevin Chen, Justin Chen, and many more.

Third, I would like to thank my colleague and ex-wife Farzana for tolerating me for nearly a decade. Although we decided to go separate ways in our life, she always stayed with me as my best friend and took care of me and comforted me in many situations when I felt lost. Last, but certainly not least, I would like to thank my parents for always being there for me and supporting me through every stage of my life.

Contents

Contents iv
List of Tables vii
List of Figures viii

1 Introduction 1
1.1 Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation 4
1.2 BigMap: Future-proofing Fuzzers with Efficient Large Maps 5
1.3 GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Processing 5
1.4 Pulley: An Algorithm/Hardware Co-optimization for In-memory Sorting 6
1.5 TGN-PNM: A Near-Memory Architecture for Temporal GNN Inference on 3D-Stacked Memory 7
1.6 Dissertation Structure 7

2 Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation 9
2.1 Introduction 9
2.2 Related Works 10
2.3 Kernel Implementation 11
2.3.1 Generic Design Decisions 11
2.3.2 Read-Only Kernels 13
2.3.3 Write Only Kernels 14
2.3.4 Mixed Kernels 15
2.4 Evaluation with Hopscotch 15
2.4.1 Bandwidth Measurement 16
2.4.2 Latency Measurement 17
2.4.3 Impact of Locality 18
2.4.4 Roofline Model and Machine Balance 18
2.5 Conclusions 20

3 BigMap: Future-proofing Fuzzers with Efficient Large Maps 21
3.1 Introduction 21
3.2 Background 23
3.2.1 American Fuzzy Lop (AFL) 23
3.2.2 Collision Rate 26
3.3 Implication of Naïve Hash Collision Mitigation Strategy 26
3.3.1 Cost of Expanding Hash-space 27
3.4 BigMap: Adaptive Two-Level Bitmap 28
3.4.1 Two-Level Bitmap Scheme 28
3.4.2 Illustrative Example 29
3.4.3 Access Patterns of the Bitmap Operations 30
3.4.4 Implementation Details 32
3.4.5 Additional Optimizations 33
3.5 Evaluation 33

3.5.1 Experimental Setup . . . 33
3.5.2 Evaluating the Impact of Map Size Variation . . . 35
3.5.3 Evaluating Coverage Metric Composition . . . 38
3.5.4 Evaluating the Scalability with Parallel Fuzzing . . . 39
3.6 Related Work . . . 41
3.7 Conclusion . . . 43

4 GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Updates and Analysis 44
4.1 Introduction . . . 44
4.2 Background on Existing Representation Formats . . . 46
4.3 GraphTango Data Structure . . . 48
4.4 GraphTango Basic Operations . . . 50
4.4.1 Edge Insertion . . . 50
4.4.2 Edge Deletion . . . 51
4.4.3 Edge Traversal . . . 51
4.5 Optimizing GraphTango . . . 51
4.5.1 Cache-Friendly Hashing Scheme . . . 51
4.5.2 Memory Allocation Scheme . . . 53
4.5.3 Parallelization . . . 55
4.5.4 Determining the TH_1 Threshold . . . 56
4.6 Evaluation . . . 57
4.6.1 Experimental Setup . . . 57
4.6.2 Analytics and Update Performance . . . 58
4.6.3 Memory Usage . . . 60
4.6.4 Impact of TH_1 Threshold . . . 61
4.6.5 Impact of Optimizations . . . 61
4.6.6 Integration with DZiG and RisGraph . . . 63
4.7 Details of Hash Function Implementation . . . 64
4.8 Conclusions . . . 65

5 Pulley: An Algorithm/Hardware Co-optimization for In-memory Sorting 67
5.1 Introduction . . . 67
5.2 Background and Motivation . . . 70
5.3 Proposed method . . . 70
5.3.1 Baseline PIM architecture . . . 70
5.3.2 Local sorting . . . 72
5.3.3 Histogram generation . . . 72
5.3.4 Prefix-sum . . . 73
5.3.5 Merging and key placement . . . 73
5.4 Evaluation . . . 73
5.4.1 Methodology . . . 73
5.4.2 Throughput . . . 74
5.4.3 Power and temperature constraints . . . 74
5.5 Conclusions and future Work . . . 74

6 TGN-PNM: A Near-Memory Architecture for Temporal GNN Inference on 3D-Stacked Memory 75
6.1 Introduction . . . 75
6.2 Background and Motivation . . . 78
6.2.1 Temporal Graph Neural Network (TGNN) . . . 78
6.2.2 3D-stacked memory . . . 80
6.3 TGN-PNM Microarchitecture . . . 81
6.3.1 Vault-level Processing Unit (VPU) . . . 81

6.3.2 Global Control Unit (GLCU) . . . 82
6.3.3 Partial-Sum Accumulation Unit (PSAU) . . . 82
6.4 Mapping TGNN Frameworks on TGN-PNM . . . 83
6.4.1 Mapping of common operations . . . 83
6.4.2 Graph storage format . . . 86
6.5 Evaluation . . . 88
6.5.1 Methodology . . . 88
6.5.2 Datasets . . . 89
6.5.3 Mapping on evaluated architectures . . . 89
6.5.4 Throughput and latency results . . . 92
6.5.5 Area estimation . . . 95
6.6 Related Work . . . 96
6.7 Conclusions and Future Work . . . 96

7 Conclusions and Future Work **98**

Bibliography **104**

List of Tables

2.1 A comparison of supported access patterns and platforms of existing memory benchmarks . . 11
2.2 Characteristics of the included kernels 12
2.3 System configuration of the evaluation platforms 13
2.4 Impact of spatio-temporal locality on bandwidth. Single Threaded. 18

3.1 Access Patterns of the Bitmap Operations 31
3.2 Benchmark Characteristics 34
3.3 Code Coverage with laf-Intel and N-gram 38

4.1 Vertex type switching steps for insertion/deletions 50
4.2 Evaluated Datasets 58
4.3 Average Memory Usage (Bytes Per Edge) 59
4.4 Impact of Optimizations on the Update Throughput (Baseline is the proposed hybrid format without any optimizations applied) 62

6.1 Arithmetic intensity (flops/byte) of various TGNN models. 79
6.2 Characteristics of the used datasets. Time encoder dimension is fixed to 100 for all datasets. 89
6.3 Configuration of the evaluated architectures. 90
6.4 Average throughput gain and latency reduction of TGN_PNM_hybrid approach across the datasets. 95
6.5 Area estimation of TGN-PNM. 95

List of Figures

2.1 (a) 1D representation of tile kernel access pattern. W is the full array width, K is the stride and L is the sequential access length. (b) 2D representation showing the tile pattern. (c) Varying spatio-temporal locality with L and K. . . . 14
2.2 Measured bandwidth of the evaluation platforms for different kernels. Working set size is mentioned with platform name. *Knights Landing 1G* effectively measures the bandwidth of the MCDRAM. . . . 16
2.3 Measured latency of the evaluation platforms with different working set size. . . . 17
2.4 Roofline plot with machine balance for Nvidia GeForce 1080 Ti GPU. This GPU has 128 SP cores vs only 4 DP cores per SM. Performance figures reflects this ratio. . . . 19
2.5 Roofline plot with machine balance for Intel Core i7 6700k CPU. The implemented kernel leverages the AVX2 vector extension. . . . 19

3.1 The generic workflow of a coverage-guided fuzzer. . . . 24
3.2 Hash collision rate drops as bitmap size is increased (derived from Equation 3.1). . . . 27
3.3 Runtime composition with varying bitmap sizes. Map operations dominate the runtime for bigger maps. The reported time is for one million test case generation. . . . 28
3.4 Steps of bitmap update operation for AFL's and BigMap's data structure. The hit counts in the coverage_bitmap are scattered in (a), while consolidated in (b). . . . 29
3.5 An illustrative example of bitmap operations on AFL's and BigMap's data structures. Value on top is the index of the bitmaps. Locations accessed at each step are highlighted in bold. (a) Execution trace and the assigned edge IDs (random). (b) AFL's data structure. Reset, classify, compare, etc., operations need to access the full bitmap. (c) BigMap's data structure. The full map is accessed only during initialization. Afterward, reset, classify, compare, etc., accesses only the used region of the coverage bitmap. Index bitmap is only accessed during the hit count update. . . . 30
3.6 Test case generation throughput of AFL and BigMap with different map sizes. AFL's throughput drops significantly as the map size is increased. Map size variation has considerably less impact on BigMap. . . . 36
3.7 Edge coverage with varying map sizes. AFL's edge coverage suffers due to throughput loss with bigger maps. Not all benchmarks are shown to improve clarity. . . . 37
3.8 Unique crashes found with varying map sizes. Going from 64kB to 256kB map shows improvement as a result of reduced collisions. AFL suffers for bigger map sizes due to throughput loss. . . . 37
3.9 (a) Throughput (normalized to the single instance) vs. the number of fuzzing instances. Dotted lines represent the individual benchmarks from (b), and the solid red line is their average. The solid black line shows 1:1 scaling as a reference. (b) Speedup attained by BigMap over AFL. The coverage map is fixed to 2MB for both (a) and (b). . . . 40
3.10 Unique crashes found with a varying number of fuzzing instances. The coverage map is fixed to 2MB. . . . 41

4.1 Example of different graph representation formats. Here, each edge e is an $\{dst, prop\}$ tuple. 47
4.2 Proposed hybrid representation format of GraphTango. . . . 49

4.3 Proposed hashing scheme. (a) Hash function to determine the index to the hash table. (b) An example probing sequence for $M = 5$ and $N = 4$. 52
4.4 Allocation and deallocation on the memory pool. Deleted pointers are shown by dashed lines and the modified pointers by red lines. 54
4.5 Load-balancing scheme of GraphTango. In the first stage, all threads go over a subset of the batch to fill bucket-chains. Afterwards, each worker thread locks and process the next available bucket-chain until all buckets are processed. 56
4.6 Comparison of the analytics throughputs. *Higher is better.* 59
4.7 Comparison of update throughputs. *Higher is better.* 60
4.8 Impact of TH_1 threshold on update throughput and memory usage. 61
4.9 Batch processing time breakdown of DZiG and RisGraph integration. *Lower is better.* 63

5.1 Parallel Radix sorting: (a) The intermediate array, which has one element per bucket and per subarray-level processing unit (SPU), (b) in step 1, each SPU generates a local histogram array, (c) in step 2, an aggregator processing unit (APU), outside memory layer (e.g., in the logic layer of 3D stack memories) performs a prefix-sum on all local histogram arrays, and (d) in step 3, each SPU determines the position (pos) of each key by deriving the bucket number and adding the prefix value of the bucket to the current index of the bucket (line 8-9). 68
5.2 Our proposed architecture: (a) The circles are subarrays, the rectangles are banks, and the pentagons are switches. Banks are connected using a dragonfly topology. (b) A bank with an SPU and three Walkers per subarray pair, (c) architecture of each SPU. (d) An example of local binary radix sorting. SPU loads one row of array $A[:]$ in Walker1. In each cycle, SPU reads one entry from Walker1 and places it in either Walker2 or Walker3, based on the binary digit being processed. Once Walker1 is fully read, SPU loads a new row from array $A[:]$ to Walker1. Once either of Walker2 or Walker3 is full, SPU writes the row in array $B[:]$. However, the SPU writes Walker1 in rows starting from the start of the array $B[:]$ but writes Walker3 in rows starting from the end of the array $B[:]$. 71
5.3 Throughput comparison of Pulley vs. Bonsai [1] and IMC [2] 74

6.1 Organization of a Hybrid Memory Cube [3, 4]. 80
6.2 TGN-PNM Architecture. 81
6.3 Partial-sum accumulation unit (PSAU). Figure drawn assuming a total of eight VPUs. 83
6.4 Mapping of dense matrix multiplication on TGN-PNM. 84
6.5 Total operand load time for the Wikipedia dataset with different memory address mapping and page policy configurations. *Lower is better.* 86
6.6 Throughput and batch processing latency for TGN-attn. 93
6.7 Throughput and batch processing latency for TGN-sum. 94

Chapter 1

Introduction

In recent years, the processing capability of computers has witnessed a remarkable increase, primarily driven by advancements in microprocessor design and parallel computing technologies. However, this rapid progress in processing power has not been matched by a similar increase in memory bandwidth [5]. The consequence of this growing performance gap is that modern processors, equipped with advanced vector units, can execute thousands of double-precision floating-point computations in the time it takes to satisfy a single last-level cache miss, leading to idle processor cores while waiting for data to be fetched from memory [6]. This disparity between processing power and memory bandwidth is exacerbated by the explosion of data-intensive applications across various domains, such as machine learning, bioinformatics, video analytics, graph processing, and database systems. These applications often operate on large datasets characterized by irregular access patterns and exhibit a low compute-to-memory ratio, making it challenging to hide the memory access latency. Consequently, traditional von Neumann architectures, when running such data-intensive workloads, face a "memory wall" scenario, where the processor's computational capacity is underutilized due to memory access bottlenecks [7, 8].

Interesting concepts are being developed to alleviate the memory wall issue. One is to move the computation near or inside memory (processing-in-memory or PIM). These approaches leverage the observation that the internal memory bandwidth can be orders of magnitude higher than the external IO interface bandwidth. For example, an 8 GB Hybrid Memory Cube (HMC) has an aggregated (i.e., read + write) external IO bandwidth of 320 GB/s via high-speed serial links [3]. On the other hand, the same cube has an internal bandwidth of 512GB/s at the logic layer, 2 TB/s at the bank level, and even higher at the subarray level (5+ TB/s) [9]. Another advantage of adopting near-data processing is to achieve *memory-capacity-proportional* bandwidth [10]. For instance, sixteen of these HMC cubes can be connected together to achieve 8 TB/s

bandwidth at the logic layer, while the external IO bandwidth remains capped at 320 GB/s. Beyond the bandwidth advantage, PIM provides energy benefits as well (e.g., data movement cost of 10.48 pJ/bit to the logic layer compared to 65 pJ/bit to the external interface [11]). For these reasons, there has been a recent influx of PIM-based approaches from both academia and industry, placing compute units at the buffer chips of DDR4 [12], banks of GDDR6 [13], and at various levels (i.e., logic layer, banks, subarrays) of 3D-stacked memories [9, 14, 15, 16, 17, 18, 19, 20, 21, 22]. Besides these emerging PIM-based accelerator architectures, there are software-based solutions that aim to circumvent the memory wall by minimizing data movement with approaches such as compression [23, 24, 25] or by improving cache utilization [26]. These approaches face trade-offs in terms of performance, power, area, cost and complexity of implementation, time-to-market, application and library support, etc. All these factors must be carefully considered when accelerating an application of interest.

In this dissertation, **I hypothesize that, to avoid being bottlenecked by the memory wall, algorithms and data structures for data-intensive applications must be designed to leverage hardware features in the memory hierarchy, and in some cases, a software-hardware co-design approach is beneficial.** Accordingly, I present five pieces of work to support my hypothesis: (i) Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation [27], (ii) BigMap: Future-proofing Fuzzers with Efficient Large Maps [28], (iii) GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Updates and Analysis [29]. (iv) Pulley: An Algorithm/Hardware Co-optimization for In-memory Sorting [18, 30], and (v) TGN-PNM: A Near-Memory Architecture for Temporal GNN Inference on 3D-Stacked Memory (planned submission).

My first work, Hopscotch, introduces a comprehensive memory benchmark suite accommodating a wide range of access patterns. The relevance of this work with my hypothesis stems from the observation that the impact of the hardware features in the memory hierarchy on a workload is not always self-evident. For instance, it is commonly assumed that sequential memory access can attain a significantly higher bandwidth compared to random accesses, primarily due to the memory controller prioritizing row buffer hits (FR-FCFS scheduling). However, with the evolution of multi-core processors, the likelihood of row buffer hits has substantially diminished. This trend is reflected in the design of HMC, which prioritizes parallelism over row buffer locality and is achieved by mapping consecutive DRAM accesses to different banks (i.e., low address interleaving) rather than on the same row and by employing a closed page policy and narrow row buffers [3, 31]. In fact, on HMC-based systems, sequential and random accesses demonstrate very similar bandwidth [4]. Consequently, optimizing an application by converting random accesses to sequential may offer limited performance benefits on an HMC-based system. To avoid such pitfalls, it is essential to use benchmarks with

different access patterns, read-write mix, working set size, and other relevant metrics to properly characterize a system's performance. The Hopscotch benchmark suite is designed with that objective in mind.

The rest of my works target accelerating four important data-intensive applications: BigMap targets coverage-guided fuzzing, GraphTango targets streaming graph processing, Pulley targets large-scale sorting, and TGN-PNM targets temporal graph neural networks. In the case of BigMap, a memory bottleneck is observed when trying to support large coverage bitmaps to mitigate hash collisions. The coverage bitmap is very sparsely filled, but the full map needs to be traversed for various operations on the map since the exact locations of the used entries are unknown. While we initially hypothesized that a PIM solution would be beneficial, we instead introduced a two-level hashing scheme that consolidates only the used portion in an auxiliary map, reducing the working set size and improving cache utilization. Our approach effectively removed the memory bottleneck and allowed an arbitrarily large map size, verified by increasing the map size from 64kB to 8MB without any noticeable impact on performance.

In our next work, GraphTango, we propose a hybrid storage format for streaming graphs that aims to minimize cache line fetches. The key insight in this work is that the typical graph workloads show irregular access patterns with a very large working set size that is unlikely to fit in the last-level cache (LLC). Even with our smallest dataset of 5 million edges, the LLC miss rate during the update phase is over 49%, indicating that the working set size is larger than the LLC. Therefore, it is unlikely that doing lookup operations or neighbor traversal of two *different* nodes will overlap in the same cache lines. With this observation, the data structure optimization primarily boils down to whether we can overlap the lookup and neighbor traversal of the *same* node in the same cache line. To that extent, we proposed a hybrid data structure where the low-degree vertices store the edges directly with the neighborhood metadata, confining accesses to a single cache line, medium-degree vertices use adjacency lists, and high-degree vertices use hash tables as well as adjacency lists. We proposed a novel hashing scheme that places consecutive probes in the same cache line, minimizing the number of cache line fetches. While this hybrid scheme provided excellent throughput gain, we improved our approach with a novel bucket-chaining-based workload balancing technique. These two works (BigMap and GraphTango) support our hypothesis that the memory bottleneck can be largely alleviated using hardware features in the memory hierarchy (namely caching and TLB optimizations for these two approaches) of traditional general-purpose architectures.

In our next work, Pulley, we targeted large-scale sorting. While the previous works are software-based techniques implemented on existing platforms, Pulley approach is a hardware-algorithm co-optimization approach that is based on our prior subarray-level PIM architecture, Gearbox [14]. In Pulley, we observed that the existing in-memory sorting algorithms are based on merge sort. The limitation of merge sort is that while the buckets can be sorted independently, the final merge step has to go through all the elements

and create a single-point merging bottleneck. In our approach, we leveraged radix sort that is scalable and does not suffer from this bottleneck. However, the radix sort requires random accesses that are not optimal for PIM. To alleviate this issue, we introduce a local-sorting step that does a one-bit radix sorting within latched row buffers. After this local-sorting step, the rest of the accesses become sequential access. With our approach, we observed 13x to 20x speedup over FPGA and in-logic-layer-based sorting accelerator.

In our last work, TGN-PNM, we target accelerating temporal graph neural networks (TGNN) by placing processing units on the vaults of an HMC-like 3D-stacked memory. One key challenge with TGNN workload is the evolving nature of the graph, making it extremely difficult to maintain workload balance at runtime. We alleviated this issue by partitioning along the feature dimension, where each vault contains a subset of features for *all* nodes. With this approach, each vault-level processing unit has the exact same amount of workload, solving the imbalance issue. Furthermore, elementwise operations do not need inter-vault communication. Operations involving reduction (e.g., dot products) require inter-vault communication but have a regular pattern and can easily be handled by an adder tree. A matrix/vector with small feature dimensions is stored in a traditional fashion to prevent DRAM column access under-utilization and is processed using a broadcast-based mechanism. These last two works (Pulley and TGN-PNM) fit our hypothesis as they try to alleviate the memory bottleneck by hardware-software co-optimization. The rest of this chapter goes into more details of these approaches.

1.1 Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation

Most memory benchmarks use either fully sequential or random access patterns [32, 33, 34, 35, 36]. However, real-world applications show a wide variety of access patterns that fall in-between these two extremes. A benchmark with a tunable access pattern is more appropriate in such cases. While some memory benchmarks provide limited tunability (e.g., Spatter [37], ApexMap [38]), they do not support individual tuning of read-only, write-only, or mixed-accesses, hindering their ability to isolate memory bottlenecks. We present Hopscotch, a micro-benchmark suite in which the access pattern of read-only, write-only, and mixed accesses can be tuned to cover a wide spatio-temporal locality spectrum. Most of these kernels are available on CPU, GPU, and FPGA platforms (FPGA porting courtesy to Mosanu et al. [39]). We provide a few additional tools with Hopscotch, namely a memory access pattern visualizer and an empirical Roofline tool for CPU and GPU.

1.2 BigMap: Future-proofing Fuzzers with Efficient Large Maps

In this work, we accelerate a popular coverage-guided fuzzer called AFL by consolidating memory accesses to a smaller cache region, improving utilization, and reducing pollution. Coverage-guided fuzzing is a powerful technique for finding security vulnerabilities and latent bugs in software. Such fuzzers usually store the coverage information in a small bitmap. The bitmap is accessed frequently; thus, the size is kept small to fit the bitmap in faster cache levels. Hash collision within this bitmap due to its small size is a well-known issue and can reduce fuzzers' ability to discover potential bugs. Prior works noted that collision mitigation with naively enlarging the hash space leads to an unacceptable runtime overhead. This work introduces BigMap, a two-level hashing scheme that enables the use of an arbitrarily large coverage bitmap with low overhead. The key observation is that the overhead stems from frequent operations performed on the full bitmap, although only a fraction of the map is actively used. BigMap condenses these scattered active regions on a second bitmap and limits the operations only to that condensed area. Further optimizations are performed by leveraging huge pages to reduce the number of page walks caused by TLB misses. We implemented our approach on top of the popular fuzzer AFL and conducted experiments on 19 benchmarks from FuzzBench and OSS-Fuzz. The results indicate that BigMap does not suffer from increased runtime overhead, even with large map sizes. Compared to AFL, BigMap achieved an average of 4.5x higher test case generation throughput for a 2MB map and 33.1x for an 8MB map. The throughput gain for the 2MB map increased further to 9.2x with parallel fuzzing sessions, indicating the superior scalability of BigMap. More importantly, BigMap's compatibility with most coverage metrics, along with its efficiency on bigger maps, enabled exploring aggressive compositions of expensive coverage metrics and fuzzing algorithms, uncovering 33% more unique crashes. BigMap makes using large bitmaps practical and enables researchers to explore a wider design space of coverage metrics.

1.3 GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Processing

In this work, we propose an optimized graph storage format for streaming graph processing that attempts to minimize cache line accesses. Streaming graph processing involves performing batched updates and analytics on time-evolving graphs. The underlying representation format of the graph largely determines the throughputs of these updates and analytics phases. Existing representation formats usually employ variations of hash tables or adjacency lists. However, a recent study showed that the adjacency-list-based approaches perform poorly on heavy-tailed graphs, and the hash table-based approaches suffer on short-tailed

graphs [40]. We propose GraphTango, a hybrid representation format that provides excellent update and analytics throughput irrespective of the graph's degree distribution. GraphTango dynamically switches among three different formats based on a vertex's degree: i) Low-degree vertices store the edges directly with the neighborhood metadata, confining accesses to a single cache line, ii) Medium-degree vertices use adjacency lists, and iii) High-degree vertices use hash tables as well as adjacency lists. In this case, the adjacency list provides fast traversal during the analytics phase, while the hash table provides constant-time lookups during the update phase. We further optimized the performance by designing an open-addressing-based hash table that focuses on maximizing cache line utilization. In addition, we developed a thread-local lock-free memory pool that allows fast growing/shrinking of the adjacency lists and hash tables in a multi-threaded environment. We further improved the performance by designing a novel bucket-chaining-based workload balancing technique. We evaluated GraphTango by integrating it with the SAGA-Bench framework and compared it with four other representation formats: Stinger, Degree-aware Robin Hood Hashing, and two adjacency list-based formats with different workload balancing schemes. GraphTango vastly outperforms these approaches, on average providing 4.5x higher insertion throughput, 3.2x higher deletion throughput, and 1.1x higher analytics throughput over the *next best* alternative. Furthermore, we integrate GraphTango with the state-of-the-art graph processing frameworks, DZiG and RisGraph. The results demonstrate that GraphTango combined with DZiG and RisGraph reduces the average batch processing time by 2.3x and 1.5x, respectively, compared to the vanilla versions of these frameworks.

1.4 Pulley: An Algorithm/Hardware Co-optimization for In-memory Sorting

In this work, we propose a subarray-level PIM architecture for accelerating large-scale sorting. Sorting is an important kernel that requires many passes on data, where each pass imposes significant data movement overhead. PIM can reduce this data movement overhead while providing high parallelism. We selected radix sorting as the sorting algorithm because it is scalable and can exploit PIM's parallelism. However, this algorithm is inefficient for current PIM-based accelerators for three reasons: (i) requiring a large intermediate array per processing unit, wasting capacity, (ii) requiring a prefix-sum operation across all the large intermediate arrays, imposing performance overhead, and (iii) requiring significant random accesses, which are costly in PIM. In this work, we propose an algorithm and hardware co-optimization for sorting that enables every group of processing elements to cooperatively share and generate an intermediate array, reducing the capacity overhead of intermediate arrays and the performance overhead of the prefix-sum operation. To prevent the shared array from becoming a bottleneck due to random accesses, we eliminate random accesses by adding a local sorting step to the radix sorting and providing efficient hardware support for this step.

On average, Pulley delivers 20× speedup compared to Bonsai, an FPGA-based sorting accelerator, and 13× speedup compared to IMC, an in-logic-layer-based sorting accelerator.

1.5 TGN-PNM: A Near-Memory Architecture for Temporal GNN Inference on 3D-Stacked Memory

In this work, we propose TGN-PNM, a PIM-based accelerator designed specifically for Temporal Graph Neural Networks (TGNNs). TGNNs are gaining increasing attention due to their ability to capture complex relationships and temporal dynamics in various domains. However, designing accelerators for TGNN workloads poses several challenges, including the lack of a standard model architecture, the absence of distinct execution phases, and the difficulty in maintaining workload balance in evolving graphs. Existing accelerators for static GNNs are not easily extendable to TGNNs. In this work, we leverages the concept of vault-level parallelism by placing a Vault Processing Unit (VPU) at each vault in a 3D-stacked memory. The VPU consists of a SIMD unit for time encoding, elementwise, and other memory-intensive operations, and a systolic array for compute-intensive dense-matrix-matrix multiplications. By placing compute units at the logic layer, our design achieves near-linear performance improvement with increasing memory stacks and exposes higher internal memory bandwidth. We address the challenges of TGNN workloads by introducing a hybrid partitioning scheme that uses traditional partitioning for small feature-dimensions, and feature-based partitioning for large feature-dimensions. We evaluated our approach against a few architectures: a high-end CPU and GPU, a subarray-level general purpose PIM architecture Gearbox, a bank-level AI accelerator Newton, and the FGPA-based TGNN accelerator tFGPA. Our evaluation demonstrated average throughput gains of 26.8x over CPU, 16.7x over GPU, 5.2x over Gearbox, 4.4x over Newton, and 10.4x over tFPGA.

1.6 Dissertation Structure

The remainder of this dissertation is organized as follows:

- Chapter 2 presents the design principles and features of Hopscotch, the micro-benchmark suite for memory performance evaluation.
- Chapter 3 delves into the BigMap approach, focusing on optimizing the memory access patterns of coverage-guided fuzzers and ensuring efficient cache utilization.
- Chapter 4 introduces GraphTango, a hybrid graph representation format tailored for streaming graph processing, emphasizing cache performance improvements.
- Chapter 5 describes Pulley, an algorithm-hardware co-optimization approach for large-scale in-memory sorting, showcasing significant speedup over FPGA and in-logic-layer sorting approaches.

- Chapter 6 presents TGN_PIM, a processing-in-memory architecture designed for accelerating temporal graph neural network inference leveragin feature-based partitioning.
- Chapter 7 summarizes the dissertation, discusses its contributions, and outlines potential future directions.

Through these chapters, we aim to provide a comprehensive exploration of memory access challenges and innovative solutions, validating our hypothesis that memory-centric optimizations, whether through hardware or software, can enhance the performance of data-intensive applications and mitigate the memory wall challenge.

Chapter 2

Hopscotch: A Micro-benchmark Suite for Memory Performance Evaluation

2.1 Introduction

Benchmarking for memory characterization is a well-studied problem. Most benchmarks use either fully sequential or random access patterns for this purpose [32, 33, 34, 35, 36]. However, real-world applications show a wide variety of access patterns that fall in-between these two extremes. A benchmark with a tunable access pattern is more appropriate in such cases. Examples of such benchmarks are Spatter and ApexMAP. Spatter shows sparse access using a scatter-gather kernel [37], while ApexMAP provides a tunable access pattern for reads [38]. Unfortunately, Spatter only supports mixed access, and ApexMAP only reads. Consider a system with excellent read bandwidth, but low write bandwidth. An evaluation with a mixed access benchmark will evaluate the system with as low performing because the write bandwidth will act as a bottleneck. However, an application with mostly read accesses (e.g., machine learning inference) will run on this system perfectly fine, which will not be properly identified by that particular benchmark. Besides, mixed access benchmarks cannot determine whether the read or the write bandwidth is the bottleneck. On the other hand, benchmarks with only read or only write accesses will show the same performance figures on systems that incorporate independent read and write channels and systems that do not. Therefore, a memory benchmark should be tunable, and should also support read-only, write-only, and mixed accesses. However, a closer inspection, as summarized in Table 2.1 reveals that no existing benchmarks show these two desirable properties simultaneously. In our attempt to fix this gap, we present Hopscotch, which is a comprehensive micro-benchmark suite for memory characterization. It introduces a tunable kernel

supporting read-only, write-only and mixed accesses. The pattern exhibited by this kernel is best described as a tiled pattern. This is common pattern in multimedia applications. Furthermore, this kernel can be tuned to change the degree of spatio-temporal locality. Hopscotch also includes various non-tunable kernels with different access patterns. Finally, these kernels are combined in interesting ways to form the micro-benchmarks. Besides the usual peak bandwidth and latency measurement, Hopscotch provides insight on caching efficiency and expected bandwidths with different access patterns. Hopscotch kernels are currently supported on CPU, GPU, and FPGA platforms. The FPGA port is carried out by Mosanu et al. as a part of the Pimulator work [39] and runs on a RISC V soft-core. Furthermore, the simplistic nature of the kernels make then easily portable to other emerging architectures as well and can be an interesting tool in evaluating PIM-based architectures. The benchmark suite is publicly available on following Github repository: https://github.com/alifahmed/hopscotch. **We published the findings of this work in [27].**

The remainder of the chapter is organized as follows. Related memory benchmarks are surveyed in Section 2.2. The design and implementation methodology of the kernels are discussed in Section 2.3. Section 2.4 evaluates three platforms using the micro-benchmarks. Section 3.7 concludes the work and discusses potential future directions.

2.2 Related Works

There exists a number of benchmarks for evaluating memory system characteristics. Few early benchmarks explored sequential access patterns for measuring peak sustainable bandwidth. Among them, STREAM covered mixed accesses [32] while STREAM2 introduced read-only and write-only kernels [41]. CacheBench added a provision for read-modify-write operations [35]. Deakin et al. extended peak sustainable bandwidth measurements to GPU [42]. Unfortunately, peak sustainable bandwidth is not a good predictor of latency bound applications. lmbench introduced a pointer-chasing access pattern to measure back-to-back-read latency [33]. A few other benchmarks contained similar random access patterns for latency and random update rate measurements [34, 43, 44]. Unfortunately, these benchmarks support only sequential and random patterns, unlike real-world applications.

The MAPS (Memory Access Pattern Signature) benchmarks, such as ApexMAP [38] and MultiMAP [45], support a wider range of access patterns using tunable kernels. ApexMAP parameterizes the regularity, working-set-size, and locality of accesses. MultiMAP uses stride and working-set-size. While tunable, these two benchmarks support only read access. A recent paper proposes Spatter [37], which covers scatter-gather access patterns with tunable sparsity. Spatter can be configured to have a combination of sequential, strided, or random reads and writes. Unfortunately, Spatter only has mixed accesses. As described earlier, mixed

Table 2.1: A comparison of supported access patterns and platforms of existing memory benchmarks

Benchmark	Read-only pattern	Write-only pattern	Mixed pattern (read, write)
STREAM [32]	–	–	(sequential, sequential)
STREAM2 [41]	sequential	sequential	(sequential, sequential)
CacheBench [35]	sequential	sequential	(sequential, sequential)
lmbench [33]	sequential, random	sequential	(sequential, sequential)
HPC-Challenge [34]	–	–	(sequential, sequential), (random, random)
MultiMAP [45]	strided	–	–
ApexMAP [38]	tunable	–	–
pmbw [43]	sequential, random	sequential, strided	–
TinyMemBench [44]	random	sequential	(sequential, sequential)
MLC [46]	sequential, random	–	(sequential, sequential), (random, sequential)
X-Mem [47] (v2.4.2)	sequential, strided, random	sequential, strided	(random, random)
GPU-STREAM [36, 42]	–	–	(sequential, sequential)
Spatter [37]	–	–	(tunable, tunable)
Hopscotch	tunable	tunable	(tunable, tunable)

access benchmarks cannot determine whether the read or the write bandwidth is being the bottleneck. Table 2.1 summarizes the access patterns supported by different benchmarks. To our knowledge, Hopscotch is the first benchmark suite that provides tunable access patterns for read-only, write-only, and mixed accesses.

2.3 Kernel Implementation

2.3.1 Generic Design Decisions

An overview of the kernels included in Hopscotch is presented in Table 2.2. Each kernel operates on fixed size arrays. The size can be changed at compile time to control where in the memory hierarchy the data reside. Each kernel iterates multiple times over the working set. The first iteration is ignored in timing calculations (warm start). Reported time is the elapsed wall clock time. The build system uses GNU gcc with "-O3 -march=native -fopenmp" flags. GPU codes are compiled with Nvidia's CUDA Complier (NVCC). NVCC

Table 2.2: Characteristics of the included kernels

Kernel Name	Operation (i = outer loop index, j = inner loop index)	Read Pattern	Write Pattern
r_seq_ind	$x = a[i]$	sequential	–
r_seq_reduce	$x \mathrel{+}= a[i]$	sequential	–
r_rand_ind	$idx = rand\_gen(i)$ $x = a[idx]$	random	–
r_rand_pchase	$p = *p$	random	–
r_stride_<k>	$x = a[i]; i \mathrel{+}= k;$	strided	–
r_tile	$x = a[i + j]; i \mathrel{+}= k;$	tunable	–
w_seq_memset	$memset(a, x, W)$	–	sequential
w_seq_fill	$a[i] = x$	–	sequential
w_rand_ind	$idx = rand\_gen(i)$ $a[idx] = x$	–	random
w_stride_<k>	$a[i] = x; i \mathrel{+}= k;$	–	strided
w_tile	$a[i + j] = x; i \mathrel{+}= k;$	–	tunable
rw_seq_inc	$a[i]$++	sequential	sequential
rw_gather	$a[i] = b[idx[i]]$	sequential + random	sequential
rw_scatter	$a[idx[i]] = b[i]$	sequential	random
rw_scatter_gather	$a[idx_1[i]] = b[idx_2[i]]$	sequential + random	random
rw_tile	$a[i + j] = b[i + j];$ $i \mathrel{+}= k;$	tunable	tunable

uses maximum optimization level for device code by default. Optimization level of host code has no impact on results.

