Accelerating Sparse Matrix Kernels with Co-Optimized Architecture

Submitted in partial fulfillment of the requirements for

The degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Fazle Sadi

B.Sc., EEE, Bangladesh University of Engineering and Technology

M.A.Sc., ECE, The University of British Columbia

M.Sc., ECE, Carnegie Mellon University

Advisors: Professor Larry Pileggi

Professor Franz Franchetti

Carnegie Mellon University

Pittsburgh, PA, USA

December 2018

Abstract

This dissertation presents an architecture to accelerate sparse matrix linear algebra, which is among the most important numerical methods for numerous science and engineering domains, such as graph analytics, in current big data driven era. Sparse matrix operations, especially for unstructured and large matrices with very few nonzeros, are devoid of guaranteed temporal and/or spatial locality making them inherently unsuitable for cache based general purpose commercial off-the-shelf (COTS) architectures. Lack of locality causes data dependent high latency memory accesses and eventually exhausts limited load buffer in COTS architectures. These render sparse kernels to run at a small fraction of peak machine speed and yield poor off-chip bandwidth saturation despite finely tuned software libraries/frameworks and optimized code. The poor utilization of compute and memory resources on COTS structures indicates significant room for improvement using existing technology. However, a computation paradigm that is not dependent on data locality is essential for sparse operations to achieve same level of performance that BLAS-like standards on COTS have delivered for dense matrix linear algebra.

An algorithm/hardware co-optimized architecture that provides a data locality independent platform for sparse matrix operations is developed in this work. The proposed architecture is founded on a fundamental principle of trading streaming bandwidth and compute to avoid high latency accesses. This principle stems from a counterintuitive insight that minimally required number of irregular memory accesses for sparse operations generally incur high traffic that is transferred at slow speed, whereas, more regular accesses can provide reduced traffic overall and faster transfer through better usage of block level data. This work finds that a scalable,

high performance and parallelizable multi-way merge network, which is absent in current literature, is the core hardware primitive required in developing our proposed architecture. Through both algorithmic and circuit level techniques, this work develops a novel multi-way merge hardware primitive that meaningfully eliminates high latency accesses for sparse operations. This work also demonstrates methodologies to avoid strong dependency on fast random access on-chip memory for scaling, which is a major limiting factor of current custom hardware solutions in handling very large problems.

Using a common custom platform, this work shows implementations of Sparse Matrix dense Vector multiplication (SpMV), iterative SpMV and Sparse General Matrix-Matrix multiplication (SpGEMM), which are core kernels for a broad range of graph analytic applications. A number of architecture and circuit level optimization techniques for reducing off-chip traffic and improving computation throughput to saturate extreme off-chip steaming bandwidth, provided by state of the art 3D stacking technology, are developed. Our proposed custom hardware is demonstrated on ASIC (fabricated in 16nm FinFET) and FPGA platforms and evaluated against state of the art COTS and custom hardware solutions. Experimental results show more than an order of magnitude improvement over current custom hardware solutions and more than two orders of magnitude improvement over COTS architectures for both performance and energy efficiency. This work is intended to contribute through a software stack provided by GraphBLAS-like [1] standards where broadest possible audience can utilize this architecture using a well-defined and concise set of matrix-based graph operations.

Acknowledgements

First and foremost, I would like to thank my supervisors Professor Larry Pileggi and Professor Franz Franchetti. I sincerely consider myself fortunate to have such well suited combination of supervision, without which this interdisciplinary research wouldn't have been possible. Larry and Franz provided me great flexibility and guidance to preserve and enjoy graduate school. At the same time they presented me ample opportunities to evolve as an independent researcher. To me both of them are great mentors.

I would also like to thank my committee members Professor James C. Hoe and Dr. Aydın Buluç for their important feedback on this thesis. I am specially thankful to James for guiding me as his own student when my project got involved with FPGAs. I have gained valuable perspectives from the insightful discussions I had with James about life within and outside academia. I am also grateful to Dr. Scott McMillan from Software Engineering Institute (SEI) for his mentorship during the last couple of years.

My first few years in graduate school would have difficult if not for my fellow group members Kaushik Vaidyanathan, Ekin Sumbul and Qiuling Zhu. Their work ethics and helpfulness allowed to me have a head start with my research. I would also like to thank Shaolong Liu, Jing Huang, Paul Brouwer, Jinglin (Kiki) Xu, Berkin Akin, Bishnu Prasad Das, Sam Pagliarini, Meric Isgenc and Joe Sweeney for their help, encouragement and friendship.

I am extremely grateful for the endless support that I received from the tightly knit group of friends at A level. I would like to individually thank Thom Popovici, Daniele Spampinato, Tze Meng low, Richard Veras, Guanglin Xu, Jiyuan Zhang and Anuva

Kulkarni. I don't think I would have such memorable experience at CMU without them. I will always cherish their friendship.

I would like to express my gratitude to DARPA, IARPA and SEI for creating and nourishing a collaborative ecosystem necessary in undertaking such interdisciplinary research. The 16nm ASIC fabrication was funded by DARPA CRAFT program and the initial research was sponsored by DARPA PERFECT program. I would like to acknowledge IARPA TIC program for sponsoring Stratix® 10 FPGAs.

Most importantly I would like to thank my parents, brother, sister, relatives and in-laws for their never-ending love and support. I will always be indebted to my parents for the unthinkable sacrifices they made to grow me as an individual.

I am also grateful to our two pet birds, Tweety and Boo. Somehow they are always able to impart their joyfulness to us during difficult moments of life.

Finally, my journey of graduate school would have been impossible without my best friend and wife Faria. It is for her that I will always cherish the memories that graduate school has gifted us.

To my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Sparse matrix algebra has proven to be an essential tool in graph analytics of various domains, such as social networks, machine learning, bioinformatics, data mining, etc. Widespread necessity for improved performance of sparse operations has led to a proliferation of software frameworks/libraries, such as OSKI [2], GraphMat [3], SuperLU [4], GraphLab [5, 6], Giraph [7], CombBLAS [8], Pregel [9], SociaLite [10], which reduces the "ninja performance gap" [11] between hand-optimized fine-tuned graph code and naively written code by researchers and developers from vastly different backgrounds. The expectation of user community is to have same level of benefits that Basic Linear Algebra Subprograms (BLAS) provided for the dense matrix algebra. However, while the high actual to peak performance ratio of dense operations on COTS platforms has been possible by BLAS like standards, it is largely due to the fact that the conceptual work horse for underlying traditional memory hierarchy of general purpose COTS platforms, i.e. temporal and spatial locality in data, is strongly

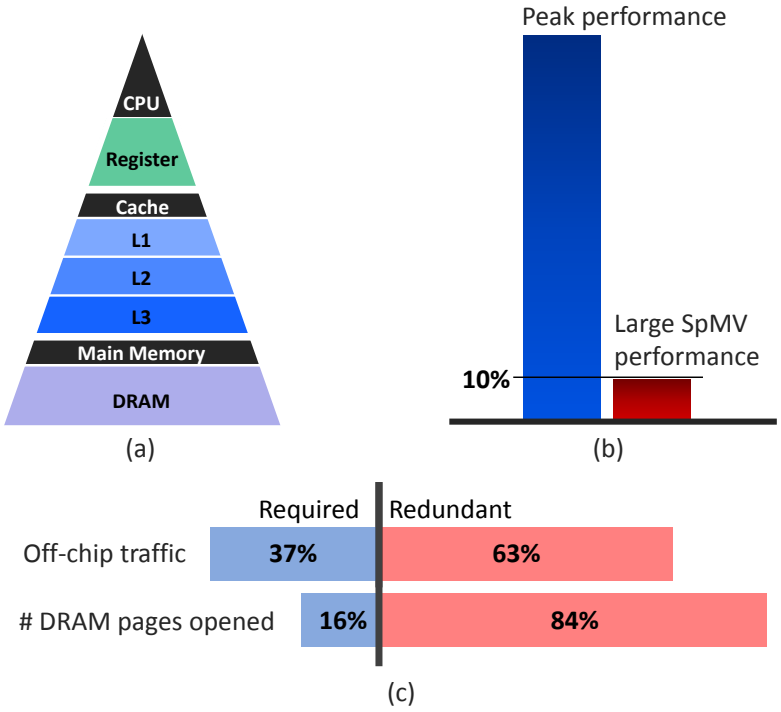


Figure 1.1: a) Traditional memory hierarchy used in general purpose architectures that is not suitable for sparse operations, b) Low peak to actual performance ratio, and c) Redundant traffic and inefficient DRAM access of SpMV on traditional architecture.

possessed by dense matrices. On the other hand, neither temporal nor spatial locality is guaranteed in sparse data. Even though locality to some extent is present in structured graphs, detection and exploitation of it often cause significant overhead, which also becomes increasingly difficult as graphs become larger and sparser. Hence, due to traditional memory hierarchy based COTS architectures not being inherently suitable for sparse operations, there is a large gap in performance between sparse matrix algebra and their dense counterpart.

Key Problem. The root cause of sparse matrix operation difficulties is data dependent random access to a huge address space. This translates to high latency load requests to DRAM. Moreover, general purpose architectures have limited depth

of load buffer that becomes filled with parallel load requests due to high latency in serving pending loads. Therefore, high enough outstanding load requests cannot be generated to hide latency despite parallel operations. Additionally, these load requests may not always be independent. As a result, sparse kernels experience low actual to peak performance ratio and poor off-chip bandwidth usage. For example, conventional implementation of Sparse Matrix dense Vector multiplication (SpMV) yields less than 10% of the peak performance on cache based COTS architectures [12–16]. Additionally, absence of locality causes redundant off-chip traffic and inefficient DRAM access due to block level data transfer. Figure 1.1(c) shows that an example SpMV incurs 63% redundant off-chip traffic that never takes part in actual computation. Furthermore, it wastes 84% costly DRAM page openings due to this redundant traffic.

From the above metrics it is evident that current CMOS and memory technology are under-utilized for sparse operations. However, it requires an architecture where efficient synergy between compute and memory access is independent of data locality and fine grained parallelism is ensured. Such architecture is absent in current literature and warrants research effort given its potential to contribute in sparse matrix algebra.

1.2 Thesis Contributions

This thesis attempts to work out a computation paradigm to replace data dependent high latency random accesses for sparse operations with sequential/streaming accesses. Our work is founded on a central counterintuitive insight that is -

Minimum number of memory accesses result in higher amount of traffic and latency bound computation, whereas more memory accesses incur less overall traffic and computation can run at much faster streaming bandwidth speed.

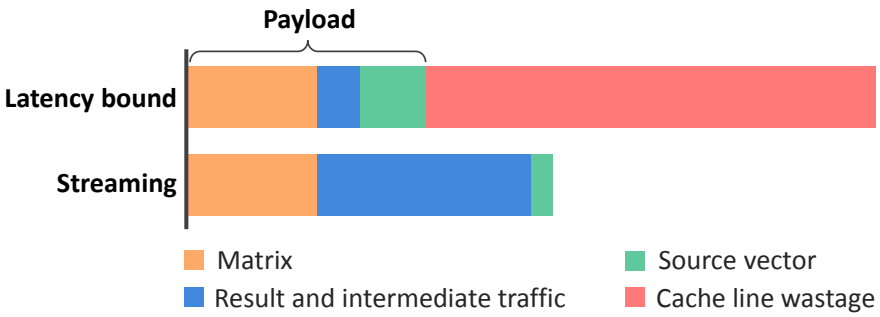


Figure 1.2: Off-chip traffic of SpMV using latency bound and full streaming algorithms.

This insight is depicted in Figure 1.2 for an example SpMV operation, where the bars show total off-chip main memory traffic for latency bound and streaming algorithms. Orange, blue and green colored portions of the bars together represent payload, i.e. data that takes part in actual computation. The red portion represents the wasted data due to cache line level block transfers from DRAM. Latency bound SpMV requires least number of load requests, hence minimum payload. On the other hand, with streaming algorithm, where all random accesses are converted to sequential accesses, SpMV generates more load requests. However, full streaming algorithm performs better because of the following reasons.