We measured the transferred bytes as seen by the user. For example, copying 1MB data would mean 1MB read and 1MB write, total 2MB. This approach is the same as STREAM. Interestingly, there are other ways as mentioned in [48]. lmbench uses the amount moved from one place to other (1MB for the prior example). Hardware, on the other hand, may move a different amount. If the write-allocate policy is active, then the data to be written is read into the cache first, making the total transferred bytes to 3MB. As mentioned, Hopscotch will report 2MB in such cases. When combining results from multiple kernel runs, geometric mean is used in case of rates (e.g., MB/s). The correctness of some kernels is sensitive to optimizations. In such cases, the *volatile* keyword is used to prevent optimization instead of unportable inline assembly.

Table 2.3: System configuration of the evaluation platforms

CPU	Core	Freq. (GHz)	Cache				Memory
			L1D	L2	L3	others	
Core i7-6700k (Skylake)	4	4.0	4 × 32 KiB, private, 8-way	4 × 256 KiB, private, 4-way	4 × 2 MiB, shared, 16-way	–	DDR4-3200 (14-14-14) 4 × 8 GB, 2 Channels
Xeon Phi 7210 (Knights Landing)	64	1.3	64 × 32 KiB, private, 8-way	32 × 1 MiB, shared, 16-way	–	16 GB MCDRAM, shared, direct map	DDR4-2666 (19-19-19) 6 × 32 GB, 6 Channels
Xeon Gold 5218 (Cascade Lake)	16	2.1	16 × 32 KiB, private, 8-way	16 × 1 MiB, private, 16-way	16 × 1.375 MiB, shared, 11-way	–	DDR4-2666 (19-19-19) 2 × 32 GB, 2 Channels

2.3.2 Read-Only Kernels

Read-only kernels in Hopscotch are designed carefully to contain only read accesses, at least from the user's perspective. These kernels can still incite writes in the hardware when evicting dirty cache lines. These stray writes can happen during the first run only. The measurements are not impacted by this because the timing for the first run is discarded.

Hopscotch contains two sequential read-only kernels. The first kernel *r_seq_reduce* uses reduction operation (x += $a[i]$) inside a loop. Although it contains data dependency, it can be efficiently distributed among multiple threads using OpenMP *reduction* directive. The second kernel *r_seq_ind* is a data-independent version that reads from the array into a register ($x = a[i]$). We prevented the compiler from optimizing away this statement using the *volatile* keyword. In our experiments, we found out that the data-independent version runs slightly faster.

There are two kernels for the random read-only pattern. The first kernel *r_rand_pchase* exhibits a pointer-chasing scheme *(p = (void**)*p)*. The array initialization routine generates a Hamiltonian cycle of pointers. Thus, it guarantees uniform distribution, and each array element is accessed only once. The pointer-chasing type operation is better suited for latency calculation since each read must complete before the next read can be issued. This ensures only one outstanding read (queue depth of one at the memory controller) at any given time. The second kernel *r_rand_ind* reads from randomly generated indices. These indices are generated in each iteration instead of being read from an array. Reading an index from an array (e.g., $x = a[b[i]]$) would result in an overlapping sequential and random type access pattern instead of purely random pattern. Our approach is similar to the HPC-challenge benchmark's *RandomAccess* kernel [34]. The random index generation routine must be kept minimal since we are considering only memory bound applications. HPC-challenge's index generation routine has a loop-carried dependency. We avoided the dependency by generating indices from the loop counter. Although the generated indices are not uniformly

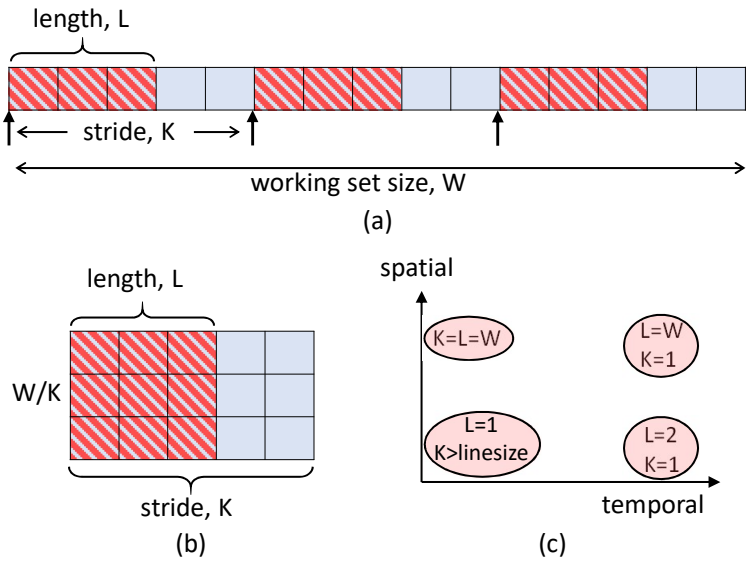


Figure 2.1: (a) 1D representation of tile kernel access pattern. W is the full array width, K is the stride and L is the sequential access length. (b) 2D representation showing the tile pattern. (c) Varying spatio-temporal locality with L and K.

distributed, they are random enough to defeat any caching and prefetching efforts. *r_rand_ind* kernel maxes out the memory controller queue unlike *r_rand_pchase* kernel.

Hopscotch also includes a read-only kernel with a tunable access pattern. Figure 2.1 depicts the structure of the kernel. It shows a tile-like pattern, which is a mixture of sequential and strided access. The kernel is parameterized using the working set size (W), stride (K), and sequential access length (L). The pattern can be made fully sequential by setting $K = L = W$, and of stride K with setting $L = 1$. Locality behavior of this kernel can be controlled with L and K as shown in Figure 2.1(c). When $L \leqslant K$, there will be no overlapping access, making the temporal locality zero. Making $L > K$ would result in overlap. The minimum overlap will happen for $K = W/2$ and $L = K + 1$. As we increase L, or decrease K, temporal locality will increase. $K = 1$ and $L = W$ will provide the maximum overlap. The locality behavior of the kernel is shown in Figure 2.1(c).

2.3.3 Write Only Kernels

Ensuring that kernels issue only writes while maintaining portability is difficult. In write-allocate, a cache line is read first in case of a write miss. Few instruction set architectures support non-temporal store instructions to indicate that the value written will not be used in the near future. This bypasses write-allocation. However, using these non-temporal store instructions with the help of inline assembly reduces portability. Most implementations of *memset* function issue such non-temporal stores if the architecture supports it [49]. Hopscotch includes two sequential write kernels. The *w_seq_memset* kernel uses *memset*, and the *w_seq_fill* kernel writes to an array using a loop. If the system uses write-allocate policy, the *w_seq_memset* kernel

shows the actual write-only bandwidth, while the *w_seq_fill* kernel shows the bandwidth for simultaneous read and write. On the other hand, if the system does not use write-allocate, then both kernels show similar results. Thus, a big gap between the results of these two kernels indicates write-allocation. The opposite is not true. Both kernels can report similar bandwidth even in the presence of write-allocate policy. This scenario can happen if the compiler is smart enough to recognize the non-temporal behavior of the write loop, or the memory supports independent read and write channels. In the latter case, the extra reads will not impact the write bandwidth significantly.

Besides the sequential kernels, Hopscotch also includes random, strided and tunable write-only kernels. However, these kernels may issue extra reads in presence of write-allocation. Unlike *memset*, there is no portable way of providing a non-temporal hint for non-sequential writes. These three kernels are implemented the same way as their read-only counterparts by replacing the read operations by writes.

2.3.4 Mixed Kernels

Hopscotch includes five kernels with mixed access. *rw_seq_inc* sequentially reads an array and increments each element. Since each element is explicitly read before updating the value, this kernel is impervious to the ambiguity caused by the write-allocate policy. This kernel can be used to measure the peak sustainable bandwidth for mixed access workloads. The next three kernels (*rw_gather*, *rw_scatter* and *rw_scatter_gather*) exhibit standard scatter and gather type accesses. *rw_gather* kernel consists of a source data array (b), a destination data array (a), and an index array (idx). The data is copied from the source array in the order given by index array and then written to the destination array sequentially. In our implementation, the index array is initialized using $idx[i] = i$, and then shuffled. Thus, the *rw_gather* kernel has a combination of sequential read of index array, random read of source array, and sequential write to the destination array. Similarly, the *rw_scatter* kernel reads sequentially from the source and the index array, and then writes to the destination array based on the index. The *rw_scatter_gather* is a combination of the previous two patterns. It has two randomized index arrays - one indicating the positions for read, and the other indicating the positions for write. Overall, the *rw_scatter_gather* kernel is a combination of sequential reads of the index arrays, random read of the source array, and random write to the destination array. Finally, the *rw_tile* kernel has a tunable pattern similar to *r_tile*, with the read replaced by a copy.

2.4 Evaluation with Hopscotch

We have evaluated three platforms with Hopscotch. Table 2.3 shows the details of the platforms. The *Skylake* platform represents a typical desktop. The *Knights Landing* platform uses a Xeon phi processor with Intel

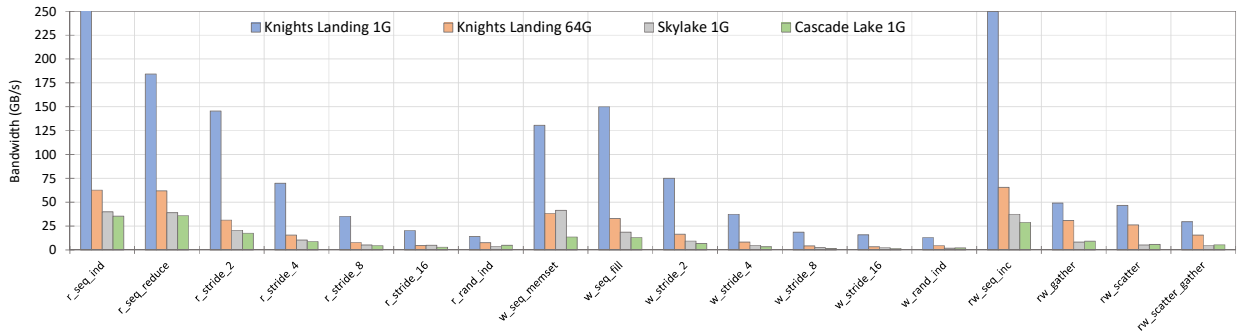


Figure 2.2: Measured bandwidth of the evaluation platforms for different kernels. Working set size is mentioned with platform name. *Knights Landing 1G* effectively measures the bandwidth of the MCDRAM.

Many Integrated Core (MIC) architecture. It has 64 low-frequency cores arranged in tiles. Each tile has two cores with a shared 1 MiB L2 cache. Additionally, it has an on-chip eight channel high-bandwidth distributed 16 GB MCDRAM. The MCDRAM can be used as a cache for the DDR memory, or as a standalone memory, or as a combination of both. In our evaluation, we have used the MCDRAM as a cache. The *Cascade Lake* platform uses a high-end server grade Xeon Scalable processor. This configuration reflects current generation servers.

2.4.1 Bandwidth Measurement

The bandwidth micro-benchmark estimates the expected bandwidth for different read and write access patterns. This benchmark runs all the non-temporal kernels. This is because reusing an array element counts it multiple times in the bandwidth measurement, while not contributing towards required bandwidth. Kernels with random access patterns are assumed to be non-temporal and included in this benchmark, although they show low amount of locality.

Figure 2.2 presents the results of running the bandwidth benchmark on our platforms. For the Skylake and Cascade Lake platforms, we have set the working set size large enough to exercise the main memory. For the Knights Landing platform, we have used 1 GB working set size to evaluate the MCDRAM and 64 GB working setize for evaluating the main memory. The *r_seq_ind* kernel gives the peak sustainable bandwidth for read. The *r_seq_reduce* kernel shows 31.6% degradation for the MCDRAM compared to the *r_seq_ind* kernel. For other platforms, the difference is only 1.2% on average. We expected *r_stride_16* to show higher bandwidth than *r_rand_ind*. Accesses from both kernels will miss, but *r_stride_16* is more prefetcher friendly. However, *r_rand_ind* has a small amount of locality making it marginally faster in two of the platforms.

For write-only accesses, the faster kernel between *w_seq_memset* and *w_seq_fill* measures the peak sustainable write bandwidth. We found *w_seq_memset* to be faster, except for the MCDRAM. This indicates

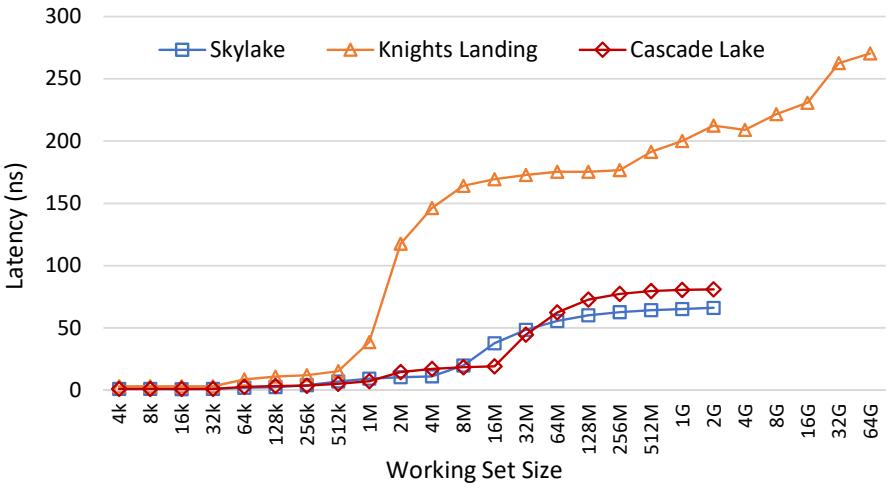


Figure 2.3: Measured latency of the evaluation platforms with different working set size.

the possibility of improvement in the *memset* function for Knights Landing architecture. For Skylake, the bandwidth of w_seq_fill is roughly half of the *memset* kernel, hinting towards write-allocation.

2.4.2 Latency Measurement

Hopscotch's latency micro-benchmark measures the worst case back-to-back-read latency of an unloaded system. Back-to-back-read latency is the time that each read takes, assuming that the instructions before and after are also cache-missing reads [33]. We have used the pointer chasing kernel (r_rand_pchase) for this benchmark. It does a series of loads ($p = *p$), where the address of the next read is the value of the ongoing read. Because of the dependency, only one read can be outstanding at a time. Since the accessed address sequence is entirely random for this kernel, prefetching schemes cannot help to improve the measured latency. Thus it measures the "worst case" latency.

Figure 2.3 shows the results of running the benchmark on our evaluation platforms. The Skylake platform shows slightly lower latency than the Cascade Lake platform. This is mainly due the the higher memory frequency and tighter timings of the memory module used in the Skylake platform. On the other hand, the Knights Landing exchanges latency for better bandwidth. The plateau from 8MB to 256MB shows the latency of the L2 cache to be approximately 173ns. The latency of MCDRAM is 210ns. Because of the distributed nature of the MCDRAM, Knights Landing has Non-Uniform Memory Access (NUMA), which means access latency will vary based on the relative location of memory and processor. As our benchmark is not NUMA aware, the measured latency can be thought of as the latency seen by the vast majority of applications.

Table 2.4: Impact of spatio-temporal locality on bandwidth. Single Threaded.

Spatial locality	Temporal locality	Bandwidth (MB/s)			
		Knights Landing 1G	Knights Landing 64G	Cascade Lake 1G	Skylake 1G
low	low	251	232	971	605
low	high	885	857	8703	5700
high	low	1727	1581	9216	4621
high	high	1755	1772	10251	6499

2.4.3 Impact of Locality

The impact of locality is evaluated using the *r_tile* kernel. This kernel is executed on four locality configurations as shown in Table 2.4. Low spatial and low temporal locality configuration is achieved by setting $K = 32$ and $L = 1$. Any value of K that results in stride wider than cache line is acceptable for this configuration. For low spatial and high temporal locality, we selected $K = 1$ and $L = 2$. Increasing the value of L will increase the number of times each element is reused. High spatial locality and low temporal locality is achieved with $K = W$ and $L = W$. This results in sequential access. Finally, setting $K = 1$ and $L = 32$ provided us with high spatial and high temporal locality. Any large value of L is acceptable. However, it should be smaller than the L1 cache size. A ratio of the high-high and the low-low configurations gives us a notion of the effectiveness of the caching system. In our evaluation, both Knights Landing platforms show similar performance, with a ratio of 7.3. The Cascade Lake and the Skylake platforms show a ratio of 10.7. The Cascade Lake and Skylake platforms also have better speedup with higher temporal locality, indicating premature cache line replacement in Knights Landing. The direct mapping of MCDRAM in Knights Landing may be one of the reasons for such behavior.

2.4.4 Roofline Model and Machine Balance

The original Roofline model shows peak theoretical performance with varying operational intensity [50]. The Roofline model is helpful in determining the maximum achievable performance of an application. It also indicates whether the application is memory-bound or compute-bound using the concept of machine balance and code balance:

$$Machine\ Balance,\ B_m = \frac{Maximum\ sustainable\ mamory\ BW\ (words/s)}{Peak\ performance\ (flops/s)}$$

$$Code\ Balance,\ B_c = \frac{Requested\ data\ (words)}{Computation\ (flops)}$$

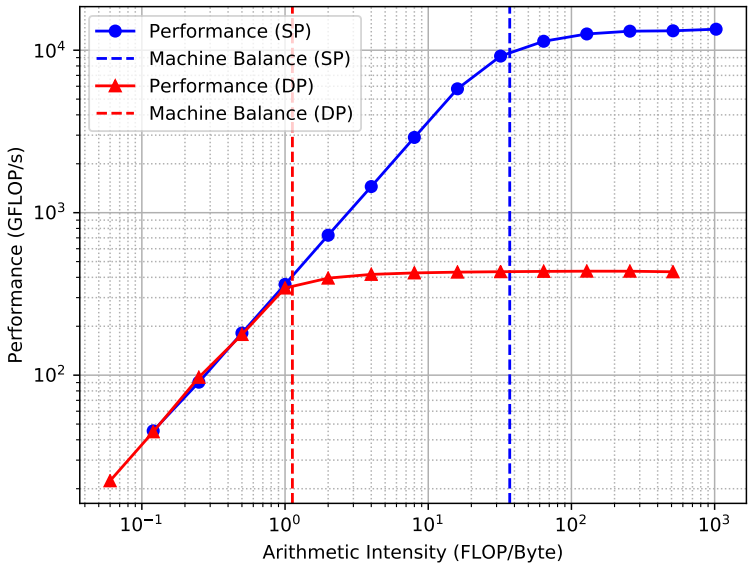


Figure 2.4: Roofline plot with machine balance for Nvidia GeForce 1080 Ti GPU. This GPU has 128 SP cores vs only 4 DP cores per SM. Performance figures reflects this ratio.

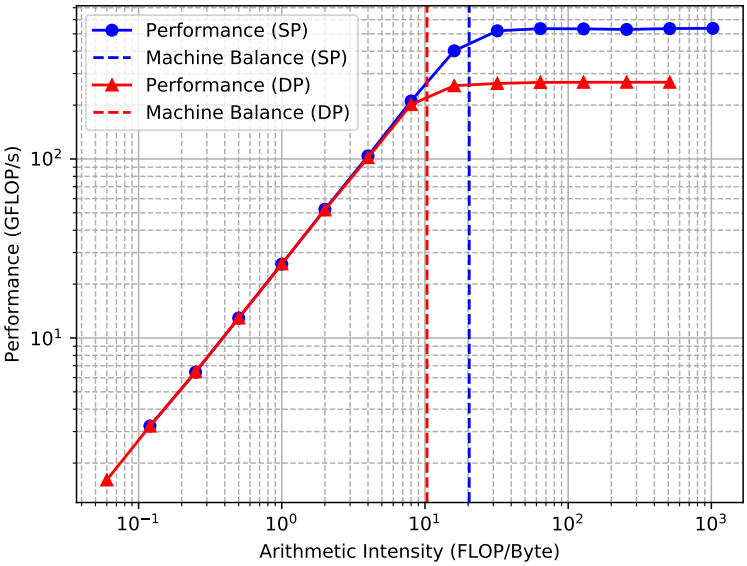


Figure 2.5: Roofline plot with machine balance for Intel Core i7 6700k CPU. The implemented kernel leverages the AVX2 vector extension.

Intuitively, machine balance denotes the capability of the machine, while code balance denotes the intensity of the applications. If $B_m < B_c$, then the volume of data requested by the application per flop will surpass the capability of the machine. In this case, the application performance will be bottlenecked by the memory bandwidth of the machine. Even having a perfect cache will not help in this scenario.

However, determining peak theoretical performance is non-trivial. One way is hand calculating based on the specification provided by the vendors. Unfortunately, the required information is not available in many

cases. Another way is to experimentally measure the peak achievable performance. Hopscotch adopts the latter approach. The kernel first loads an element of the working set into a register. Then it repetitively performs balanced multiply-add operations until the requested operational intensity is reached. Finally, the updated value is stored back. These steps are performed on all of the elements of the working set, and repeated for different operational intensity. Partial unrolling is done to keep all the fused-multiply-add (FMA) units busy without register spilling. Hopscotch provides a Roofline benchmark for both CPU and GPU platforms.

Hopscotch additionally displays the machine balance on the Roofline plot. Machine balance is the ratio of peak performance and peak memory bandwidth [32]. Machine balance is a property of the platform, while the operational intensity is a property of the application. An application will be memory-bound if the operational intensity of that application is lower than the machine balance. Otherwise, it will be compute-bound.

Figure 2.5 shows the Roofline plot generated by Hopscotch for an Nvidia GTX 1080 Ti GPU. This GPU has 28 streaming multiprocessors (SM). Each SM has 128 single precision cores and 4 double precision cores. These cores can execute one FMA instruction each clock. With boost core clock of 1582 MHz, the peak single precision performance is $(28 * 128 * 1582 * 2) = 11339$ GFLOP/s, and peak double precision performance is $(28 * 4 * 1582 * 2) = 354$ GFLOP/s. Actual performance can exceed these numbers with GPU Boost. This feature allows higher core clocks if thermal and power budget permit. As we can see in Figure 2.5, the reported performance with Hopscotch kernel is very close to the theoretical peak performance.

2.5 Conclusions

We presented Hopscotch, which is a micro-benchmark suite for memory characterization. It contains kernels with different access patterns for read-only, write-only, and mixed accesses. Furthermore, we introduced a kernel with tunable spatio-temporal locality. We also demonstrated how these kernels are combined into micro-benchmarks revealing important memory characteristics. Bandwidth, latency and caching efficiency are measured using the benchmark suite for three different platforms. We also demonstrated and validated the empirical roofline tool on CPU and GPU platforms.

As for the future direction, it would be interesting to do a principle component analysis with different performance metrics as the features to identify potential overlaps between the kernels and also to determine the gaps that are not covered by the current kernels. Another interesting direction to pursue would be to port the kernels to a few PIM-based architectures. Extending the kernels to enable evaluating memory controller fairness (e.g., detecting starvation) and prefetching schemes would be interesting as well.

Chapter 3

BigMap: Future-proofing Fuzzers with Efficient Large Maps

3.1 Introduction

Real-world applications usually have a large codebase, making it difficult to detect security vulnerabilities while providing a vast attack surface to the adversaries. Fuzzing techniques are geared towards automatically generating test vectors to expose these vulnerabilities or to improve the code coverage in general. Among different types of fuzzer, black-box fuzzers blindly generate random test vectors without resorting to any form of program analysis. Consequently, these fuzzers scale very well with program size and are easily parallelizable but are unlikely to find rare bugs. On the other end of the spectrum, white-box fuzzers can do a directed and exhaustive search of the coverage space with symbolic execution, but are prohibitively slow to be useful for large, real-world applications [51, 52, 53, 54]. Coverage-guided grey-box fuzzers fill the middle ground and so far have been most successful in finding software bugs. At the time of this writing, Google's OSS-Fuzz platform uncovered over 20,000 vulnerabilities on 300 projects [55] with the help of three coverage-guided grey-box fuzzers - libFuzzer [56], Honggfuzz [57], and American Fuzzy Lop (AFL) [58].

As the name suggests, coverage-guided fuzzers use some form of a coverage metric to track and guide their test generation process. For example, libFuzzer and Honggfuzz use basic block coverage. AFL, on the other hand, tracks edge hit counts with the help of a coverage bitmap. Each edge encountered while executing a test case is dynamically assigned to a location on this bitmap to store and update the hit count. This coverage bitmap is accessed very frequently and should occupy faster cache levels to maximize the test case generation throughput. For this reason, the size of the bitmap has historically been kept small (the default

size is 64kB for AFL). Due to the bitmap's size limitation and the randomness of the location assignment, it is possible to have hash collisions, where two or more edges point to the same location on the bitmap. Hash collisions introduce ambiguity in coverage feedback and can severely limit the fuzzer's ability to find bugs [59]. Our work seeks a better understanding and efficient mitigation of this issue.

The straightforward way for reducing hash collisions is to expand the hash space (i.e., increase coverage bitmap size). Prior works noted that naïvely enlarging the bitmap can severely diminish the test case generation throughput, potentially resulting in lower code coverage within the same time budget [59]. We investigated the reason behind the throughput drop with larger bitmaps. We observed that for large bitmaps, most of the time is spent doing a few specific operations (e.g., reset, classify, compare, and hash) on the bitmap. These operations are performed on the full bitmap, although only a small fraction of the bitmap is actively used for storing coverage statistics. This type of access pattern is inefficient and heavily pollutes the processor's data cache, ultimately lowering the throughput. In this work, we introduce BigMap, a *two-level* bitmap scheme that optimizes these map operations. BigMap adds an extra level of indirection to bitmap accesses to condense randomly scattered coverage metrics in a sequential bitmap, vastly improving cache locality behavior. Furthermore, the map operations now only need to be performed on the used portion instead of the full bitmap. Overall, our proposed approach enables using large maps without sacrificing throughput.

We integrated BigMap into AFL and conducted experiments with benchmarks from FuzzBench [60] and OSS-Fuzz [55]. With AFL's carefully tuned default map size of 64kB, BigMap demonstrated identical throughput, despite adding an extra level of indirection. The throughput gain over AFL increased with map size, with up to 13.6x (average of 4.5x) for a 2MB map and up to 114x (average 33.1x) for an 8MB map. BigMap also demonstrated better scalability with concurrent fuzzing instances, achieving an average of 9.2x higher throughput than AFL for a 2MB map and up to 12 parallel instances. The higher throughput resulted in uncovering 37% more unique crashes on average.

Interestingly, BigMap is compatible with any coverage metric (not just edge hit count) as long as it uses some form of a coverage bitmap. This property, along with the efficiency of BigMap with large maps, enables exploring aggressive compositions of coverage metrics and algorithms previously thought infeasible. To demonstrate this capability of BigMap, we selected a few large applications from OSS-fuzz [55] as seed benchmarks. Their discoverable edges are further amplified by enabling laf-intel transformations[1] [61] and then combining it with a more expressive coverage metric, N-gram [62]. This combination resulted in fuzzing harnesses with over 600k discoverable edges (over 5.5 million static edges). To put it into context, typical

[1]Laf-intel transforms multi-byte comparisons into a cascade of single-byte comparisons. Laf-intel also deconstructs switch statements and strcmp/memcmp functions into if-else statements.

real-world applications have around 1k - 50k discoverable edges [63, 64, 65]. After mitigating hash collisions on these benchmarks with the help of BigMap, we saw a 33% increase in the number of unique crashes. In summary, we make the following contributions:

- We investigate the shortcomings of enlarging bitmap to mitigate hash collision and identify the following key reasons: frequent operations on the full coverage map and excessive cache pollution.
- We leverage our findings in designing BigMap. BigMap introduces a two-level mapping scheme to limit the operations on the used region of the map. With this adaptive technique, the bitmap can be made arbitrarily large without sacrificing speed.
- We extend base AFL with our proposed method. Compared to AFL, we see an average of 4.5x higher test case generation throughput for a 2MB map and 33.1x for an 8MB map.
- We evaluate the scalability of BigMap with concurrent fuzzing instances. Compared to AFL, we see an average throughput gain of 9.2x for a 2MB map. Furthermore, BigMap was able to uncover 37% more unique crashes.
- BigMap's enables the aggressive composition of coverage metrics. We evaluate the composition of two well-known coverage metrics, laf-intel and N-gram, and find that unique crash coverage improved by 33%.

AFL fuzzer with integrated BigMap is open-sourced at: https://github.com/alifahmed/BigMap. **We published the findings of this work in [28].**

3.2 Background

3.2.1 American Fuzzy Lop (AFL)

AFL is one of the most popular fuzzers currently available. Many prior works [59, 66, 67] built upon AFL, including our work. AFL uses an evolutionary algorithm for fuzzing. Figure 3.1 illustrates this flow. In general, this workflow is applicable to other coverage-guided fuzzers as well. At the beginning of the process, AFL instruments the target application and populates the seed pool with user-provided seed inputs. Afterwards, AFL enters a fuzzing cycle: i) Selects a seed from the seed pool for mutation. ii) Mutates the seed to generate many new test cases. A seed is usually mutated tens of thousands of times before moving to the next seed. iii) Executes the generated test cases and checks the coverage feedback. If any test case crashes or hangs, it is reported to the user. If a test case covers an interesting path dictated by the fitness function (e.g., improves coverage), it is added to the seed pool as a potential candidate for future mutations. Otherwise, the test case is discarded. iv) After finishing with the current seed, the flow goes to (i) and selects a new seed for

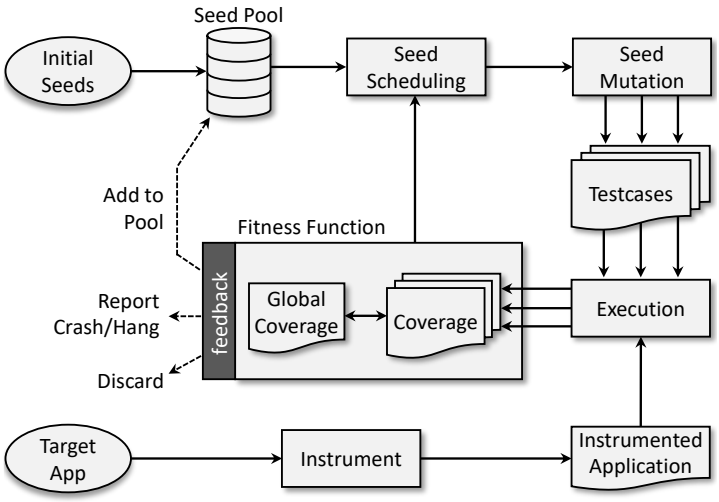


Figure 3.1: The generic workflow of a coverage-guided fuzzer.

mutation. Fuzzing cycle continues until the user interrupts, or some other criteria is met (e.g., coverage goal or time budget).

Seed Scheduling and Mutation

This section is kept short because our approach is orthogonal to the seed scheduling and mutation strategy. Seed scheduling policy determines which seed from the seed pool will be fuzzed next. AFL prioritizes the seeds based on their execution speed and input file length. Short input files are preferred because a mutation is more likely to touch important control structures and not just redundant data blocks on a smaller file [63]. As for mutating the seed, AFL applies a few deterministic (i.e., not random) mutation steps followed by random mutations. The mutation steps involve bit-flips, block substitution, splicing, etc. The deterministic mutation steps usually take a long time to finish. It is a common practice to skip this deterministic stage and directly apply random mutations for shorter runs (e.g., 24 hours).

Execution and Coverage Feedback

AFL collects the coverage of a test case with the help of the instrumented target. The exact execution path is not tracked. Instead, a coarse-grained edge hit count is used as the coverage metric [63]. The edges are identified as a hash of the (source_block, destination_block) tuple. Listing 3.1 shows the necessary steps.

```
1 B_X , B_Y = random % MAP_SIZE <COMPILE TIME>
2 E_XY = (B_X >> 1) ⊕ B_Y
3 coverage_bitmap[E_XY]++
```

Listing 3.1: Instrumentation capturing the hit counts of E_{XY}.

Here MAP_SIZE is the size of the coverage bitmap. B_X and B_Y are the source and the destination basic block IDs, respectively. E_{XY} is the ID corresponding to the $X \rightarrow Y$ edge. Basic block IDs are assigned at compile time following a discrete uniform distribution over the [0..MAP_SIZE) range. On the other hand, edge IDs are calculated at runtime and also falls within [0..MAP_SIZE). The shift operation in the edge ID calculation makes it possible to preserve the directionality of the edges (e.g., $E_{XY} \neq E_{YX}$). It also helps in properly identifying distinct tight loops (e.g., $E_{XX} \neq E_{YY} \neq 0$). AFL sports an alternative technique for getting edge IDs that leverages the *trace-pc-guard* coverage sanitizer of the Clang compiler [68]. In this method, the Clang compiler itself instruments static edges without any need to instrument at the basic block level. Unfortunately, this method cannot detect indirect edges as the target basic block information is unavailable at compile time.

Irrespective of how the edge IDs are generated, they act as an index to the coverage bitmap. The corresponding byte at that index stores the desired statistics (e.g., hit count for vanilla AFL) of that particular edge. The following steps are performed to collect the coverage of individual test cases:

- **Bitmap reset:** The coverage bitmap is a shared data structure and is used by all the test cases. Thus, before executing a test case, the coverage bitmap is cleared to remove any artifact of previous runs. A simple memset to zero does this job.
- **Bitmap update:** The instrumented target executes the test case and records the edge hit counts on the bitmap.
- **Bitmap classify:** The exact hit counts are converted to coarse hit counts by mapping them into buckets. The buckets used by AFL are: [1], [2], [3], [4-7], [8-15], [16-31], [32-127], [128,∞]. Hit counts that fall into different buckets are considered as an interesting change in the control flow. Change within the same bucket is ignored. Bucketing also mitigates the impact of accidental hash collisions.
- **Bitmap compare:** After the classify step, the modified bitmap is compared with a global coverage bitmap that keeps track of all the edges covered so far. Newly discovered edges, if any, are added to the global coverage map at this point. If the test case crashes/hangs instead, it is compared to a global crash/hang coverage bitmap.
- **Bitmap hash:** If the test case is considered interesting, a hash of the bitmap is calculated and saved for rapid comparison in the future.

Since these bitmap operations are performed for every test case (expect bitmap hash, which is performed for every *interesting* test case), it is crucial to minimize the time spent on these operations. One way to facilitate faster bitmap operations is to keep the bitmap size small. This limitation on map size leads to a high number of hash collisions. As stated earlier, collisions introduce ambiguity in coverage feedback and may

result in discarding interesting test cases. This work's primary objective is to enable large coverage bitmaps (thus reducing hash collisions) without incurring associated runtime overhead.

3.2.2 Collision Rate

In our work, the severity of the hash collision is quantified using the ***collision rate*** metric. Consider drawing n keys from a hash space of size H. Among the n draws, if c number of key matches with one of the previously drawn keys, then the collision rate is defined as c/n (where $c < n$). If the key draw follows a discrete uniform distribution, then the collision rate can be expressed using Equation 3.1.

$$CollisionRate(H, n) = 1 - \frac{H}{n}\left[1 - \left(\frac{H-1}{H}\right)^{n}\right] \tag{3.1}$$

Equation 3.1 is consistent with how AFL generates the block and edge IDs. Here, the hash space size H is analogous to the coverage bitmap size, and the number of drawn keys n is equivalent to the number of generated IDs.

Note that the collision rate does not indicate the actual number of keys with collision. Consider an example where the following keys are sequentially drawn: $\{4, 2, 5, 3, 2\}$. Here, the collision rate is $1/5$ and not $2/5$. Although the given collision rate definition does not account for all the colliding keys, we have used it to remain consistent with the existing literature [59, 63].

3.3 Implication of Naïve Hash Collision Mitigation Strategy

Hash collisions can be completely avoided by assigning unique IDs to every discoverable edge. Otherwise, traversing two (or more) different edges will update the same location in the coverage bitmap. Unfortunately, assigning unique IDs may not always be possible. AFL's default bitmap size is 64kB, where each byte stores the statistics of an edge. Thus, even in the best scenario, at most 64k edges can be assigned with different IDs. Any more than that, and collision will be unavoidable. The birthday problem suggests that the collision is likely to occur with significantly less than 64k edges [69]. Assuming a uniform distribution of the edge IDs within the 64kB bitmap range, the probability of having at least one collision is ~50% after assigning only 300 IDs.

Similar to edge IDs, block IDs are also randomly generated within the [0..MAP_SIZE) range (Listing 3.1). Thus, it is quite possible to have more than one basic block with the same ID. Edges originating from or entering these colliding blocks will point to ambiguous locations in the coverage bitmap. Bucketing the

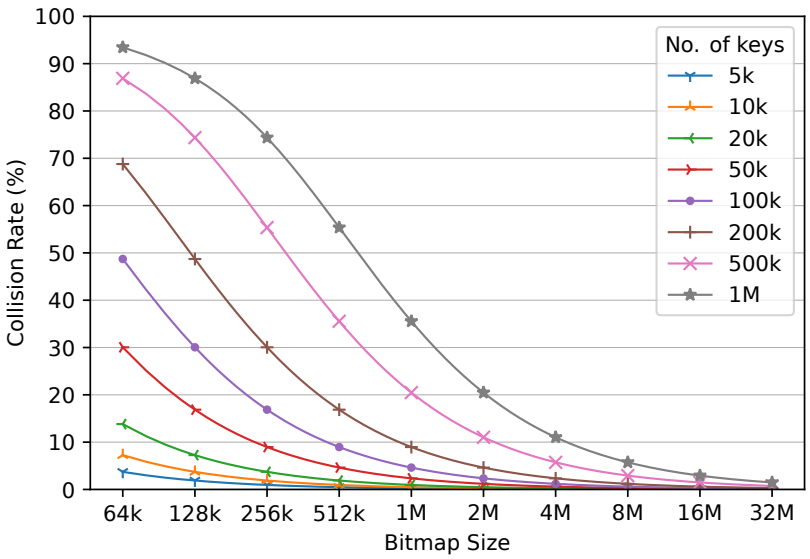


Figure 3.2: Hash collision rate drops as bitmap size is increased (derived from Equation 3.1).

hit counts provides some protection against such accidental hash collisions. Having too many collisions still severely limits the fuzzer's ability to guide its fuzzing process by providing incorrect coverage feedback.

The straightforward way of reducing hash collisions is to expand the hash space (i.e., use a larger bitmap). Figure 3.2 shows the collision rates with different bitmap sizes and the number of keys drawn (derived from Equation 3.1). The keys here are analogous to the discoverable edges and blocks. For real-world applications, the number of discoverable edges usually ranges from 1k to 50k. As a result, a 64kB map is subjected to ~30% collision rate. Using more thorough coverage metrics like full/partial path coverage [62], context-sensitive edge coverage [67], or branch condition transformations [61] can make the required number of IDs go well over 500k. These techniques can be stacked, further increasing the collision rate. We need a much larger map than 64kB if we want to explore these techniques without worrying about hash collisions.

3.3.1 Cost of Expanding Hash-space

The bitmap should be much larger than the number of required IDs to keep the collision rate in check. Unfortunately, increasing bitmap size also increases the runtime overhead of the bitmap operations. Figure 3.3 shows the runtime composition for six benchmarks with 64kB, 2MB, and 8MB bitmap sizes. For the small 64kB map, the fuzz target's execution time dominates the overall runtime. The costs of bitmap operations are negligible at this point. On the other hand, the runtime is dominated by the bitmap operations for the larger 8MB map. The **classify**, **compare**, and **reset** operations require iterating through the full bitmap for *every* test case. As a result, they are impacted most by the increase in bitmap size. Bitmap **hash** operation also needs to go through the full bitmap but is only performed on the *interesting* test cases. Therefore, the

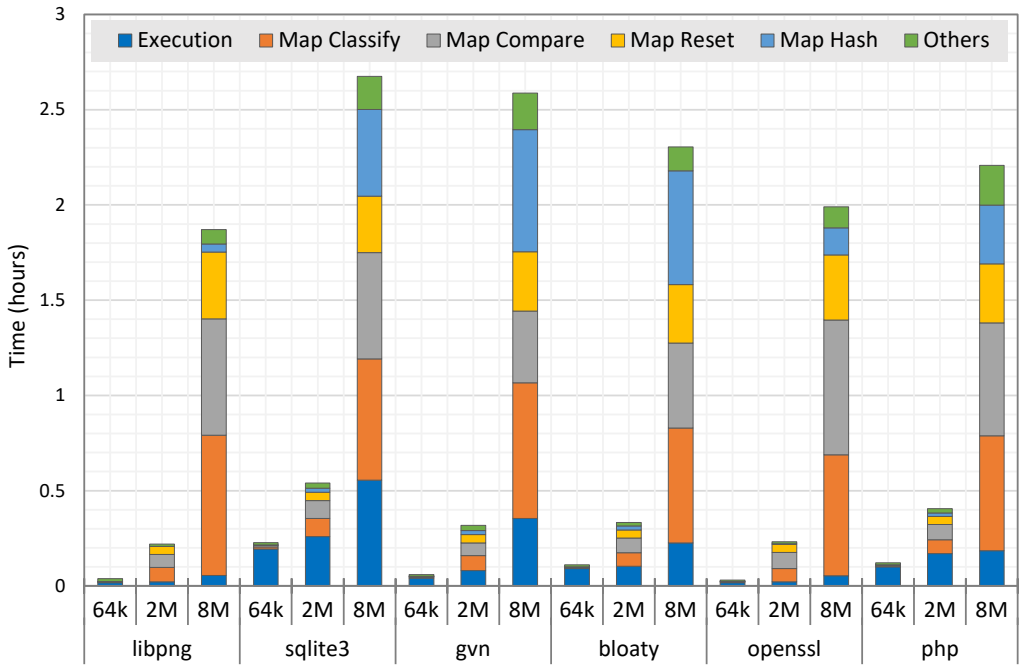


Figure 3.3: Runtime composition with varying bitmap sizes. Map operations dominate the runtime for bigger maps. The reported time is for one million test case generation.

overhead of hash operation varies considerably depending on the benchmark. There are a few other bitmap operations not shown in this figure, simply because they are too infrequent to have any perceivable impact on the runtime.

3.4 BigMap: Adaptive Two-Level Bitmap

In the AFL's data structure for coverage tracking, the keys are randomly distributed throughout the bitmap. Figure 3.4(a) shows an example where the edge ID E_{XY} is used as the key to access the coverage map. In this example, only five of the twelve locations are modified. However, since there is no information on exactly where these modified locations are, the bitmap operations like reset, classify, compare, etc., have to traverse the complete map. We propose the use of a two-level bitmap scheme to consolidate these scattered accesses.

3.4.1 Two-Level Bitmap Scheme

In our proposed scheme, the consolidation process is carried out during the bitmap update phase by maintaining three data structures: i) A *coverage_bitmap* that holds the coverage statistics. ii) An *used_key* that points to the next available space in the coverage_bitmap. iii) An *index_bitmap* that maps an edge ID to a location in the coverage_bitmap. Figure 3.4(b) demonstrates the update steps. First, we query index_bitmap[E_{XY}] to get

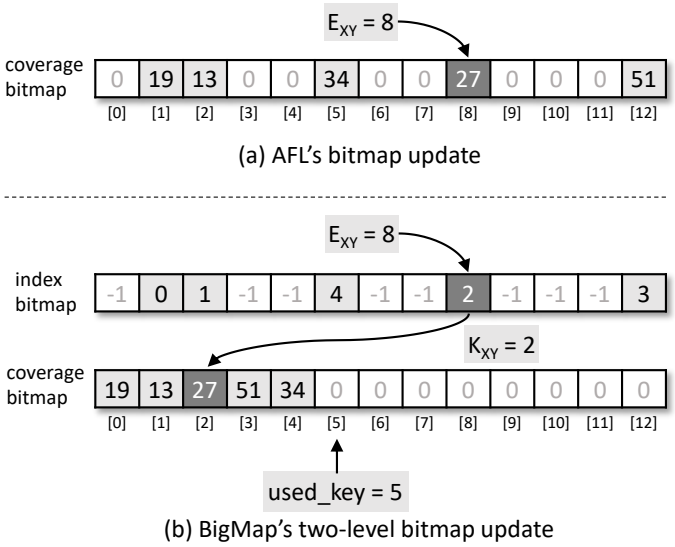


Figure 3.4: Steps of bitmap update operation for AFL's and BigMap's data structure. The hit counts in the coverage_bitmap are scattered in (a), while consolidated in (b).

the location of the stored hit count. If the edge is encountered for the first time, we will get an invalid location (-1 in our implementation). In this case, the index_bitmap[E_{XY}] is assigned to the next available location in the coverage_bitmap (= used_key). Once we have the location, the hit count in the coverage_bitmap is incremented.

As depicted in Figure 3.4, BigMap's scheme makes the coverage statistics contained within the first used_key locations, unlike AFL. Therefore, all the bitmap operations (except bitmap update) need to iterate over the [0..used_key) range instead of the full bitmap. As a result, the runtime of the map operations will depend on how many edges are discovered instead of how big the coverage bitmap is. An interesting aspect of this solution is its adaptive nature, where the default bitmap size can be arbitrarily large irrespective of the target application's size. Applications with a large number of discoverable edges will benefit from hash collision mitigation, and applications with few discoverable edges will not incur any significant overhead despite having large map structures. This flexibility helps in situations when it's difficult to assess the optimal map size in advance.

3.4.2 Illustrative Example

Figure 3.5 shows a step by step example of how the map operations are performed. We will focus on BigMap and will contrast it with AFL towards the end of this section.

At the beginning of the fuzzing session, BigMap initializes the index_bitmap to -1, indicating none of the edges are assigned any location yet. The hit counts in coverage_bitmap are also set to zero. This initialization

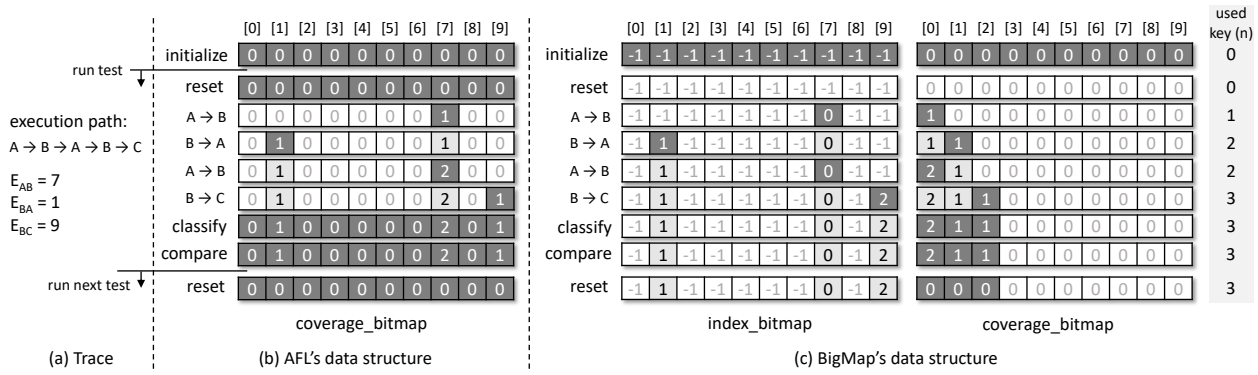


Figure 3.5: An illustrative example of bitmap operations on AFL's and BigMap's data structures. Value on top is the index of the bitmaps. Locations accessed at each step are highlighted in bold. (a) Execution trace and the assigned edge IDs (random). (b) AFL's data structure. Reset, classify, compare, etc., operations need to access the full bitmap. (c) BigMap's data structure. The full map is accessed only during initialization. Afterward, reset, classify, compare, etc., accesses only the used region of the coverage bitmap. Index bitmap is only accessed during the hit count update.

is performed a single time during the whole fuzzing campaign, and it is the only time BigMap accesses the full bitmaps. At this point, the index_bitmap and the coverage_bitmap are ready to capture the test case's coverage information. The used portion (none for the first run) of the coverage_bitmap is reset before each test is executed. During execution, the index_bitmap is updated as new edges are being discovered. Corresponding locations in the coverage_bitmap is also updated. After the execution is finished, the hit counts are bucketed and compared with the global coverage maps. If the test case is deemed interesting by the fuzzer, additional bitmap operations such as hashing, rank update, etc., may be performed. These steps read/modify only the used portion of the coverage_bitmap as well.

A few things to note here. The index_bitmap is touched only during the update phase. It is not accessed at any other phase, including reset. Therefore, the same edge will point to the same coverage_bitmap location for all the test cases. Also, the update phase is the only stage where AFL's data structure is more efficient. AFL's structure does one data access per edge compared to two accesses of BigMap's structure. Fortunately, the extra access shows good cache locality, as will be discussed in the next section.

3.4.3 Access Patterns of the Bitmap Operations

AFL's Data Structure

Table 3.1(a) summarizes the access patterns for AFL's data structure. The bitmap update does sparse access over the coverage_bitmap. These accesses correspond to the IDs of the encountered edges. It has a high temporal locality because the same edges are likely to be traversed again within the same program execution

Table 3.1: Access Patterns of the Bitmap Operations

(a) AFL's Data Structure

Map Operation	Bitmap	Access to	Temporal locality	Spatial locality	Cache pollution
Update	Coverage	Used map[2]	High	Low	Low
Others[1]	Coverage	Full map	Low	High	High

(b) BigMap's Data Structure

Map Operation	Bitmap	Access to	Temporal locality	Spatial locality	Cache pollution
Update	Index	Used map[2]	High	Low	Low
	Coverage	Used map[2]	High	High	None
Others[1]	Index	None	–	–	None
	Coverage	Used map[2]	High	High	None

[1] Bitmap reset, compare, classify, hash etc.

[2] Corresponds to the highlighted cells in the example of Figure 3.5.

(e.g., edges inside loops or common functions). The same edges are also likely to be traversed across different executions due to the overlap of execution paths.

The rest of the bitmap operations iterates the full map. Most of these locations do not contain any useful information, therefore causes heavy cache pollution. In turn, the cache pollution may trigger the eviction of useful data to slower cache levels or memory. For example, pollution may prevent keeping common edge locations in L1/L2 cache across consecutive executions.

BigMap's Data Structure

Table 3.1(b) shows the access patterns for BigMap's data structure. During the update operation, BigMap's structure makes two accesses per edge, first to the index_bitmap and then to the coverage_bitmap. Access to the index_bitmap is scattered and is identical to the pattern of AFL's data structure. On the other hand, access to the coverage_bitmap has a high spatial and temporal locality. The spatial locality stems from the fact that the edge hit counts are now residing in close vicinity. The rest of the bitmap operations do sequential access to the coverage_bitmap, exhibiting high spatial and temporal locality. We infer high temporal locality for BigMap's structure and not for AFL's structure. This is because AFL's structure has a high reuse distance as it accesses the full map. Overall, BigMap's structure demonstrates vastly improved cache locality behaviors compared to AFL.

3.4.4 Implementation Details

The BigMap approach requires minor modifications of AFL's instrumentation to support the two-level bitmap update. The new instrumentation is shown in Listing 3.2.

```
1 B_X , B_Y = random % MAP_SIZE <COMPILE TIME>
2 E_XY = (B_X >> 1) ⊕ B_Y
3 if (index_bitmap[E_XY] == −1)
4       index_bitmap[E_XY] = used_key++
5 K_XY = index_bitmap[E_XY]
6 coverage_bitmap[K_XY]++
```

Listing 3.2: BigMap instrumentation for map update.

Here, lines 1, 2, and 6 are identical to AFL's instrumentation scheme (Listing 3.1). Lines 3-5 are added to query and modify the index_bitmap. Since these instructions are executed for every edge, overhead can be a big concern. Given the rarity of new edge discovery, most of the time, the overhead will consist of one branch condition check (at line 3) and one extra access to the index bitmap (at line 5). The branch condition outcome is highly skewed towards not-taken and will be predicted correctly by the branch predictor almost always. Furthermore, the access to the index_bitmap is amenable to hit the L1 or L2 cache, making the access time negligible. The index_bitmap update (at line 4) will be invoked only when a new edge is discovered for the first time. Interestingly, while the index_bitmap is indexed by the edge ID, it does not necessarily have to be the case. In fact, any coverage metric can be used in edge ID's place, trivializing the integration process.

The modification in the instrumentation takes care of the bitmap update operation. Further adjustments are required to support the rest of the bitmap operations. It primarily involves changing the iteration count from the full map size to *used_key*. Bitmap hash operation is an exception to this rule. AFL uses CRC32 for calculating bitmap hash. If we always calculate hash in the [0..used_key) range, it might lead to wrong hash values. Consider the following example with three test case executions:

Execution Path	used_key	coverage_bitmap	Bitmap Hash
P1: $A \rightarrow B \rightarrow C$	2	{1,1,0,0,...}	crc32({1,1})
P2: $A \rightarrow B \rightarrow C \rightarrow D$	3	{1,1,1,0,...}	crc32({1,1,1})
P3: $A \rightarrow B \rightarrow C$	3	{1,1,0,0,...}	crc32({1,1,0})

The hash of first case is $crc32(P1) = crc32(\{1,1\})$. Here, $\{1,1\}$ are the hit counts up to the used_key. While executing the second case, the used_key will be incremented to 3. Therefore, the hash of third case will be calculated as $crc32(P3) = crc32(\{1,1,0\})$. The first and third paths are essentially the same. However,

the calculated hash values do not match because $crc32(\{1,1\}) \neq crc32(\{1,1,0\})$. To avoid such discrepancy, BigMap calculates the hash up to the last non zero value in the coverage_bitmap.

3.4.5 Additional Optimizations

We carried out a few additional optimizations to make our implementation faster. These optimizations are orthogonal to the two-level bitmap scheme and can be adopted by any AFL based fuzzers. First, we merged the bitmap classify and compare steps. The bitmap compare operation almost always follows the classify operation. Because these operations are carried out in the same region of the bitmap, they can be easily merged. This merging allows more efficient use of cache and cuts the cost of (compare + classify) to half. The second optimization is to replace normal reset operation with a non-temporal version. The reset operation happens just before the execution and can pollute cache with regions of the bitmap that are never used. Using non-temporal stores prevents this pollution. This optimization is only beneficial to the vanilla AFL because BigMap already limits the map operations to the used region. Our final optimization is to allocate the index and coverage bitmap using the OS-provided facility for huge pages. There are limited numbers of slots on L1/L2 DTLB, and a large bitmap can consume many of them, resulting in frequent page-walks caused by DTLB misses. Allocating the bitmaps on a huge page reduces these overheads.

3.5 Evaluation

We evaluated our proposed approach in three steps. First, we demonstrated that BigMap could support larger maps without sacrificing test generation throughput, unlike standard AFL. Second, we investigated BigMap's ability to support coverage metric compositions and whether that leads to practical benefits in terms of improved code coverage. This step also acts as a justification for using large coverage maps. Finally, we evaluated the scalability of both fuzzers with respect to the number of concurrent fuzzing instances.

3.5.1 Experimental Setup

System Configuration

The experiments were conducted on a system with two Intel Xeon E5645 CPUs (totaling 12 physical cores) clocked at 2.40GHz. Each fuzzing instance was pinned to a separate physical core with a private 32kB L1 data cache, 256kB unified L2 cache, and a shared 12MB L3 cache. Fuzzers were run for 24 hours. Because the run time is relatively short, the deterministic fuzzing step is skipped, and the fuzzers were configured to run in persistent mode. Persistent mode enables feeding multiple inputs in a loop and does not have any

Table 3.2: Benchmark Characteristics

Benchmark	Number of seeds	Discovered edges[1]	Collision rate[2] (%)	Static edges[3]	Version
zlib	77	722	0.55	875	v1.2.11
libpng	1	1,218	0.92	2,987	v1.6.35
systemd	6	2,314	1.74	53,453	v245
libjpeg	1	2,928	2.20	9,542	v2.0.4
mbedtls	1	5,377	3.99	10,942	v2.21.0
proj4	43	6,379	4.71	7,830	v6.3.1
harfbuzz	58	8,930	6.51	10,021	v2.6.4
libxml2	1	9,422	6.86	50,327	v2.9.10
openssl	2,241	10,297	7.46	45,989	v1.0.2u
bloaty	94	10,536	7.62	89,658	v1.0
curl	31	12,728	9.11	62,523	v7.68.0
php	2,782	20,260	13.98	123,767	v7.4.3
sqlite3	1,256	40,948	25.64	45,136	v3.31.1
licm	101	64,317	36.29	977,899	v10.0.1
gvn	140	65,781	36.89		
strength-reduce	122	76,065	40.83		
indvars	174	82,105	42.98		
loop-vectorize	345	108,231	51.06		
instcombine	1,046	131,677	56.90		

[1] Maximum edge coverage among all fuzzing configurations.

[2] With a 64kB map.

[3] Derived using SanitizerCoverage [68].

fork() call or initialization overheads, thus significantly boosts the test execution rate. This setup is adopted from FuzzBench [60]. As for the instrumentation mode, we used the *afl-clang-fast* that leverages an LLVM compiler pass to inject the instrumentation code. This mode is faster than the gcc-based or coverage-sanitizer based alternatives [63]. Optimizations mentioned in Section 3.4.5 applied to both AFL and BigMap.

Benchmarks

We used 19 benchmarks in our experiments. The characteristics of these benchmarks are given in Table 3.2. The first 13 benchmarks are taken from FuzzBench [60]. These benchmarks are relatively small and have low collision rates. The remaining six benchmarks are LLVM optimization passes collected from OSSFuzz [55]. These benchmarks share the same LLVM-opt binary [70], and different fuzzing harnesses are selected via

command-line arguments. The LLVM-opt binary itself has a high number of static edges. Collectively, the benchmarks span a wide range of discoverable edges (∼1k - 131k) and collision rates.

Performance Metrics

The following metrics are used to evaluate the performance of our approach:

- ***Test case generation throughput or execution rate:*** Denotes the number of test cases evaluated by the fuzzer per unit time. Everything else being equal (e.g., seed selection and mutation strategy), a fuzzer with a higher throughput is expected to give better coverage.
- ***Unique crashes:*** AFL has a built-in deduplication mechanism for finding unique crashes. AFL considers a crash unique if it covers an edge unseen by the previous crashes or does *not* cover an edge common in all the previous crashes. This mechanism requires maintaining a local and global *crash-coverage* bitmap, making it inherently biased towards larger maps. To avoid this bias, we resorted to Crashwalk [71], which takes the hash of the call stack and the faulting address for deduplicating crashes.
- ***Edge Coverage:*** While crash coverage is the proper way of quantifying a fuzzer's performance, crashes in a program are typically sparse. Therefore, in addition to crash coverage, we also report edge coverage. Intuitively, a fuzzer that covers more edges are also likely to discover more bugs. To get the edge coverage, we collected the output corpus of the fuzzers and subjected them to a bias-free independent coverage build.

3.5.2 Evaluating the Impact of Map Size Variation

We claimed that BigMap performs efficiently regardless of the size of the coverage bitmap. This section validates the claim by comparing BigMap's performance with AFL for four different map sizes: 64kB, 256kB, 2MB, and 8MB. In this experiment, we used an average of three runs to reduce the variations introduced by random mutation steps.

Impact on Test Case Generation Throughput

The test case generation throughput of AFL and BigMap is shown in Figure 3.6. As expected, AFL's throughput dropped dramatically as the map size is increased. On average, AFL's throughput went from 4,400/sec for a 64kB map to only 125/sec for an 8MB map. BigMap handled large maps gracefully without any significant drop, and the average throughput remained consistently above 4,100/sec irrespective of the map size.

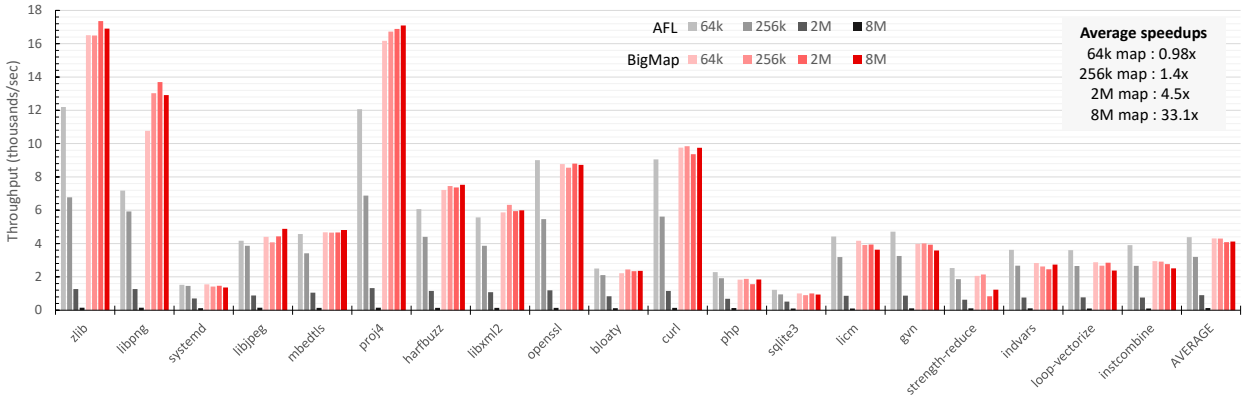


Figure 3.6: Test case generation throughput of AFL and BigMap with different map sizes. AFL's throughput drops significantly as the map size is increased. Map size variation has considerably less impact on BigMap.

For the 64kB map, BigMap usually outperformed AFL for smaller benchmarks (e.g., zlib, libpng, proj4), while AFL performed better on larger benchmarks (e.g., sqlite3, indvars, instcombine). This outcome is because only a tiny portion of the 64kB map is used for the small benchmarks. In such cases, BigMap gained the advantage by traversing the used region of the map. On the other hand, larger benchmarks almost completely filled the 64kB map. As a result, BigMap and AFL performed nearly the same during the classify, compare, and reset stages, but BigMap is ultimately slightly slower due to the extra indirection overhead during the update stage. One thing to note here is that the nearly full map also implies very high collision rates, suggesting the use of maps bigger than 64kB would be beneficial. Other factors impacting the throughput include the working-set size and access-pattern of the benchmark itself.

For larger maps, BigMap universally provided higher throughput than AFL. The 8MB map, in particular, incurred an extremely high performance hit for AFL. This performance hit is because, with an 8MB map, the combined size of the local and global coverage maps exceeded the last-level cache capacity of our experimental setup. Note that for a few benchmarks (e.g., libpng, proj4, libjpeg etc.), BigMap attained higher throughput at larger map sizes. We attribute this behavior to the various non-deterministic steps applied throughout the fuzzing process. As mentioned before, we have aggregated multiple runs to reduce the impact of randomness. Still, a fuzzing run can produce test cases that exercise longer (or shorter) execution paths more frequently relative to other runs.

On average, BigMap attains 0.98x, 1.4x, 4.5x, and 33.1x higher throughput for 64kB, 256kB, 2MB, and 8MB maps, respectively. We conclude that BigMap might not be an attractive choice for small map size of 64kB. However, if larger map is required (e.g., for reducing hash collisions or to support complex coverage metrics), then BigMap clearly provides superior test generation throughput.

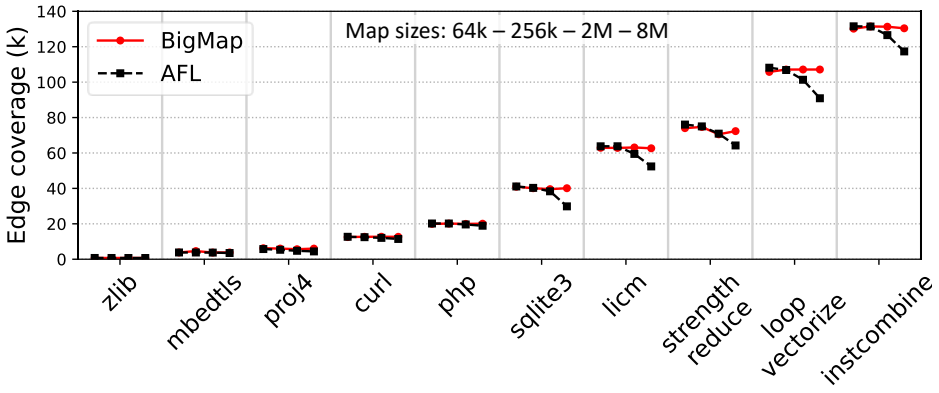


Figure 3.7: Edge coverage with varying map sizes. AFL's edge coverage suffers due to throughput loss with bigger maps. Not all benchmarks are shown to improve clarity.

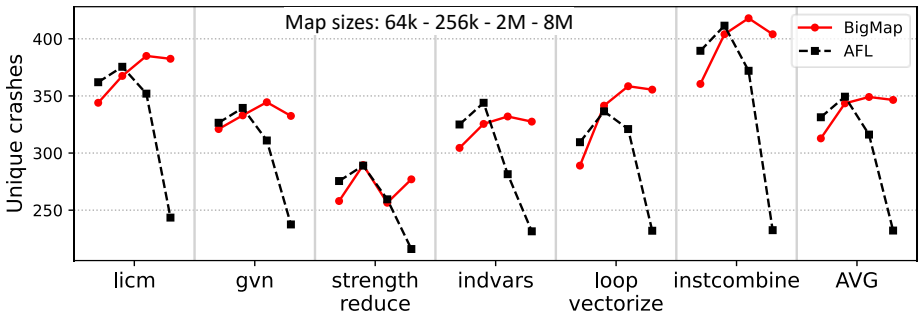


Figure 3.8: Unique crashes found with varying map sizes. Going from 64kB to 256kB map shows improvement as a result of reduced collisions. AFL suffers for bigger map sizes due to throughput loss.

Impact on Edge Coverage and Unique Crashes

Figure 3.7 shows the edge coverage with map size increase. During a fuzzing campaign, the rate of discovering new edges is initially high and then flattens out as time progresses. Our results indicate that BigMap reached the plateau for all of the benchmarks within the 24 hour time budget. AFL performed identically for small benchmarks. However, AFL's low throughput on bigger maps prevented it from reaching the plateau for benchmarks with a higher number of discoverable edges. Note that mitigating the hash collision was not particularly beneficial at improving the edge coverage. A public report available on FuzzBench indicates that the metric edge coverage has a relatively small variance across a wide range of fuzzers [72]. Furthermore, AFL authors noted that edge count bucketing provides some protection against collisions [63]. We hypothesize that the inherent small variance and the binning process made the edge coverage relatively insensitive to collisions.

Crashes, on the other hand, are extremely sparse and do not follow any simple pattern. We were able to find crashes on the bloaty and the LLVM benchmarks. For bloaty, we found one unique crash on all configurations except for AFL 8MB. The number of unique crashes found on the LLVM benchmarks is given in Figure 3.8. From this figure, it is evident that AFL performed its best with a 256kB map. The smaller

Table 3.3: Code Coverage with laf-Intel and N-gram

Benchmark (n-gram + laf-intel)	Collision rate		Edge coverage		Unique crash	
	64kB	2MB	64kB	2MB	64kB	2MB
loop-unswitch	70.6	4.9	214,437	211,697	276	325
sccp	71.1	5.2	218,473	226,084	261	324
earlycase	75.3	5.8	260,008	255,295	279	382
loop-prediction	75.8	6.2	265,740	270,806	202	265
loop-rotate	76.2	6.2	271,383	269,534	276	384
irce	77.0	6.0	281,479	262,675	226	245
licm	78.5	7.1	301,490	312,943	284	433
gvn	79.0	7.3	309,262	324,302	295	367
simplifycfg	79.1	7.4	311,143	325,526	285	412
strength-reduce	83.1	8.4	387,462	373,813	250	307
indvars	84.0	9.3	409,555	414,217	271	342
loop-vectorize	87.2	11.2	512,991	510,469	233	362
instcombine	86.9	13.1	588,397	602,669	295	434
AVERAGE	78.8%	7.5%	333,217	335,387	264	352

64kB map prevented finding more crashes due to collisions, while larger maps of 2MB and 8MB caused excessive runtime overhead, leading to low crash coverage. Interestingly, the *optimal* map size is unknown beforehand and may vary with the target application. Therefore, to find the most crashes, AFL has to run the target application with different map sizes (or have access to an oracle). However, testing with multiple map sizes will consume valuable compute time that may have been better utilized otherwise (e.g., longer runs or multiple instances with co-operative fuzzing). Finding the optimal map size is less of an issue for BigMap as we can choose an arbitrarily large map size with little runtime penalty. This adaptive nature of BigMap makes it an attractive choice.

3.5.3 Evaluating Coverage Metric Composition

Previously we mentioned how BigMap's efficiency with large maps enables an aggressive composition of multiple coverage metrics. In this section, we investigate one such scenario by stacking laf-intel [61] and N-gram [62]. Because the original AFL does not have in-built support for laf-intel and N-gram, we implemented BigMap on a community-maintained version of AFL called AFLPlusPlus [73]. We used all the LLVM fuzzing harnesses available on OSS-Fuzz as benchmarks. The laf-intel transforms each multi-byte comparison into a series of single-byte comparisons. The switch statements and strcmp/memcmp functions are also

deconstructed into multiple if-else statements. With laf-intel applied, the resulting LLVM-opt has around 5.5 million static edges. N-gram does not increase the number of static edges but provides a more thorough coverage metric. Unlike AFL's default edge coverage with (src_block, dst_block) tuple, N-gram gets partial path coverage by hashing the last N blocks. We choose $N = 3$ (i.e., the hash of the last three blocks) for this experiment. With stacked laf-intel and N-gram applied, the covered edges vary between 212k - 603k (∼87% collision rate). While laf-intel and N-gram independently showed improvement in terms of edge/crash coverage in small benchmarks, they were not applied to such large benchmarks previously due to excessive hash collisions. To our knowledge, this is the first time the combination of laf-intel and N-gram is applied to benchmarks of this scale. Also, note that the purpose of this experiment is not to scrutinize the effectiveness of the N-gram or laf-intel themselves, rather show that BigMap can effortlessly support such combinations with high map pressure. Similar to the setup of Section 3.5.2, we took an average of three runs to reduce the variation caused by random mutation steps.

The results of the experiment are shown in Table 3.3. Here, both the 64kB and 2MB version employs BigMap. By mitigating collision with a bigger map, the unique crashes found improved by 33% on average. However, the edge coverage remained unaffected, similar to what we observed in our previous experiment. We conclude that for large applications and/or when applying extensive coverage metrics, crash coverage can benefit from collision mitigation.

3.5.4 Evaluating the Scalability with Parallel Fuzzing

Every fuzzing instance uses one CPU core. As a result, a system with n physical cores can run n concurrent fuzzing instances with virtually little performance penalty (assuming there is minimal contention for other system resources) while gaining about n times more throughput. This linear scaling property is achievable by programs with a small memory footprint, where the program and the used portion of their bitmap fit within faster cache levels. In this section, we evaluate how scaling fares when a large bitmap (i.e., 2MB) is used. For this experiment, we ran 4, 8, and 12 concurrent instances in the master-secondary configuration. In this configuration, a single master instance performs the deterministic fuzzing steps before proceeding to random fuzzing. The rest are secondary instances that skip the deterministic step. The output corpus is periodically synchronized between these instances. This configuration is standard for all real-world parallel fuzzing sessions.