1. Despite total payload is higher than what it is for latency bound algorithm, overall traffic is less in streaming SpMV due to elimination of excess portion of cache line transfers, which gets evicted before any use.

2. As modern DRAM offers significantly greater streaming bandwidth than high latency random access bandwidth, the overall reduced traffic is transferred at significantly higher speed and computation runs much faster than latency bound SpMV. This trend in DRAM is also likely to continue as memory technology

trend over past two decades has shown improvement in streaming bandwidth at a rate that is at least square of the improvement rate in latency [17].

Therefore, ideal solution for sparse operation acceleration is to eliminate latency bounded accesses and pay with streaming bandwidth and more compute. However, turning this solution into practical implementation is not trivial. While it is theoretically possible to implement full streaming algorithms on traditional COTS architectures, advantages of eliminating latency boundedness are generally offset by one of these two reasons - a) fine grained parallelism requirement introduces computational complexity and eventually renders the entire operation to become compute bound, and b) off-chip traffic overhead of COTS compatible streaming algorithm becomes prohibitively large diminishing benefits of sequential access over random access.

This works presents an algorithm/hardware co-optimized architecture that can practically eliminate all off-chip high latency random accesses while being able to harness the benefits. A major finding of this dissertation is that a scalable and high throughput multi-way merge hardware primitive is the key in building such architecture. By developing a novel set of methodologies for implementing such multi-way merge network, this work provides a common platform that is co-optimized for off-chip traffic aware streaming algorithms for fundamental sparse matrix operations. This work demonstrates implementations of these following core kernels.

1. SpMV. We primarily focus on Sparse Matrix dense Vector multiplication (SpMV) as it is a dominant and versatile component used in numerous operations, such as large-scale linear solvers, and possibly the most notorious bottleneck causing very low fractions of peak processor performance. SpMV is also a difficult kernel to accelerate due to huge dense address space of dense vectors and required random access.

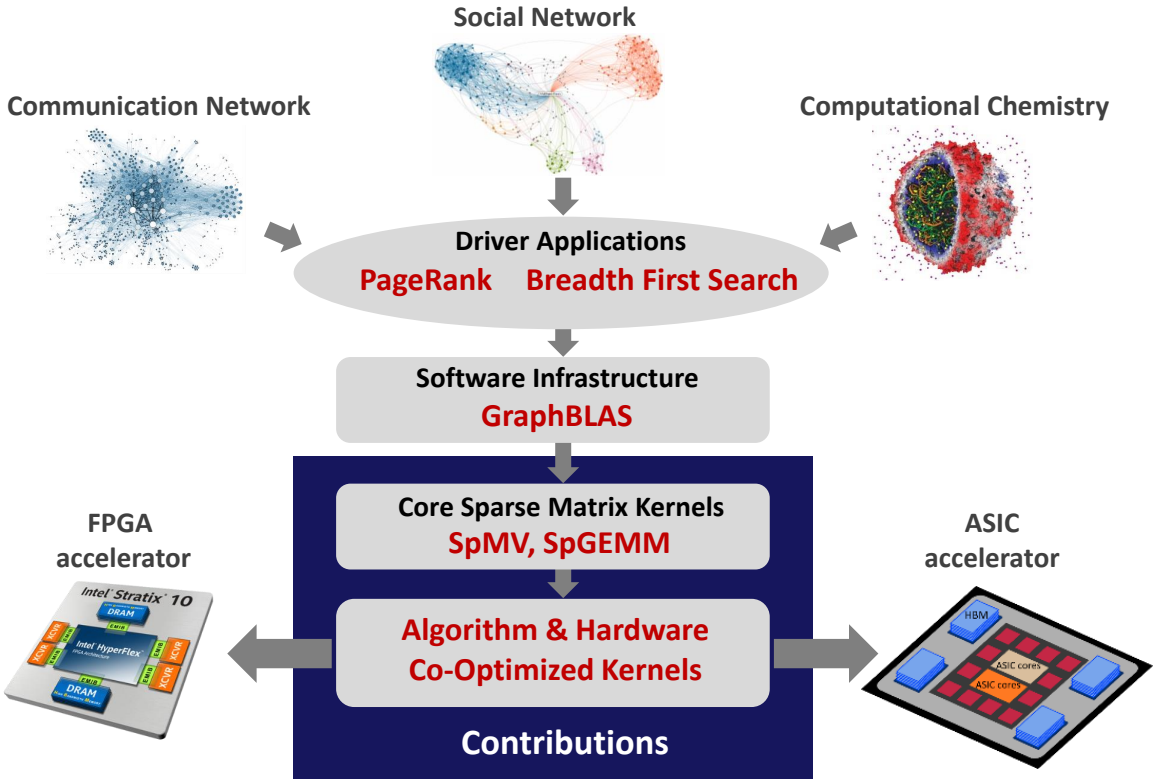


Figure 1.3: Contribution of this dissertation in graph analytics acceleration ecosystem.

This work demonstrates how our developed multi-way merge primitive can be leveraged to implement a SpMV algorithm possessing efficient data transfer characteristics, which is otherwise compute bound on COTS architectures and, hence, commonly discarded.

2. Iterative SpMV - PageRank. A large class of SpMV applications, such as PageRank, conjugate gradient, are conducted in iterative manner. This works demonstrates how iterative SpMV can be practically implemented yielding higher performance and efficiency than a single run using our proposed custom architecture. This work elaborates iterative SpMV implementation using PageRank as an example.

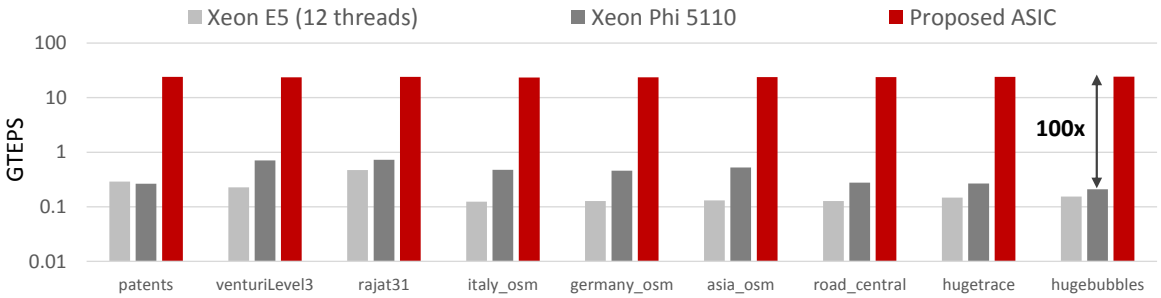
3. SpGEMM. Another class of graph analytics require sparse matrix multiplication with another sparse matrix or a set of sparse vectors, such as Betweenness

Centrality [19], All Pairs Shortest Path [20], Breath-First Search [21], which require Sparse General Matrix-Matrix multiplication (SpGEMM) as key operation. This work further demonstrates how off-chip traffic aware streaming SpGEMM can be implemented using a sparse accumulator requiring only sequential access, which is implemented using the multi-way merge hardware primitive developed in this work.

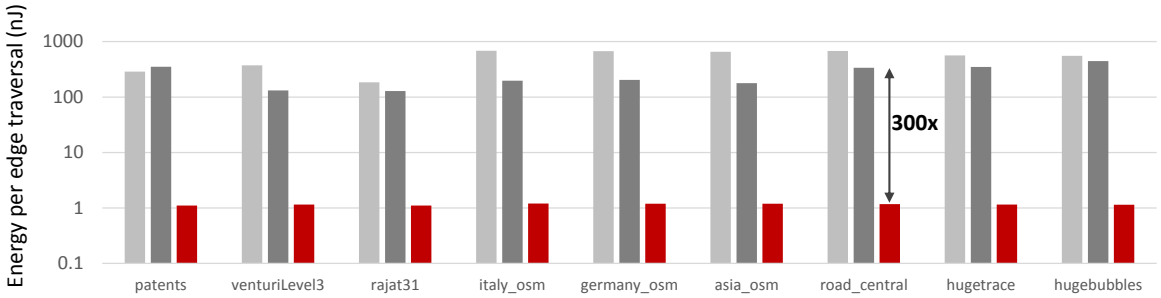
Our developed architecture is intended to contribute in the lower part of software-hardware stack within the ecosystem provided by GraphBLAS-like standards, as shown in Figure 1.3, where wide array of applications can be mapped using few fundamental sparse kernels. This solution has shown orders of magnitude improvement in performance and efficiency over state of the art solutions, an example of which is shown in Figure 1.4. For practical demonstration, our proposed architecture has been implemented on ASIC, which is currently under fabrication in 16nm technology and shown in Figure 1.5, and FPGA custom platforms.

1.2.1 Approach and Scope

In this dissertation we have taken a top-down approach, i.e architecture last, unlike most approaches in the literature that first assumes a given architecture characteristics and develop accelerator on top of it, which is a bottom-up approach. Our approach is to start with investigating the algorithmic properties that are required to yield high performance and efficiency. Later we have developed custom hardware to precisely address the requirements derived from the algorithm. Hence, this accelerator is termed as a 'co-optimized' solution. While developing these hardware primitives our priority is on scalability, technology trends and highest utilization of critical resources, such as off-chip bandwidth and on-chip storage.



(a) Performance comparison for SpMV acceleration.



(b) Energy efficiency comparison for SpMV acceleration.

Figure 1.4: Example of performance and energy efficiency improvement of proposed ASIC architecture.

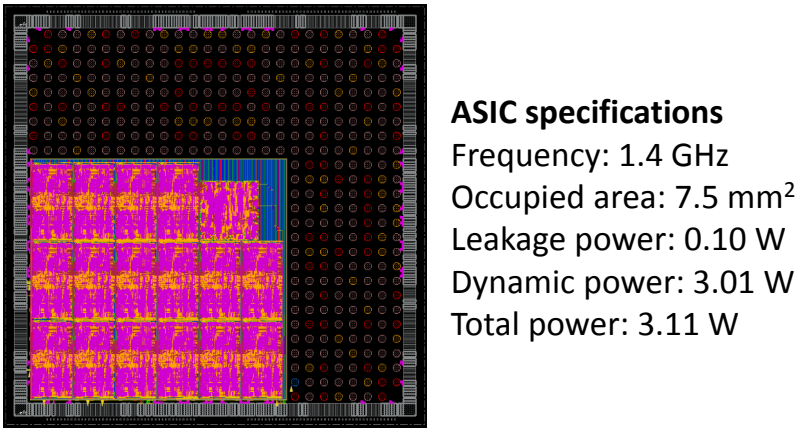


Figure 1.5: Custom ASIC for proposed accelerator.

We have focused on shared memory architecture in this work. It is because of the observation in memory technology trend that DRAM industry has been able to dramatically lower the cost [22] while storage capacity of single main memory

sub-system has increased substantially. Main memory system with storage capability in the order of tera-bytes is common now a days, which can easily store working data set for very large (multiple billion node) graphs. Hence, shared memory architecture should be able to handle very large sparse matrix kernels if computational core can support data set of this scale.

In this work we have exclusively considered large graphs ($\sim$ billion nodes) for which working data set is too large to fit on the fast on-chip memory, such as last level cache (LLC), Block RAM (BRAM) and scratchpad. We have assumed no structure in the data pattern, i.e. data is devoid of both temporal and spatial locality. In this study our focus is on graphs with high sparsity, e.g. average degree < 10. Furthermore, we have actively avoided costly preconditioning of the input data and complicated sparse formats in an effort to eliminate related overheads.

1.2.2 Effort vs Benefit

A practical concern in developing custom architecture is that often the impact is too narrow to justify the effort and cost. As there is no consensus on what the "building blocks" of graph processing should be [23], standard graph libraries provide hundreds of functions. It would be impractical to design custom hardware for all of them from the perspective of both usability and cost. However, a wide range of graph algorithms can be represented with sparse matrix algebra using just a handful of routines [24] and GraphBLAS [1] standard has been developed on this principle. GraphBLAS uses sparse matrix as the primary data structure and defines a concise set of sparse matrix routines that has been found to be useful for wide range of graph algorithms. These routines are defined on sparse matrices and, hence, work regardless of the complexity of

the underlying graph. Standardization of data structure and narrow set of operations allow wide range of audience to leverage decades of developed knowledge in linear algebra. While GraphBLAS provides the programming environment for a broadest possible audience, this work facilitates the underlying implementation by a single hardware platform that accelerates the fundamental sparse matrix kernels.