Figure 3.9(a) shows the resultant throughput. Each benchmark's throughput is normalized to the corresponding single-run version to visualize the scaling effect better. The black line is added as a theoretical reference for 1:1 scaling, where k instances gain k times the throughput. The bold red line is the average

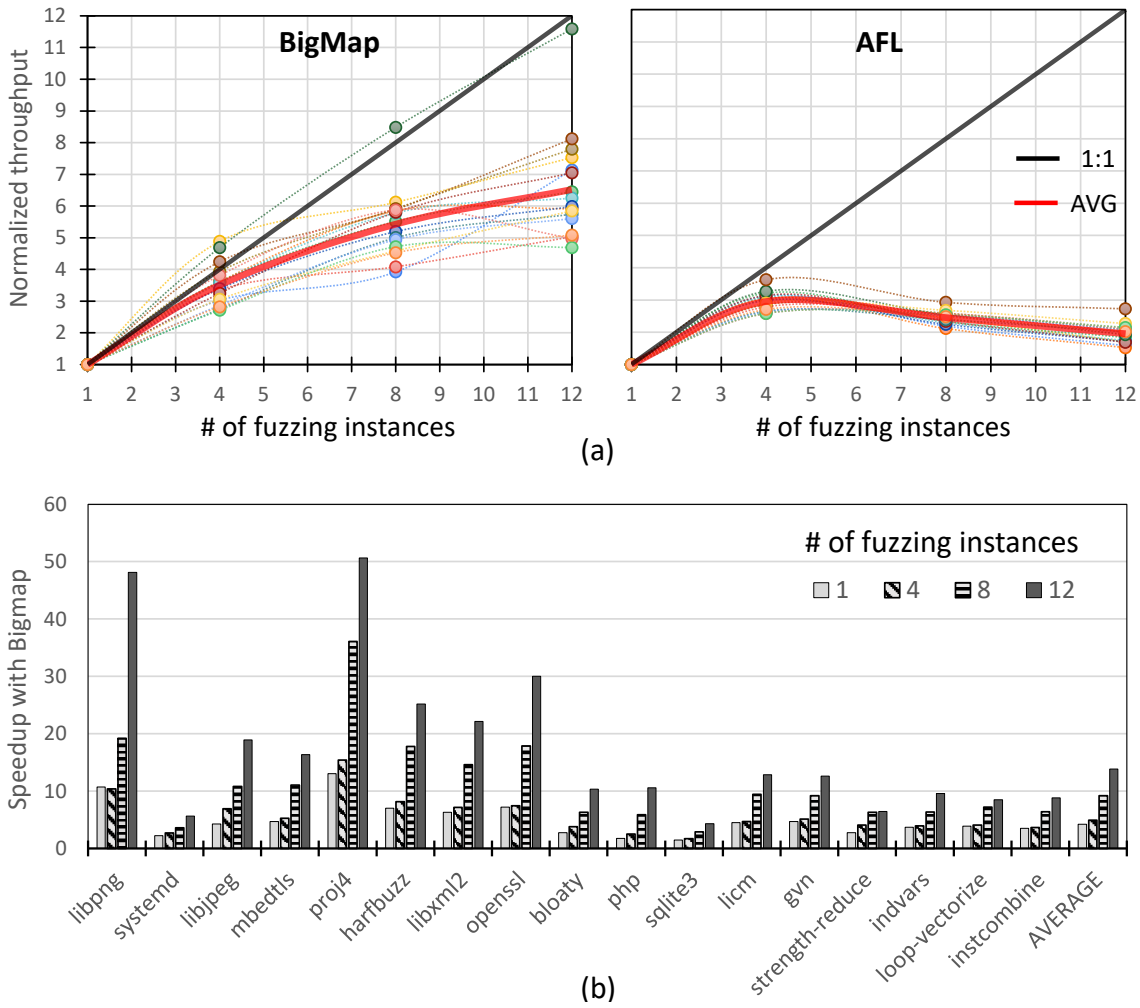


Figure 3.9: (a) Throughput (normalized to the single instance) vs. the number of fuzzing instances. Dotted lines represent the individual benchmarks from (b), and the solid red line is their average. The solid black line shows 1:1 scaling as a reference. (b) Speedup attained by BigMap over AFL. The coverage map is fixed to 2MB for both (a) and (b).

execution rate across all benchmarks. It is evident that both BigMap and AFL cannot maintain 1:1 scaling with large maps. The reason is, with multiple instances and large maps, the working set is much more likely to exceed the last-level cache capacity. Note that the last-level cache is shared across all the fuzzing instances. BigMap performs relatively well since it does not access the full map, therefore having a smaller effective memory footprint. AFL scales poorly. The throughput of AFL has a negative slope above four instances, meaning the total number of executions actually went down as the number of instances was increased. The per benchmark speedup attained by BigMap is given in Figure 3.9(b). This speedup is measured by taking the ratio of *total* test cases generated by BigMap and AFL with equal number of instances. As AFL scales poorly with the number of instances compared to BigMap (demonstrated in Figure 9(a)), it is expected for

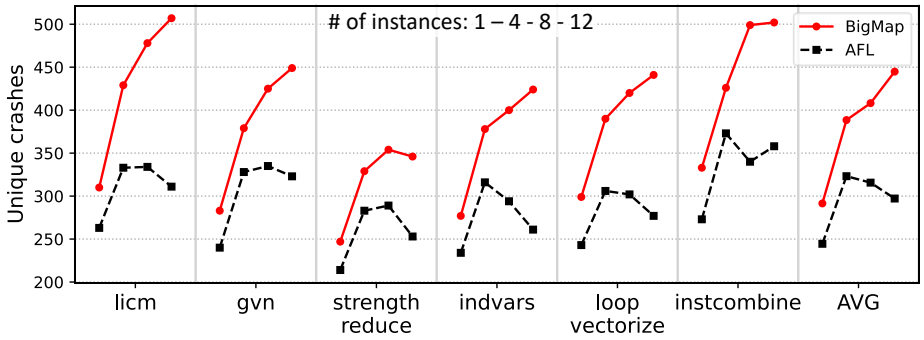


Figure 3.10: Unique crashes found with a varying number of fuzzing instances. The coverage map is fixed to 2MB.

the speedup to show a super-linear behavior. On average, BigMap achieved a speedup of 4.9x, 9.2x, and 13.8x for the 4, 8, and 12 concurrent runs, respectively.

Figure 3.10 depicts a similar trend in the number of unique crashes found. AFL suffers due to the drop in execution throughput. For 4, 8, and 12 instances, BigMap found 20%, 36%, and 49% more unique crashes on average. If we compare the best configurations available on our hardware (e.g., 12 instances for BigMap and 4/8 instances for AFL), BigMap shows an average speedup of 9.2x and uncovers 37% more crashes.

3.6 Related Work

Fuzzing as an evolutionary process was first introduced by Sidewinder in 2006 [74]. Since then, most successful fuzzers have followed this path [58, 73, 56, 57, 75, 66, 67, 76, 77, 78]. A critical component in this evolutionary process is the fitness function that determines what inputs will be used as seeds for future fuzzing rounds. AFL and AFL based fuzzers [58, 73, 66] use coarse edge hit counts as the fitness function. Any test vector that exercises a yet-unseen edge or a seen edge with a different hit count is considered an interesting input. On the other hand, libFuzzer based fuzzers [56, 57, 75] leverage compiler support such as SanitizerCoverage [68] to utilize basic block coverage as the fitness function. Angora [67] combines function calling context with edge coverage to differentiate between interesting test cases covering the same sets of edges but have unique execution paths. PerfFuzz [79] considers both execution count and code coverage. Ankou [78] queries behavioral similarity between a new test case and the current seeds in the seed pool to determine if it should be considered as interesting. All of these approaches use some form of code coverage as the fitness function. BigMap is orthogonal to these approaches and can be adopted to improve their fitness functions' accuracy by reducing collision.

In addition to seed selection, fuzzers can leverage coverage information for scheduling seeds from the seed pool. AFL schedules "favored" entries more frequently, and these entries are determined based on the

edge coverage. AFLFast [66] selects seeds that cover the least frequently traveled paths. VUzzer [75] uses control-flow graphs to model the execution path and prioritizes inputs that visit deeper blocks. Cerebro [80] employs a multi-objective algorithm that takes code coverage, complexity, and execution time into account during scheduling the seed. FairFuzz [81] prioritizes seeds based on rare branch coverage. The intuition being that rare branches are more likely to hide hard to trigger bugs. NeuFuzz [82] trains a deep neural network model to differentiate between a vulnerable path from a clean path and prioritizes the vulnerable one. AFLGo [83] measures branch distance to select seeds that are closer to predetermined targets. Since these approaches use coverage feedback in their scheduling mechanism, hash collisions can obscure the seeds' priority.

The expressiveness of the coverage metric is another factor that influences the collision rate. Angora's context-sensitive coverage puts up to eight times more pressure on the bitmap [67]. Coverage metrics such as N-gram (hash of last N branches), memory-access-aware branch coverage, and memory-write-aware branch coverage also exhibits higher map pressure than simple edge coverage [62]. Control-flow transformations such as laf-intel [61] or CmpCov [84] can increase the map pressure as well, necessitating collision mitigation.

Fuzzers do not need to fixate to a particular coverage metric or scheduling algorithm. An ensemble of different fuzzing mechanisms is proven to be an effective strategy [62, 85]. Ensemble fuzzers run multiple fuzzing instances with different metrics and periodically cross-pollinate the inputs. However, unlike BigMap, they do not stack the coverage metrics together, which is still subjected to increased hash collisions. Comparing BigMap with ensemble fuzzing is not covered in this work and can be an interesting avenue for future research.

CollAFL [59] is the state-of-the-art technique for mitigating hash collisions in coverage bitmap. It leverages static analysis to distribute edge IDs with a link-time compiler pass. Blocks with a single incoming edge are assigned IDs statically. For other blocks, injected instrumentation generates the IDs at runtime. It adapts to indirect edges by considering all blocks with no incoming edges as the potential branching target. One shortcoming of CollAFL is that it cannot be extended for coverage metrics other than the block or edge coverage (e.g., N-gram, Angora). In addition, it expands the bitmap to fit all the statically assigned IDs. Our experimental findings (presented in Table 3.2) indicate that only a fraction of the static edges are visited during a fuzzing campaign, making the increase in map size a source of unnecessary runtime overhead. While both BigMap and CollAFL aim to solve the hash collision issue, they are orthogonal mitigation techniques. BigMap can be used independently of CollAFL to reduce hash collisions. It can also be used in combination with CollAFL to completely eliminate collisions while providing more efficient access to the map. Furthermore, BigMap supports any form of coverage metric as long as it is recorded in a coverage bitmap, making it applicable to a wide variety of fuzzers.

3.7 Conclusion

We investigated the common belief that enlarging bitmaps to mitigate hash collisions necessarily results in the deterioration of both throughput and quality in fuzzing campaigns. Our key observation is that the primary source of overhead stems from frequent map operations performed on the full bitmap, although only a fraction of the map is under active use. Although our initial assumption was that PIM-based architecture would be a good fit to accelerate such a workload, we found a software-based solution that optimizes memory access behaviors of the coverage-bitmap operations. Consequently, we proposed BigMap, a two-level bitmap that adds an extra level of indirection to limit the map operations on the map's active regions. This approach reduced the effective working set size, improved cache utilization and reduced cache pollution. We further improved our approach by incorporating huge pages for the coverage-bitmaps, reducing the page walks caused by TLB misses. Our evaluation results showed 0.98x-33.1x throughput gain over AFL as we increased the map size from 64kB to 8MB. BigMap also demonstrated better scalability with the number of concurrent fuzzing instances. Furthermore, BigMap's compatibility with most coverage metrics, along with its efficiency on large maps, enabled exploring aggressive compositions of coverage metrics and fuzzing algorithms, uncovering 33% more unique crashes. By making the use of large maps practical and open-sourcing BigMap, we hope to enable and spur further research into the design space of coverage metrics.

Chapter 4

GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Updates and Analysis

4.1 Introduction

Streaming graph processing involves performing batched updates and analytics on a time-evolving graph. The update phase handles modifications to the graph topology (e.g., insertion/deletion of edges and nodes), while the analytics phase runs the necessary algorithms on the graph. This is a common scenario in many real-world graph applications such as social network analysis [86, 87], bioinformatics [88, 89], recommendation systems [90, 91], routing and navigation [92], knowledge discovery [93], sensor networks [94], etc. The focus of streaming graph processing is fundamentally different from static graph processing. *Static graphs* are constructed only once, and the construction cost gets amortized over time. Therefore, the overall performance of a static graph processing framework is primarily determined by the analytics throughput. In the case of *streaming graphs*, the graph topology can change very frequently. Hence, both update and analytics throughput is critical for streaming graphs [40].

The most common operation during the update phase is *edge lookup*. The lookup is performed before insertion to avoid duplicate edges[1] and before deletion to find the location of the target edge. On the other hand, the most common operation during the analytics phase is the *neighborhood traversal* of a given vertex. The performance of a streaming graph processing framework is critically dependent on how efficiently the graph storage format can support these lookup and traversal operations. Existing storage formats for streaming graphs usually employ variations of adjacency lists or hash tables [97, 96, 40, 95, 98]. Approaches based on adjacency lists [95, 40] provide high update throughput on short-tailed graphs[2] but suffer in heavy-tailed graphs as it requires linear lookup through the edge array [40]. On the other hand, hash-based approaches [97, 96] offer constant-time lookup, providing better update throughput on heavy-tailed graphs. However, they perform poorly on short-tailed graphs because the overhead of hash calculation and several random accesses becomes more expensive than conducting a simple linear search. Furthermore, edges are stored in hash tables relatively sparsely to mitigate collisions. As a result, edge traversal becomes inefficient and negatively impacts their analytics phase's throughput. None of the existing approaches can efficiently handle both short-tailed and heavy-tailed graphs.

This work proposes GraphTango, a streaming graph representation format that provides excellent performance regardless of the graph's degree distribution. Our key idea is to adaptively switch the underlying data structure based on the vertex degree: i) *Type1 vertex*: Low-degree vertices where the edges are stored within the same cache line as the neighborhood metadata. Update and edge traversal thus requires only one cache line access, unlike other approaches. ii) *Type2 vertex*: Medium-degree vertices that store edges as adjacency lists. The degree is too high for this type to fit all edges in a cache line, but small enough so that linear search performs better than hashing. iii) *Type3 vertex*: High-degree vertices that store edges as adjacency lists, along with hash tables storing indexes to the adjacency lists. In this case, the adjacency list provides optimal edge traversal during the analytics phase, while the hash table provides constant-time lookup during the update phase. The hash tables are not accessed during the analytics phase, avoiding any potential cache pollution. To improve the cache access pattern of the hash table, we designed an open-addressing-based hash table with double hashing that fully utilizes every fetched cache line. Our proposed hashing scheme minimizes cache line fetches and is especially beneficial if the hash tables do not fit into the last level cache (LLC), which is often the case for real-world graph workloads[3]. With this hashing scheme, updates for Type3 vertices are performed with only three cache line accesses for more than 99.2% of the cases. In addition, we

[1] In accordance with the prior works [40, 95, 96, 97], edges are inserted only after a lookup to avoid duplicate edges.

[2] Following prior work [40], we define heavy/short-tailed graph with respect to an update batch: heavy-tailed graphs have high *maximum degree* within a batch. Short-tail is the opposite.

[3] Even with our smallest dataset of 5M edges, the LLC miss rate during the update phase is over 49%, indicating that the working set size is larger than the LLC.

developed a thread-local lock-free memory pool that allows fast growing and shrinking of the adjacency lists and hash tables in a multi-threaded environment.

We evaluated GraphTango by integrating it with the SAGA-Bench [40] benchmarking framework. SAGA-Bench integration ensures that all approaches use the same algorithm implementations via a common API. Therefore, any performance improvement comes purely from the data structure standpoint. SAGA-Bench comes with four representation formats: AdListShared, AdListChunked, Stinger [95], and DegAwareRHH [97], each of which is shown to excel in different algorithm and dataset combinations [40]. Details of these formats can be found in Section 4.2. For update operations, GraphTango consistently performed best across all datasets. On average (maximum), GraphTango demonstrates 4.5x (6.6x) higher insertion throughput and 3.2x (5.0x) higher deletion throughput over the *next best* approach. As for analytics, GraphTango offers 1.1x (1.6x) higher throughput than the *next best* approach. Unlike prior approaches, GraphTango provides excellent update and analytics throughput for both short-tailed and heavy-tailed graphs.

Being a storage format, GraphTango is orthogonal to most full-fledged graph processing frameworks and can easily replace the underlying storage formats of those frameworks. To demonstrate, we integrated GraphTango with the state-of-the-art graph processing frameworks DZiG [99] and RisGraph [100]. *DZiG + GraphTango* reduced the overall batch processing runtime by 2.3x (5.2x) on average (maximum) compared to the original DZiG. *RisGraph + GraphTango* reduced the overall batch processing runtime by 1.5x (1.9x) on average (maximum) compared to the original RisGraph.

GraphTango is publicly available on GitHub, both as a standalone framework and as an integration with SAGA-Bench, DZiG, and RisGraph: `https://github.com/alifahmed/graphTango.git`. **We made the findings of this work publicly available by putting it into the arXiv [27]. It is currently under review for IJPP, 2024 [101].**

4.2 Background on Existing Representation Formats

Figure 4.1 illustrates how various graph representation formats store vertices and edges. While these examples store only the outgoing edges, the concept is also applicable if storing incoming edges.

Compressed Sparse Row (CSR) is one of the most commonly used formats for *static* graphs [102, 103, 104, 105]. As shown in Figure 4.1(b), CSR organizes data in an *edge array* and an *index array*. Edges are stored in the edge array in ascending order - all edges of vertex v_i appear before any edge of v_{i+1}. The index array stores the position of the first edge of every vertex. CSR is widely used for static graphs because it provides a compact representation, increasing spatial locality while traversing the graph. However,

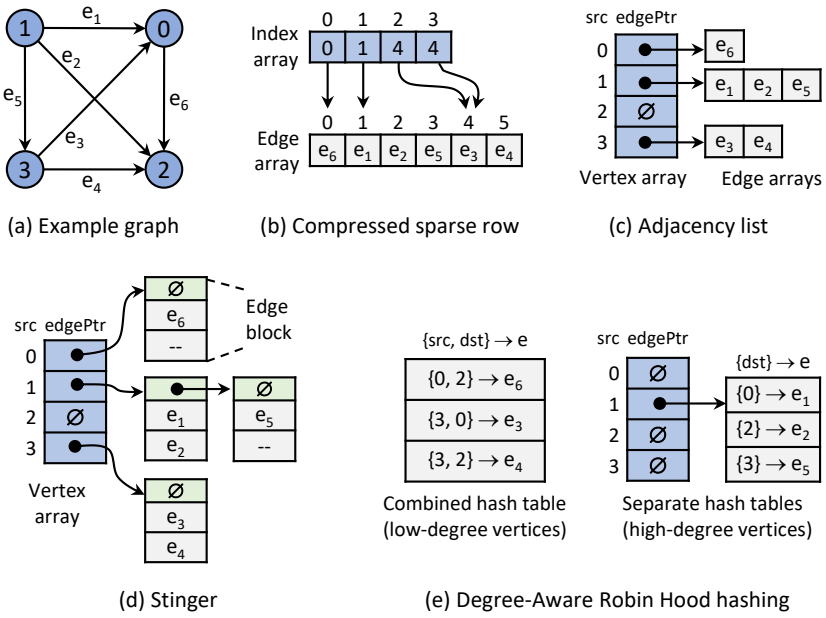


Figure 4.1: Example of different graph representation formats. Here, each edge e is an $\{dst, prop\}$ tuple.

inserting or deleting an edge requires reconstructing both the edge array and the index array, making CSR unsuitable for dynamic graphs.

Adjacency List stores the edges of every vertex in separate arrays (Figure 4.1(c)). A *vertex array* stores the pointers to these edge arrays. **These edge arrays are assumed to be memory-contiguous (like *std::vector*), rather than a linked list of edges.** This important distinction is used throughout the rest of the dissertation. As each edge array can grow/shrink independently, insertion and deletion operations only modifies the edge array of the corresponding vertex. This property makes adjacency lists a common choice for dynamic graph frameworks [40, 98]. Another advantage of adjacency lists is that the edge traversal during the analytics phase has a sequential access pattern, leading to excellent analytics throughput for vertex-centric algorithms. The downside of adjacency lists is that the edges are not stored in any particular order within an edge array. Therefore, finding an edge requires a linear search through the corresponding edge array, leading to poor update throughput on high-degree vertices.

In adjacency-list-based approaches, parallel updates on multiple vertices are realized in two ways. The first scheme is the shared style multithreading (referred as ***AdListShared***), where the vertex array additionally contains a lock for every vertex. Any thread can process updates on any vertex by acquiring the corresponding lock first. This approach provides fine-grained parallelism. However, if most updates are targeted towards the *same* vertex, it can cause lock contention and is often the case for heavy-tailed graphs. The alternative scheme groups source vertices into chunks and assign each chunk to a fixed thread (referred as ***AdListChunked***). Chunked style multithreading is lock-free. However, it is prone to workload imbalance if the chunks have a high disparity in the number of edges they contain.

Stinger [95] is an adjacency-list-based representation format. As illustrated in Figure 4.1(d), Stinger stores the edges as linked lists of *edge blocks*. Each edge block can accommodate a fixed number of edges (default is 16). Parallelism in Stinger is achieved by acquiring locks on the edge blocks. The capacity of the edge blocks presents a trade-off between performance and storage requirements. Using smaller capacity edge blocks increase parallelism but makes graph traversal inefficient by increasing the amount of pointer-chasing accesses. On the other hand, larger blocks lead to many unused slots for low-degree vertices. Besides, like adjacency lists, Stinger also suffers from linear lookups on high-degree vertices, stagnating the update throughput.

Degree-Aware Robin Hood Hashing (DegAwareRHH) [97] is a hash-based format. As shown in Figure 4.1(e), DegAwareRHH maintains two types of hash tables based on the vertex degree. Edges corresponding to low-degree vertices are stored in a combined hash table to improve data locality. On the other hand, each high-degree vertex maintains its own hash table. Both of these hash tables use Robin Hood hashing [106], which minimizes probing distance. For parallelism, DegAwareRHH leverages chunked-style multithreading similar to AdListChunked. The constant time lookup enabled by the hash tables makes DegAwareRHH suitable for the update phases on heavy-tailed graphs. However, the sparse storage of edges in the hash table makes DegAwareRHH's edge traversal inefficient, negatively impacting the analytics throughput.

4.3 GraphTango Data Structure

Figure 4.2 gives an overview of the GraphTango data structure[4]. GraphTango organizes the vertex data in two arrays: one for storing the vertex properties ($vProp$) and the other for storing neighborhood metadata of the vertex ($edgeMeta$). These arrays are indexed using vertex id. Neighbors of each vertex are stored as an $e_x = \{dst, [prop]\}$ tuple, where $e_x.dst$ is the destination vertex id, and $e_x.prop$ is an optional edge property (e.g., the weight of the edge).

The $edgeMeta$ array is aligned to a page boundary[5], and each element of the array is of cache line size. Therefore, accessing any field of $edgeMeta[i]$ will bring the rest of the fields into the cache. The deg field holds the current degree of the corresponding vertex. Depending on the degree, a vertex will fall into one of the following three categories:

Type1 Vertex: These are low-degree vertices with $deg \leq TH_0$. As illustrated in Figure 4.2(b), we store the edges directly with the metadata for Type1 vertices. The threshold TH_0 denotes the number of edges

[4]The description assumes storing only outgoing edges for clarity. In our implementation, we stored both incoming and outgoing edges for directed graphs.

[5]To clarify, only the $edgeMeta$ array itself is page boundary aligned, not the edge arrays or hash tables it may point to.

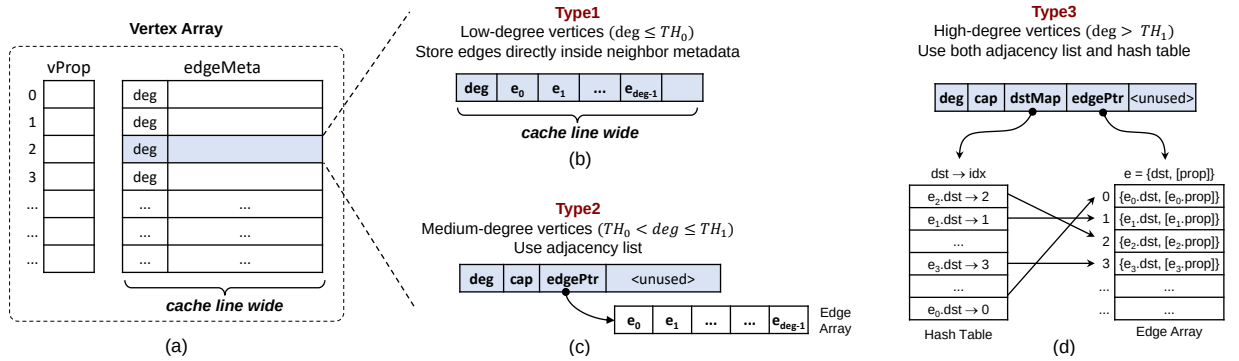


Figure 4.2: Proposed hybrid representation format of GraphTango.

that can fit inside the metadata and is defined as:

$$TH_0 = \left\lfloor \frac{CACHE\_LINE\_SIZE - sizeof(deg)}{sizeof(e)} \right\rfloor$$

For example, $TH_0 = 7$ for a typical cache line size of 64 bytes and edges of 8 bytes. The advantage of storing edges with metadata is that all edges are brought into the cache as soon as we access the vertex during the update or analytics phase. When searching for a specific edge, we need to do a linear search. However, the search is extremely fast, as all accesses will be cache hits.

Type2 Vertex: These are medium-degree vertices with $TH_0 < deg \leq TH_1$, where TH_1 is a user-configurable threshold. Edges for this type of vertices are stored in adjacency lists, as shown in Figure 4.2(c). To support adjacency lists, $edgeMeta$ additionally maintains the current capacity (cap) and a pointer to its edge array ($edgePtr$).

Like Type1 vertices, Type2 also requires a linear search when looking for a specific edge. As the linear search on the edge array is prefetcher-friendly and has good spatial locality, it offers better performance than hash-based search up to a certain point (i.e., tuned using the TH_1 threshold). However, the linear nature of the search becomes a performance bottleneck for higher-degree vertices. Hash-based search is preferable in such cases, as explained below.

Type3 Vertex: These are high-degree vertices with $deg > TH_1$. Figure 4.2(d) illustrates this scenario. Here, we maintain *both an adjacency list and a hash table for each Type3 vertex*. The hash table maps an edge's destination vertex id ($e_x.dst$) with its location in the corresponding adjacency list. Maintaining both hash table and adjacency list comes with the following benefits: i) The hash table enables constant-time lookups during the *update phase*. ii) The adjacency list provides fast and efficient traversal during the *analytics phase*. Prior hash-based approaches suffer from low analytics throughput due to inefficient edge

Table 4.1: Vertex type switching steps for insertion/deletions

(a) Insertions triggering type switch or capacity doubling

Direction	New capacity	Alloc new edge array	Edge copy size	Dealloc old edge array	Rehash
Type1 → Type2	nextPow2(TH_0)	✓	deg (=TH_0)	X	X
Type2 → Type2	cap * 2	✓	deg	✓	X
Type2 → Type3	cap * 2	✓	deg (=TH_1)	✓	✓
Type3 → Type3	cap * 2	✓	deg	✓	✓

(b) Deletions triggering type switch or capacity halving

Direction	New capacity	Alloc new edge array	Edge copy size	Dealloc old edge array	Rehash
Type3 → Type3	cap / 2	✓	deg	✓	✓
Type3 → Type2	cap / 2	✓	deg (=TH_1)	✓	X
Type2 → Type2	cap / 2	✓	deg	✓	X
Type2 → Type1	TH_0	X	deg (=TH_0)	✓	X

traversal [40]. GraphTango is free of this issue because it uses only the adjacency lists for edge traversal and does not require accessing the hash tables during the entirety of the analytics phase.

4.4 GraphTango Basic Operations

4.4.1 Edge Insertion

The edge insertion procedure is as follows: (i) Retrieve the edge metadata - $edgeMeta[srcId]$. (ii) If the current *deg* reaches the current capacity , we double the capacity. The exact steps for capacity doubling will depend upon the current and new type, as demonstrated in Table 4.1(a). In general, capacity doubling involves allocating memory for the larger edge array, copying current edges to the new edge array, and freeing the old array. For Type3, the hash table is also rehashed. The amortized cost of capacity doubling is O(1) [107]. (iii) Search for a duplicate edge using *dst*. As mentioned earlier, for Type1 and Type2, it will involve doing a linear search, and for Type3, the search will be performed using the hash table. (iv-A) If the edge is found, update the property and return. (iv-B) If the edge is *not* found, add the edge at the end of the edge array and increment *deg*. For Type3, we also create an entry in the hash table pointing to the location.

4.4.2 Edge Deletion

The edge deletion procedure is as follows: (i) Retrieve the edge metadata - $edgeMeta[srcId]$. (ii) Search for existing edge using *dst*. (iii-A) If the edge is *not* found, return. (iii-B) If the edge is found, delete the entry from the edge array and hash table (for Type3) and decrement *deg*. We do a ***compaction step*** here to fill the gap. It involves moving the last entry of the edge array to the deleted entry's position and updating the corresponding hash table record. The compaction step is simple and is of constant time complexity. (iv) If the *deg* becomes 1/4th of the capacity, we halve the capacity. The steps for capacity halving is given in Table 4.1(b). Similar to the capacity doubling during insertion, the amortized cost of capacity halving is also O(1) [107].

4.4.3 Edge Traversal

As we store the edges in consecutive memory for all three vertex types[6], the edge traversal API simply returns a cursor (i.e., position of the iterator) for indexing to the: i) $edgeMeta[vid]$ for Type1 vertices, or ii) $edgeMeta[vid].edgePtr$ for Type2/Type3 vertices. GraphTango's traversal mechanism is essentially the same as an adjacency list for Type2/Type3 vertices. As for Type1, GraphTango has a better access pattern as it requires one less indirection.

4.5 Optimizing GraphTango

4.5.1 Cache-Friendly Hashing Scheme

The hash table used by the Type3 vertices can be realized in several ways. The most convenient approach is to use *std::unordered_map*. Unfortunately, this approach is not ideal for our purpose because the C++ standard [108] effectively limits the collision resolution of *std::unordered_map* to separate chaining[7]. With separate chaining, the hash table is constructed as an array of buckets. Each bucket points to a linked list of colliding elements (i.e., keys that hashed to the same bucket). The issue with separate chaining is that it involves multiple random accesses - one to access the bucket and one or more for traversing the linked list. Each of these random accesses is a potential cache miss if the hash table does not fit into the cache. An alternative to separate chaining is open addressing, where all elements are stored in the hash table itself, eliminating the need for linked lists traversals. Prior hash-based graph representation formats [97, 96] leveraged open-addressing-based Robin Hood hashing [106] that minimizes probing distance. For

[6]Even after deletions, our compaction step ensures that all valid edges of a vertex are stored in consecutive memory.

[7]This constraint is a side effect of mandating *pointer stability*, which means that an iterator must remain valid upon inserting or deleting elements.

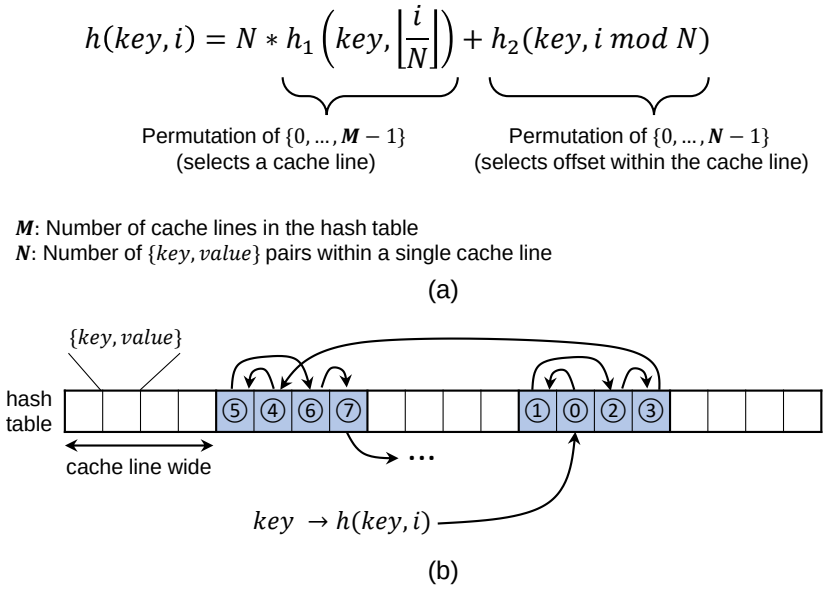


Figure 4.3: Proposed hashing scheme. (a) Hash function to determine the index to the hash table. (b) An example probing sequence for $M = 5$ and $N = 4$.

GraphTango, we designed a more cache-friendly open-addressing-based hash table that minimizes the number of cache line accesses, making it especially suitable for real-world graph workloads where the hash table is unlikely to fit into the cache.

The key idea of our hashing scheme is to limit the probes within a single cache line until it is fully searched, before moving onto a different cache line. Figure 4.3 illustrates this hashing scheme. The hash table itself is composed of an array of $\{key, value\}$ pairs. The index of the i-th probe to the hash table is given by the following hash function:

$$h(key, i) = N \cdot h_1\left(key, \left\lfloor \frac{i}{N} \right\rfloor\right) + h_2(key, i \ mod \ N)$$

Here, N is the number of $\{key, value\}$ pairs that can fit within a single cache line. The purpose of $h_1\left(key, \lfloor \frac{i}{N} \rfloor\right)$ is to select a cache line for probing and returns the base index of that selected cache line. Note that the $\lfloor \frac{i}{N} \rfloor$ parameter remains the same for every N consecutive probes, thereby selecting the same cache line. As $h_1()$ should eventually explore all cache lines in the hash table, it must be a permutation of $\{0, 1, ..., M-1\}$, where M is the number of cache lines in the hash table. On the other hand, $h_2(key, i \ mod \ N)$ determines the offset within the cache line and must be a permutation of $\{0, 1, ..., N-1\}$. Any hash function conforming to this permutation requirement can be used to implement $h_1()$ and $h_2()$. In GraphTango, we used double hashing for $h_1()$ to avoid primary/secondary clustering. $h_2()$ uses linear probing to make hash computation simpler. Our hash function is very cheap to compute, with the reference implementation having

two multiplications and eight other simple arithmetic/logical instructions. This is because we ensure that both N and M are powers-of-two, converting expensive modulus and division operations to simple shifts. Further optimization is possible by leveraging SIMD instructions to do a parallel comparison on all entries mapped to the same cache line. However, as discussed later, GraphTango demonstrates short probing distance, making iterative comparison just as performant. **Interested readers can find the implementation details of these hash functions in the Appendix 4.7.**

Insertions and deletions to the proposed hash table are similar to other open-addressing-based hash tables. Each location of the hash table can contain either: i) a valid $\{key, value\}$ pair, or ii) an *empty* marker, or iii) a *deleted* marker (i.e., tombstone). We used two reserved values as the empty and deleted marker instead of using dedicated tag storage. During both insertion and deletion, the table is probed (using the hash function) until the *key* or an empty marker is found. If the *key* is found: i) For insertion, the corresponding *value* is updated. ii) For deletion, the entry is marked as deleted. Instead of *key*, if an empty marker is found: i) No action is required for deletion. ii) For insertion, the $\{key, value\}$ pair is inserted to the location of the first encountered delete marker, or to the current location if no delete marker was encountered.

When using with GraphTango, the hash tables' initial capacity is set to twice the capacity of the corresponding adjacency lists. Upon inserting/deleting edges, both the hash tables and the adjacency lists can grow/shrink in size (see Section 4.4), but the capacity ratio always remains 2. This property sets the maximum load factor (α) of the hash table to 0.5. Assuming uniform hashing[8], the theoretical average probing distance is: i) $\frac{1}{1-\alpha} = 2$ for an unsuccessful search. This is often the case for edge insertions in the absence of duplicates. ii) $\frac{1}{\alpha} ln \frac{1}{1-\alpha} = 1.39$ for a successful search (e.g., deleting existing edges). In GraphTango, we can fit eight $\{key, value\}$ pairs within a single cache line (i.e., $N = 8$). As a result, the hash table needs to access only one cache line as long as the probing distance remains ≤ 8, which provides a large slack over the theoretical average probing distances. We empirically observed the same trend with our graph datasets, where over 99.2% of the insertions had a probing distance ≤ 8. Therefore, almost all edge insertion operations for *Type3* vertices require only three cache line accesses: i) one for retrieving $edgeMeta[srcId]$ metadata that contains hash table and adjacency list pointers, ii) one for searching the hash table, and iii) one for indexing to the adjacency list.

4.5.2 Memory Allocation Scheme

As discussed in Section 4.4, GraphTango requires frequent growing/shrinking of adjacency lists and hash tables. Calling $malloc()/free()$ in every such instance can cause high runtime overhead and memory fragmentation.

[8]Double hashing can demonstrate performance very close to the ideal scenario of uniform hashing [107, 109].

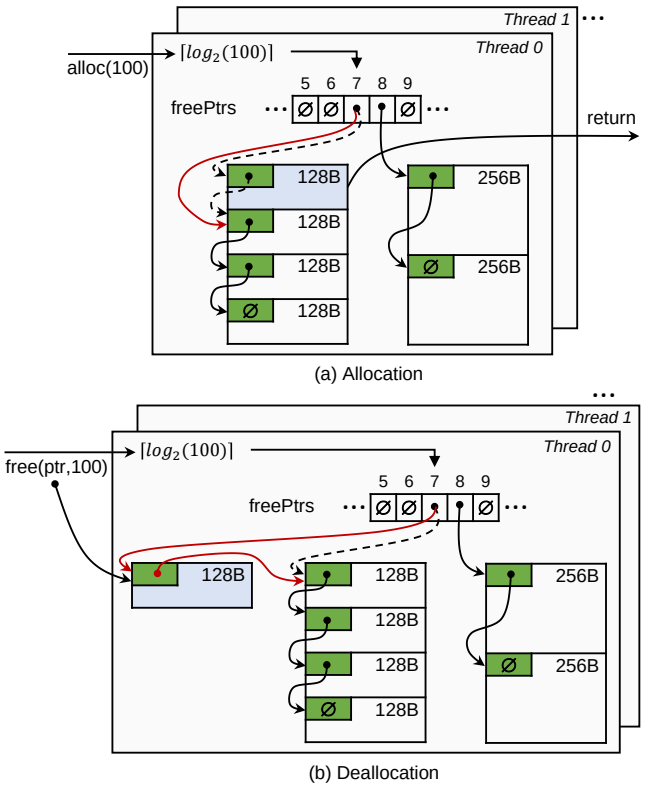


Figure 4.4: Allocation and deallocation on the memory pool. Deleted pointers are shown by dashed lines and the modified pointers by red lines.

We avoid this issue by designing a fast thread-local lock-free memory pool that supports O(1) allocation and deallocation.

Figure 4.4 illustrates the data structure of the memory pool. This memory pool allocates chunks in power-of-two sizes. Individual linked lists of available chunks are maintained for each valid size. The heads of the linked lists are stored in the $freePtrs$ array. The first 8 bytes of each chunk (highlighted green) hold the pointer to the next free chunk of the same size. This way, no extra storage beside the $freePtrs$ array is required to hold the pointers. However, it limits the minimum chunk size to 8 bytes in a 64-bit machine.

Allocation steps are shown in Figure 4.4(a). For an allocation request of sz bytes, the pool will return a chunk of size $newSz = 2^k$, where $k = \lceil log_2(sz) \rceil$ (i.e., nearest power of two that is $\geq sz$). The allocation proceeds as follows: i) Find the first available free chunk of $newSz$. This is simply given by $ret = freePtrs[k]$. ii) If ret is not a null pointer, then it points to a free chunk. In this case, we update $freePtrs[k]$ to point to the next free chunk, and return ret. iii) If ret is a null pointer, this indicates no chunk of the requested size is available. In this case, a large memory block is allocated (of size $max(4MB, newSz)$ and aligned to the page boundary) and then split into a linked list of $newSz$ byte chunks. $freePtrs[k]$ is set to point to the first chunk. At this point, we have free chunks of $newSz$. Therefore, repeating step (ii) will complete the

allocation. Note that, once allocated, the full chunk can be used to store data, including the space initially used to hold the pointer to the next chunk.

Deallocation steps are shown in Figure 4.4(b). Unlike the standard $free(ptr)$, we provide the size of the allocated chunk as an additional parameter - $free(ptr, sz)$. Using the sz parameter, we can directly index to the $freePtrs$ array and add ptr as a free chunk, as shown in Figure 4.4(b).

An advantage of the proposed memory pool is that the most recently deallocated chunk will be allocated first, thereby being more likely to reside in the cache. Furthermore, each thread maintains its own $freePtrs$ array. As a result, no lock is required when multiple threads are trying to allocate/deallocate simultaneously. A minor downside is that one thread cannot allocate free chunks from another thread's pool. We found it to be of little consequence in practice because the maximum amount of unused space per thread is O(blockSize). Also, note that the $\lceil log_2(sz) \rceil$ calculation used to index $freePtrs$ is very cheap to perform. It only requires *count leading zero* (clz) and shift instructions.

4.5.3 Parallelization

As discussed in Section 4.2, both shared and chucked style multithreading approaches have shortcomings. In shared style multithreading, the lock granularity is a single node. Therefore, every update requires attaining a lock and contributes to the increased overhead. Besides, if many updates within a batch involves the same node, then shared style multithreading suffers severe lock contention as many threads try to lock that particular node. On the other hand, there is no lock needed in chunked style multithreading, but it suffers from workload imbalance because the thread to node mapping is fixed. Therefore, even if a thread is free and here are updates left to process, that thread cannot process that update if it is not mapped to that particular thread. Besides, all threads need to go over the full batch to check if the updates belong to that thread.

To solve these issues, in GraphTango, we propose a novel workload balancing technique using the concept of bucket-chaining. Figure 4.5 shows this scheme. Our approach essentially increases the lock granularity of the shared style multithreading by first creating a bucket-chain. In this first step, all the threads go over a subset of the batch, and distribute the updates within that portion into separate buckets. In this stage, each bucket set is local to the thread, so no locking is necessary. Which bucket a particular update goes is decided based on the node id. Note that the access pattern of each thread in this step is sequential through the batch, as well as sequential when inserting an update to a bucket (i.e., always inserted on the top position of the bucket).

Once all updates are assigned a bucket, we start the second phase of processing the updates. In this phase, all threads will try work stealing on the bucket-chains to process them. The bucket-chains are formed

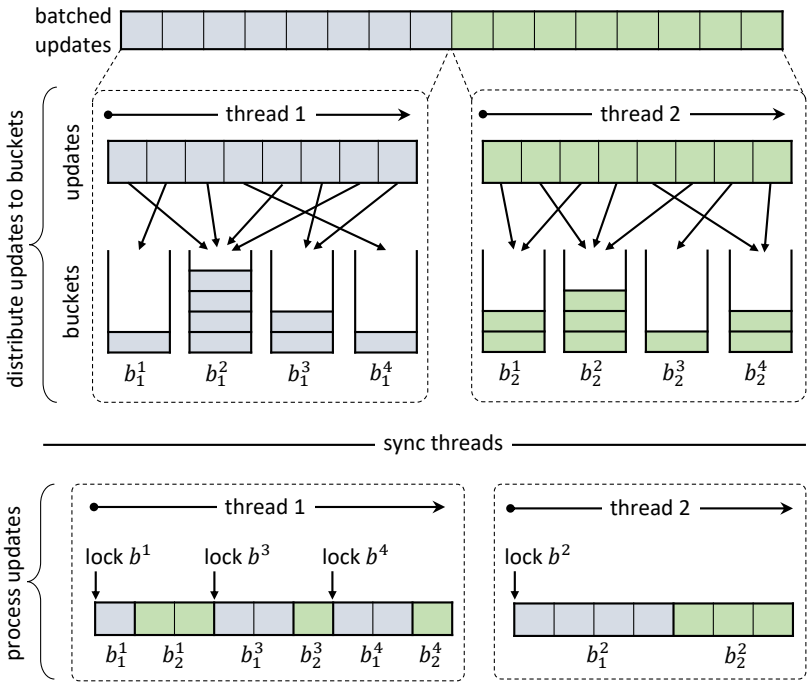


Figure 4.5: Load-balancing scheme of GraphTango. In the first stage, all threads go over a subset of the batch to fill bucket-chains. Afterwards, each worker thread locks and process the next available bucket-chain until all buckets are processed.

by linking the first bucket of first thread with first bucket of second thread and so on. In this way, the same indexed nodes will fall in the same bucket-chain, even if they were distributed by a different thread during the first step. This ensures that all updates of a nodes are processed in the same order as they appeared within the batch. Also, if a thread finished processing their current bucket-chain, it can immediately start processing other updates by acquiring the lock of the next bucket-chain. Our approach solves ***all*** the issues faced by the other two approaches and provides excellent load balancing.

4.5.4 Determining the TH_1 Threshold

Unlike the TH_0 threshold, which is fixed for a given cache line size and edge element size, the TH_1 threshold is flexible and has a moderate impact on performance and memory usage (Section 4.6.4). As mentioned before, TH_1 should be set to a value for which $O(TH_1)$ linear search through the edge array is likely to perform better than $O(1)$ hash table lookup. The following equation provides an estimate and can be used as a rule of thumb for selecting TH_1:

$$TH_1 = 2^{\lceil log_2(3 \times edgesPerCacheLine) \rceil} \tag{4.1}$$

This equation sets TH_1 to a value roughly corresponding to four cache line accesses for Type2 vertices. This is slightly above the three cache line accesses of Type3 vertices, as Type2 vertices have favorable sequential access patterns and do not incur hash calculation overheads. As an alternative, we provide a microbenchmark program (*graph dataset agnostic*) with GraphTango that empirically finds a suitable TH_1 threshold.

4.6 Evaluation

4.6.1 Experimental Setup

A) Platform: The experiments are conducted on an AMD Ryzen 3900x @ 3.8GHz machine with 12 physical cores, 64MB of LLC, and 32GB of DDR4 DRAM. Hyper-threading and turbo-boost were disabled for better reproducibility. All experiments are performed with 12 cores.

B) Implementation: We evaluated GraphTango by integrating it with the SAGA-Bench [40] benchmarking framework. SAGA-Bench comes with four representation formats - AdListShared, AdListChunked, Stinger [95], and DegAwareRHH [97]. Details of these formats can be found in Section 4.2. SAGA-Bench integration facilitates fair comparison, because all approaches must use the exact same algorithm implementations through a common API . The source code is compiled with gcc-9.3.0 and -03 flag.

Both vertex id and edge property are considered to be of 64-bits size. Therefore, GraphTango has $TH_0 = 7$ for unweighted graphs and $TH_0 = 3$ for weighted graphs. TH_1 is set to 32 following the tuning carried out in Section 4.6.4. This TH_1 value also matches the value provided by Equation 4.1.

C) Profiling Methodology: Graph datasets are first randomly shuffled to break any existing ordering of edges. This is done to reflect the realistic scenario where edge updates are unlikely to occur in any pre-defined order. The shuffled dataset is inserted in batches of 1M edges until the full graph is built and then deleted[9] in batches of 1M edges until no edges are left to delete. This batch size is similar to prior works [96, 40]. **Analytics is performed on the graph after every batch of insertions and deletions. Reported throughputs are the geometric mean of the per-batch throughputs.** GraphTango dynamically switches between vertex types as edges are inserted/deleted. As discussed in Section 4.4, this switching may involve memory allocation/deallocation, copying, or rehashing. **This switching overhead is included in the reported results.**

D) Datasets: We have used four real-world datasets in our experiments: Orkut, LiveJournal, Wiki-topcats (referred as Wiki), and Wiki-talk (referred as Talk). Orkut and LiveJournal are online social media

[9]The vanilla SAGA-Bench does not support edge deletions. We added deletion support for all representation formats by closely following the corresponding papers or from their source code if available.

Table 4.2: Evaluated Datasets

Dataset	Vertices (million)	Edges (million)	Max degree[1]		Vertex mapping[2]		
			in	out	Type1	Type2	Type3
Orkut	3.0	117.2	329	329	27.2%	38.6%	34.2%
LiveJournal	4.8	69.0	237	332	63.0%	26.0%	11.0%
Wiki	1.8	28.5	8,504	154	57.8%	33.9%	8.3%
Talk	2.4	5.0	665	20,088	98.3%	1.2%	0.5%

[1] Per-batch maximum degree with batch size of 1 million edges
[2] For $TH_0 = 7$ and $TH_1 = 32$

networks, Wiki is a dataset of Wikipedia hyperlinks, and Talk is the Wikipedia communications network. These datasets are part of the SNAP dataset collection [110]. All these datasets are directed except for Orkut. Properties of these datasets are given in Table 6.2. Orkut and LiveJournal have a much lower per-batch maximum degree compared to Wiki and Talk. Consequently, **Orkut and LiveJournal are characterized as short-tailed graphs while Wiki and Talk are heavy-tailed graphs.**

E) Algorithms: We used four algorithms in our experiments: i) Breath-First Search (BFS), ii) Page Rank (PR), iii) Single-Source Shortest Path (SSSP), and iv) Connected Components (CC). Vertex centric incremental compute model is used for these algorithms, where the computation is constrained within the region affected by the update phase instead of the whole graph. The implementations of these algorithms are directly taken from SAGA-Bench without any modification.

4.6.2 Analytics and Update Performance

A) Analytics Throughput: Figure 4.6 shows the analytics throughput of the representation formats. As mentioned in Section 4.6.1(C), the analytics phase is conducted multiple times as we gradually build the graph. Reported values are the geometric mean of per-batch throughputs. GraphTango outperforms other approaches in every dataset and algorithm combinations. Compared to the *next best* approach (i.e., AdListShared for BFS, SSSP, CC and AdListChunked for PR), GraphTango provides an avg (max) speedup of 1.1x (1.6x). As all these approaches are using the exact same algorithm implementation, their relative performance is primarily determined by their edge traversal efficiency. Adjacency-list-based approaches perform well in this regard, because their edge traversal consists of mostly sequential accesses. GraphTango also uses adjacency lists for medium- and high-degree vertices (Type2 and Type3). For low-degree vertices (Type1), GraphTango has a better access pattern, as it requires one less indirection (i.e., does not need pointer chasing to find the corresponding edge array), thereby offering higher throughput. Stinger, despite using coarse-grained adjacency lists, suffers due to additional pointer chasing between edge blocks. Overall, GraphTango provides

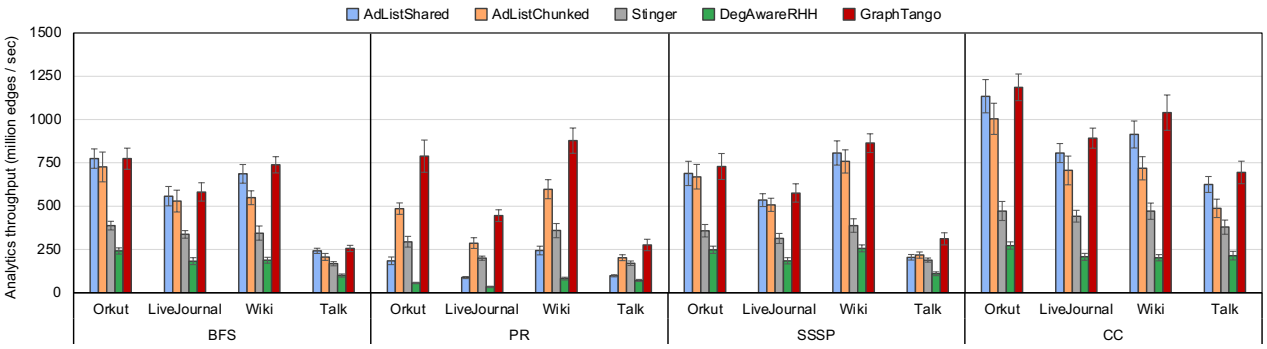


Figure 4.6: Comparison of the analytics throughputs. *Higher is better.*

Table 4.3: Average Memory Usage (Bytes Per Edge)

Dataset	AdList-Shared	AdList-Chunked	Stinger	DegAware RHH	GraphTango
Orkut	13.3	12.0	32.7	43.8	33.6
LiveJournal	16.3	12.9	48.9	57.0	34.6
Wiki	15.7	12.7	44.1	62.9	34.4
Talk	44.8	21.9	230.2	74.6	116.9

an avg (max) speedup of 1.8x (5.1x) over AdListShared, 1.3x (1.6x) over AdListChunked, 2.0x (2.7x) over Stinger, and 5.2x (14.0x) over DegAwareRHH.

B) Update Throughput: Figure 4.7 shows the update (edge insertion and deletion) throughput. Note that the updates are interleaved with analytics phases (see Section 4.6.1.C). The algorithm choice of the analytics phase has little impact on the update throughput, and the reported values are the average across the four algorithms. Here, GraphTango outperforms other approaches by a large margin. Adjacency-list-based approaches perform well on short-tailed graphs. On these graphs, GraphTango provides an avg (max) speedup of 2.5x (2.7x) over the next best approach AdListShared. On the other hand, hash-based DegAwareRHH performs best on the heavy-tailed graphs. Interestingly, AdListShared performed even worse than AdListChunked for heavy-tailed graphs. This is due to the lock contention of shared-style multithreading on AdListShared. On heavy-tailed graphs, GraphTango provides an avg (max) speedup of 6.5x (6.6x) over the next best approach DegAwareRHH. Notably, other approaches are suitable for either short- or heavy-tailed graphs. GraphTango's hybrid nature makes it consistently the best-performing irrespective of the graph's degree distribution.

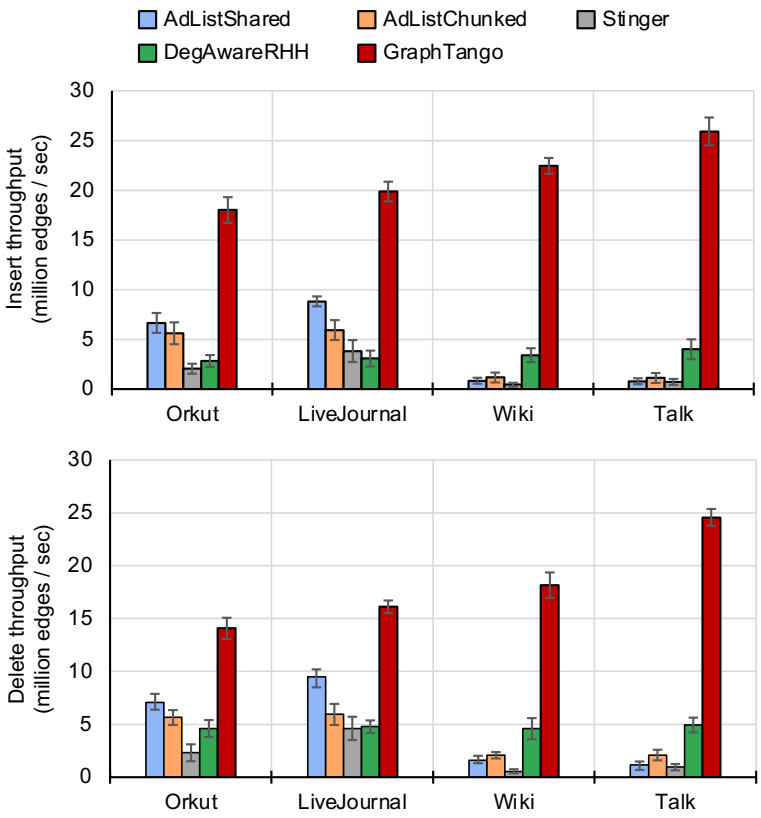


Figure 4.7: Comparison of update throughputs. *Higher is better.*

4.6.3 Memory Usage

Table 4.3 shows the average memory usage per edge. The AdListShared and AdListChunked are most efficient in terms of memory usage. Compared to AdListChunked - Stinger, DegAwareRHH, and GraphTango require 5.1x, 4.1x, and 3.4x more memory on average, respectively. For DegAwareRHH, the high memory usage is caused by: i) Sparse storage of edges in hash tables (to reduce collision), and ii) Robin Hood hashing mechanism that requires storing the probe distance for each entry. On the other hand, Stinger and GraphTango have a relatively high initial capacity (16 for Stinger and TH_0 for GraphTango) that remains mostly unused for low-degree vertices. This scenario is especially noticeable for the Talk dataset, where more than 96% of the vertices have a degree ≤ 3, leading to high memory usage. Although GraphTango has higher memory usage compared to the simple adjacency-list-based approaches, the update throughput benefit is significant, especially on heavy-tailed graphs (19.3x to 32.8x speedup on Wiki and Talk datasets). If needed, one way to reduce memory usage of GraphTango is to increase the TH_1 threshold, as discussed in Section 4.6.4. Compared to Stinger and DegAwareRHH, GraphTango requires less memory as well as provides much higher performance.

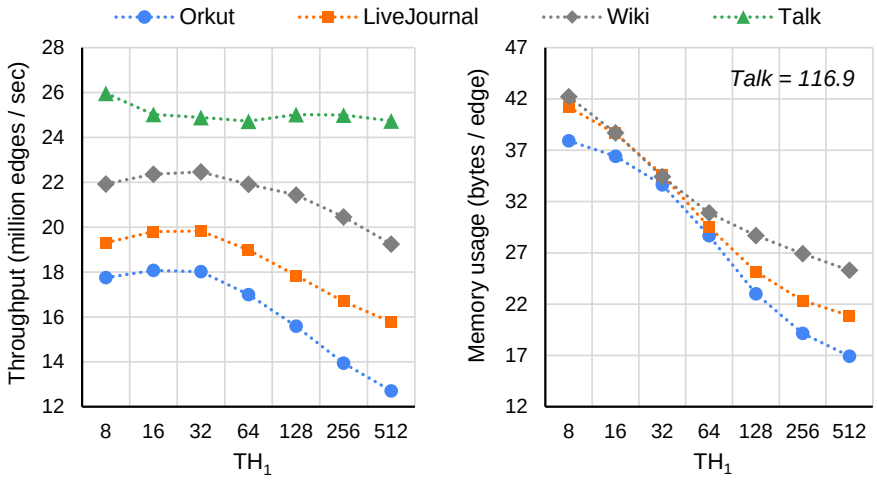


Figure 4.8: Impact of TH_1 threshold on update throughput and memory usage.

4.6.4 Impact of TH_1 Threshold

Figure 4.8 shows the impact of the TH_1 threshold on update throughput and memory usage. Analytics throughput is not shown because the choice of TH_1 does not impact the analytics performance. TH_1 of 16 and 32 provides the best throughput on three out of the four datasets. We used TH_1 of 32 in all other experiments, as it has lower memory usage.

The TH_1 threshold controls the ratio between Type2 and Type3 vertices. Increasing TH_1 maps more higher-degree vertices to Type2 instead of Type3. As Type2 vertices requires linear search during updates, it eventually becomes a performance bottleneck for $TH_1 > 32$. On the other hand, Type2 vertices do not require maintaining a hash table and thus require less memory than Type3. As a result, increasing TH_1 reduces the memory usage. On average, increasing TH_1 from 8 to 512 reduces the memory usage by 1.9x.

The Talk dataset is an outlier showing negligible variation with TH_1. This is because, for the Talk dataset, 98.3% of the vertices are mapped to Type1 (refer to Table 6.2), leaving only 1.7% of the vertices that can be affected by changing TH_1.

4.6.5 Impact of Optimizations

The purpose of this section is to isolate the contribution of the proposed hybrid format as well as the memory allocation and hashing scheme optimizations. Table 4.4 shows our findings. In this table, the STail and HTail columns show the normalized update throughput over the *baseline* configuration for short-tailed and heavy-tailed graphs, respectively. We show only the update throughput because the memory pool and hashing optimizations have a negligible impact on the analytics throughput.

Table 4.4: Impact of Optimizations on the Update Throughput (Baseline is the proposed hybrid format without any optimizations applied)

Configuration	Format	Allocation Scheme	Hashing Scheme	Speedup	
				STail	HTail
baseline	Hybrid	malloc[2]	std_map[3]	1.00	1.00
next best[1]	-	-	-	0.76	0.29
opt pool	Hybrid	proposed	std_map[3]	1.12	1.14
opt hash	Hybrid	malloc[2]	proposed	1.71	1.70
GT_Tessil	Hybrid	proposed	Tessil[4]	1.40	1.60
GT_RHH	Hybrid	proposed	RHH[5]	1.35	1.45
GT_Abseil	Hybrid	proposed	Abseil[6]	1.34	1.43
GraphTango	***Hybrid***	***proposed***	***proposed***	**1.89**	**1.79**
DegAwareRHH	DegAware	malloc[2]	RHH	0.29	0.29
DegAwareCFH	DegAware	malloc[2]	proposed	0.40	0.38

[1] AdListShared for STail and DegAwareRHH for HTail.
[2] glibc version 2.31.
[3] std::unordered_map with libstdc++ version 6.0.28.
[4] tsl::robin_map from [111], version 1.0.1.
[5] Robin Hood hashing implementation from [112], version 3.11.5.
[6] Google's Abseil flat_hash_map version LTS 20211102 [113].

The *baseline* configuration implements our proposed hybrid format (i.e., Type1, Type2, and Type3 mapping of vertices) but uses sub-optimal malloc for memory allocation and std::unordered_map for hashing. We can observe that using only the hybrid format is sufficient to provide a better performance than the *next best* approach (AdListShared for STail and DegAwareRHH for HTail). Compared to the next best approach, the *baseline* offers 1.3x (3.4x) higher speedup for STail (HTail) graphs. The *opt pool* configuration shows the benefit of the proposed memory allocation scheme. On average, the proposed pool provides 1.13x better performance over the *baseline*. On the other hand, the *opt hash* configuration shows the benefit of the proposed cache-friendly hashing scheme, offering 1.71x speedup over std::unordered_map. With both optimizations enabled, GraphTango provides an average speedup of 1.84x over the *baseline*.

A valid concern at this point is whether we can combine other hashing schemes with our hybrid format to get even better performance. To answer this question, we tried three other open-addressing-based hash-table implementations: i) *GT_Tessil*: The Robin Hood hashing variation of Tessil (tsl::robin_map) [111]. This is the fastest hash table implementation according to the benchmark results published in [114]. ii) *GT_RHH*: Another fast implementation of Robin Hood hashing from [112, 114]. Although it is slightly slower than Tessil, it consumes significantly less memory. iii) *GT_Abseil*: Google's Abseil flat_hash_map [113, 115]. The max load factors of these approaches are set equal to ours (= 0.5) for a fair comparison. On STail (HTail) graphs, the proposed hashing scheme provides 1.35x (1.12x) speedup over *GT_Tessil*, 1.4x (1.23x) speedup

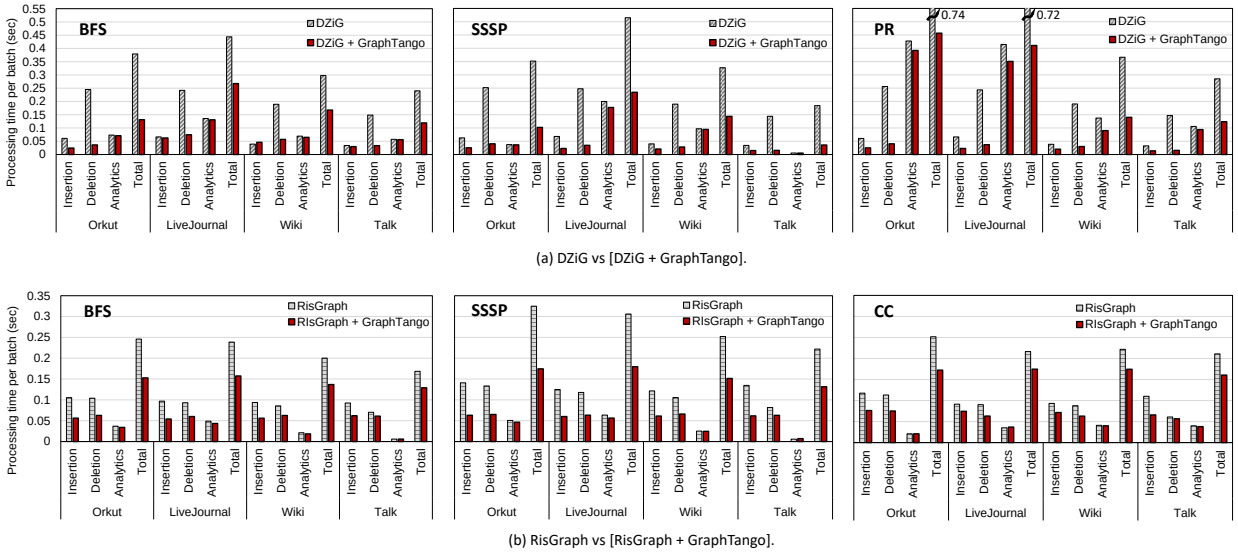


Figure 4.9: Batch processing time breakdown of DZiG and RisGraph integration. *Lower is better.*

over *GT_RHH* and 1.41x (1.25x) speedup over *GT_Abseil*. Because our hashing scheme tries to minimize cache line access, it is especially suitable for graph workloads where the hash table is unlikely to reside in the cache.

Finally, we evaluate whether DegAwareRHH can leverage our hashing scheme to outperform GraphTango. *DegAwareCFH* denotes this configuration. *DegAwareCFH* provides 1.37x (1.31x) better throughput over the vanilla DegAwareRHH. However, GraphTango still outperforms it by 4.7x for both STail and HTail graphs.

4.6.6 Integration with DZiG and RisGraph

This section demonstrates that full-fledged graph processing frameworks can leverage the GraphTango format to improve their performance further. We selected two state-of-the-art graph processing frameworks DZiG [99] and RisGraph [100] for this purpose. We modified their publicly available source code [116, 117] and replaced their storage format with GraphTango. We run the datasets on BFS, PR, and SSSP for DZiG. CC is omitted because its implementation is unavailable in the framework's repository. For the same reason, PR is omitted in case of RisGraph.

Figure 4.9(a) shows the comparison results between DZiG and DZiG+GraphTango. DZiG internally uses adjacency list as graph storage. For this reason, analytics time for DZiG and DZiG+GraphTango is similar in most cases. Interestingly, the insertion time is also comparable in some cases. For example, LiveJournal and Wiki datasets for BFS. This is because the original DZiG's edge insertion does not check for duplicate edges.

Therefore, the edge insertion becomes as simple as adding an element to the end position of an array [10]. On average, GraphTango provides a 1.9x reduction in insertion time *even though it also checks for duplicate edges*. For deletion, unmodified DZiG performs 6x worse on average. We identified two reasons: i) Unlike insertions in DZiG that do not search for duplicates, delete operations require a linear search through the neighbor list, incurring higher runtime cost, and ii) DZiG performs a quicksort on the batch based on the source and destination vertex ids to distribute them among the threads. As we use fixed mapping of vertices in GraphTango, sorting costs are avoided. Overall, DZiG+GraphTango provides an average of 2.3x reduction in total batch processing time compared to the original DZiG.

Figure 4.9(b) shows the comparison between RisGraph and RisGraph+GraphTango. RisGraph uses a hybrid graph storage format that uses adjacency list for low/medium degree vertices and adjacency list along with hash table for high degree vertices. Unlike GraphTango, RisGraph does not differentiate between low and medium degree vertices and uses the same data structure for both. Furthermore, RisGraph uses Google's dense hash map and does not attempt to minimize the number of cache accesses as GraphTango does with its proposed cache-friendly hashing scheme. Due to these differences, RisGraph+GraphTango provides on average 1.5x reduction in total batch processing time compared to the vanilla RisGraph.

4.7 Details of Hash Function Implementation

Given these parameters,

M = Number of cache lines in the hash table

N = Number of $\{key, value\}$ pairs within a cache line

Our proposed hash function is of the following form:

$$h(key, i) = N \cdot h_1\left(key, \left\lfloor \frac{i}{N} \right\rfloor\right) + h_2(key, i \ mod \ N)$$

Here, $h_1()$ selects a cache line inside the hash table array, and $h_2()$ selects an offset within the cache line. Therefore, $h_1()$ must be a permutation of $\{0, 1, ..., M-1\}$ to ensure that all cache lines are eventually selected. Similarly, $h_2()$ must be a permutation of $\{0, 1, ..., N-1\}$ to explore all $\{key, value\}$ pairs within a cache line. Any $h_1()$ and $h_2()$ that meet the permutation requirement can be used. For GraphTango, we

[10]There is a flag to enable duplicate edge insertion checking. But that checking is done by sorting the batch as a pre-processing step, thereby incurring heavy overhead.

used the following:

$$h_1(k,x) = (h_3(k) + x \cdot h_4(k))\ mod\ M$$
$$h_2(k,x) = (k + x)\ mod\ N$$
$$h_3(k) = \lfloor (A \cdot k\ mod\ 2^w)/2^{w-m} \rfloor$$
$$h_4(k) = \lfloor (A \cdot k\ mod\ 2^w)/2^{w-2m} \rfloor\ or\ 1$$

Here, w is the key width in bits, A is a large constant, and $m = log_2(M)$. We use double hashing for $h_1()$ to negate primary/secondary clustering. It is computed with the help of two pairwise independent hashing functions, $h_3()$ and $h_4()$. $h_3()$ and $h_4()$ are computed with multiplicative hashing. As for $h_2()$, we used simple linear probing. Although seemingly complex, the hash can be computed cheaply as we ensure both N and M are powers of two. The following code snippet shows how to calculate the hash value for a 32-bit key:

```
u32 h(u32 key, u32 i){
    u32 y = key * A;
    u32 h3 = y >> (32 - logM);
    u32 h4 = (y >> (32 - (logM << 1))) | 1;
    u32 h1 = (h3 + (i >> logN) * h4) & (M - 1);
    u32 h2 = (key + i) & (N - 1);
    return (h1 << logN) + h2;
}
```

Note that the code does not need any expensive division/modulus operation. When compiled on an x86_64 machine with gcc 9.3.0 and -O3 flag, it resulted in 2 multiplications and 8 other simple arithmetic/logical instructions.

4.8 Conclusions

Existing streaming graph representation formats can only support either short-tailed or heavy-tailed workloads efficiently. This work proposes GraphTango, which aims to solve this issue by adaptively switching formats based on the current degree of a vertex. We also propose a cache-efficient hashing scheme and a fast memory pool. These optimizations work in synergy with GraphTango to provide excellent update and analytics throughput regardless of the graph's degree distribution. Our evaluation on the SAGA-Bench showed that on average (maximum), GraphTango provides 4.5x (6.6x) higher insertion throughput, 3.2x (5.0x) higher deletion throughput, and 1.1x (1.6x) higher analytics throughput over the *next best* approach. Currently

GraphTango is designed with shared memory systems in mind. It would be interesting to extend this approach for distributed systems.