Additionally, there are issues over CMOS technology trend that has made custom hardware even more relevant for data intensive operations such as large sparse matrix kernels. With slowing down of Moore's Law, i.e. doubling of transistors in every 18 months, the free ride of achieving higher performance with new generation of technology node is not true anymore. On the other hand, non-Dennard technology scaling [25] has imposed a severe constraint on power budget for the entire gamut of system design. This has led to the "Dark Silicon" phenomenon, that allows to utilize only a fraction of the transistors available on-chip. Specialized hardware can mitigate these problems and provide both performance and energy benefits.

Thus the effort in development of custom architecture for sparse matrix operation is worth the possible impact both in terms timeliness and broad range of audience.

1.2.3 Key Developments

The key developments of this work are as following.

1. We present a study on SpMV streaming algorithms and identify an algorithm that conducts in two separate phases and has the potential to achieve significant performance and efficiency improvement for large problems. We demonstrate that despite having superior data communication characteristics, this algorithm is abandoned in literature due to computational requirements, which is not well

suited for general purpose processors. This work elaborates on how the proposed SpMV algorithm can be exploited by invoking various architecture and circuit level techniques enabled by custom hardware.

2. This work makes a finding that a scalable and high throughput multi-way merge network is a fundamental hardware primitive for both SpMV and SpGEMM operations. However, multi-way merge hardware scheme with these required properties is not available in current literature. We present a thorough investigation on the improvement of multi-way merge operation both at hardware implementation and algorithmic level. This work demonstrates novel circuit level techniques and parallelization scheme particularly suitable for sparse matrix operations that achieve scalability and high performance. As multi-way merge is a core operation in various other applications, our developed multi-way merge merge network scheme can have broader impact.

3. As off-chip communication directly affects performance and efficiency of sparse kernels, this work emphasizes on developing various techniques that reduce off-chip data transfers for both SpMV and SpGEMM. We demonstrate effective meta-data compression scheme that significantly reduces main memory traffic. Furthermore, we present a new block traversal scheme for 2D partitioned data for SpGEMM that becomes increasingly more beneficial in terms of main memory traffic as matrix becomes larger and sparser.

4. This thesis demonstrates practical methodologies in properly utilizing extreme off-chip bandwidth for sparse operations that is offered by state of the art 3D stacked main memory technology. Keeping various computational constraints

and limited power budget in mind, this work makes a comprehensive effort in optimizing traffic and achieving high sustained throughput.

5. We show methods to enhance performance for a number of common cases in sparse kernel applications. One of these cases is SpMV operation for iterative applications, such as PageRank. We demonstrate a technique that reduces off-chip traffic and almost doubles the computation throughput of iterative SpMV. Another case is when the input graph contains nodes with disproportionately large number of neighbors that are commonly found in power-law graphs. This work shows that by detecting these nodes and processing them using specially designed pipeline provide additional benefits. This work makes active effort in ensuring that these optimization techniques do not cause considerable overhead in terms of either hardware resource or data structure.

6. In this work we thoroughly demonstrate a streaming sparse accumulator as the computation engine for efficient SpGEMM, which uses the same hardware primarily designed for SpMV. We show that streaming sparse accumulator eliminates the requirements of complex data structure and expensive hardwares that are otherwise required for sparse accumulation. Furthermore, we demonstrate that streaming sparse accumulator enables off-chip traffic reduction opportunities.

7. A key contribution of this work is to demonstrate scalable acceleration methodologies that can handle large problems. One of the major constraints of state of the art sparse kernel accelerators is strong dependence on the on-chip fast memory to scale. This work presents computation methodologies that are resource aware and can handle significantly larger matrices (~ billion nodes vs few million

nodes) with considerably less on-chip memory relative to custom accelerators in literature.

8. The proposed custom hardware solution in this work is suitable for implementation in both ASIC and FPGA platforms. We have fabricated a 16nm FinFET based ASIC chip as a practical demonstration of our proposed accelerator. We further ported the ASIC design to Intel® Stratix® 10 FPGA. Our proposed accelerator in both ASIC and FPGA platforms demonstrates multiple orders of magnitude improvement over general purpose architectures in terms of both performance and energy efficiency. Comparison results with recent custom accelerators also demonstrate significant improvement in these metrics and considerably higher scalability.

1.3 Background

Matrix algebra has been a part of graph analytics since its inception due to the duality between canonical representation of graphs as a set of edges and vertices and a matrix [1]. However, it is until recently that sparse matrix operations based on already existing mature knowledge-base of linear algebra are found highly productive by graph analytics community [24, 26]. This renewed interest in sparse matrix algebra, substantially triggered by the success of Google PageRank algorithm [27], is attributed to recent advancements in sparse matrix data structures and algorithms along with numerous high-impact applications.

We will discuss the general difficulties related to sparse matrix operations next. Later in this section, we will also discuss relevant research efforts in literature for accelerating these operations

1.3.1 Sparse Matrix Kernel Acceleration Challenges

Fundamental challenges in achieving scalability, performance and efficiency for sparse matrix primitives in general are detailed below.

Low FLOP to memory access ratio. Fundamental sparse matrix kernels, such as SpMV and SpGEMM, are data intensive and have low flop to memory access ratio in general. This means for few computations it requires a relatively large quantity of data transfers. In modern architectures, data access is much more expensive than compute in terms of both energy and performance. For example, a double precision FP operation requires 0.5-50pJ, while reading the operands from cache memory at 20mm distance consumes 500pJ. Energy consumption is ever higher (6000pJ) if data is randomly accessed from off-chip main memory [28–31]. In terms of performance, modern microprocessors require around 200 cycles to randomly access main memory whereas FP multiplier takes only four cycles [32].

Inefficient Computation. Sparsity implies meta-data, huge address space and complicated load-store, which cause inefficiency in general purpose computing. As found by the authors of [33], among all the instructions of sparse matrix operation more than 94% are responsible for traversing the graph, e.g. finding relevant neighbors of a node, and loading arguments for computation. This also incurs a high energy overhead. While an arithmetic operation requires 0.5-50pJ, scheduling instructions in modern core consumes 2000pJ [29, 34].

Poor Bandwidth Usage. Off-chip memory bandwidth is one of the most important resources of modern systems and sparse operations on general purpose processors generally render bandwidth utilization to be poor. For latency bound sparse operations data is transferred from off-chip memory in large blocks, most of which remain unused. On the other hand, off-chip bandwidth bound sparse kernels often cause repetitive transfers of same traffic as the working data set is too large for on-chip memory. This incurs redundant traffic and negatively affects both performance and energy efficiency.

On-chip Memory Dependency for Scaling. This is possibly the most important constraint in handling large graphs for both general purpose and custom architectures. Data structure of sparse matrix requires additional vectors as meta-data and random accesses to these are required for fast traversal. Hence, it is a common technique for custom accelerators to store large portion of meta-data, such as vertex and edge properties, in the fast on-chip random access memory. This severely limits scaling capability as larger problem requires larger on-chip memory. Modern SOCs and micro-processors already use around 90% of available silicon area for on-chip memory [35]. With Moore's law currently hitting physical limits, radical increase in on-chip memory capacity is not likely in foreseeable future. Hence, state of the art shared memory custom accelerators have reported to only handle graphs with few million nodes [16, 33, 36], whereas multi-billion node graphs are now commonplace.

1.3.2 Related Work in Literature

Majority of the research efforts to improve sparse matrix operations has been dedicated in the software paradigm targeted for general purpose architectures. A commonality among many of these efforts is the focus in trying to extract and exploit temporal/spatial

locality and execute by fine grained parallelism. Using sophisticated sparse formats to capture and to provide ease of access to neighboring data is one approach in exploiting locality. Another approach is to preprocess or precondition the matrix to induce locality using techniques such as register blocking and matrix reordering, which are expensive operations in general.

General Purpose Architecture

Authors of [37] have proposed to use blocked column-major and blocked row-major Compressed Sparse Row (CSR) formats, namely BCM-CSR and BRM-CSR. These formats improve SpMV performance of structured sparse matrices as long as the data set fits in LLC. However, the authors reported sharp decline in performance when the data set becomes larger than LLC. Authors of [13] used register, cache and TLB level blocking to accelerate SpMV for multi-core environments. Recently, GPUs got much attention of researchers where they have explored many sophisticated approaches, such as fine grained parallel decomposition [38], model based auto-tuning [39] and transformation of matrix representation and tiling to increase temporal locality [40]. In [40], authors have utilized column re-ordering according to column length to accelerate SpMV based PageRank application on GPU. Researchers of [39] have proposed to blocked ELLPACK format for GPUs. In Blocked ELLPACK nonzeros are stored in blocks and ELL format is used to index the blocks, which introduces high memory overhead. Authors of [41] also proposed several complicated data formats, such as hybrid ELL/COO and packet (PKT) format, for SpMV on GPU. One-warp-one-row (1W1R) method is used in [41] where it was reported to improve performance by reducing the divergence among rows. In an IBM technical report [42], the authors have shown how compile-time and run-time optimizations on GPU can

accelerate SpMV. In [43], SpMV has been implemented using one-thread-one-row (1T1R) method on GPU. However, this method suffers from the variable length of the rows, i.e. different number of nonzeros per row. The authors have explored scan-based method to overcome the inefficiency of this work. Cell processors have been used to accelerate SpGEMM and SpMV in [44] by SIMD style implementation on eight synergistic processing elements (SPEs). Even though SPEs have neither caches nor efficient word-granularity gather/scatter support, the authors reported that the task parallelism afforded by the SPEs, the eight independent load store units, and the ability to stream nonzeros via direct memory access units overcame the limitations. This work uses blocked compressed sparse row (BCSR) format and register blocking to exploit locality.

Custom Architecture - FPGA

Among custom hardware platforms, FPGAs are more popularly pursued due to more accessibility and cost effectiveness. For example, researchers of the works [45–48] have implemented custom SpMV acceleration hardware using FPGAs. The maximum reported performance improvement in [45] is 29x over CPU. While graph dimension of most FPGA implementations in literature are limited to fractions of a million nodes due to full vector storage requirement in on-chip Static Random Access Memory (SRAM), authors of [45] reported to handle 9M node graph by using multiple off-chip SRAM blocks. FPGA accelerator in [36] takes an edge centric graph processing approach for PageRank acceleration, which can be implemented using SpMV, where the authors reported maximum 2.3M node graph using 8.4MB SRAM. The work in [48] and [47] report maximum performance of 1.4giga floating point operations per second (GFLOPS) and 4GFLOPS accordingly. These performances are very much comparable to those

for GPUs. GraphOps, a FPGA based modular hardware library for accelerating graph analytics presented in [49], focuses on optimizing graph layout for efficient layout in off-chip main memory and reported to operate on 16M node graph for SpMV and PageRank. However, the authors reported a poor throughput of 37 Million Edges Per Second (MEPS) and only 16% off-chip bandwidth utilization. In general, FPGAs are reported to have better efficiency than general purpose architecture, however, considerable performance improvement with FPGA for large scale SpMV is absent in the literature.

Custom Architecture - ASIC

A handful of ASIC accelerators are available in literature for sparse kernel acceleration, such as [16,18,33,50]. Authors of [16] has implemented SpMV and Sparse Matrix Sparse Vector multiplication (SpMSpV) kernels on 14nm simulated ASIC for accelerating machine learning workloads, such as classification, regression, recommendation and clustering. This work reported highest graph of 1.6M nodes, 117x improvement in performance per watt against multi-threaded software implementation on CPU and 60GB/s aggregated throughput using 4 independent accelerator blocks. Authors of [18] accelerated SpMV operation for support vectors machine (SVM) using an 14nm simulated ASIC similar to [16]. This work reported 14.7x and 22.9x performance improvement against baseline SpMV software running on Intel® Atom and Xeon core respectively. Energy efficiency improvement against Atom and Xeon core is 9x and 20x respectively. The authors of [33] presented another ASIC simulated using sub-28nm technology node for graph analytics. This work reported to achieve 1.7x-6.5x speedup and 50x-100x energy efficiency in comparison with software graph analytics frame work on 16-core Haswell Xeon CPU for applications such as PageRank, breadth-first

search (BFS), single source shortest path (SSSP) and collaborative filtering (CF). This work uses a large 32MB Embedded DRAM (eDRAM) scratchpad that is also very energy inefficient. The authors reported the eDRAM energy consumption to be 90% of what is consumed by the entire system. Despite using a large scratchpad, this work efficiently supports only 8M node graph. It is because this accelerator stores multiple tables related to all vertex and edge properties in the scratchpad for fast random access. The work in [50] presents a simulated 32nm ASIC along with 3D stacked architecture for SpGEMM acceleration. This work uses multiple Content Addressable Memory (CAM) blocks for random sparse accumulation during blocked SpGEMM computation. However, CAM is very expensive in terms of both silicon space and power. Furthermore, resultant output matrix of this work is not sorted.

1.4 Dissertation Outline

In this dissertation we will initially present the discussion of off-chip traffic aware algorithm and custom architecture development for SpMV kernel. In the later part of this thesis we will demonstrate how the developed architecture enables traffic aware algorithm implementation for SpGEMM, which can also be construed as sparse matrix multiplication with a set of sparse vectors, i.e. SpMSpV. The chapters of this dissertation are organized as following.

Chapter 2 will provide a study on streaming and non-streaming SpMV algorithms. We will elaborate our proposed off-chip traffic aware two phased algorithm, namely Two-Step SpMV, which can efficiently eliminate high latency accesses for SpMV. We will also present the comparative benefits and challenges related to Two-Step SpMV and demonstrate why a scalable multi-way merge hardware is essential for SpMV.

Chapter 3 will thoroughly demonstrate the development of scalable multi-way merge hardware which is essential for sparse matrix operation acceleration. We will drive the discussion using SpMV kernel as it poses more difficulties in achieving performance. This chapter will elaborate circuit level details and techniques that have been devised to improve performance and scalability of multi-way merge tree. Furthermore, a parallelization method will be illustrated that does not require more on-chip memory to scale. We will explain how this parallelization method dictates the partitioning scheme of the matrix. This chapter also progressively discusses available multi-way merge implementation techniques in literature and why those are not suitable for sparse matrix kernel acceleration.

Chapter 4 will detail the implementation of Two-Step SpMV kernel on custom hardware. We will present accumulation techniques to avoid bubbles in pipelined adders in case of large number of consecutive collisions. Furthermore, this chapter will demonstrate our proposed meta-data compression scheme and special hardware to efficiently process high degree nodes in power-law graphs. We will also show efficient technique for this accelerator implementation where the main memory sub-system has multiple channels.

Chapter 5 will present an optimization technique by iteration overlap for iterative SpMV operation using an example of PageRank application. We will demonstrate how throughput is increased and traffic is reduced by iteration overlap optimization. The second half of this chapter will elaborate on-chip fast memory usage of our accelerator and the effects of memory management on custom hardware solutions' scalability and efficiency. This discussion will show how our proposed architecture is capable of handling significantly larger graphs than what is possible with present architectures despite using less or similar fast memory.

Chapter 6 will present experimental results for SpMV kernel with our proposed accelerator implemented on both ASIC and FPGA platforms. We will show comparison against multiple general purpose architectures as benchmarks, which are CPU, co-processor (Xeon Phi) and GPU. Furthermore, we will compare against other custom hardware solutions available in literature. We will use Intel® MKL for implementation on CPU and co-processor. For the rest of benchmarks, we will use the reported metrics from relevant publications. As performance metrics we will use speedup in execution time and GTEPS. As energy efficiency metric, we will use energy per edge traversal.

Chapter 7 will describe how the developed accelerator can be used for SpGEMM operation. We will demonstrate how an efficient streaming sparse accumulator can be implemented using the multi-way merge network primarily developed for Two-Step SpMV implementation. We will further discuss the advantage of streaming sparse accumulator over current SpGEMM acceleration techniques using both custom and general purpose hardware. Additionally, we will present a new block traversal scheme for 2D partitioned matrix that significantly reduces off-chip traffic for SpGEMM computation. Lastly, we will present experimental results and compare with general purpose and custom hardware benchmarks in terms of performance and energy efficiency.

Chapter 8 will discuss possible directions of future work and present concluding remarks.

Chapter 2

Two-Step SpMV Algorithm

Contents

In this chapter, streaming algorithms for Sparse Matrix dense Vector multiplication (SpMV) operation on shared memory system are explored. We will elaborate our proposed streaming algorithm, namely Two-Step SpMV, that enables full main memory streaming by efficiently eliminating high latency accesses and reduces off-chip traffic for large problems. This algorithm also serves as an exemplary rationale behind the

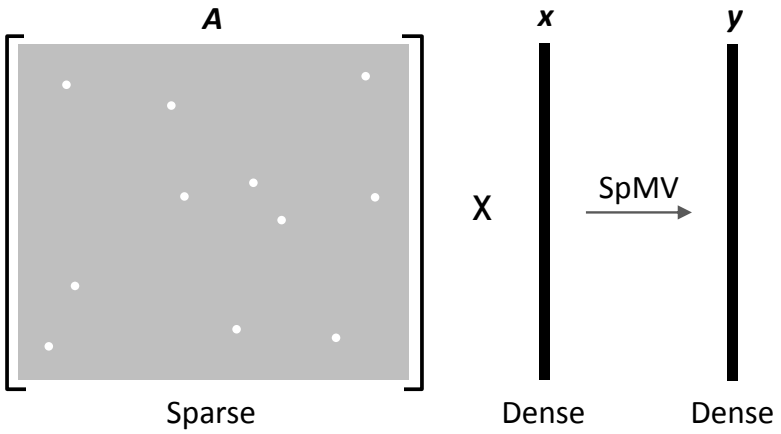


Figure 2.1: Sparse Matrix dense Vector multiplication (SpMV) operation.

development of scalable multi-way merge as a core hardware primitive for our proposed architecture in this work.

This chapter will also demonstrate comparative analysis of Two-Step algorithm against other bandwidth bound and latency bound algorithms. For evaluation and demonstration, we assume the Disk Access Machine (DAM) model [51] with two levels of memory hierarchy, on-chip storage (fast access) and off-chip main memory (slow access with block transfer). In this work we have focused on sparse matrices that are significantly larger than system's fast on-chip storage, e.g. last level cache (LLC), scratchpad, etc. Furthermore, we assume matrices with high sparsity where exploitation of temporal/spatial locality or any structure in the nonzero pattern is difficult.

2.1 SpMV Operation on Large Matrices

SpMV operation can be represented as $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{y}$, where $\boldsymbol{A}$ is a sparse matrix and $\boldsymbol{x}$ and $\boldsymbol{y}$ are respectively the source and resultant vectors as shown in Figure 2.1. SpMV operation has low floating point operation (FLOP) to memory access ratio, i.e. for

few computations we require a relatively large number of memory accesses. This is because for only two floating point operations, i.e. the multiplication and addition, three memory accesses to $\boldsymbol{A}$, $\boldsymbol{x}$ and $\boldsymbol{y}$ are required. Given current technologies, our compute capability significantly exceeds memory access speed, which is termed as memory-wall [52, 53]. Hence, resource requirement for SpMV operation is intrinsically imbalanced with respect to available system resources.

More importantly, SpMV requires random access to a large address space corresponding to either source vector $\boldsymbol{x}$ or reslutsnat vector y. When SpMV problem set become significantly bigger than fast memory, which is true for numerous real world problems, these random accesses are translated to high latency main-memory random accesses. As will be explained in detail later, random access to main memory makes SpMV latency bound for large problems that is substantially inefficient for both performance and energy. Matrix partitioning can help in overcoming this, however, at the cost of increased computational complexity and payload (data that actually takes part in computation).

Due to sequential main memory access, streaming SpMV algorithms are generally main memory bandwidth bound. Since off-chip main memory bandwidth is one of the most scarce resources of current architectures, performance and energy efficiency of SpMV operation critically depend on proper usage of it. Proper usage of off-chip memory bandwidth implies full streaming access and minimization of off-chip traffic. In this work, we propose a streaming SpMV algorithm that guarantees streaming main memory access and incurs relatively less off-chip traffic. This algorithm is named as Two-Step SpMV. We remain oblivious to computation related micro-architectural constraints in developing Two-Step SpMV and exclusively focus on data transfer characteristics.

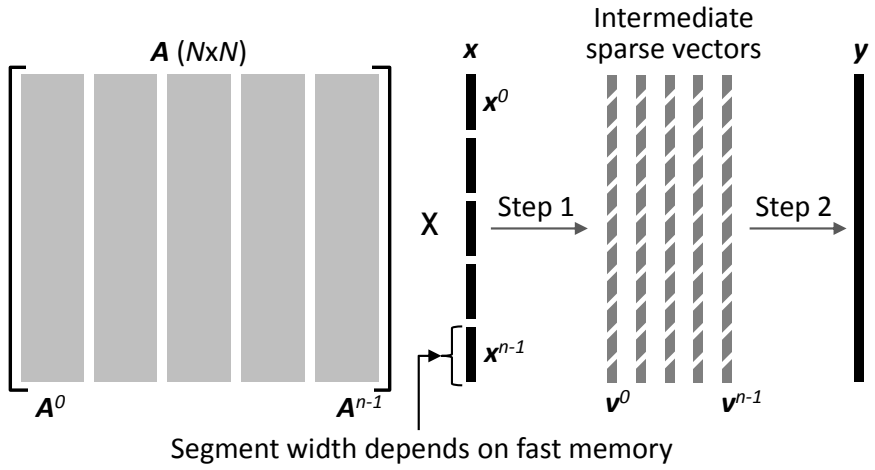


Figure 2.2: Two-Step SpMV algorithm.

2.2 Proposed Two-Step SpMV

As the name suggests, Two-Step SpMV is conducted in two separate steps, which is depicted in Figure 4.1. This algorithm is fundamentally depended on matrix partitioning into 1D column-blocks and multi-way merge operation. For Two-Step SpMV, initially the source vector $\boldsymbol{x}$ is first divided into multiple segments and matrix $\boldsymbol{A}$ is partitioned into vertical stripes, i.e. column blocks, as shown in Figure 4.1. The stripes of $\boldsymbol{A}$ is stored in a row major sparse format, e.g. Row Major Coordinate (RM-COO) or CSR as are depicted in Figure 2.3. Depending on the sparsity of matrix stripes one row-major format might be preferable to other. A detail discussion on preferable sparse formats is given in Chapter 4. Dimensional width of the stripes of $\boldsymbol{A}$ is same as the source vector segment length, which is directly proportional to the fast storage that can be randomly accessed in constant time. This means, segment width of $\boldsymbol{x}$ is determined such that fast memory storage is capable of holding the entirety of it.

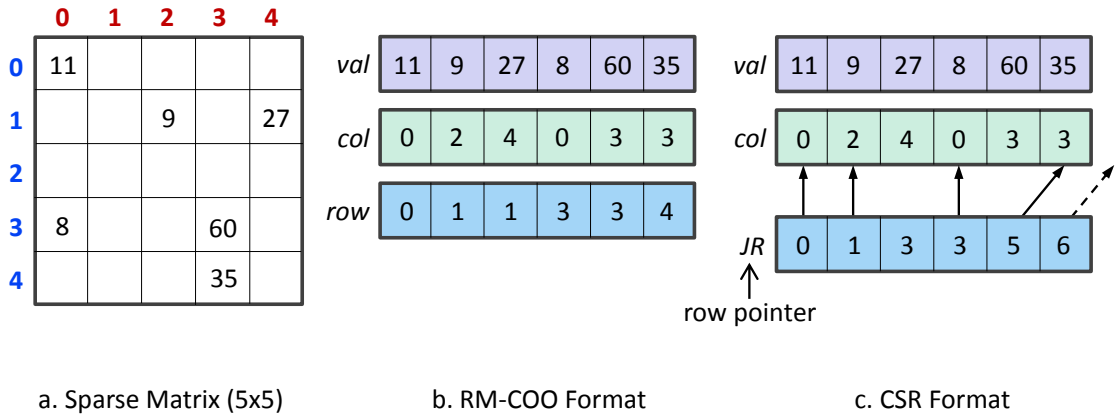


Figure 2.3: Row major sparse formats.