Chapter 5

Pulley: An Algorithm/Hardware Co-optimization for In-memory Sorting

5.1 Introduction

Sorting is widely used and appears in many big data applications and database operations, such as index creation, sort-merge joins, and user-requested output sorting. Accordingly, many studies have focused on accelerating sorting using FPGA and ASIC [1, 2]. Two factors limit the performance of these accelerators. First, these accelerators employ merge-based sorting algorithms, where all the data should eventually pass through a single merging point, causing a bottleneck. Second, these approaches impose significant data movement overhead because they move data between memory and processing units in several passes (for datasets too large for the accelerators' SRAM buffers), where each pass performs only a few operations per loaded datum from memory. Since data movement in current systems can be orders of magnitude costlier than arithmetic and logic operations, the data movement overhead dominates the total execution time and energy consumption. Processing-in-memory (PIM) architectures alleviate this data movement overhead by processing data inside the memory.

Recent PIM-based accelerators also provide high parallelism by placing one or several ALUs per memory segment (e.g., one ALU per memory subarray or a few ALUs per bank) inside memory layers. We refer to these PIM-based accelerators as in-memory-layer accelerators. New high-bandwidth interconnects (e.g.,

```
1  //The intermediate array
2  Struct bucket{
3        int hist;
4        int prefix;
5        int index;
6  };
7  //nSPU=number of SPUs
8  //nBkt=number of buckets
9  bucket intmdt[nSPU][nBkt];
```

(a)

```
1  //Step 1: Local Histogram
2  Input: keys[],mask,nShift,
3  SPU_ID //SPU's ID
4  nSPU//number of SPUs
5  Output: intmdt[][]=0
6  //--------------------------------
7  If (SPU_ID<nSPUs){
8    for(j=0;j< keys.length;j++){
9      bkt=(keys[i]& mask)>>nShift;
10     intmdt[SPU_ID][bkt].hist++;}}
```

(b)

```
1  //Step 2: Prefix-sum
2  Input: intmdt[][],t=0
3  APU_ID//Aggregator PU's ID
4  Output: intmdt[][]
5  --------------------------------
6  If (PU_ID== APU_ID){
7    for(bkt=0;bkt<nBkt;bkt++){
8      //for each PU
9      for(i=1;i<nSPUs;i++){
10       intmdt.prefix[i][bkt]=t
11       t+=intmdt.hist[i][bkt];}}}
```

(c)

```
1  //Step 3:Key placement
2  Input:intmdt[][],keys[],mask,nShift
3  Output:sortedKeys[]
4  --------------------------------
5  If (SPU_ID<nSPUs){
6     for(j=0;j< keys.length;j++){
7        bkt=(keys[j]& mask)>>nShift;
8        pos=intmdt[i][bkt].prefix+
9              intmdt[i][bkt].index;
10       intmdt[i][bkt].index++;
11       sortedKeys[pos]=key[j];}}
```

(d)

Figure 5.1: Parallel Radix sorting: (a) The intermediate array, which has one element per bucket and per subarray-level processing unit (SPU), (b) in step 1, each SPU generates a local histogram array, (c) in step 2, an aggregator processing unit (APU), outside memory layer (e.g., in the logic layer of 3D stack memories) performs a prefix-sum on all local histogram arrays, and (d) in step 3, each SPU determines the position (pos) of each key by deriving the bucket number and adding the prefix value of the bucket to the current index of the bucket (line 8-9).

NVLink [118]) provide all-to-all connections between multiple devices and increase the connectivity of multi-device PIMs, providing even higher capacity and parallelism. Thus, it is crucial to employ scalable algorithms such as radix sorting that can exploit the high parallelism and high connectivity of multi-device in-memory-layer PIM-based accelerators.

In this work, my contributions are the following: i) Developing a cycle-accurate simulation framework for Pulley and conducting the experiments, ii) Identifying the capacity issue of storing the histogram in local subarrays for large radix, and brainstorm with the primary author to devise a solution, and iii) Identifying the issue of maintaining stable sort during the local-sorting phase.

Radix sorting splits the k bits of keys into smaller d-bit digits, and sorts data in $\lceil k/d \rceil$ passes. In each pass, the algorithm partitions the keys into $radix = 2^d$ distinct buckets and places a key in a bucket in three steps. The first step is *Histogram generation*, where each processing unit generates a histogram array by counting the number of keys in each bucket. In the second step, the algorithm performs *Prefix-sum* operations across all local histogram arrays generated by all processing elements. Finally, in the third step, *Key placement*, each processing unit uses the prefix-sum results to find the address of each key in the pass's sorted output and writes the key in the correct address. This last step moves keys among memory segments, memory stacks,

and devices, introducing significant data movement overhead. Therefore, to reduce data movement overhead for sorting, we need to reduce the number of passes by employing large radixes.

However, implementing large-radix sorting in PIM is challenging for three reasons. First, large-radix sorting requires reserving a large histogram array per processing element, wasting the capacity. Second, the Histogram generation step introduces random accesses to the histogram array. Random access to a large array is very costly in PIM because memory reads the data at a row granularity, only a few Kbits. If the histogram array does not fit in one row, each random access to the histogram array may need to load a new row, imposing significant performance and energy overhead. Third, the Prefix-sum step with large radix imposes significant performance overhead because, in current in-memory-layer accelerators, the prefix-sum operation across memory segments should be performed using a core far from memory segments. The core moves all the histogram values and prefix-sum values between the memory segments and the core.

In this work, we address these challenges. To this end, we employ a baseline PIM architecture, Fulcrum [9], which has one lightweight processing unit per two subarrays. Fulcrum [9] uses a version of radix sorting that dose not calculate the length of each bucket and assumes (i) all buckets have almost the same length, and (ii) a bucket in each pass can always fit in one subarray. These assumptions are not true for sorting gigabytes of real-world data, where data is unevenly distributed among buckets and buckets surpass the capacity of one subarray. In this work, we address these inefficiencies of Fulcrum by enabling radix sorting that uses histogram and prefix-sum values for calculating the exact length of each bucket and the exact position of each key within each bucket. More importantly, we enable large radix sorting, which reduces the number of passes.

We, therefore, propose an algorithm/hardware co-optimization by which every group of processing units can cooperatively generate a shared intermediate array. In our algorithm, first, each processing unit locally sorts its keys. We optimize the local sorting by exploiting an efficient sequential mechanism for dichotomizing keys and proposing hardware support that enables filling and processing the two buckets in binary radix sorting from two different directions, eliminating the histogram generation step for the local binary radix sorting (more details in Section 5.3.2). Next, in our algorithm, each processing unit iteratively generates a small part of the histogram array (e.g., 256 elements) and then reduces this small part in the large shared intermediate array. The local sorting step enables the part-by-part histogram generation, providing two benefits: (i) reducing the size of the intermediate array per processing unit and (ii) eliminating random accesses to the shared intermediate array. Since our proposed method requires only one histogram array per group of processing elements, our proposed method also decreases the overhead of prefix-sum operations on histogram arrays.

In summary, this work makes the following contributions:

- Proposing the first in-memory-layer approach for gigabyte sorting.
- Reducing the number of required passes by enabling large-radix sorting.
- Exploiting the high parallelism of recent PIMs and all-to-all connectivity of recent interconnections.
- Evaluating the effect of our proposed method against a near-HBM FPGA-based approach [1] and an in-logic-layer-based sorting accelerator [2].
- Releasing the source code of our simulator [119].

We published the findings of this work in [18] and filed a patent [30].

5.2 Background and Motivation

Figure 5.1 shows the structure of the intermediate array, and the pseudo-code of three steps of the radix sorting mapped to a PIM-based accelerator, which has one processing unit per subarray (SPU) and one aggregator core, far from the subarrays. Figure 5.1 (a) shows that each element of the intermediate array has three fields: (i) histogram value (hist), prefix-sum value (prefix), and index. The index field is used in the Key placement step and keeps the current index of the bucket. To save the memory space, the prefix-sum can be performed in place, redcuing the number of fields to two fields. Accordingly, if we uses $radix = 2^{16}$, each SPU requires at least 512 KB ($2^{16} \times 2 \times 4$) memory space for the intermediate array.

In addition to the capacity overheads, operations on intermediate arrays impose performance overhead due to (i) random accesses and (ii) the prefix-sum operation. Line 10 in Figure 5.1 (b) and line 8-10 in Figure 5.1(d) show the random access to the intermediate array. Figure 5.1 (c) shows the prefix-sum operation on intermediate arrays, where an APU moves many histogram values and prefix-sum values between subarrays and the APU. Hence, the overhead of the prefix-sum operations is on the order of $n \times r$, where n is the number of subarray-level processing units, and r is the number of buckets. In this work, we reduce these capacity and performance overheads.

5.3 Proposed method

5.3.1 Baseline PIM architecture

Figure 5.2 illustrates the architecture of our proposed method, which uses 3D-stacked memories, such as HMC/HBM. (The proposed architecture can be modified and customized for DIMM as well.) In our 3D-stacked memory, each memory stack has a few layers; within each layer, banks are connected through an interconnection network with a dragon-fly topology, and within each bank, subarrays are connected through

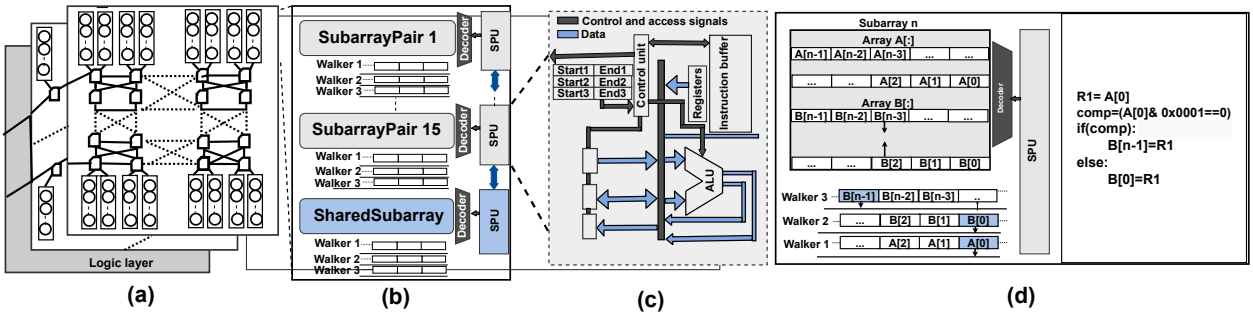


Figure 5.2: Our proposed architecture: (a) The circles are subarrays, the rectangles are banks, and the pentagons are switches. Banks are connected using a dragonfly topology. (b) A bank with an SPU and three Walkers per subarray pair, (c) architecture of each SPU. (d) An example of local binary radix sorting. SPU loads one row of array $A[:]$ in Walker1. In each cycle, SPU reads one entry from Walker1 and places it in either Walker2 or Walker3, based on the binary digit being processed. Once Walker1 is fully read, SPU loads a new row from array $A[:]$ to Walker1. Once either of Walker2 or Walker3 is full, SPU writes the row in array $B[:]$. However, the SPU writes Walker1 in rows starting from the start of the array $B[:]$ but writes Walker3 in rows starting from the end of the array $B[:]$.

a line interconnection topology. Every two banks in a layer form a group, and a through-silicon via (TSV) connects groups in different layers (which are horizontally aligned) to form a vault. The memory stack also has one logic layer.

We use a subarray-level PIM approach, Fulcrum/Gearbox[9, 14], as the baseline PIM architecture. In Fulcrum, every subarray pair has one simplified sequential processing unit (Figure 5.2 (b)) and each vault has a core in the logic layer. Each subarray-level processing unit (SPU) has a few registers, an 8-entry instruction buffer, a controller, and an ALU (Figure 5.2 (c)). (The design is motivated by the characteristics of memory-intensive applications, where there are few simple operations per loaded datum in each step of the process.) The core in the logic layer has several roles, including (i) broadcasting the eight instructions to all SPUs at the beginning of each step and (ii) performing aggregation operations.

In Fulcrum, every pair of subarrays also has three row-wide buffers, called Walkers. The Walkers load an entire row from the subarray at once, but the processing units sequentially access and process one word at a time. The sequential access is enabled by using a one-hot-encoded value, where the set bit in this value selects the accessed word. Therefore, to sequentially process the row, the processing unit only needs to shift the one-hot encoded value, making sequential processing highly efficient. We chose Fulcrum as the baseline architecture because the three Walkers provide three parallel efficient sequential assess, enabling an efficient mechanism for dichotomizing keys into two groups, which is the main operation in binary radix sorting.

Fulcrum also cannot efficiently move data among memory banks and assumes that data is already is bucketed among banks, during the data transfer among the host and the accelerator. This assumption is not true in many scenarios. The second version of Fulcrum, Gearbox [14], adds the interconnection and hardware

support for moving data between banks and subarrays. We employ this capability in Key placement step to send keys to their destination subarray.

To reduce the number of passes, we use the radix of 2^{16}. Therefore, for 32-bit/64-bit keys we require two/four passes of bucketization on data, where each pass comprises four steps: (i) Local sorting, (ii) Histogram generation, (iii) Prefix-sum, (iv) Merging and key placement. In the following subsection, we will explain our contribution in each step.

5.3.2 Local sorting

The three Walkers with shift-based sequential access mechanisms are highly efficient for binary radix sorting, where we need to dichotomies an array of keys into two buckets (Bucket0 and Bucket1). To this end, we load the key array row-by-row in Walker1, and employ Walker 2 as Bucket0 and Walker3 as Bucket1. Then, as shown in Figure 5.2 (d), in each clock cycle, the SPU shifts the one-hot-encoded value to read one key from Walker1 and writes it to either Walker2 or Walker3 based on the digit being processed by shifting the one-hot-encoded value of the corresponding Walker. The binary radix sorting is efficient because it requires no random access. The only problem is that, with non-uniformly distributed keys, the size of each binary bucket can be very different in each pass. To address this issue, instead of reserving a large space for each binary bucket, we propose to reserve a space that is almost the size of the key array. Then, we design a hardware controller that starts Bucket0 from the bottom of the space and fills it upward and starts Bucket1 from the end of the space and fills it downward (Figure 5.2 (d)). The reverse ordering of keys in Bucket1 can violate stability, a requirement for radix sorting. To maintain stability, we store the end address of Bucket0 as a metadata to enable distinguishing the two buckets. Then, in the next pass, our controller processes Bucket1 from end to start.

5.3.3 Histogram generation

In this step of our proposed method, 15 processing units in a bank cooperatively generate one large intermediate array in the lower subarray in the bank. The step comprises three substeps. First, each SPU generates the histogram values of the first 256 buckets. Second, all SPUs reduce the histogram values of each of the 256 buckets in the lower subarray. Third, all SPUs go to the first substep, to generate the histogram values of the next 256 buckets, until the histogram values of all the 2^{16} buckets are generated.

As we explained, the second substep requires reducing the histogram values of 256 buckets. To perform this operation, we propose Cooperative operations, where 15 processing elements cooperate to reduce their histogram values in the last subarray of the bank.

Assuming the histogram array in i^{th}subarray pair is $Hist[i][:]$, the Cooperative reduction is as follows: the i^{th} SPU receives a value from $(i-1)^{th}$ SPU, adds this value to the histogram value of thej^{th} bucket ($Hist[i][j]$), and passes the result to the $(i+1)^{th}$ SPU.

5.3.4 Prefix-sum

In a 3D-stacked memory, in our configuration, 256 subarrays share a bus (TSVs). A naive PIM-based approach performs prefix-sum operations on all histogram values in 256 subarrays, imposing significant overhead for reading and writing these values through the shared bus. Since we reduced the number of intermediate arrays, the overhead of prefix-sum decreases to the overhead of prefix-sum on only 16 histogram arrays in a vault. (The cores in the vaults also aggregate their prefix-sum arrays.)

5.3.5 Merging and key placement

The process of finding the exact position of each key is very similar to the original radix sorting, as shown in Line 11 of Figure 5.1 (d). In this step, keys in the 15 subarrays are sent to the lower subarray, where the shared histogram array resides. Then, the SPU at the bank level derives the position of the key in the sorted output array and sends the key and its address through the interconnection toward the destination subarray. Gearbox [14] adds hardware support for transferring data elements from one subarray to another. We employ this capability for sending each key to the its destination subarray.

5.4 Evaluation

5.4.1 Methodology

Pulley targets sorting gigabytes of data, where the capacity of one memory stack is not enough. Given that new interconnection technologies, such as NVLink [118], provide a high-bandwidth fully-connected topology among multiple devices, we can increase the capacity of our accelerator by connecting multiple devices. We evaluated Pulley in a 6-device setting, where each device has four stacks of 8-GB memories, providing 192 GB capacity. We chose to place four stacks per device to ensure that the power consumption of each device is less than 300 Watt. We chose the 6-device setting because the second generation of NVLink allows 6 links per device. (The third and the fourth generations allow 12 and 18 links per device.)

For each stack, we follow the configurations of Fulcrum [9]. We developed an in-house event-accurate simulator for Pulley and released the source code of the simulator [119].

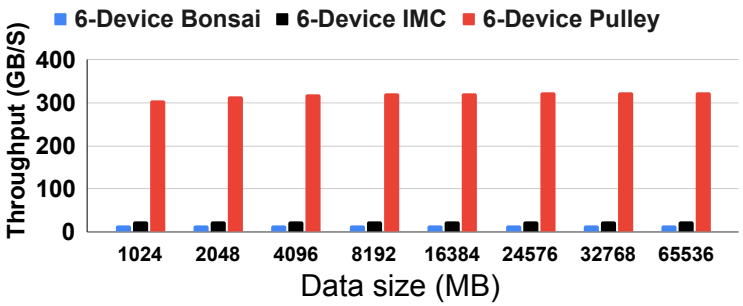


Figure 5.3: Throughput comparison of Pulley vs. Bonsai [1] and IMC [2]

5.4.2 Throughput

We compared our proposed method against an (i) state-of-the-art near-HBM FPGA-based sorting accelerator (Bonsai [1]) and an in-logic-layer sorting accelerator (IMC-Sort [2]). The two evaluated approaches are the most related works that can support sorting gigabytes of data. Pulley, on average, deliver 20× speedup compared to Bonsai and 13× speedup compared to IMC.

5.4.3 Power and temperature constraints

We evaluated the energy consumption of the memory elements and interconnected elements in Pulley using CACTI3DD [120] and evaluated the energy consumption of processing units using the RTL synthesize. The average power consumption of Pulley per stack is 38.6 watts. Our average power density is 540 mW/mm2, which is under the power density budget of a PIM-based accelerator with a high-end server active cooling (1214 mW/mm2 [121, 122, 123]) and under the power budget of the PCIe peripheral interface (300 Watts per device and 75 per stack).

We could not evaluate performance per watt against Bonsai [1] and IMC because Bonsai provides no energy number, and IMC [2] only provides energy numbers normalized to CPU (we do not know what the absolute energy consumption of this method is).

5.5 Conclusions and future Work

This work motivates providing hardware support for sharing intermediate arrays in sorting. As future works, we envision investigating the benefits of shared intermediate arrays for other important kernels such as graph processing and database operations. We also optimized operations on the shared intermediate by providing hardware support for co-operative operations. Future works can investigate what other applications can benefit from these operations.

Chapter 6

TGN-PNM: A Near-Memory Architecture for Temporal GNN Inference on 3D-Stacked Memory

6.1 Introduction

Graph Neural Networks (GNNs) have gained significant attention in various domains, including social network analysis [124], biology [125, 126, 127], and recommendation systems [128, 129], owing to their remarkable capability to capture complex relationships and interactions among entities. GNNs typically learns the graph representation via a message passing mechanism that aggregates the neighborhood information into low-dimensional node embeddings. Afterwards, these embeddings are leveraged for performing various downstream inference tasks on the graph, such as node classification [130, 131], link prediction [132, 133, 134], and graph clustering [135].

In user facing production environments, these inference tasks on GNNs are often times subjected to very stringent latency/throughput constraints. Subsequently, researchers have proposed a plethora of accelerator architectures, primarily focusing on the inference tasks on static GNNs [136, 137, 138, 139, 140, 141, 129]. These accelerators leverage the observation that inference tasks on prevailing static GNN models (e.g., GCN) are primarily composed of two distinct execution phases: aggregation and combination [136]. The aggregation phase uses the graph structure and neighbor interactions to recursively update the feature vectors of nodes. In contrast, the combination phase transforms the aggregated features of each node through neural network

layers to compute the node embeddings. Among these two phases, the aggregation phase shows the typical pitfalls of graph processing behavior; this phase is data-intensive with highly irregular access patterns, and low compute intensity. On the other hand, the combination phase shows a regular access pattern with high compute density. Accordingly, existing static GNN accelerators aim to optimize one or both of these phases. However, many of these optimizations are inherently tied to the static nature of the graph (e.g., storing the graph in specialized compressed formats), and cannot be easily extended to accelerate *temporal GNN* (TGNN) [142, 143], where the graph topology is no longer static as the nodes and edges evolves over time. Most real-life systems of interactions falls into this temporal category, with studies showing that including the temporal information for graph representation learning can boost prediction accuracy [144, 145, 146].

Despite the importance of TGNNs, if it very challenging to design an accelerator targeting TGNN workloads for various reasons. First is the lack of standard model architecture. Unlike GCN, which is the prevailing model for static GNNs, there is no general consensus about which temporal GNN model is the best, and often times boils down to accuracy-complexity tradeoff. Therefore, it is desirable that the proposed accelerator is flexible to accommodate design choices that might arise in the future. Second, unlike static GNN's aggregation/combination phases, models for TGNNs often times cannot be decomposed into distinct phases of rigid execution patterns (e.g., aggregation phase itself may require neural network layers, such as in temporal graph attention [143]). Furthermore, based on the TGNN model architecture, batch size, and dataset, the operational intensity of the execution phases can vary widely and can contain both memory-bound and compute-bound kernels (more discussion on this topic on Section 6.2.1). It is critical to handle both types of kernels efficiently to extract maximum performance. Third, due to the evolving nature of the graph, it is difficult to maintain proper workload balance. This issue is less prominent in static graphs, where pre-processing steps can by applied on the graph to reorganize vertices to ensure balance and extract maximum locality [147, 148, 149]. This pre-processing cost is a one time cost for static graphs and gets amortized over time, but becomes prohibitively expensive on dynamic graphs. Researchers have proposed node migrations techniques that can partially solve the workload imbalance on dynamic graphs [98, 150]. Unfortunately, migration based methods introduce additional data-movements to relocate the migrating nodes. This data-movement overhead is significant for GNN workloads, as each node is usually associated with a large number of features. Migration-based techniques also require dictionary lookups to determine a node's current location. If caching is used, then maintaining coherence also becomes an issue. Because of these aforementioned challenges, accelerators designed specifically for TGNN is extremely rare. We are aware of only one prior accelerator that targets TGNN workloads [151], mapping the TGN framework [144] on an FPGA (referred as tFPGA). However, we observed that this approach can only support a few numbers of compute

units due to FPGA resource limitation and is vastly outperformed by in-/near-memory-processing-based approaches.

In this chapter, we propose TGN-PNM, which is a near-memory architecture for accelerating TGNN workloads. TGN-PNM exploits vault-level parallelism by placing one Vault Processing Unit (VPU) at every vault in the logic-layer of a 3D-stacked memory. Each VPU contains a SIMD unit for common memory-intensive operations (e.g., BLAS level 1 and 2 kernels, time encoding, and other elementwise operations) and a systolic array for compute-intensive operations (e.g., BLAS level 3 kernels). Placing the compute units at the logic layer enables obtaining performance that is memory-capacity-proportional (i.e., linear performance improvement can be obtained by increasing the number of memory stacks) and also exposes the internal memory bandwidth that can be an order of magnitude higher than the bandwidth seen by external I/O links [10]. While it is possible to obtain a finer-grained parallelism by placing the compute units at the banks or subarrays, as proposed by a few prior generic accelerator architectures [22, 152, 9, 14, 20, 19, 16], it comes with several pitfalls. First, any extra logic in the compute units gets multiplied by the number of banks/subarrays. Therefore, these compute units cannot accommodate complex logic, such as transcendental time encoding functions needed for TGNN kernels. These functions needs to be handled separately, for example, by the host or by placing compute cores in the logic layer, thereby introducing a lot of data-movement overhead. Placing compute units at the bank-/subarray-level also imposes a stricter requirement on the data layout and moving the intermediate results to conform to this requirement often times becomes a bottleneck. Finally, gates in the DRAM dies are usually larger and slower than their counterpart in the logic die. These reasons motivate us to place the compute units at the logic layer, rather than in the banks/subarrays.

In our approach, if we partition the graph in the traditional manner, where each vault holds a subset of nodes, the primary bottleneck arises from significant inter-vault communication during neighbor aggregation. To overcome this limitation and improve workload balance, we introduce a feature-dimension partitioning scheme. The key idea of this approach is to allocate each vault with a subset of the features of *all* nodes. This design choice takes advantage of the fact that nodes in GNNs typically possess numerous features (ranging from hundreds to thousands). With this partitioning scheme, all elementwise operations, including operand fetch for neighbor aggregation, become localized within each vault. Consequently, inter-vault communication is only required for dot-product reduction operations during matrix-vector and matrix-matrix multiplications, as each vault produces only a portion of the output vector or matrix. This reduction can be handled efficiently with a simple pipelined tree-adder. Additionally, each vault only needs to contain a subset of the weights and does not need to duplicate the weights across all the vaults. Another advantage of our proposed architecture is that the compute units within all vaults can work in lockstep, leveraging a single instruction queue and a

unified fetch/decode unit. We extended our approach with a broadcasting mechanism and hybrid partitioning scheme, both of which helps towards efficiently handling small feature vectors.

We evaluated our approach against a few architectures: a high-end CPU and GPU, a subarray-level general purpose PIM architecture Gearbox [14], a bank-level AI accelerator Newton [22], and the FPGA-based TGNN accelerator tFPGA [151]. Our evaluation demonstrated average throughput gains of 26.8x over CPU, 16.7x over GPU, 5.2x over Gearbox, 4.4x over Newton, and 10.4x over tFPGA.

6.2 Background and Motivation

6.2.1 Temporal Graph Neural Network (TGNN)

Evolving graphs can be expressed in two manners: Discrete-time dynamic graphs (DTDG) and continuous-time dynamic graphs (CTDG). In DTDG, the dynamic graph is represented as a sequence of static graph snapshots, $G(t) = \{G_{t_1}, G_{t_2}, ..., G_t\}$. However, DTDG is a coarse-grained representation where the exact event timestamps between subsequent snapshots are lost. This loss of temporal information can lead to comparatively lower accuracy during inference [145]. On the other hand, CTDG is more fine-grained and can represent the dynamic graph as an ordered sequence of timestamped events, $G(t) = \{\delta_1, \delta_2, ..., \delta_k\}$, where each event δ_i denotes the addition/deletion of a node or an edge. Typically, CTDG is a multigraph, which means that there can be more than one edge between a pair of nodes, pertaining to multiple interaction events between the nodes at different times. In this paper, we focus on the continuous time representation of the temporal graph.

A few prior works proposed neural network models for learning representations over CTDG [153, 154, 144, 145, 155, 146, 156]. Among these approaches, TGN [144] proposes a generic message-passing-based modular framework that can be tuned to mimic many of these other approaches. In TGN, each node is composed of raw node features $\boldsymbol{v}_i$ that denote the static properties of the node, node memory or state $\boldsymbol{s}_i(t)$ that captures the history of temporal interactions, and dynamic node embeddings $\boldsymbol{z}_i(t)$ that combine the node memory with the spatial information (neighborhood or graph topology). Each interaction event involving a node produces a message. An interaction between nodes v_i and v_j therefore will produce two messages:

$$\boldsymbol{m}_i(t) = \{\boldsymbol{s}_i(t^-)||\boldsymbol{s}_j(t^-)||\Phi(\Delta t_i)||\boldsymbol{e}_{ij}\} \tag{6.1}$$

$$\boldsymbol{m}_j(t) = \{\boldsymbol{s}_j(t^-)||\boldsymbol{s}_i(t^-)||\Phi(\Delta t_j)||\boldsymbol{e}_{ij}\} \tag{6.2}$$

Table 6.1: Arithmetic intensity (flops/byte) of various TGNN models.

	Memory updater function					Embedding function				
		Batch size = 1		Batch size = 32			Batch size = 1		Batch size = 32	
	Type	Wiki	GDELT	Wiki	GDELT	Type	Wiki	GDELT	Wiki	GDELT
TGN-attn [142]	GRU	0.50	0.50	15.04	15.63	attn (1 layer)	3.11	2.62	83.03	69.68
TGN-sum [142]	GRU	0.50	0.50	15.04	15.63	sum	0.71	0.48	0.77	0.49
TGAT [145]	-	-	-	-	-	attn (2 layers)	28.09	26.60	116.97	151.61
JODIE [146]	RNN	0.05	0.05	13.47	14.82	time projection	0.50	0.50	0.97	0.97
tFPGA [151]	GRU	0.50	0.50	15.04	15.56	simple attn	1.03	0.74	1.17	0.78

Here, $||$ denotes the concatenation operation, $\boldsymbol{s}(t^-)$ denotes the node state just before the event, $\Phi(\Delta t)$ is the vector encoding of elapsed time since the last state update of that particular node, and $\boldsymbol{e}_{ij}$ is the edge embedding of that interaction. The time encoding function is usually implemented as $\Phi(\Delta t) = cos(\Delta t \boldsymbol{w} + \boldsymbol{b})$, following prior works such as Time2Vec [157] and TGAT [145]. In batch processing, one node can receive multiple messages within a batch, which is aggregated into a single message per node by a message aggregator function. The message aggregator function can be implemented in various ways. The common implementation simply keeps the most recent message as the aggregated message [144, 151, 156]. The aggregated message is then used to update the node states as follows:

$$\boldsymbol{s}_i(t) = mem(\bar{\boldsymbol{m}}_i(t), \boldsymbol{s}_i(t^-)) \tag{6.3}$$

$$\boldsymbol{s}_j(t) = mem(\bar{\boldsymbol{m}}_j(t), \boldsymbol{s}_j(t^-)) \tag{6.4}$$

Here, $\bar{\boldsymbol{m}}(\boldsymbol{t})$ is the aggregated message and $mem()$ is a learnable function for updating the memory, e.g., a recurrent neural network such as LSTM or GRU. Finally, the dynamic embedding is derived by aggregating over the k-hop temporal neighborhood $\mathcal{N}_i^k$:

$$\boldsymbol{z}_i(t) = emb(\boldsymbol{v}_i, \boldsymbol{v}_j, \boldsymbol{s}_i(t), \boldsymbol{s}_j(t), \boldsymbol{e}_{ij}, \Delta t) | \forall j \in \mathcal{N}_i^k \tag{6.5}$$

Here, $emb()$ is a learnable function and can be realized in several ways. One is to simply use the node's current state, $\boldsymbol{z}_i(t) = \boldsymbol{s}_i(t)$. However, this approach can lead to memory staleness if the node's state has not updated in a while [158, 144]. JODIE [146] uses a linear time projection of the node's state to avoid this issue, $\boldsymbol{z}_i(t) = (1 + \Delta t \boldsymbol{w}) \circ \boldsymbol{s}_i(t)$. TGAT [145] and TGN [142] uses a multi-head graph attention mechanism over the temporal neighborhood as the embedding function. It is also viable to simply use the average of neighbor states as the embedding while retaining relatively high accuracy, as shown by TGN-sum [142].

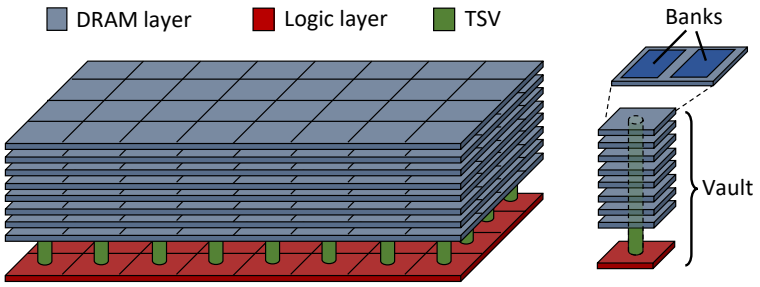


Figure 6.1: Organization of a Hybrid Memory Cube [3, 4].

Table 6.1 summarizes the arithmetic intensity of these functions for different datasets, model-architectures, and batch sizes. It can be observed that the arithmetic complexity of the same kernels (e.g., embedding function) can differ vastly among different models. Furthermore, even within the same model and same function, arithmetic complexity can change widely depending on the data reuse opportunity. Therefore, a TGNN accelerator must be flexible to efficiently accommodate kernels with different arithmetic intensity.

6.2.2 3D-stacked memory

In 3D-stacked memory, multiple DRAM dies (e.g., 4 or 8) are stacked vertically and may contain an optional buffer or logic die. Notably, two distinct variations of 3D-stacked memory technology have emerged that are promising in terms of PIM architectures [159, 121]: Hybrid Memory Cube (HMC) [3] and High Bandwidth Memory (HBM) [160]. In HBM, the external IO interface of the memory stack is implemented through DDR physical channels. The HBM memory stacks are tightly integrated with the host die on a silicon interposer, with the memory controllers residing on the host die. In contrast, the HMC specification places the memory controllers within a logic die part of the HMC memory stack. Figure 6.1 shows the organization of HMC. In HMC, memory layers are partitioned vertically to form mostly independent vaults. Each memory layer in each vault contains multiple memory banks. All banks within a vault are connected using through-silicon vias (TSVs) that act as the shared bus to carry DRAM address and command signals to these banks. Each vault also contains a memory controller in its logic layer. DRAM transistors in the memory layers have traditionally been designed for low cost and leakage. The logic die, on the other hand, uses high-performance transistors [11]. HMC 2.1 specification supports up to 32 vaults. Four external SerDes IO links are connected to these memory controllers using a crossbar switch at the logic layer. Note that this crossbar switch only connects the IO links to the vaults, but the vaults themselves do not communicate directly (e.g., send memory access requests) with each other. These IO links can be used to connect to the host or can be used to connect to other HMC cubes to increase capacity. In this chapter, we will mostly refer to memory specifications and terminologies pertaining to HMC, but the concept can be easily extended to HBM memory by placing the

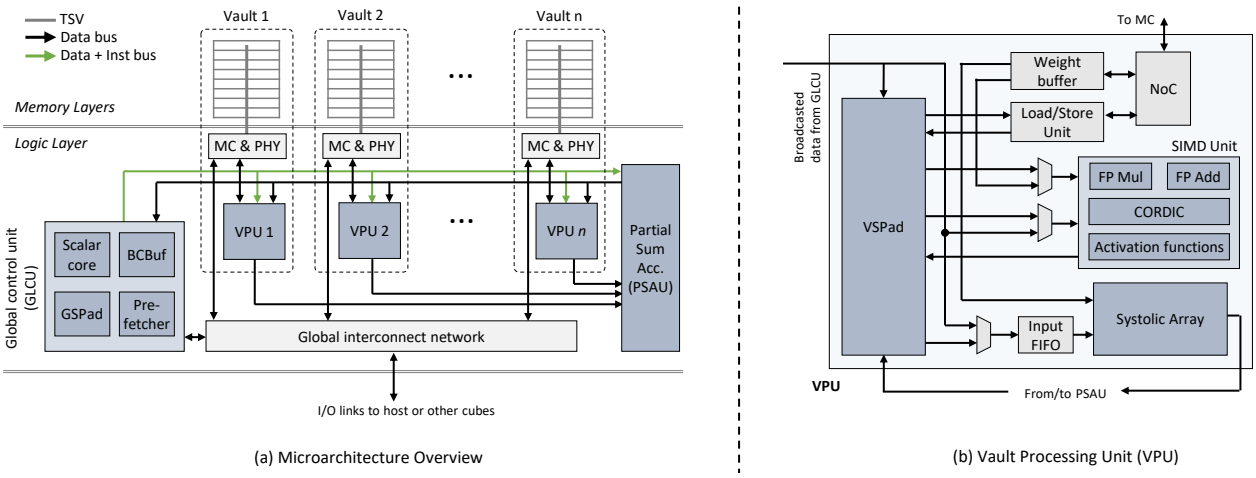


Figure 6.2: TGN-PNM Architecture.