A pseudocode of Two-Step SpMV algorithm is given in Pseudocode 1. We discuss the separate steps of Two-Step SpMV below. At the beginning of SpMV operation, we assume that matrix stripes and source vector segments are resided in main memory.

Step 1

In this step of Two-Step algorithm, partial SpMV is conducted between the stripes of matrix ($\boldsymbol{A}^k$) and the corresponding vector segments ($\boldsymbol{x}^k$). At first, one vector segment is streamed out from main memory and stored in the fast storage. Afterwards, pertinent matrix stripe is streamed out from main memory to the computation core. Each nonzero in the matrix stripe is multiplied with the corresponding source vector element and accumulated with any existing partial results within the same matrix stripe of $\boldsymbol{A}$. As the matrix blocks are is stored in row major format, they are also sequentially traversed in increasing order of row indices of the nonzeros. Hence this partial SpMV operation on sparse matrix block produces a sparse vector that have nonzero elements sorted in ascending row indices of the matrix nonzeros, which are the

Pseudocode 1: Two-Step algorithm for large SpMV.

```
1  STEP 1
2  for k = 0 to n − 1 do
3  |  Stream in Matrix Column Block A^k
4  |  u ← 0
5  |  for All rows A^k_{i,:} with nnz > 0 do
6  |  |  for Each non-zero A^k_{i,j} in A^k_{i,:} do
7  |  |  |  Random access to vector segment x^k
8  |  |  |  u_i ← A^k_{i,j} · x^k_j + u_i
9  |  |  end
10 |  end
11 |  Sparsify u to v^k
12 |  Stream out v^k
13 end
14 STEP 2
15 for i = 0 to N − 1 do
16 |  for k = 0 to n − 1 do
17 |  |  Stream in v^k
18 |  |  y_i ← y_i + v^k_i [Multiway Merge]
19 |  end
20 end
21 Stream out y
```

meta-data (index) for the sparse vector nonzero elements. Thus, each sparse vector is effectively a sorted list and each nonzero in the sparse vector is a key-value pair where the key is the position index, i.e. the meta-data.

Since during the partial SpMV of $\boldsymbol{A}^k$ and $\boldsymbol{x}^k$ the intermediate sparse vector $\boldsymbol{v}^k$ is generated sequentially, it can be streamed out to main memory. In the first step, this process is conducted for all n matrix stripes. Therefore, after step 1 we end up with n intermediate sparse vectors residing in main memory. These intermediate vectors are stored in main memory as they are too large to fit in the fast on-chip storage.

Step 2

The second step of Two-Step SpMV is essentially a large multi-way merge operation among all the sorted lists, i.e. n intermediate sparse vectors. For this multi-way merge operation, $\boldsymbol{v}^k$ for any value of k is accessed sequentially. Hence, in step 2 all the intermediate vectors are streamed out from main memory to the computational core and merged to form the final resultant dense vector $\boldsymbol{y}$. As $\boldsymbol{y}$ is also generated in sequential manner, it can streamed to main memory.

2.2.1 Advantages of Two-Step SpMV

The main advantage of Two-Step SpMV is that it guarantees full main memory streaming access and incurs less off-chip traffic than other streaming and non-streaming algorithms for large problems. Therefore, Two-Step SpMV facilitate proper use of main memory bandwidth. We will elaborate on why this algorithm causes less traffic in later part of this chapter. Additionally, Two-Step enables various optimization opportunities such as meta-data compression for off-chip traffic reduction and throughput augmentation in iterative applications, which will be discussed in detail in Chapter 4 and Chapter 5. Furthermore, Two-Step SpMV does not require preconditioning (preprocessing) of the matrix and is not dependent on any exploitation of nonzero locality pattern. Hence, for large problems with high sparsity that are devoid of spatial and temporal locality or when preconditioning is expensive, Two-Step algorithm is especially advantageous.

In the following sections, we will evaluate Two-Step SpMV with other non-streaming and streaming SpMV algorithms and demonstrate the benefits of this algorithm in terms of off-chip data transfer characteristics.

2.3 Evaluation of Two-Step SpMV

As mentioned before, SpMV has a low FLOP to memory access ratio and, hence, its performance and efficiency are primarily dictated by memory access behavior. In this section, we evaluate our proposed Two-Step SpMV by exploring the fundamental differences in memory access characteristics for various families of SpMV algorithms while remaining oblivious to the compute requirements. We assume the DAM model [51] with two levels of memory hierarchy, fast on-chip storage and main memory (DRAM) with slow access and transfer in large blocks. Furthermore, we assume high sparsity where there is no exploitable spatial or temporal locality in the matrix data.

2.3.1 Non-Streaming SpMV

SpMV can be conducted by random access to either the source vector $\boldsymbol{x}$ or the resultant vector $\boldsymbol{y}$. Without any loss of generality, we assume random access to $\boldsymbol{x}$ for non-streaming SpMV. In this algorithm, the matrix is not partitioned and resultant vector elements are computed by direct inner product using the formulation $\boldsymbol{y}(i) = \sum_{j=0}^{N-1} \boldsymbol{A}(i,j)\boldsymbol{x}(j)$. Here, i and j are the row and column indices of matrix $\boldsymbol{A}$ and N is the dimension of $\boldsymbol{x}$. For large problems, $\boldsymbol{x}$ is significanly larger than the on-chip fast memory, e.g. LLC of CPU. As there is no locality in data, for every matrix element multiplication a cache miss is almost always unavoidable and request for $x(j)$ is forwarded to off-chip main memory. This phenomenon for an architecture with LLC and DRAM main memory is depicted in Figure 2.4.

In non-streaming SpMV every access to DRAM for $\boldsymbol{x}(j)$ is random and causes a new page (row buffer) to be opened every time due to large address space of $\boldsymbol{x}$. This causes high latency for almost every access and makes this algorithm main memory

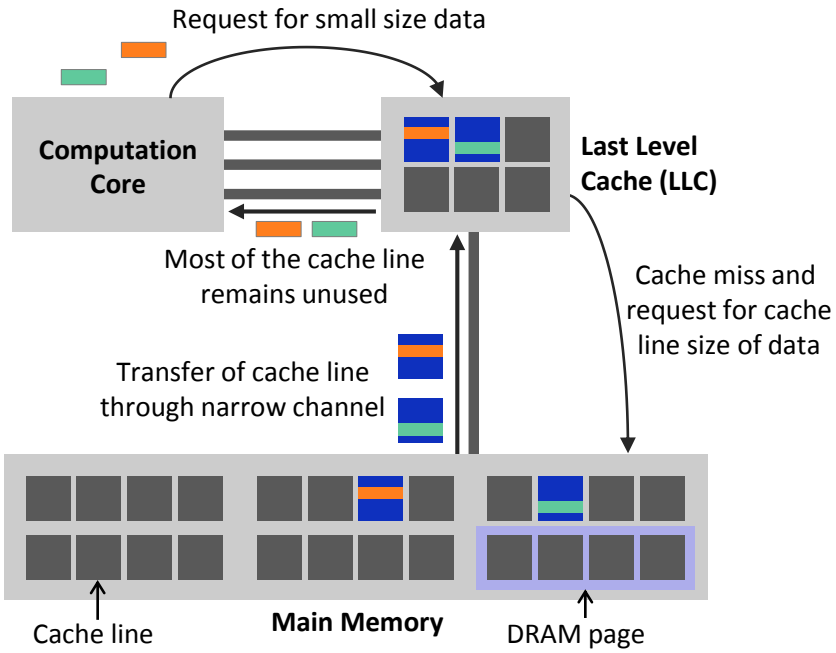


Figure 2.4: Redundant data transfer and random access in non-streaming SpMV.

latency bound. More importantly, data from main memory is transferred to LLC in cache line size blocks. As there is no locality in the data request, most of the cache line remains unused and eventually gets evicted. Thus a large amount of off-chip traffic that is transferred between DRAM and LLC using the scarce main memory bandwidth is wasted. An example off-chip traffic volume between DRAM and LLC for non-streaming and Two-Step SpMV is shown in Figure 2.5 for a randomly generated Erdos Rényi sparse matrix of size 80M×80M, average degree of 3 and 64B cache line. The fast storage size is varied to demonstrate its insignificance on the overall off-chip data transfer. It can be noticed that the redundant data (gray region) of non-streaming SpMV makes the overall data transfer significantly greater than Two-Step. This redundant data is due to the portions of cache line size blocks of $\boldsymbol{x}$ that are transferred

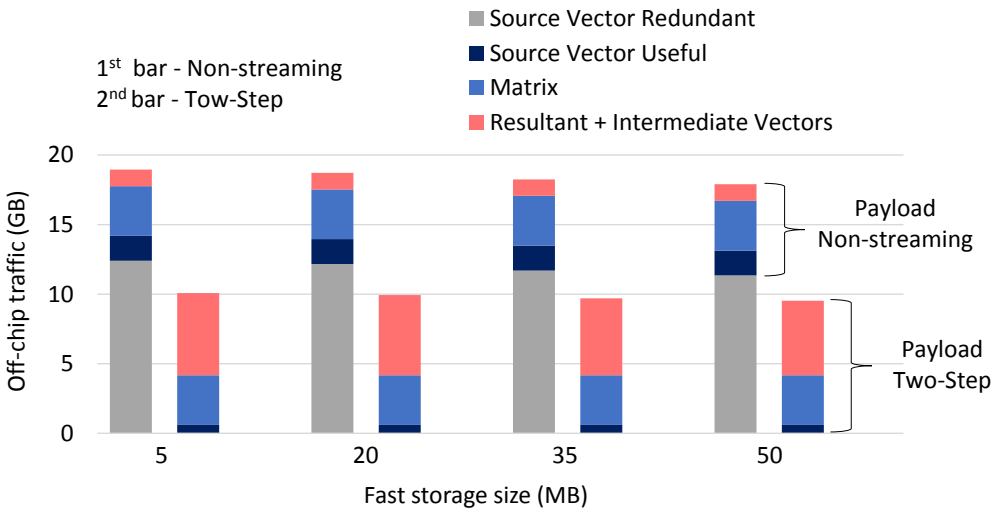


Figure 2.5: Off-chip traffic comparison between non-streaming and Two-Step SpMV.

to LLC and get evicted without ever being used. On the other hand, the blue and red colored regions represent the payload, i.e. the data that actually take part in computation.

A noteworthy observation in Figure 2.5 is that the payload for Two-Step SpMV is consistently greater than the latency bound non-streaming algorithm, despite incurring overall less traffic. This is because the intermediate sparse vectors ($\boldsymbol{v}^k$s) of Two-Step SpMV have to make a full round trip to and from main memory. Nevertheless, Two-Step SpMV provides several optimization opportunities to reduce the payload, which will be discussed in detail in Chapter 4.

2.3.2 Streaming SpMV

In this section we will compare the off-chip data transfer characteristics of two fundamental ways of conducting streaming SpMV. A high level mathematical model for off-chip traffic between the fast and slow memory levels in a single core scenario

in presented. However, this analysis is also applicable to multi-core shared memory scenarios.

An efficient SpMV kernel should be memory bandwidth bound [38] and, generally, the measure of success for a streaming SpMV algorithm is the fraction of peak bandwidth that can be achieved. Here, we consider the algorithms that achieves full main memory streaming access (i.e. utilizes peak bandwidth). Therefore, our measure of success would be the execution time, which is inversely proportional to the volume of off-chip data that is transferred between the memory hierarchy levels.

For streaming SpMV algorithms, large sparse matrices are commonly partitioned into blocks for which working data set fits in the fast memory. This avoids random access to main memory and facilitates parallelization. Figure 2.6 depicts a $N \times N$ matrix that is 2D partitioned into $m \times n$ blocks. A square matrix is considered to simplify the calculation without any loss of generality. Sparse matrix $\boldsymbol{A}$ has in the order of hN nonzeros, where h is the average degree. We denote S_m as the size of matrix element (including meta-data) and S_v as the size of source and resultant vector elements.

Independent of the sparse matrix block storage format and computation method, there are basically two ways to traverse the matrix blocks for conducting streaming SpMV. As Figure 2.6 shows, one of the ways is to traverse the blocks in row-major direction, while the another is to traverse in column-major direction. We name the family of algorithms following former as Row-major Block traversed Algorithms (RBA) and later as Column-major Block traversed Algorithms (CBA). Our proposed Two-Step SpMV falls under the family of CBA.

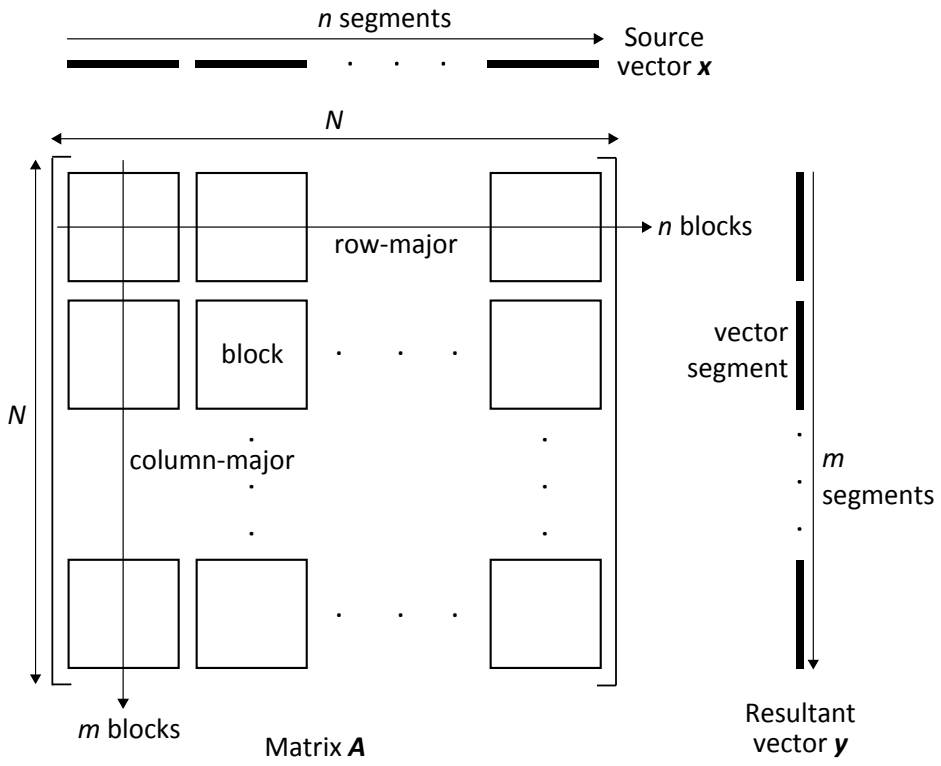


Figure 2.6: Matrix blocking and different ways of block traversal.

Row-major Block traversed Algorithms (RBA)

For RBA, as used in [54], the matrix block and relevant segment of the source vector ($\boldsymbol{x}$) can be streamed to the fast memory from DRAM. Block level SpMV can be conducted by sequential access to segment of $\boldsymbol{x}$. The partial resultant vector for each block is accumulated with the existing results from previous blocks as we traverse a row of blocks in the the 2D partitioned matrix. Thus, the entire source vector has to be streamed from DRAM for traversal of one row of the matrix blocks. As mentioned before, source vector $\boldsymbol{x}$ is significantly larger than fast memory storage and, hence, cannot be re-used for following block row traversal. On the other hand, the resultant vector portion needs to be stored in fast storage for random access since it is updated while a row of matrix blocks is traversed. Therefore, resultant vector segment

dimension (N/m) is dictated by the on-chip storage size. The off-chip traffic for RBA is shown in Table 2.1. For example, to compute on each block row of the matrix we need to stream hNS_m/m amount of matrix data from DRAM to computation core. For the entire computation on matrix $\boldsymbol{A}$, hNS_m amount of matrix data is streamed into the core from DRAM. Using the information of Table 2.1, we can calculate the total off-chip traffic for RBA (D^r) from the formulation given in Equation 2.1.

Table 2.1: Off-chip traffic volume between DRAM and on-chip memory for RBA (row-major block access)

Data source (direction w.r.t. chip)	Traffic per block row	Total data
Matrix (in)	hNS_m/m	hNS_m
Source vector (in)	NS_v	mNS_v
Resultant vector (out)	NS_v/m	NS_v

$$D^r = hNS_m + NS_v + mNS_v \tag{2.1}$$

Column-major Block traversed Algorithms (CBA)

In CBA, matrix blocks can be streamed from DRAM to computation core similar to RBA. However, in this case segment of source vector $\boldsymbol{x}$ is randomly accessed as the m matrix blocks in column-major direction are traversed. Therefore, the fast storage is occupied by the source vector segment. Another important difference of CBA from RBA is that each column of matrix blocks will produce an intermediate resultant vector. In the worst case (when there is no reduction), an intermediate vector will have the same number of elements as the total Number of Non-Zeros (NNZs) in all the matrix blocks in that column. The intermediate resultant vector needs to

be updated/accumulated as we move to the next column of matrix blocks. As the intermediate resultant vector becomes more dense each time it is updated, storing it on chip does not remain feasible. There are multiple ways this intermediate result can be updated. One way is to store the intermediate vector in DRAM and update its elements by randomly accessing DRAM, but this will cause inefficient bandwidth usage and make SpMV CBA a latency bound non-streaming algorithm. Another way is to stream the entire intermediate result from DRAM to computation core every time a new column of matrix blocks is traversed and conduct accumulation. Despite guaranteed DRAM streaming, this method will incur redundant off-chip traffic as most nonzeros in the intermediate result is not required to be updated. From the perspective of off-chip traffic entirely, a better way is to stream out the n intermediate vectors (one for each column block of $\boldsymbol{A}$) to DRAM as they are produced. Afterwards, stream back all the n intermediate vectors from DRAM to chip and apply reduction operations on them to get the final resultant vector $\boldsymbol{y}$. Table 2.2 presents the off-chip traffic for the operations in CBA. Further, we can deduce the total data transfer for CBA (D^c) as shown in Equation 2.2.

Table 2.2: Off-chip traffic volume between DRAM and on-chip memory for CBA (column-major block access).

Data source (direction w.r.t. chip)	Traffic per block column	Total data
Matrix (in)	hNS_m/n	hNS_m
Source vector (in)	NS_v/n	NS_v
Intermediate vec. (out & in) (assuming no reduction)	$2hNS_v/n$	$2hNS_v$
Resultant vector (out)	-	NS_v

$$D^c = hNS_m + NS_v + (2h+1)NS_v \tag{2.2}$$

Comparison: RBA vs CBA

From Equation 2.1 and Equation 2.2 we see that the first two terms (hNS_m and NS_v) are identical. The third terms in the equations make these algorithms distinct from each other. For RBA, mNS_v appears due to reading the entire source vector from DRAM m times. On the other hand, $(2h+1)NS_v$ represents the intermediate results of CBA that make a round-trip from computation core to and from DRAM. By investigating further, we can see that the key factors are m and $(2h+1)$ for RBA and CBA respectively. If $m < (2h+1)$, then RBA is preferable as it will have less data transfer than CBA. Otherwise, CBA is preferable when $m > (2h+1)$.

Table 2.3: Speedup of CBA over RBA on typical systems.

System	On-chip/DRAM (MB/GB)	N	m	Execution time (s)		Gain with CBA
				RBA	CBA	
GPU (Tesla GP100 w/ HBM2)	4/16	250M	500	2.76	0.05	50x
FPGA (Stratix 10 w/ HBM2)	16/32	500M	250	2.00	0.08	26x
Desktop (Skylake Core i7)	8/64	1B	1e3	236.7	2.35	100x
Server (Haswell Xeon E7)	45/1500	23B	4.2e3	7.7e3	12.6	417x

We can derive m from system configuration, matrix dimension (N) and sparsity (h). For any system with DRAM capacity of C^{DRAM}, the largest matrix dimension N that it is able to handle, given h, can be calculated from Equation 2.3. We assume

that the matrix, source vector and resultant vector occupy the main memory entirely.

$$C^{DRAM} = \overbrace{hNS_m}^{\text{matrix}} + \overbrace{NS_v}^{\text{source vector}} + \overbrace{NS_v}^{\text{resultant vector}}$$
$$\Rightarrow N = \frac{C^{DRAM}}{(hS_m + 2S_v)} \tag{2.3}$$

RBA needs to store $\frac{1}{m}$th portion of the resultant vector in the fast on-chip storage. Therefore, we can express the capacity of the on-chip memory C^{chip} as the following.

$$C^{chip} = \frac{NS_v}{m}. \tag{2.4}$$

Replacing N from Equation 2.3 to Equation 2.4 gives us closed form expression for m as shown in Equation 2.5.

$$m = \frac{C^{DRAM}}{C^{chip}} \frac{S_v}{(hS_m + 2S_v)} \tag{2.5}$$

From the above equation we see that m is directly proportional to the ratio of slow(DRAM) and fast(on-chip) memory storage size. This ratio for typical systems is generally much greater than the pertinent sparsity parameter, i.e. average degree h, of matrices for large SpMV problems. This means m is generally much larger than $2h + 1$. For example, h can be in the order of $1 \rightarrow 10$ for large sparse matrices (e.g. social network graphs such as *Twitter lists*, *YouTube* from KONECT library [55]). On the other hand, m is in the order of thousands for systems capable of handling large SpMV problems.

For a number of practical systems, we have explored the value of m, largest matrix dimension N and execution time of both the algorithms (assuming off-chip data is

transferred at peak bandwidth) for $h = 3$. Table 2.3 summarizes our findings. We see that m is in the order of hundreds for systems with relatively small DRAM size. For systems with large DRAM capacity, m is in the order of thousands, which is significantly larger than $(2h + 1) = 7$. Therefore, RBA incurs much more off-chip traffic than CBA, specially for large SpMV problems. For example, a Haswell server with 1.5 TB DRAM can operate on 23B× 23B matrix with average degree of 3. If RBA is used, the source vector of 23B elements is transferred from DRAM to chip $m = 4200$ times (600x larger than $2h + 1$), whereas for CBA the source vector is transferred only once. As the overhead for CBA, due to intermediate results, is considerably less than transferring source vector m times, we get 417x improvement in execution time with CBA. Furthermore, m is directly proportional to matrix dimension N, which makes RBA less scalable than CBA. Therefore, our analysis shows that column-major matrix block traversal based streaming SpMV algorithms, such as our proposed Two-Step algorithm, are preferable for large SpMV problems as it can meaningfully eliminate high latency random accesses without incurring prohibitively large off-chip traffic overhead.

2.4 Challenges

2.4.1 Multi-way Merge for Two-Step SpMV

Thus far we remained oblivious to the computation requirement of Two-Step SpMV algorithm. While our proposed Two-Step can be theoretically be implemented on COTS architectures like CPU or GPU, it may be very inefficient. This is because of the second step of the algorithm where thousands of sorted lists with millions of

elements have to be merged to produce one resultant dense vector. Multi-way merge of this extent is essentially compute-bound [45,56] and difficult to accomplish with COTS architectures efficiently due to bad scaling behavior. Difficulties in implementation of high performance/throughput multi-way merge operation is one of the key reasons for discarding algorithms similar to Two-Step SpMV despite having efficient memory access behavior. Additionally, when the matrix gets larger multi-way merge operations becomes exponentially resource intensive. One of the key contributions of this work is the development of a scalable and large multi-way merge custom hardware that is essential for efficient implementation of Two-Step SpMV for large problems.