VPUs after the memory controller on the host die, similar to the approach proposed in [161]. For our specific application, however, using HMC offers a few advantages over HBM: i) Placing the VPUs in the logic layer of HMC results in lower energy for data movement compared to placing them on the host die, as the data do not have to travel through the silicon interposer. ii) HMC is optimized for random accesses (by using short row buffers, closed page policy, consecutive address to different banks), while HBM is for sequential accesses (wider row buffers, open page policy, consecutive address to the same row). Consequently, HMC is better aligned with the irregular access patterns of graph workloads.

6.3 TGN-PNM Microarchitecture

Figure 6.2(a) presents the microarchitecture of our proposed approach. We primarily add three components besides the interconnect network and the memory controllers that are already present in the logic layer of an HMC-like memory: i) Vault-level Processing Units (VPUs) at the logic layer of every vault, ii) one Global Control Unit (GLCU), and ii) one Partial-Sum Accumulation Unit (PSAU). Next, we will describe the contents and connectivity of each of these components.

6.3.1 Vault-level Processing Unit (VPU)

The VPUs are the primary computing units in our design. Figure 6.2(b) shows the organization of a VPU. Each VPU contains a $S_u \times S_u$ systolic array of fused-multiply-accumulators, a S_u-lane SIMD unit, a local scratchpad memory (VSPad), and a weight buffer. In the case of the systolic array, instead of using a single $S_u \times S_u$ array, it is realized as a group of four arrays, each with the dimension of $S_u \times S_a$, where $S_a = S_u/4$. These arrays can be configured to work cooperatively or independently: as a single $S_u \times S_u$ array, two

$S_u \times 2S_a$ arrays, or four $S_u \times S_a$ arrays. The concept of this segmented systolic array is borrowed from the work of Yan et al. [136] and is leveraged to support dense matrix multiplication with small feature dimensions efficiently. The output of the systolic array is sent to the PSAU for accumulation.

The SIMD unit contains S_u pipelined multipliers and adders, one 16-stage CORDIC functional unit for time encoding, and one activation function unit. The activation functions are realized using a lookup table. The CORDIC unit consumes S_u operands simultaneously but processes them sequentially (i.e., it has an initiation interval of 16 cycles). As both the systolic array and the SIMD unit work on S_u data elements at a time, the access granularity to the VSPad is fixed to $size(elem) * S_u$. These functional units' operands can come from the VSPad, weight buffer, result bus, or the global broadcast buffer. All the VPUs operate in a lockstep controlled by the GLCU. VPUs have very limited outside visibility - they can only load/store data from their local vault (by using the address broadcasted by the GLCU) and send data out only to the PSAU for reduction. VPUs cannot communicate directly with each other. We map the dense matrix multiply on the systolic array. All the other operations are usually mapped to the SIMD unit. Mapping is disscued in more detail in Section 6.4.1.

6.3.2 Global Control Unit (GLCU)

The primary objective of the GLCU is to drive the VPUs by broadcasting instructions. It also contains a scalar core, a scratchpad memory (GSPad, local to the GLCU), a data broadcast buffer (BCBuf), and a prefetcher. The scalar core can be used for operations not supported by the VPUs or for complex reductions (e.g., softmax in graph-attention). It has access to the full memory stack (via the global interconnect network), the GSPad, and the BCBuf. The function of the broadcast buffer BCBuf is to provide a common operand to all the VPUs. In our mapping scheme, discussed later in Section 6.4.1, BCBuf is used for time encoding and handling low-dimension GEMM. Width of the BCBuf is S_u elements. GLCU also contains a prefetcher that can fill the GSPad or VSPad by fetching a node's associated data. Using the broadcast mechanism, we can issue memory read/write requests to every vault in every clock cycle. With this approach, we can easily saturate the memory controller queue without resorting to maintaining multiple threads, as done in a few prior PIM approaches [152, 162].

6.3.3 Partial-Sum Accumulation Unit (PSAU)

As discussed earlier, partitioning across the feature dimension requires only inter-vault communication when performing dot-product reductions during GEMV and GEMM operations. The PSAU handles this reduction. It contains a pipelined parallel adder tree to reduce the results from all the VPUs' systolic arrays. A few entry

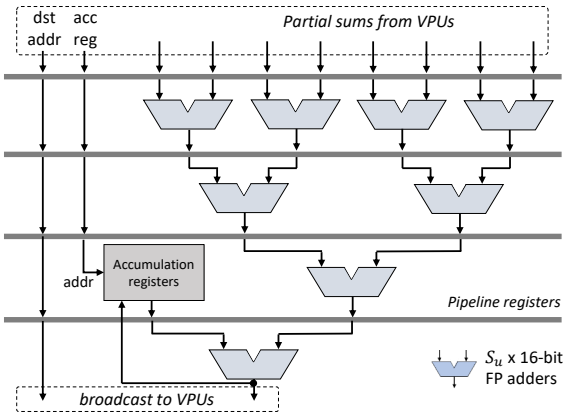


Figure 6.3: Partial-sum accumulation unit (PSAU). Figure drawn assuming a total of eight VPUs.

accumulation registers store the partial results. The final output is broadcast to all the VPUs and stored in either one of the VSPads or the GSPad, depending on the destination address. An alternative to running the vaults in lockstep and doing partial sum across vaults is to make the VPUs completely independent. However, it requires either having a copy of the model parameters in each vault or a high amount of inter-vault traffic.

To summarize, the VPUs can get their operands from i) their local scratchpad memory (VSPad), ii) their local DRAM stack (must load to the VSPad first), and iii) the broadcast buffer (BCBuf). After doing the intended computation, VPUs can send the result to the following: i) local scratchpad (VSPad) as the intermediate operands for later stages, ii) local DRAM stack store, or iii) PSAU for reduction. The result of the reduction by PSAU can be sent back to one of the VPU's scratchpad or the GLCU's scratchpad.

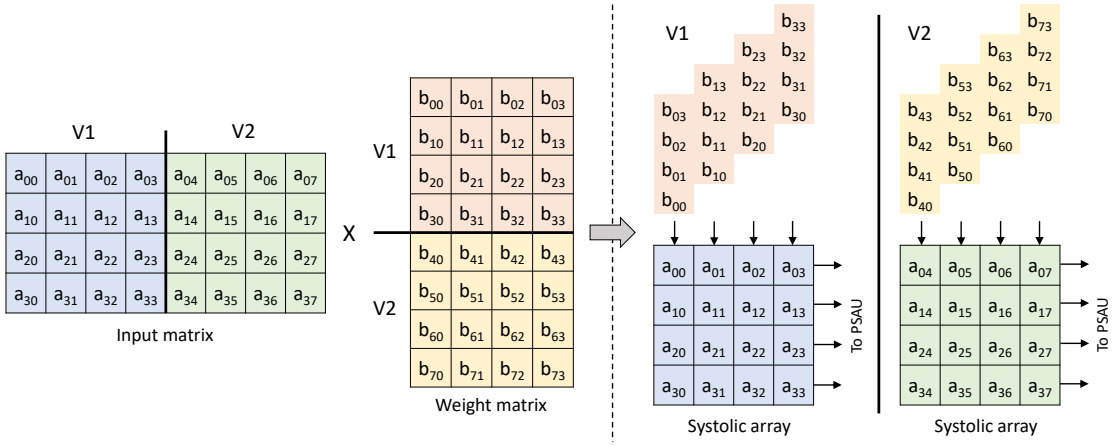
6.4 Mapping TGNN Frameworks on TGN-PNM

In this section, we discuss a potential graph storage format and mapping schemes for the common operations pertaining to TGNN frameworks.

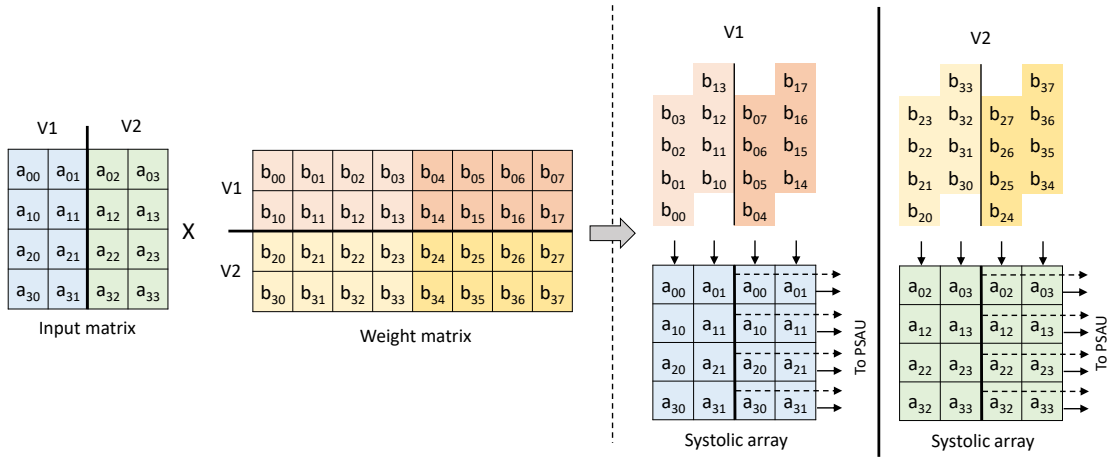
6.4.1 Mapping of common operations

Dense matrix multiplication

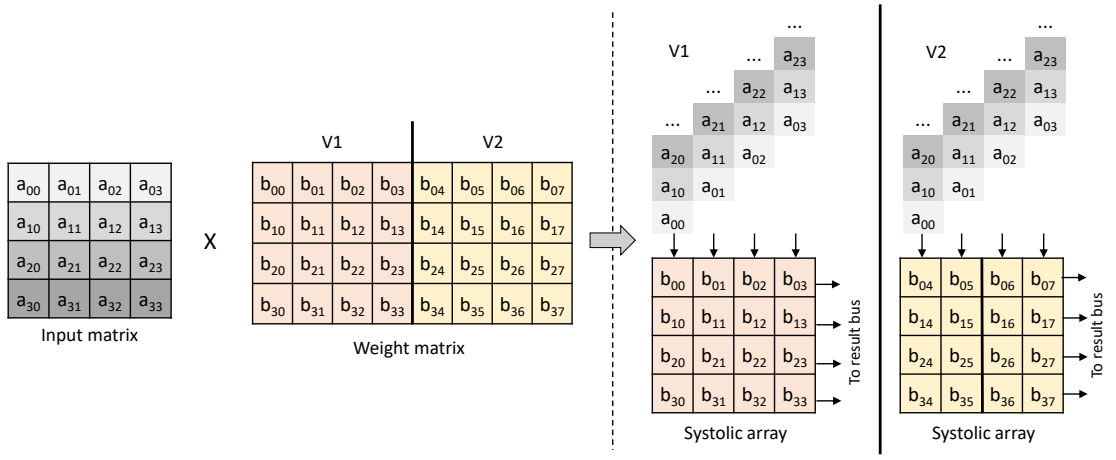
We always process dense matrix multiplication using the systolic array. Figure 6.4 demonstrates our mapping scheme. Depending on the current input location and input's feature dimension, we map the operation into one of the three possible scenarios: i) In the first scenario, input is currently stored in VSPad (i.e., using feature-based partitioning) and the number of input-features within the vault is $\geq S_u$. In this scenario, all lanes of the systolic array will be occupied without resorting to any special strategy. We use input-stationary



(a) Scenario 1: Input stored in VSPad, with per-vault feature-dimension $\geq S_u$



(b) Scenario 2: Input stored in VSPad, with per-vault feature-dimension $< S_u$



(c) Scenario 3: Input stored in the broadcast buffer (BCBuf).

Figure 6.4: Mapping of dense matrix multiplication on TGN-PNM.

dataflow on the systolic array for this case. If the input matrix is $(m \times n)$ and the weight matrix is $(n \times k)$, then m is usually the batch size (or batch size * the number of neighbors) or the partitioned output of an intermediate matrix. n-dimension is also partitioned across vaults. Therefore, these values are relatively small when compared to k. Using input-stationary dataflow enables streaming the weights along k, thereby providing higher reuse of the inputs. Furthermore, we use two sets of registers to enable loading the next set of inputs and biases while processing the current set. Outputs of the systolic arrays are sent to the PSAU for reduction, after which the final result can be stored in the VSPad or BCBuf, depending on the output dimension. ii) In the second scenario, input is stored in the VSPad; however, the per-vault feature dimension is smaller than S_u. In this case, using the former scheme will leave some of the systolic array lanes unused. To avoid this, we use a smaller segment of the systolic array[1]. To fully utilize the available lanes, the inputs are duplicated P times, and the weight matrix is also folded P times. This approach fully occupies all the MAC units, maximizing the available compute bandwidth. The maximum value of P is the number of segments. While increasing the number of segments means that we can support narrower input matrices efficiently, it also means that the output of the systolic array produces a higher number of outputs per clock (i.e., $P.S_u$), and the PSAU had to be widened accordingly. In our implementation, we support four segments. Any higher value resulted in unacceptable area overhead for PSAU. iii) In this scenario, the input is stored in the BCBuf and broadcasted to all VPUs, where the corresponding results are calculated and stored back on the same vault. As we are broadcasting the inputs, we use weight stationary dataflow in this scenario.

Time encoding

As mentioned earlier in Section 6.2, time encoding is usually implemented as $\Phi(\Delta t) = cos(\Delta t \boldsymbol{w_t} + \boldsymbol{b_t})$. Depending on the task, Δt denotes either the time elapsed since the last update of a node or the incident time of an edge. $\boldsymbol{w_t}$ and $\boldsymbol{b_t}$ are the learnable weights and biases. Each VPU contains a subset of these weights and biases in its weight buffer. Timestamps are stored in the memory in a traditional fashion (discussed in Section 6.4.2) and fetched in the GSPad for processing. The scalar core computes the Δt of a batch and then puts the results in the BCBuf. Then, the Δt of S_u elements are broadcasted to the VPUs. The SIMD units on the VPUs will multiply the broadcasted value with the stored weights, add bias, and then send it to the CORDIC unit to calculate the resultant time encoding.

[1]The idea of a segmented systolic array is borrowed from [136], where a systolic module is formed by combining several narrower arrays. Refer to Section 6.3.1) for more detail.

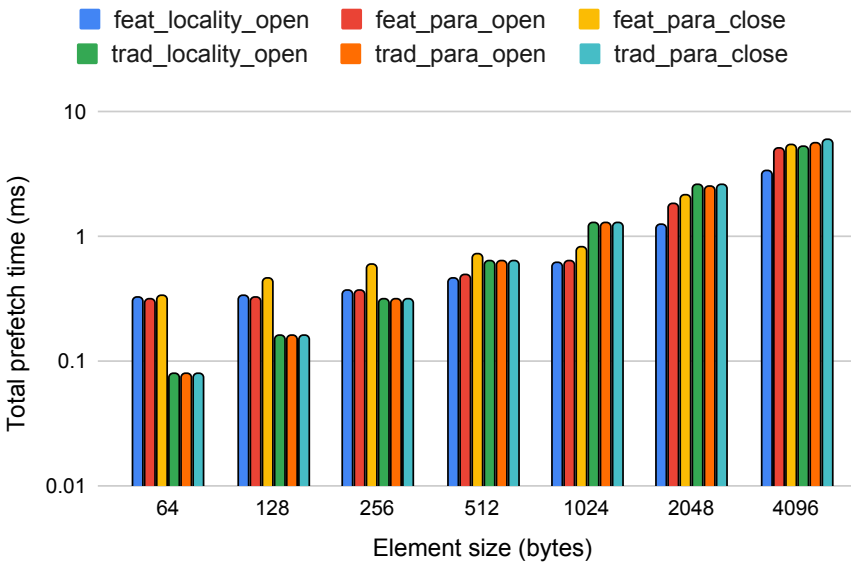


Figure 6.5: Total operand load time for the Wikipedia dataset with different memory address mapping and page policy configurations. *Lower is better.*

Elementwise operations

Elementwise operations are performed using the SIMD unit. Note that using the broadcasting mechanism for elementwise operations is not efficient. This is because the same value gets broadcast to every VPU, but we do not have any reuse opportunity in elementwise operations. Therefore, we store both operands in the VSPad for elementwise operations. One potential shortcoming of our approach is that, if the dimension of the operands is small, then the SIMD lanes will be underutilized.

6.4.2 Graph storage format

Static GNN accelerators typically use compressed formats such as CSR to improve locality characteristics. However, such formats are unsuitable for dynamic graphs, as any change in the graph topology (e.g., addition of nodes or edges) requires reconstructing the whole graph from scratch. To support this dynamic behavior efficiently, we store the graph in an adjacency list format and limit the maximum number of neighbors a node can have[2]. This constraint is fairly common in existing TGNN frameworks and accelerators [144, 151]. To maintain a fixed number of neighbors, the adjacency list of each node is realized using a circular queue. More specifically, we stored the following metadata for each node: $\{head, tail, ts, eid_0, eid_1, ..., eid_K, ets_0, ets_1, ..., ets_K\}$. Here, *head* and *tail* are used to determine the location for new edge insertion in the circular queue. *ts* is the timestamp of the last update. eid_i and ets_i are i-th edge's neighbor node id and incident timestamp,

[2]One reason that limiting the number of neighbors does not hamper the model accuracy by a large margin is that the impact of temporal interactions of the discarded neighbors is already captured and summarized in the node's memory.

respectively. Note that partitioning along the feature dimension is impossible for these graph metadata, as these are mostly scalar values and are also shared by all the vaults. It is possible to partition these graph metadata along the node dimension so that each vault contains all metadata associated with a subset of the nodes. However, real-world graphs usually demonstrate power-law degree distribution, indicating that some of the vaults will get a disproportionately high number of accesses if we partition along the node dimension, causing severe load imbalance. Therefore, these metadata are stored in a traditional fashion (i.e., consecutive memory address is mapped to either in the same DRAM row to maximize row buffer locality or in different banks to maximize parallelism) and fetched to the global scratchpad (GSPad) instead of the vault's local scratchpad (VSPad).

On the other hand, vector data, such as weights and the states of nodes and edges, can be partitioned along their feature dimension so that each vault contains a subset of features for all nodes. However, if the feature dimension is small, it may be beneficial to use traditional memory address mapping for these vector data as well. Figure 6.5 shows the operand prefetch time with different partitioning and address mapping schemes. We used an 8 GB HMC 2.1 memory stack for this experiment with 32 vaults, 32B DRAM column width, and 256 B wide row buffers. Here, the x-axis shows the data size associated with each node. We tried three paging-policy/address-mapping-scheme combinations: i) optimized for row buffer locality, where consecutive addresses are mapped to the same row ($x\_locality\_open$), ii) optimized for parallelism, where consecutive addresses are mapped to different banks, and keeping the row buffer open (x_para_open), and iii) optimized for parallelism with closed page policy (x_para_closed). Figure 6.5 suggests that traditional mapping performs best for per-node data size $\leq 256B$. For larger data, feature-based scheme $feat\_locality\_open$ performs best. One interesting observation is that feature-based partitioning performs better in some cases, even when each vault's portion of the feature may not fully occupy a DRAM column width (i.e., for an element size of 512B, translating to a per-vault element size of 16B). One reason is that the memory requests per vault case of feature-based partitioning are perfectly balanced. Another factor impacting the traditional scheme was the additional latency of traversing the global interconnect. Therefore, we propose a *hybrid* partitioning where the data is stored using a traditional scheme if the associated data size $\leq 256B$ (feature-dimension ≤ 128 with FP16) and using a feature-based scheme otherwise. An additional consideration is the intended operation on the data. If it is an elementwise operation, then using the traditional scheme is not viable because broadcasting is not optimal for elementwise operations. In this case, the data is stored using the feature-based scheme.

6.5 Evaluation

In this section, we compare the performance of our method against five other architectures: (i) a high-end CPU server as the baseline, (ii) NVIDIA A100 tensor core GPU, (iii) subarray-level general purpose PIM architecture Gearbox [14], (iv) bank-level PIM-based machine learning accelerator Newton [22], and (v) an FPGA-based TGNN accelerator (we refer to this approach as *tFPGA* in our paper) [151]. A summary of the configuration of these platforms is provided in Table 6.3. Due to the lack of TGNN-specific accelerators except for tFPGA, we choose to compare against semi-general purpose accelerators such as Gearbox and Newton. Besides, as one of these is subarray-level, and the other is bank-level, it highlights the trade-offs of the vault-level accelerator we proposed. The rest of the section is organized as follows. First, we discuss our evaluation methodology. Second, we discuss the other architectures and how we mapped the TGNN application on them in detail. Third, we present the performance comparison. Finally, we provide the area estimation.

6.5.1 Methodology

We evaluated our approach by mapping two variants of the TGNN model architecture proposed in [144]: TGN-attn and TGN-sum. Both of these models use GRU as the memory updater function. TGN-attn uses a single-layer multi-head graph-attention mechanism for neighbor aggregation, while TGN-sum uses the average of the neighbor states. As a result, TGN-attn has both memory-intensive (collecting neighbor state data) and compute-intensive (creating key, value, and query matrices for calculating self-attention weights) stages. On the other hand, TGN-sum's neighbor aggregation is memory-bound (refer to Section 6.2.1 for more details). We set the maximum number of neighbors to 10 and batch size to 200, following the original model implementation in [144]. However, as discussed later in Section 6.5.3, Gearbox and Newton only support GEMV operation and perform GEMM by repeating GEMV on every input column. We set the batch size to one for these two approaches as they do not benefit from batching.

We implemented our approach on an HMC 2.1-like memory stack with 32 vaults and 16 banks per vault. The TSV width is 64 bits, clocked at 2 GHz, attaining per vault memory bandwidth of 16 GB/s (total 512 GB/s for a single stack). The logic layer is clocked at 1.2 GHz. S_u is set to 16 (i.e., 16×16 systolic array and 16-lane SIMD units in each VPU). In our implementation, we use 16 bit brain floating-point (BF16) format for all the operands. We dropped the support for denormalization and restricted rounding modes to reduce the area required by the FP units.

The performance measures are collected by building cycle-accurate simulation models for the VPUs and the GLCU and then feeding the resulting memory request trace to a modified version of DRAMSim3

Table 6.2: Characteristics of the used datasets. Time encoder dimension is fixed to 100 for all datasets.

Dataset	$\|V\|$	$\|E\|$	$\|\boldsymbol{v}_i\|$	$\|\boldsymbol{e}_{ij}\|$	$\|\boldsymbol{s}_i(t)\|$	Max weight dim	Total weight (MB)
Wikipedia	9K	157K	-	100	100	516 x 944	5.95
Reddit	11K	672K	-	100	100	516 x 944	5.95
GDELT	9K	1913K	413	186	413	1242 x 1680	19.4

[163]. As for the performance metrics, we measured batch processing latency and throughput. We define batch processing latency as the elapsed time between receiving a batch to process and writing back the corresponding dynamic node embeddings to memory. We use the number of processed events per unit time for throughput measurement. The objective is to maximize the throughput and minimize the latency for the TGNN inference task.

6.5.2 Datasets

We conduct the experiments on three real-world datasets: Wikipedia [146, 164], Reddit [146, 164], and GDELT [153]. Details of these datasets are given in Table 6.2. Wikipedia and Reddit are bipartite graphs consisting of user edits on Wikipedia pages and posts on subreddits. GDELT dataset is a reduced version of a temporal knowledge graph dataset introduced in [156]. Among these datasets, Wikipedia and Reddit do not have any raw node features. The time encoding and the nodes' memory dimensions are configurable hyperparameters. We used 100 as the time encoder dimension. On the Wikipedia and Reddit datasets, we used 100 as the memory dimension following the prior works [144, 145, 146]. On GDELT, the memory dimension is set equal to the raw node feature dimension of 483.

6.5.3 Mapping on evaluated architectures

As mentioned earlier, we evaluated our approach against five other architectures. This section goes into the implementation details and mapping schemes on these architectures.

CPU and GPU

For performance evaluation on the CPU and GPU platforms, we profiled the open-source implementation of the TGN model architecture [165], which is written using the PyTorch Geometric library. However, on the GPU platform, we have used FP16 instead of BF16 because the GRU cell of PyTorch does not support BF16 on CUDA as of version 2.0.1. We have not included the host-GPU data transfer times in the measurements to provide a fair comparison. The reported results are an average of five runs.

Table 6.3: Configuration of the evaluated architectures.

Architecture	Parameters
TGN-PNM	8GB HMC 2.1, 32 vaults, 256B row buffers, TSV BW 16GB/s, 16-lane bfloat16 SIMD and 16x16 systolic array per vault, 32KB VPU SPad, 1MB global SPad, logic layer freq 1.2GHz.
CPU	Server with two AMD EPYC 7742 64-core @ 2.25GHz (total 256 hardware threads), 1024GB DDR4, 8 memory channels, peak memory BW 409.6GB/s.
GPU	NVIDIA A100 SXM, 80GB HBM2e, peak memory BW 2039GB/s, peak compute rate for 16-bit FP is 624 TFLOPs. Host is the same as the baseline CPU server.
Gearbox [14]	8GB HMC 2.1, 32 vaults, 256 subarrays per vault, 256B row buffers, 49ns row activation time, 8192 subarray-level ALUs @ 164MHz, TSV BW 16GB/s per vault, logic layer cores ARM Cortex-A35 @ 600MHz, 128KB scratchpad memory shared by the cores.
Newton [22]	8GB HBM2e-like, 16 pseudo channels, 16 banks per channel, 1024B row buffers, 49ns row activation time, 16 MAC units per bank.
tFPGA [151]	Xilinx U200 FPGA @ 250MHz, 77GB/s DDR4 memory, two CUs, four 8×8 systolic arrays and one 16-lane multiply-add tree per CU.

Gearbox [14]

Gearbox places scalar processing units at the subarrays of a 3D-stacked memory. These processing units support word-level arithmetic and logic operations. In each subarray, three latched row buffers (called Walkers) act as the source/destination registers. Gearbox also contains an ARM core in each vault's logic layer, primarily for reduction operations. Although the processing units of Gearbox can operate only on a single word per cycle, Gearbox can attain high performance by leveraging massive subarray-level parallelism.

Authors of Gearbox implemented GEMV operation by mapping each row of the matrix to a subarray and then broadcasting the input vector elements one by one to all subarrays. The input vector itself is stored in a shared buffer at the logic layer. The subarray-level processing units (APLUs) perform the MAC operation. This approach does not require partial sum accumulation across subarrays. However, given the large number of subarrays (8192 subarrays in an 8GB HMC 2.1 stack), this approach only makes sense for a very large matrix. For example, the authors used a matrix of dimension 25600x19200 for evaluating performance. The matrix size is often much smaller in practical TGNN datasets (refer to Table 6.2) and causes extreme under-utilization of the processing units. In our evaluation, we used an alternative scheme, where every [$subarrays\_in\_vault$ X $elems\_in\_dram\_row$] slice of the matrix will be mapped to a vault. A full matrix is thus potentially distributed across multiple vaults. This approach can process a subset of columns in parallel while maintaining DRAM row buffer locality. Furthermore, all the vaults are cooperatively processing a single event at a time, thus minimizing the processing latency of events. Additionally, this scheme does not require duplication of matrices and also maintains proper event ordering. One concern with this approach is that

it requires accumulating partial sums across vaults. Our evaluation shows that the inter-vault partial sum accumulation add moderate overhead (25% - 37%). But even with this overhead, the latency improvement is substantial.

As for mapping the rest of the operations, the GEMM kernel is implemented by repeating GEMV for each column of the input matrix. Although this approach appears inefficient as it does not utilize any form of cache blocking to leverage the reuse opportunities, it will not negatively impact the attainable throughput. This is because the machine balance of Gearbox is extremely low ($\sim$0.03 flop/byte with 164MHz ALUs and 49ns row activation time). As a result, Gearbox is fundamentally compute bound even for GEMV kernels and, by extension, on GEMM kernels. Tiling for increasing reuse will not provide any improvement in throughput unless we increase the number of processing units. Therefore, GEMM is implemented simply by repeating GEMV. The TGNN framework does not require GEMM operations unless we are batching the queries. Since GEMM in Gearbox does not provide any advantages over multiple individual GEMV operations, query batching is, in fact, undesirable as it increases the query latency without providing any throughput benefit.

The time encoding and most of the activations used by LSTM/GRU and GAT require calculating transcendental functions. As Gearbox ALPU does not support these functions, these operations are handled by the cores at the logic layer. ReLU activations are mapped to ALPUs as they are simple comparisons. Elementwise operations are cooperatively handled by the cores in the logic layer as well.

The performance of Gearbox is estimated by leveraging the simulation framework of Pulley [119]. Here, we use an analytical model for regular operations (e.g., GEMV) and simulation otherwise (e.g., feature aggregation of neighbors). We assumed that all operands, except for the weight matrices, are loaded into the shared scratchpad memory before processing.

Newton [22]

Newton is a near-memory accelerator proposed by SK Hynix and primarily targets acceleration of the GEMV operations of machine learning workloads. Newton puts several MAC units in a SIMD fashion (number of MAC units matched with DRAM columns) in every bank of an HMB2E-like memory stack. The weight matrix is stored in a chunk-interleaved manner, where the first matrix row's first chunk is followed by the second matrix row's first chunk, and so on. The input vector is stored in a global buffer and is broadcasted to the bank-level compute units a single chunk at a time, where the result gets reduced by a parallel adder tree. The width of the chunk is made the same as the DRAM row width to take advantage of the spatial locality. The first chunk of all the matrix rows is processed first, followed by the second chunk of all matrix rows, etc. This approach provides maximum reuse of the input vectors. Similar to Gearbox, GEMM operations are performed with repeated GEMV. Elementwise operations, activations, and time encodings are performed by

the host. Performance estimation is derived using the performance model provided by the authors in the original paper [22].

tFPGA [151]

This approach proposes a model-architecture co-design for accelerating the TGN framework [142] on FPGA. The design consists of multiple independent computing units. Each compute unit supports memory state updates using GRU and embedding using a simplified version of the graph-attention mechanism. The GRU is implemented using three 8×8 systolic arrays[3] for efficient GEMM operations corresponding to the update, reset, and memory gates of GRU. For the embedding, unlike the multi-head attention mechanism used by TGN that involves computing the key, value, and query matrices, tFPGA uses a simplified approach that only considers the temporal separation of the neighbors for calculating the attention weights, thereby eliminating a major portion of the computations. This embedding function is realized using a 16-lane multiply-adder tree for feature aggregation and one 8×8 systolic array for feature transformation. For neighbor sampling, tFPGA samples a fixed number of the most recent neighbors, similar to our approach. Time encoding is realized by a coarse-grained loop-up table. Further optimizations are done by pipelining all the stages and using a dedicated edge prefetcher. The HLS code of this approach is open-sourced [166]. However, we faced compilation issues when trying to generate the FPGA bitstream using the published code and, therefore, opted to use the performance model provided by the authors in their paper.

6.5.4 Throughput and latency results

Figure 6.6 presents the throughput and latency results for the TGN-attn model, and Figure 6.7 presents the results for the TGN-sum model. Here, TGN_PNM_feat uses only the feature-based partitioning scheme, while TGN_PNM_hybrid uses the hybrid partitioning scheme. TGN_PNM_hybrid provides the best throughput and latency across the benchmarks in both models. Table 6.4 summarizes the average throughput gain and latency reduction observed by the TGN_PNM_hybrid across the datasets.

Our results show that CPU and GPU perform worst, both in terms of latency and throughput. High latency in the case of GPU is expected as GPU architecture is optimized primarily for throughput and not latency. We attribute the low throughput of GPU for this particular workload to the small batch size. Increasing the batch size from 200 to 1000 increased the throughput of GPU by 3.7x on average while having a moderate impact on latency, which is increased by 1.4x. However, increasing batch size to improve throughput may not be feasible in practical scenarios, as user-facing interactive applications tend to have strict latency

[3]Authors have evaluated tFPGA on two FPGAs: Xilinx Alveo U200 and Xilinx ZCU104, with different numbers of compute elements due to FPGA resource constraint. We use the configuration of the more powerful Xilinx Alveo U200 for evaluation.

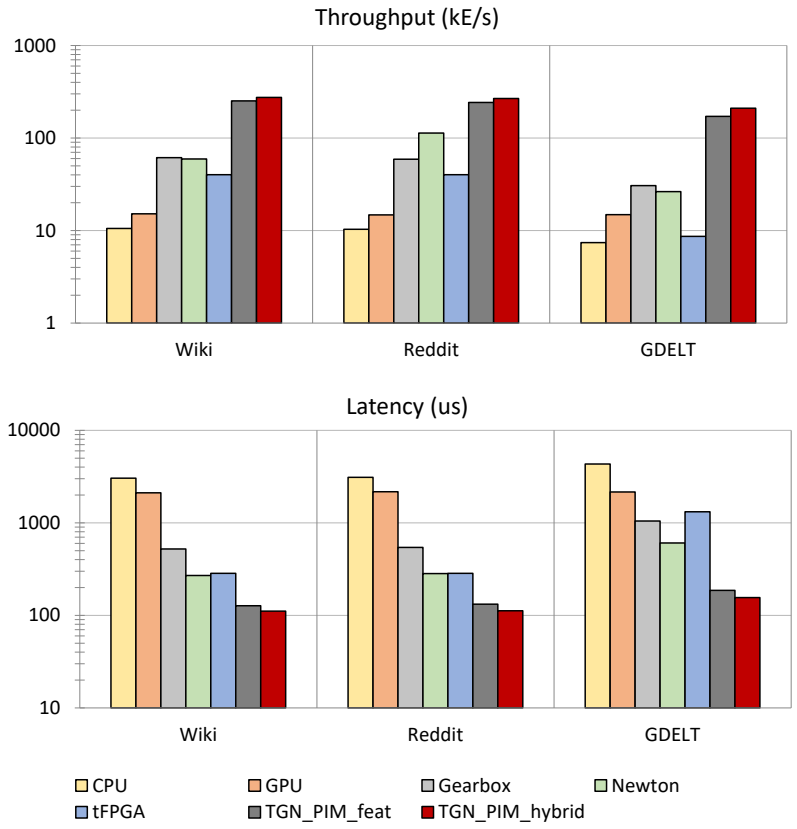


Figure 6.6: Throughput and batch processing latency for TGN-attn.

constraints. Furthermore, increasing the batch size also means that the graph state will update less frequently, and therefore, embedding will be performed using stale data and can negatively impact accuracy. Besides, the maximum batch size is limited by the capacity of the on-chip buffers for tFPGA and our approach.