2.4.2 Multi-way Merge for other Sparse Matrix Operations

The second phase of Two-Step SpMV is essentially a sparse accumulation, which is also a fundamental operation for Sparse General Matrix-Matrix multiplication (SpGEMM). Multi-way merge primitive offers a way of streaming sparse accumulation using only sequential memory access instead of random sparse accumulation, which is commonly practiced in literature. In Chapter 7, we will describe in detail how SpGEMM can be benefited from streaming sparse accumulation in terms of hardware efficiency and off-chip traffic reduction of large problems. Furthermore, SpGEMM can be considered as sparse matrix multiplication with a set of sparse vectors, i.e. Sparse Matrix Sparse Vector multiplication (SpMSpV). Therefore, an scalable multi-way merge network can serve as the core hardware primitive for an architecture to accelerate a range of fundamental sparse matrix operations.

2.5 Summary

This chapter elaborated our proposed Two-Step SpMV algorithm that possesses full streaming access behavior and incurs significantly less off-chip traffic than the streaming algorithm that is generally adopted in literature. We have also shown that Two-Step SpMV produces overall less off-chip traffic than latency bound algorithm despite having more payload. Two-Step SpMV also presents several opportunities to further reduce off-chip traffic, which will be described in Chapter 4. However, to handle large problems Two-Step SpMV requires a scalable and high performance multi-way merge network that is difficult to efficiently implement in both general purpose and custom hardware. This multi-way merge operation requirement is one of the main reasons for algorithms similar to Two-Step SpMV not being adopted by researchers despite having better memory access behavior. Furthermore, in Chapter 7 we will demonstrate how multi-way merge is also the core operation for other fundamental sparse matrix operations, such as SpGEMM. Next in Chapter 3 we will thoroughly elaborate the development of a scalable and high performance multi-way merge hardware required for our proposed custom architecture.

Chapter 3

Scalable Multi-way Merge

Contents

In Chapter 2 we have seen that a meaningful implementation of the Two-Step SpMV algorithm largely depends on the hardware design necessary for achieving performance

and efficiency. The most important hardware kernel required for Two-Step SpMV implementation is a scalable multi-way merge network. Later in Chapter 7 we will also see that multi-way merge operation is fundamentally important to accelerate other sparse matrix operations, such as SpGEMM and SpMSpV.

This chapter will progressively demonstrate the development of our proposed novel multi-way merge network, namely Hybrid Comparison Look Ahead Merge (HCLAM), which incurs significantly less resource consumption as scaled to handle larger problems. We will further demonstrate a parallelization scheme, namely Parallelization by Radix Pre-sorter (PRaP), which enables streaming throughput increase of the merge network without prohibitive demand of on-chip memory. This parallelization scheme is the main contributor is saturating extreme off-chip streaming bandwidth of 3D-stacked main memory used in the proposed accelerator of this dissertation. We will show circuit level details of proposed and state of the art multi-way merge schemes to elaborate fundamental techniques that enable high performance and scalability.

3.1 Scalability

From the perspective of hardware implementation and performance, scalability can be viewed from two distinct aspects, which are

1. Problem scaling

2. Technology scaling.

Figure 3.1 shows how problem scaling and technology scaling affect a multi-way merge binary tree implementation. Problem scaling is the phenomenon when the input data set becomes larger. For example, when the matrix dimension, i.e. number of nodes in

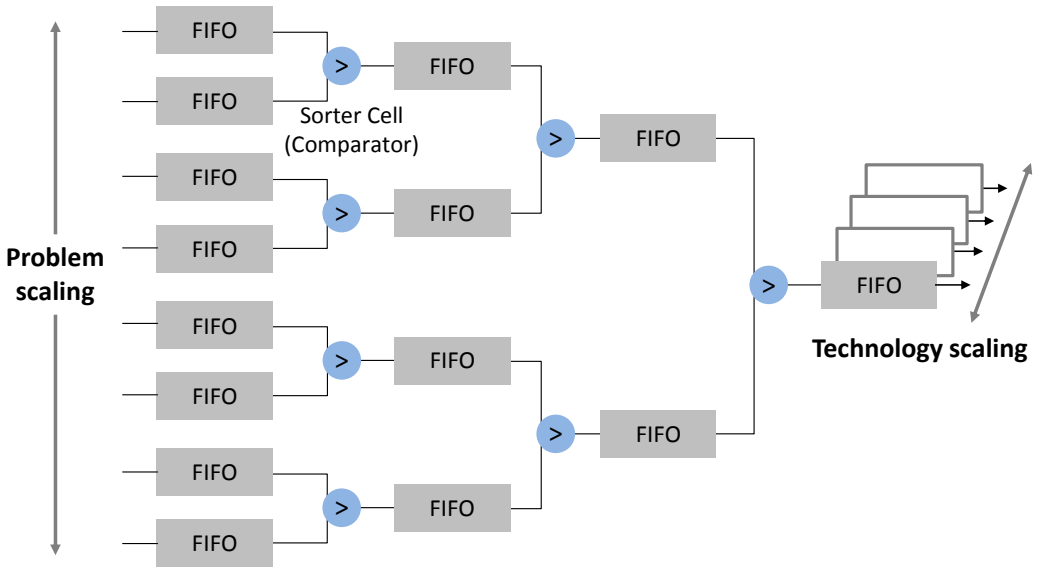


Figure 3.1: Binary tree implementation of naive multi-way merge network.

graph, become larger for SpMV, more lists are required to be merged in the second step of the Two-Step algorithm. As a result, required number of sorter cells and FIFOs grow exponentially. Hence for practical feasibility, hardware design should be able to handle growing matrix size without prohibitively requiring more hardware resources or sacrificing performance.

On the other hand, design of any algorithm or hardware is expected to take full advantage of new technologies with extended capabilities. For example, 3D stacked HBM technology enables extreme off-chip bandwidth. As many sparse matrix kernels should ideally be memory bandwidth bound, accelerators in this domain are expected to properly utilize the extreme bandwidth offered by this new technology. For a single multi-way merge tree hardware, where the maximum output rate is one element per cycle, delivering enough throughput to saturate such high bandwidth is another major challenge. Additionally, maintaining balanced throughput for multiple DRAM channels pose further challenges.

Multi-way merge solution that can practically address both of these scalability issues is absent in the literature. One of the major contributions of this work is to develop a multi-way merge hardware design that address both problem and technology scaling, while being practically feasible for custom hardware platforms, such as ASIC and FPGA. In the following sections, we separately discuss the challenges, implications and solutions for both problem and technology scaling in developing the multi-way merge hardware kernel.

3.2 Problem Scaling

3.2.1 Register FIFO based Multi-way Merge

Figure 3.2 depicts a basic hardware binary tree for merging K sorted lists, hence a K-way merge network. For this particular example, $K = 8$. Each element of the lists is a key-value pair, which we will refer as a 'record' from now on. By 'read' we mean accessing a certain entry in the FIFO and by 'dequeue' we mean updating the counter of the FIFO that is being read. Data 'write' and 'queue' both means the same in this context, that is making an entry in the buffer and updating the counter of a FIFO.

The basic building blocks of this tree are sorter cell (comparator) and FIFO. Each sorter cell compares the keys from the two connected FIFOs and dequeues the record (key-value pair) with the smaller key. To improve clock frequency, the merge tree is further divided into pipelined stages by storing the output of sorter cells in pipeline registers. Total number of stages can be calculated as $S = \log K + 1$ and stage number starts from '0' at leaf level. The most straight-forward hardware implementation uses register based FIFOs and a total of $K - 1$ sorter cells. For any particular pipeline

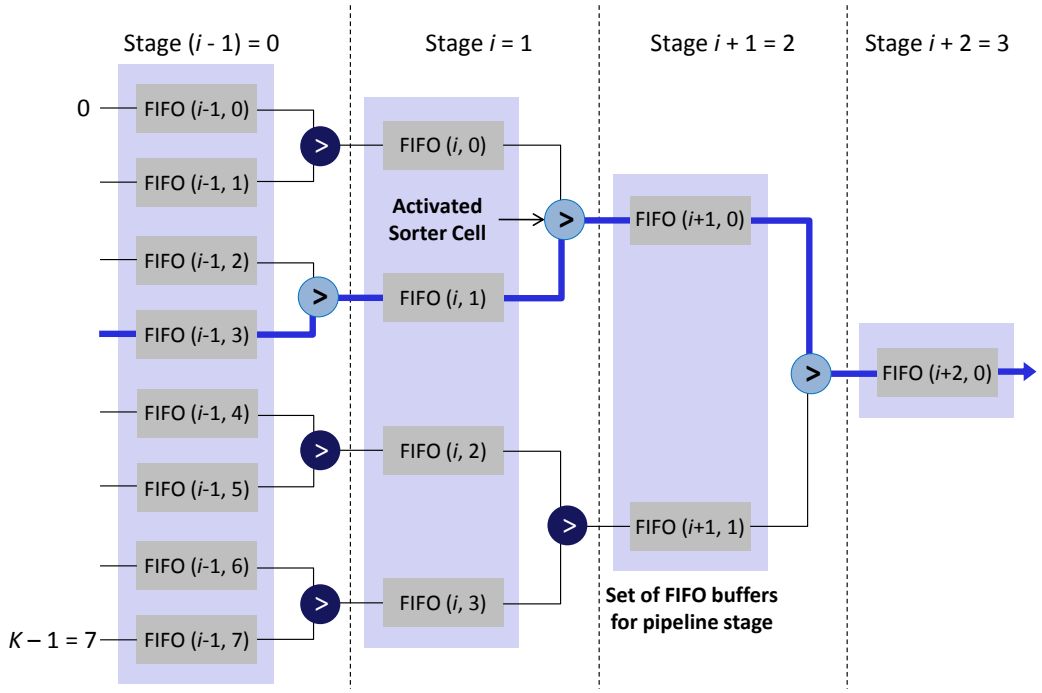


Figure 3.2: Pipelined multi-way merge binary tree implemented using independent register based FIFOs (IRFM). Highlighted blue line represents activated path in a given clock cycle.

stage and at any given clock cycle, only one sorter cell remains active in steady state. A record is dequeued from a FIFO only if the connected sorter in the next stage is active and has the smaller key. Similarly, record is queued to a FIFO only if it is not entirely full. We name this implementation as Independent Register FIFO based Merge (IRFM).

For IRFM, we define D_{FIFO} as the FIFO depth and L_d as the total number of clock cycles required beginning from the issuance of read requests to a pair of FIFOs to when the output record is ready to be queued in the destination FIFO of next pipeline stage. Additionally, we define the number of clock cycles to generate the read address, $raddr_i$ of stage i as L_a. As $raddr_i$ is dependent on the output record of stage $i+1$, minimum $L_d + L_a$ clock cycles are required for a record to travel from one pipeline stage

to the next. Hence, maximum throughput of this multi-way merge implementation, R^{max}, is 1 element per $L_d + L_a$ cycles as shown in Equation 3.1. From here on, we will also refer to the duration of $L_d + L_a$ clock cycles as a working cycle of period (T_w). To maintain R^{max} in steady state, D_{FIFO} has to be minimum 2. The reason is that it takes T_w time to dequeue a record from any FIFO and it takes another T_w time to replenish that FIFO. As a result, for consecutive accesses to any particular FIFO without introducing bubble, minimum two records have to be queued in all FIFOs during initialization. Here we are assuming that each FIFO has one read port and one write port independent from each other.