TGN_PNM_hybrid provides a substantial performance gain over the subarray-level and bank-level PIM architectures. Note that all these three architectures has almost the same number of MAC units: TGN-PNM has $(16 * 16 + 16) * 32 = 8704$ MACs, Gearbox has 8192 ALUs, and Newton also has 8192 MACs. Despite having similar number of MAC units, for the TGN-attn model our approach has 5.2x higher throughput than subarray-level Gearbox and 4.4x higher throughput than bank-level Newton. There are a few key advantages of our approach that enables this throughput gain: i) VPUs of TGN-PNM runs at a much higher clock frequency than both Gearbox and Newton's compute units. This is becuase DRAM transistors in the memory layers are designed for low cost and leakage. The logic die uses high-performance transistors [11]. Although, note that we have used 164 MHz for Gearbox, which the authors reported for a 32-bit ALU, not 16-bit. Therefore, the attainable frequency of Gearbox could be higher for 16-bit FP. ii) Another advantage of TGN-PNM over the other two approaches is handling time encoding. In case of Gearbox and Newton,

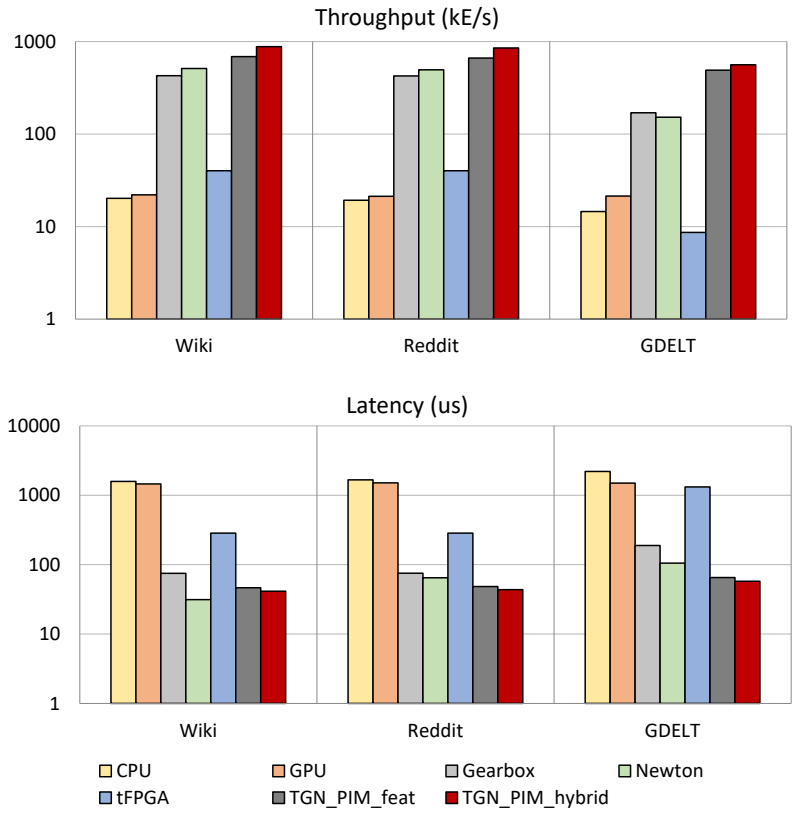


Figure 6.7: Throughput and batch processing latency for TGN-sum.

their processing elements have to be extremely simple to meet the strict area/power overhead budget of only about 20% [22] or otherwise loose capacity [152]. Thus, those two approaches cannot accommodate units for transcendental functions needed by time encoding. As a result, time encoding needs to be performed by host in case of Newton and at the logic layer in case of Gearbox. The time encoding itself is not a bottleneck, however, as time encoding sits at the intermediate stages, data have to move frequently in/out of their subarrays/banks. iii) As Newton only supports broadcast mechanism (one of the operands in their bank-level SIMD unit always comes from the broadcast buffer), it cannot handle elementwise operations efficiently. In case of Gearbox, elementwise operations may require re-layouting the data in the intermediate steps. iv) In case of Newton, underutilization can occurs if the matrix dimension is less than the total number of banks [22], which is often the case for the node states. Despite these shortcomings, Gearbox and Newton performs fairly closely to our approach for the more memory-intensive workload of TGN-sum, only trailing by 2.4x and 2.2x, respectively. Finally, our experiment shows that using hybrid storage scheme (i.e., feature-based partitioning for large matrix/vectors and traditional for others) improves the throughput slightly by 12-23% over the feature-based only approach of TGN_PNM_feat.

	Avg. througput gain		Avg. latency reduction	
	TGN-attn	TGN-sum	TGN-attn	TGN-sum
CPU	26.8	42.1	27.6	38.2
GPU	16.7	34.8	17.2	31.6
Gearbox	5.2	2.4	5.4	2.2
Newton	4.4	2.2	2.9	1.3
tFPGA	10.3	31.1	3.8	10.1
TGN_PNM_feat	1.14	1.23	1.17	1.12

Table 6.4: Average throughput gain and latency reduction of TGN_PNM_hybrid approach across the datasets.

Component	Count	Per unit area (mm^2)	Total area (mm^2)
Scalar core (ARM Cortex A-35)	1	0.68	0.68
Partial Sum Acc. Unit	1	2.25	2.25
Global NoC	1	1.43	1.43
GSPad	1	1.03	1.03
Memory controller and DDR PHY	32	0.18	5.90
Systolic array	32	0.25	8.08
SIMD: FP add & mul	32	0.02	0.51
SIMD: CORDIC	32	0.03	0.96
SIMD: Activation	32	0.15	4.68
VSPad	32	0.32	10.31
Total:			**35.83**

Table 6.5: Area estimation of TGN-PNM.

6.5.5 Area estimation

The area of processing elements and control logic are derived by synthesizing RTL models on the SAED 14 nm node using the Synopsys Design Compiler. The area of the SRAM buffers and scratchpad memories are modeled using CACTI-3DD [120] on a 32 nm node and then scaled to 14 nm. The memory controller and interconnect areas are modeled using McPAT [167]. The resulting area estimation for the major components are given in Table 6.5. Note that although the total amount of memory is the same for GSPad and VSPad (32kB * 32 = 1MB), VSPad requires a much larger area as it has to accommodate many read/write ports. The total estimated area of these components is $35.83mm^2$, which is 53% of the total die area of $68mm^2$ of an HMC stack [11]. This leaves around $32mm^2$ for the components that we haven't accounted for, such as I/O circuits, memory built-in self-test (MBIST), and features to support testing and debugging.

6.6 Related Work

With the emergence of machine learning workloads, a lot of hardware accelerators have been proposed by researchers targeting either the compute-intensive [168, 169, 170, 171, 172] or memory-intensive [152, 22] kernels of neural networks. Unfortunately, these approaches are not specifically designed for irregular access patterns exhibited by the neighborhood aggregation of graph neural networks. On the other hand, there are many hardware accelerators tailored for graph analytics workloads [103, 102, 10, 173, 174, 175]. However, these approaches cannot handle the compute-intensive portion of the temporal GNN workloads efficiently.

A few works cater to the unique hybrid nature of the GNN workloads. HyGCN [136] proposed an ASIC accelerator for static GCN, where the aggregation is scheduled on a series of SIMD units and node embeddings are processed by a collection of configurable systolic arrays. AWB-GCN [137] improved upon HyGCN by adding workload balancing mechanism for power-law graphs by distribution smoothing and row remapping. GCoD [138] proposed an algorithm/hardware co-design with separate micro-architectures for dense and sparse matrix. StreamGCN [140] targets streaming processing of many small graphs. FlowGNN [139] introduced support for edge embeddings. Recently, a PIM-based GNN accelerator has been proposed that accelerates the memory-bound kernels on the PIM side and delegates compute-bound kernels to the GPU [176]. Besides, a few general purpose PIM architectures can handle the GNN workloads efficiently if the graph is stored in specific sparse formats [9, 14]. However, these aforementioned approaches are only applicable to static GNNs where the graph topology does not change over time.

There is only one prior accelerator that we are aware of specifically targeting temporal GNN [151]. Authors in this work proposed an algorithm-hardware co-optimization, where they mapped the TGN framework [144] on a HBM-enabled FPGA. Optimizations proposed by this approach include hardware pipeline stages, look-up table based time-encoding function, double buffering and prefetching mechanisms. However, this approach can only accommodate a small number of MAC units due to FPGA resource constraint, limiting the potential speedup. We evaluated against this approach in Section 6.5 and observed vastly superior performance.

6.7 Conclusions and Future Work

In this paper, we proposed TGN-PNM, a near-memory architecture for accelerating TGNN workloads. In our approach, we placed a SIMD unit for memory-intensive operations and a systolic array for GEMM operations at the vault-level. Bottleneck arising from inter-vault communication during neighbor aggregation is avoided by partitioning the graph along the feature dimension, facilitating near perfect workload balance as well, which is very difficult to achieve on evolving graphs. Out evaluation against a few other architectures revealed that

near-/in-memory approaches performs the best for TGNN-type workloads. One interesting future direction would be to explore if it is beneficial to combine vault-level approach with bank-/subarray-level approach, where we keep the systolic array in the logic-layer for compute-intensive kernels, but move the processing of memory-intensive kernels closer to the memory.

Chapter 7

Conclusions and Future Work

Due to the ever-increasing gap between the memory bandwidth and processing capability of the modern processors, as well as the explosion of data-intensive applications, it is imperative to find a solution to the "memory wall" problem. In my thesis, I hypothesized that to avoid being bottlenecked by the memory wall, algorithms and data structures for data-intensive applications must be designed to leverage hardware features in the memory hierarchy, and in some cases, a software-hardware co-design approach is beneficial. In this dissertation, I presented five pieces of work in support of my hypothesis and made the following contributions.

My first work, Hopscotch, proposes a comprehensive memory benchmark suite that contains a carefully selected set of kernels with vastly different access patterns. The key contribution of this work is to enable identifying and isolating the impact of various memory centric hardware features on the system's performance. We extended the reach of Hopscotch by porting many of its kernels to GPU and FPGA [39] platforms. Furthermore, we provided a tool for empirically measuring the roofline plot of CPU and GPU, which can be troublesome to derive manually by going through the specification documents.

To support my hypothesis, I next target four memory-intensive workloads where the performance is bottlenecked by the available memory bandwidth. I propose solutions to circumvent the memory bottleneck by leveraging memory centric hardware features and hardware/software co-design. Accordingly, my second work, BigMap, focuses on fuzzing applications. We observed that increasing the coverage bitmap size to mitigate hash collision makes the fuzzer memory-bound, thereby hampering its ability to discover potential bugs within a given time budget. We overcame this bottleneck by introducing a two-level hashing scheme that consolidates bitmap traversal into a small area, reducing working set size and, in turn, cache pollution. We also leveraged the huge page support to reduce page walks caused by TLB misses, helping to improve the

performance further. We found that with our proposed two-level hash, it is possible to completely avoid the memory bottleneck even with an extremely large bitmap.

In my third work, I selected streaming graph processing as the target memory-bound application to accelerate. Graph applications are notorious for their irregular access patterns. Streaming graphs takes it further because the graph topology can change rapidly, making locality-improving steps such as vertex reordering infeasible. To tackle the issue, I proposed GraphTango, which aims to minimize the number of cache line accesses by introducing a hybrid storage format based on vertex degree and designing a cache-friendly hashing scheme. This hashing scheme maps subsequent probes to the hash table to occur within the same cache line if possible, thereby improving the cache locality characteristics of the applications. We proposed other improvements, such as a lock-free memory pool and a novel bucket-chaining-based workload balancing technique. With our proposed method, we observed up to 6.6x higher throughput over the next-best approach.

In my fourth work, we proposed Pulley, which targets large-scale sorting problems. Our key observation was that prior in-memory sorting algorithms used merge sort that eventually faced an unavoidable single-point merging bottleneck. We accelerated the application by hardware/algorithm co-optimization, where we leveraged one of our prior subarray-level PIM-based approaches and extended it to support LSB-first radix sort. One key challenge was random accesses when sorting within a local subarray. We proposed a design that eliminates the random accesses by introducing a pre-sorting step. With Pulley, we effectively removed the single-point merging bottleneck faced by the prior approaches and improved the performance by 13x-20x.

In my last work, TGN-PNM, we proposed a near-memory accelerator for temporal graph neural networks (TGNN). In one of our attempts, we tried accelerating TGNN by mapping it to our prior subarray-level PIM architecture Fulcrum [9, 14]. While it provided excellent performance, the Fulcrum-based approach was insufficient for a more compute-heavy TGNN workload. We proposed a vault-level near-memory architecture that contains a systolic array for compute-intensive GEMM operations and uses a SIMD unit for other operations. We added support for a feature-based partitioning scheme to avoid heavy inter-vault traffic. Our evaluation demonstrated average throughput gains of 26.8x over CPU, 16.7x over GPU, 5.2x over Gearbox, 4.4x over Newton, and 10.4x over tFPGA [151].

Overall, these works have shown that when an application is bottlenecked by memory bandwidth, it is possible to derive algorithmic or data structure changes to promote better utilization of memory-centric hardware features, such as cache hierarchy or TLBs, to reduce or eliminate the bottleneck (BigMap and GraphTango work). Otherwise, performing hardware/software co-optimization may be necessary to alleviate the memory bottleneck (Pulley and TGN-PNM work).

There are a few prominent future directions for my dissertation. The first is to explore the combination of complementary memory-centric architectures. For example, combining bit-serial architectures, such as

SIMDRAM [19] with word-level architectures, such as Fulcrum [9]. Both approaches have unique strengths and weaknesses and are suitable for accelerating different operations. Another example would be to combine vault-level PNM with bank-/subarray-level architectures. Second, it would be interesting to do a principal component analysis on Hopscotch kernels with different performance metrics as the features to identify potential overlaps between the kernels and also to determine the gaps that are not covered by the current kernels.

Publications

Peer-reviewed conference and journals

- Hopscotch: A micro-benchmark suite for memory performance evaluation.

  Alif Ahmed, Kevin Skadron.

  MEMSYS, 2019.

- BigMap: Future Proofing Fuzzers with Efficient Large Maps

  Alif Ahmed, Jason D. Hiser, Anh Nguyen-Tuong, Jack W. Davidson, Kevin Skadron.

  DSN, 2021.

- Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators.

  Marzieh Lenjani, **Alif Ahmed**, Mircea Stan, Kevin Skadron.

  ISCA, 2022.

- Pulley: An Algorithm/Hardware Co-Optimization for In-Memory Sorting.

  Marzieh Lenjani, **Alif Ahmed**, Kevin Skadron.

  CAL, 2022.

- PiMulator: a Fast and Flexible Processing-in-Memory Emulation Platform.

  Sergiu Mosanu, Mohammad Nazmus Sakib, Tommy Tracy, Ersin Cukurtas, **Alif Ahmed**, Preslav Ivanov, Samira Khan, Kevin Skadron, Mircea Stan.

  DATE, 2022.

Patents

- Methods, systems, and circuits for co-optimization for in-memory sorting.

  Marzieh Lenjani, **Alif Ahmed**, Kevin Skadron.

  U.S. Patent Application 63/366,125, 2022.

Under submission

- GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Updates and Analysis.
  Alif Ahmed, Farzana Ahmed Siddique (joint 1st author), Kevin Skadron.
  IJPP, 2024.
- Efficient Top-K Algorithm On GPU For Small K Values.
  Yiqing Yang, **Alif Ahmed**, Guoyin Zhang, Yanxia Wu, Kevin Skadron
  IJHPCA, 2024.

Planned submission

- TGN-PNM: A Near-Memory Architecture for Temporal GNN Inference on 3D-Stacked Memory. In this paper we would propose our ideas for accelerating temporal GNN as discussed in chapter 6.
  Alif Ahmed, Felix Lin, Jundong Li, Kevin Skadron

Prior publications and patents

- QUEBS: Qualifying event based search in concolic testing for validation of RTL models.
  Alif Ahmed, Prabhat Mishra.
  ICCD, 2017.
- Directed test generation using concolic testing on RTL models.
  Alif Ahmed, Farimah Farahmandi, Prabhat Mishra.
  DATE, 2018.
- Scalable Hardware Trojan Activation by Interleaving Concrete Simulation and Symbolic Execution.
  Alif Ahmed, Farimah Farahmandi, Yousef Iskander, Prabhat Mishra.
  ITC, 2018.
- Hardware Trojan Detection Using ATPG and Model Checking.
  Jonathan Cruz, Farimah Farahmandi, **Alif Ahmed**, Prabhat Mishra.
  VLSI Design, 2018.
- Automated Activation of Multiple Targets in RTL Models using Concolic Testing.
  Yangdi Lyu, **Alif Ahmed**, Prabhat Mishra.
  DATE, 2019. **(Best paper nominee)**
- Cache Reconfiguration using Machine Learning for Vulnerability-aware Energy Optimization.
  Alif Ahmed, Yuanwen Huang, Prabhat Mishra.

TECS, 2019.

- Efficient cache reconfiguration using machine learning in NoC-based many-core CMPs.

  Subodha Charles, **Alif Ahmed**, Umit Y Ogras, Prabhat Mishra.

  TODAES, 2019.

- **Patent:** Device and Method for Controlling Data Request.

  Seung-Beom Lee, **Alif Ahmed**, Joongbaik Kim, Kwon Soon-Wan.

  US Patent 10990444, 2021.

Bibliography

[1] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. Bonsai: High-performance adaptive merge tree sorting. In *ISCA*, 2020.

[2] Zheyu Li, Nagadastagiri Challapalle, Akshay Krishna Ramanathan, and Vijaykrishnan Narayanan. IMC-Sort: In-Memory Parallel Sorting Architecture using Hybrid Memory Cube. In *GLSVLSI*, 2020.

[3] Hybrid Memory Cube Consortium. Hybrid memory cube specification 2.1. https://www.hybridmemorycube.org/, 2015.

[4] Ramyad Hadidi, Bahar Asgari, Burhan Ahmad Mudassar, Saibal Mukhopadhyay, Sudhakar Yalamanchili, and Hyesoon Kim. Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube. In *Proceedings of the IEEE International Symposium on Workload Characterization* , pages 66–75, 2017.

[5] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 5th edition, 2011.

[6] J. Dongarra. An Overview of High Performance Computing and Benchmark Changes for the Future. *NIST NSCI Seminer*, 2016.

[7] S. McKee. Reflections on the memory wall. In *Computing Frontiers*, 2004.

[8] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH*, pages 20–24, 1995.

[9] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical in-situ Accelerators. In *HPCA*, 2020.

[10] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[11] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 symposium on VLSI technology (VLSIT)*, pages 87–88. IEEE, 2012.

[12] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. *arXiv preprint arXiv:2105.03814*, 2021.

[13] Yongkee Kwon, Kornijcuk Vladimir, Nahsung Kim, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, et al. System architecture and software stack for gddr6-aim. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–25. IEEE, 2022.

[14] Marzieh Lenjani, Ahmed Alif, Mircea R. Stan, and Kevin Skadron. Gearbox: A Case for Supporting Accumulation Dispatching and Hybrid Partitioning in PIM-based Accelerators. In *ISCA*, 2022.

[15] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, et al. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 350–352. IEEE, 2021.

[16] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 273–287, 2017.

[17] Lingxi Wu, Rasool Sharifi, Marzieh Lenjani, Kevin Skadron, and Ashish Venkat. Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching. 2021.

[18] Marzieh Lenjani, Alif Ahmed, and Kevin Skadron. Pulley: An algorithm/hardware co-optimization for in-memory sorting. *Computer Architecture Letters*, pages 109–112, 2022.

[19] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[20] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[21] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. To pim or not for emerging general purpose processing in ddr memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 231–244, 2022.

[22] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and TN Vijaykumar. Newton: A dram-maker's accelerator-in-memory (aim) architecture for machine learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–385. IEEE, 2020.

[23] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412. IEEE, 2015.

[24] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, M Arif Rahman, and Mircea R Stan. An overflow-free quantized memory hierarchy in general-purpose processors. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 203–215. IEEE, 2019.

[25] Po-An Tsai and Daniel Sanchez. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–242, 2019.

[26] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.

[27] Alif Ahmed and Kevin Skadron. Hopscotch: a micro-benchmark suite for memory performance evaluation. In *Proceedings of the International Symposium on Memory Systems*, pages 167–172, 2019.

[28] Alif Ahmed, Jason D Hiser, Anh Nguyen-Tuong, Jack W Davidson, and Kevin Skadron. Bigmap: Future-proofing fuzzers with efficient large maps. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 531–542. IEEE, 2021.

[29] Alif Ahmed, Farzana Ahmed Siddique, and Kevin Skadron. Graphtango: A hybrid representation format for efficient streaming graph updates and analysis. *arXiv preprint arXiv:2212.11935*, 2022.

[30] Marzieh Lenjani, Alif Ahmed, and Kevin Skadron. Pulley: An algorithm/hardware co-optimization for in-memory sorting. *U.S. Patent Application 63/366,125,2022*, 2022.

[31] Rajeev Balasubramonian. *Innovations in the memory system.* Springer Nature, 2022.

[32] J. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *TCCA*, 1995.

[33] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX*, pages 279–294, 1996.

[34] P. Luszczek et al. The HPC Challenge (HPCC) benchmark suite. *SC*, 2006.

[35] P. Mucci and K. London. Low level architectural characterization benchmarks for parallel computers. *U. Tennessee, Tech. Rep. UT-CS-98-394*, 1998.

[36] T. Deakin and S. Smith. GPU-STREAM: Benchmarking the Achievable Memory Bandwidth of Graphics Processing Units. In *SC*, pages 3202–3216, 2015.

[37] P. Lavin et al. Spatter: A Benchmark Suite for Evaluating Sparse Access Patterns. *arXiv preprint arXiv:1811.03743*, 2018.

[38] E. Strohmaier and H. Shan. Apex-Map: A Global Data Access Benchmark to Analyze HPC Systems and Parallel Programming Paradigms. In *SC*, 2005.

[39] Sergiu Mosanu, Mohammad Nazmus Sakib, Tommy Tracy, Ersin Cukurtas, Alif Ahmed, Preslav Ivanov, Samira Khan, Kevin Skadron, and Mircea Stan. Pimulator: A fast and flexible processing-in-memory emulation platform. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1473–1478. IEEE, 2022.

[40] Basak et al. Saga-bench: Software and hardware characterization of streaming graph analytics workloads. In *ISPASS*, pages 12–23, 2020.

[41] J. McCalpin. STREAM2. *URL: https://www. cs. virginia. edu/stream/stream2*, 2019.

[42] T. Deakin et al. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In *HiPC*, pages 489–507, 2016.

[43] G. Wrigley et al. Memory benchmarking characterisation of ARM-based SoCs. *Computer Research and Modeling*, pages 607–613, 2015.

[44] S. Siamashka. TinyMemBench. *URL: https://github.com/ssvb/tinymembench*, 2019.

[45] A. Snavely et al. A Framework for Performance Modeling and Prediction. In *SC*, pages 1–17, 2002.

[46] V. Viswanathan et al. Intel Memory Latency Checker v2. *URL: https://software. intel. com/en-us/articles/intelr-memory-latency-checker*, 2019.

[47] M. Gottscho et al. X-Mem: A Cross-Platform and Extensible Memory Characterization Tool for the Cloud. In *ISPASS*, pages 263–273, 2016.

[48] J. McCalpin. STREAM Benchmark Reference. *URL: https://www.cs. virginia. edu/stream/ref. html*, 2019.

[49] U. Drepper. What Every Programmer Should Know About Memory. *Red Hat*, 2007.

[50] S. Williams et al. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab, 2009.

[51] Cristian Cadar et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[52] Alif Ahmed and Prabhat Mishra. QUEBS: Qualifying event based search in concolic testing for validation of RTL models. In *ICCD*, pages 185–192, 2017.

[53] Yangdi Lyu, Alif Ahmed, and Prabhat Mishra. Automated activation of multiple targets in rtl models using concolic testing. In *DATE*, pages 354–359, 2019.

[54] Alif Ahmed, Farimah Farahmandi, and Prabhat Mishra. Directed test generation using concolic testing on rtl models. In *DATE*, pages 1538–1543, 2018.

[55] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. In *URL: https://github.com/google/oss-fuzz/*, 2020.

[56] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity Development*, pages 157–157, 2016.

[57] Robert Swiecki. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. *URL: https://github. com/google/honggfuzz*, 2020.

[58] Michal Zalewski. American fuzzy lop, v2.52b. In *URL: https://lcamtuf.coredump.cx/afl/*, 2020.

[59] Shuitao Gan et al. CollAFL: Path sensitive fuzzing. In *Security and Privacy*, pages 679–696, 2018.

[60] Fuzzbench: Fuzzer benchmarking as a service. In *URL: https://github.com/google/fuzzbench*, 2020.

[61] laf-intel: Circumventing fuzzing roadblocks with compiler transformations. In *URL: https://clang.llvm.org/docs/SanitizerCoverage.html*, 2020.

[62] Jinghan Wang et al. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *RAID*, pages 1–15, 2019.

[63] Michal Zalewski. Technical whitepaper for afl-fuzz. In *URL: https://github.com/google/AFL/blob/master/docs/technical_details.txt*, 2019.

[64] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Security and Privacy*, 2019.

[65] Ankou benchmark sources. In *URL: https://github.com/SoftSec-KAIST/Ankou-Benchmark*, 2019.

[66] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, pages 489–506, 2017.

[67] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Security and Privacy*, pages 711–725, 2018.

[68] Clang sanitizer coverage. In *URL: https://clang.llvm.org/docs/SanitizerCoverage.html*, 2020.

[69] Joseph Naus. Probabilities for a generalized birthday problem. *Journal of the American Statistical Association*, pages 810–815, 1974.

[70] opt - llvm optimizer. In *URL: https://llvm.org/docs/CommandGuide/opt.html*, 2020.

[71] Ben Nagy. Crashwalk: Bucket and triage on-disk crashes. In *URL: https://github.com/bnagy/crashwalk*, 2020.

[72] Fuzzbench report. In *URL: https://www.fuzzbench.com/reports/2020-08-23/index.html*, 2020.

[73] Marc Heuse et al. American fuzzy lop plus plus (afl++). In *URL: https://github.com/AFLplusplus/AFLplusplus*, 2020.

[74] Shawn Embleton, Sherri Sparks, and Ryan Cunningham. Sidewinder: An evolutionary guidance system for malicious input crafting. *Black Hat, August*, 2006.

[75] Sanjay Rawat et al. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, pages 1–14, 2017.

[76] Insu Yun et al. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX*, pages 745–761, 2018.

[77] Dongdong She et al. Neuzz: Efficient fuzzing with neural program smoothing. In *Security and Privacy*, pages 803–817, 2019.

[78] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference.

[79] Caroline Lemieux et al. Perffuzz: Automatically generating pathological inputs. In *SIGSOFT*, pages 254–265, 2018.

[80] Yuekang Li et al. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *ESEC/FSE*, pages 533–544, 2019.

[81] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *ASE*, pages 475–485, 2018.

[82] Yunchao Wang et al. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, pages 36340–36352, 2019.

[83] Marcel Böhme et al. Directed greybox fuzzing. In *CCS*, pages 2329–2344, 2017.

[84] Mateusz Jurczyk. Comparecoverage. In *URL: https://github.com/googleprojectzero/CompareCoverage*, 2020.

[85] Christopher Salls et al. Exploring abstraction functions in fuzzing. In *CNS*, pages 1–9, 2020.

[86] Han et al. Chronos: a graph engine for temporal graph analysis. In *EUROSYS*, pages 1–14, 2014.

[87] Cheng et al. Kineograph: taking the pulse of a fast-changing and connected world. In *EUROSYS*, pages 85–98, 2012.

[88] Compeau et al. How to apply de bruijn graphs to genome assembly. *Nature biotechnology*, 29:987–991, 2011.

[89] Zerbino et al. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18:821–829, 2008.

[90] Grewal et al. Recservice: Distributed real-time graph processing at twitter. In *HotCloud*, 2018.

[91] Eksombatchai et al. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *WWW*, pages 1775–1784, 2018.

[92] Jounsup Park and Klara Nahrstedt. Navigation graph for tiled media streaming. In *ICME*, pages 447–455, 2019.

[93] Braun et al. Knowledge discovery from social graph data. *Procedia Computer Science*, 96:682–691, 2016.

[94] Borgman et al. Drowning in data: digital library architecture to support scientific use of embedded sensor networks. In *JCDL*, pages 269–277, 2007.

[95] Ediger et al. Stinger: High performance data structure for streaming graphs. In *HPEC*, 2012.

[96] Wole Jaiyeoba and Kevin Skadron. Graphtinker: A high performance data structure for dynamic graph processing. In *IPDPS*, pages 1030–1041, 2019.

[97] Iwabuchi et al. Towards a distributed large-scale dynamic graph data store. In *IPDPSW*, pages 892–901, 2016.

[98] Andrew McCrabb and Valeria Bertacco. Optimizing vertex pressure dynamic graph partitioning in many-core systems. *IEEE Transactions on Computers*, 70:936–949, 2021.

[99] Mariappan et al. Dzig: Sparsity-aware incremental processing of streaming graphs. In *EUROSYS*, pages 83–98, 2021.

[100] Feng et al. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *SIGMOD*, pages 513–527, 2021.

[101] Alif Ahmed, Farzana Ahmed Siddique, and Kevin Skadron. Graphtango: A hybrid representation format for efficient streaming graph updates and analysis. *IJPP (under submission)*, 2024.

[102] Hu et al. Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas. In *ICCAD*, pages 1–9, 2021.

[103] Ham et al. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO*, pages 1–13, 2016.

[104] Sundaram et al. Graphmat: High performance graph analytics made productive. *arXiv:1503.07241*, 2015.

[105] Gui et al. A survey on graph processing accelerators: Challenges and opportunities. *JCS&T*, 34:339–371, 2019.

[106] Celis et al. Robin hood hashing. In *SFCS*, pages 281–288, 1985.

[107] Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[108] ISO/IEC JTC 1/SC 22 technical committee. C++ standard. https://www.iso.org/standard/79358.html, 2020.

[109] Donald E Knuth. The art of computer programming, volume 3: Searching and sorting. *Addison-Westley Publishing*, 1973.

[110] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection. https://snap.stanford.edu/data/, 2014.

[111] Thibaut Planchon. Tessil github repository. https://github.com/Tessil/robin-map, 2022.

[112] Martin Ankerl. Robin hood hashing github repository. https://github.com/martinus/robin-hood-hashing, 2022.

[113] Google. Abseil github repository. https://github.com/abseil/abseil-cpp, 2022.

[114] Martin Ankerl. Hashmap benchmarks. https://martin.ankerl.com/2019/04/01/hashmap-benchmarks-01-overview/, 2019.

[115] Matt Kulukundis. Designing a fast, efficient, cache-friendly hash table, step by step. *CPPcon. Standard C++ Foundation*, 2017.

[116] Mariappan et al. Dzig: Sparsity-aware incremental processing of streaming graphs. https://github.com/pdclab/graphbolt/tree/eurosys21-artifact, 2021.

[117] Feng et al. Risgraph github repository. https://github.com/thu-pacman/RisGraph, 2021.

[118] NVLink AND NVSwitch, The Building Blocks of Advanced Multi-GPU Communication . https://www.nvidia.com/en-us/data-center/nvlink/.

[119] FulcumV3. https://github.com/MarziehLenjani/FulcrumV3.

[120] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *DATE*, 2012.

[121] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *HPCA*, 2021.

[122] Yasuko Eckert, Nuwan Jayasena, and Gabriel H Loh. Thermal feasibility of die-stacked processing in memory. In *WoNDP*, 2014.

[123] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. TOP-PIM: Throughput-oriented programmable processing in memory. In *HPDC*, 2014.

[124] Federico Monti, Fabrizio Frasca, Davide Eynard, Damon Mannion, and Michael M Bronstein. Fake news detection on social media using geometric deep learning. *arXiv preprint arXiv:1902.06673*, 2019.

[125] Emanuele Rossi, Federico Monti, Michael Bronstein, and Pietro Liò. ncrna classification with graph convolutional networks. *arXiv preprint arXiv:1905.06515*, 2019.

[126] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.

[127] Kirill Veselkov, Guadalupe Gonzalez, Shahad Aljifri, Dieter Galea, Reza Mirnezami, Jozef Youssef, Michael Bronstein, and Ivan Laponogov. Hyperfoods: Machine intelligent mapping of cancer-beating molecules in foods. *Scientific reports*, 9(1):9237, 2019.

[128] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[129] Hongxia Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3165–3166, 2019.

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

[131] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[132] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *Advances in neural information processing systems*, 28, 2015.

[133] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. *Advances in neural information processing systems*, 30, 2017.

[134] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31, 2018.

[135] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31, 2018.

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

[137] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.

[138] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan Lin. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 460–474. IEEE, 2022.

[139] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1099–1112. IEEE, 2023.

[140] Atefeh Sohrabizadeh, Yuze Chi, and Jason Cong. Streamgcn: Accelerating graph convolutional networks with streaming processing. In *2022 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE, 2022.

[141] Mingi Yoo, Jaeyong Song, Jounghoo Lee, Namhyung Kim, Youngsok Kim, and Jinho Lee. Sgcn: Exploiting compressed-sparse features in deep graph convolutional network accelerators. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–14. IEEE, 2023.

[142] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs 2020. *arXiv preprint arXiv:2006.10637*, 2006.

[143] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962*, 2020.

[144] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

[145] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962*, 2020.

[146] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1269–1278, 2019.

[147] Priyank Faldu, Jeff Diamond, and Boris Grot. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–13. IEEE, 2019.

[148] Vignesh Balaji and Brandon Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 203–214. IEEE, 2018.

[149] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31. IEEE, 2016.

[150] Andrew McCrabb, Eric Winsor, and Valeria Bertacco. Dredge: Dynamic repartitioning during dynamic graph execution. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[151] Hongkuan Zhou, Bingyi Zhang, Rajgopal Kannan, Viktor Prasanna, and Carl Busart. Model-architecture co-design for high performance temporal gnn inference on fpga. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1108–1117. IEEE, 2022.

[152] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, et al. Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–26. IEEE, 2021.

[153] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. Do we really need complicated model architectures for temporal networks? *arXiv preprint arXiv:2302.11636*, 2023.

[154] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 international conference on management of data*, pages 2628–2638, 2021.

[155] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. Dyrep: Learning representations over dynamic graphs. In *International conference on learning representations*, 2019.

[156] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. Tgl: A general framework for temporal gnn training on billion-scale graphs. *arXiv preprint arXiv:2203.14883*, 2022.

[157] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. Time2vec: Learning a vector representation of time. *arXiv preprint arXiv:1907.05321*, 2019.

[158] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *The Journal of Machine Learning Research*, 21(1):2648–2720, 2020.

[159] Christian Weis, Norbert Wehn, Loi Igor, and Luca Benini. Design space exploration for 3d-stacked drams. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[160] JEDEC. High bandwidth memory 3 specification. https://www.jedec.org/standards-documents/docs/jesd238a, 2023.

[161] Ivan Fernandez, Ricardo Quislant, Eladio Gutiérrez, Oscar Plata, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. Natsa: a near-data processing accelerator for time series analysis. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 120–129. IEEE, 2020.

[162] UPMEM. https://www.upmem.com/.

[163] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[164] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Snap dataset collection: Jodie. https://snap.stanford.edu/jodie/, 2019.

[165] Rossi et al. TGN github repository. *URL: https://github.com/twitter-research/tgn*, 2020.

[166] Hongkuan Zhou, Bingyi Zhang, Rajgopal Kannan, Viktor Prasanna, and Carl Busart. Github repository, model-architecture co-design for high performance temporal gnn inference on fpga. https://github.com/zjjzby/TGNN-FPGA-IPDPS2022, 2022.

[167] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and many-core architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480, 2009.

[168] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH computer architecture news*, 44(3):367–379, 2016.

[169] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[170] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *44th International Symposium on Computer Architecture*, 2017.

[171] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

[172] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92. IEEE, 2019.

[173] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543. IEEE, 2018.

[174] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557. IEEE, 2018.

[175] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pages 457–468. IEEE, 2017.

[176] Hai Jin, Dan Chen, Long Zheng, Yu Huang, Pengcheng Yao, Jin Zhao, Xiaofei Liao, and Wenbin Jiang. Accelerating graph convolutional networks through a pim-accelerated approach. *IEEE Transactions on Computers*, 2023.