In IRFM read address generation is trivial as 'not-full' status any FIFO in stage $i+1$ can be independently propagated to stage i to generate $raddr_i$. For many practical implementations using register FIFOs $L_d + L_a = 1$ and throughput of 1 element per cycle is achievable using $D_{FIFO} = 2$. However, we are assuming that data stream is not interrupted at the leaf level of the binary tree (i.e. input stage of the pipeline). Due to various technical reasons, in practical implementations it is common to have occasional interruptions in data stream at the leaf level of the tree. In such case, it is beneficial to have $D_{FIFO} > 2$ to maintain R^{max}.

$$R^{max} = \frac{1}{L_d + L_a} \text{ records per clock cycle} \tag{3.1}$$

3.2.2 Block Memory based Multi-way Merge

As k grows, the logic required for the sorter cells and FIFOs grows exponentially. As hardware resources are limited, this becomes one of the key prohibiting factors in

implementing large multi-way merge network. As a solution to this, implementation design as depicted in Figure 3.3 can be used. We name this implementation as *Scheme-1*. As it is required to read from two FIFOs simultaneously in every stage, the FIFOs in Figure 3.2 are logically mapped to set words in two separate memory blocks. All the even numbered FIFOs are mapped to a memory block ($B^{ID} = 0$) and all the odd numbered ones are mapped to the other memory block ($B^{ID} = 1$). Every FIFO in Figure 3.3 works as a sorted list that can be considered as the input data for that particular pipeline stage. Stage i handles $k_i = 2^{(S-i-1)}$ such sorted lists, where S is the total number of stages. Among the lists in stage i, the even numbered $k_i/2$ lists are represented by one memory block ($B_i^{ID} = 0$) and the odd numbered $k_i/2$ lists are represented by the other memory block ($B_i^{ID} = 1$).

In any given working cycle t_w, two records, namely $d_{i,\,tw}^{out0}$ and $d_{i,\,tw}^{out1}$, are read from the same address of both memory blocks in stage i. However, only the record with smaller key, $d_{i,\,tw}^{min} = min\{d_{i,\,tw}^{out0},\ d_{i,\,tw}^{out1}\}$, is stored in the pipeline register. This record works as the input data $d_{i+1,\,tw+1}^{in}$ to stage $i+1$ in the next working cycle $t_w + 1$. Hence, for every pipeline stage *Scheme-1* require 2 reads and 1 write in the pertinent memory blocks. It is important to note that the FIFO that dequeued the record with smaller key in stage i at working cycle t_w needs to be replenished at the working cycle $t_w + 1$ to maintain steady throughput.

Here we are assuming that every block memory has one read port and one write port for depiction purpose, which is common for SRAM. However, this scheme can be generalized and the fundamental principles do not depend on the number ports of the memory blocks.

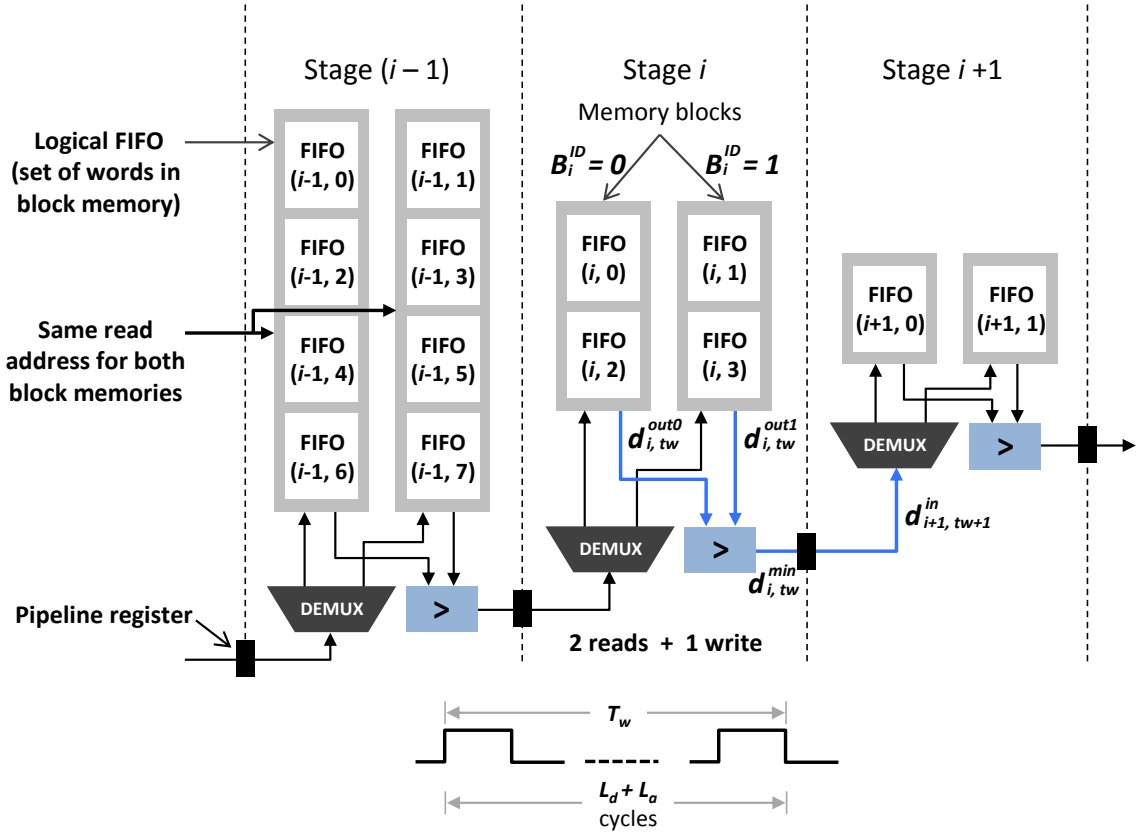


Figure 3.3: Block memory based multi-way merge network (*Scheme-1*).

3.2.3 Block Memory based Merge: Advantages and Challenges

The first advantage of *Scheme-1* is that now only $\log K$ sorter cells are required instead of $K-1$. Secondly, a single SRAM block can be used instead multiple separate register based FIFOs. This significantly reduces the silicon area required for buffer storage needed in multi-way merge network as SRAM cells (8 transistors) are much smaller that registers (19 transistors). However, as described below, there are a number of potential issues that can render *Scheme-1* to be inefficient.

1. **Performance:** The block memory in each stage, as depicted in Figure 3.3, can be conceptually conceived as a collection of several logical FIFOs sharing a common port for data input and output. Due to this shared port, these logical FIFOs cannot independently enact themselves along the relevant branch of the multi-way merge binary tree. Hence additional control logic has to be introduced. As found in the literature [57, 58], this control logic can potentially become the critical path and reduce throughput by introducing cycle delays (bubbles) in the pipeline.

2. **Scalability:** With increasing k, the FIFO buffer requirement grows exponentially for all multi-way merge binary tree. To make it worse, the depth of each logical FIFO in *Scheme-1* is required to be increased to partially compensate for the latency and additional control logic delay described above. Hence, efficient on-chip memory management is imperative to scale multi-way merge network.

3. **Latency:** Due to monolithic decoder and SRAM technology, the read and write latency of on-chip block memory is generally much higher than of register based FIFOs'. This latency significantly reduces the performance of *Scheme-1* based multi-way merge implementations found in the literature.

In the following sections we will discuss how these issues are addressed by the literature and our proposed techniques.

3.2.4 Block Memory based Merge: Current Solutions

As shown in Figure 3.3, a logical FIFO is a set of words that is part of the block memory. Simultaneous access to two logical FIFOs at the same address of the memory

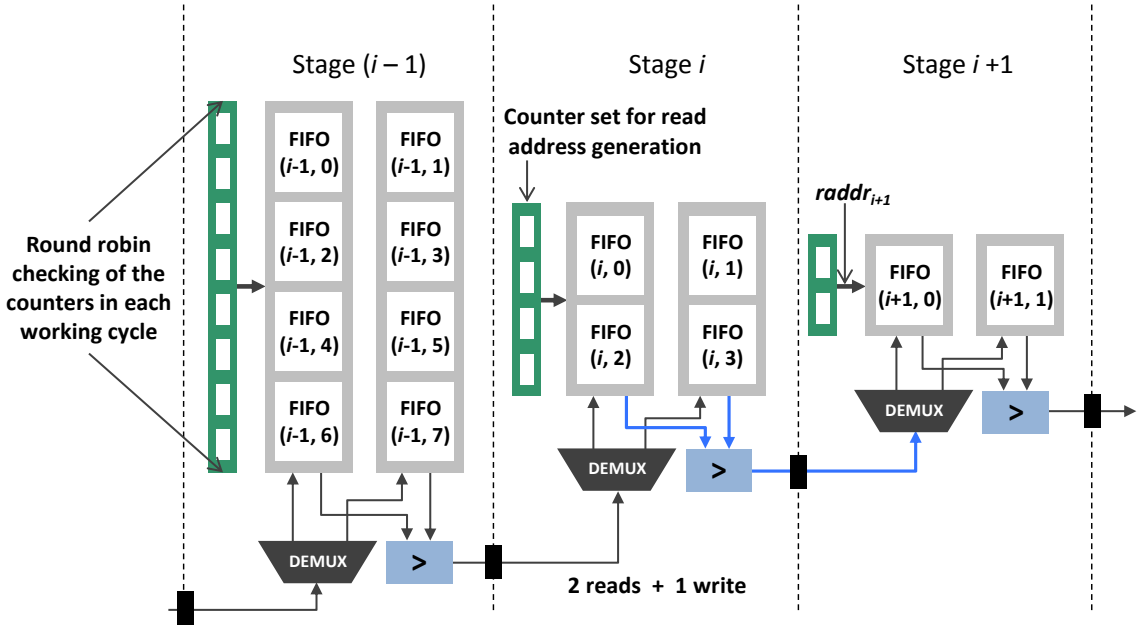


Figure 3.4: Block memory based multi-way merge network (*Scheme-1a*).

blocks is required at any given working cycle. However, generating this address is not as trivial as the register FIFO based implementation shown in Figure 3.2. In Figure 3.2, FIFOs in every branch of the tree can be independently checked for whether data needs to be replenished. However, in *Scheme-1*, it is not possible to independently check the state of all the logical FIFOs as they are part of a single memory block.

Only a handful of block memory based high performance multi-way merge hardware are available in the literature. In [57], the authors have proposed a high throughput merge-sort hardware using a similar implementation as *Scheme-1*. To find out the address of the logical FIFO in each stage that needs to be replenished, the authors of this work proposed to maintain individual counter for each logical FIFO, as depicted in Figure 3.4. We name this implementation as *Scheme-1a*. The counter keeps track of number of records that is queued in the FIFO. In every working cycle, all the counters in a pipeline stage is checked for 'not-full' status in steady state. For a k-way merge,

the first and the largest pipeline stage will have k such counters. As k increases, required time to check all the counters, i.e. $raddr$ generation time L_a in Equation 3.1, also increases proportionally. As a result the working cycle duration T_w increases and performance decreases. The authors reported to be able to implement a 64-way merge hardware using *Scheme-1a* until the $raddr$ generation time becomes prohibitively large. Hence, *Scheme-1a* is not scalable and performance is bounded by the size of the largest stage.

Another way to generate the read address of stage $i-1$ is using the read address of the output record of stage i, which we name as *Scheme-1b*. A practical implementation of *Scheme-1b* is depicted in Figure 3.5. The main idea of this scheme is to use the FIFO address where d_i^{min} was dequeued from at work cycle t_w, i.e. $raddr_{i,\,tw}$ to partially generate the read address (except Least Significant Bit (LSB)) of stage $i-1$ at work cycle t_w+1, i.e. $raddr_{i-1,\,tw+1}$. The source block identifier ($b_{i,\,tw}^{ID,\,min}$) of d_i^{min} is used to generate the LSB of $raddr_{i-1,\,tw+1}$. A read queue is used for each stage to store the generated read address to handle the scenario when the target FIFO is empty. The read queue also helps in reducing stalls by storing multiple read addresses. In Figure 3.5, we have shown all the pipeline registers that are practically used to improve clock frequency. For example, the numbered registers $f1$, $f2$, $f3$, $f4$ and $f5$ are pertinent to stage i and involved in a single work cycle to transfer a record to stage $i+1$. The numbers mentioned in the registers refer to the clock cycle at which (during rising edge) they are triggered. Register $f1$ is read address input and $f2$ & $f3$ are data output registers for the block memory that introduce 2 clock cycles delay. Hence, $L_d = 2$ for *Scheme-1b*. On the other hand, $raddr_i$ is generated using d_{i+1}^{min} that passes through registers $f4$ and $f5$. Therefore, it takes two clock cycles to generate the address, i.e. $L_a = 2$. According to Equation 3.1, the maximum steady state throughput

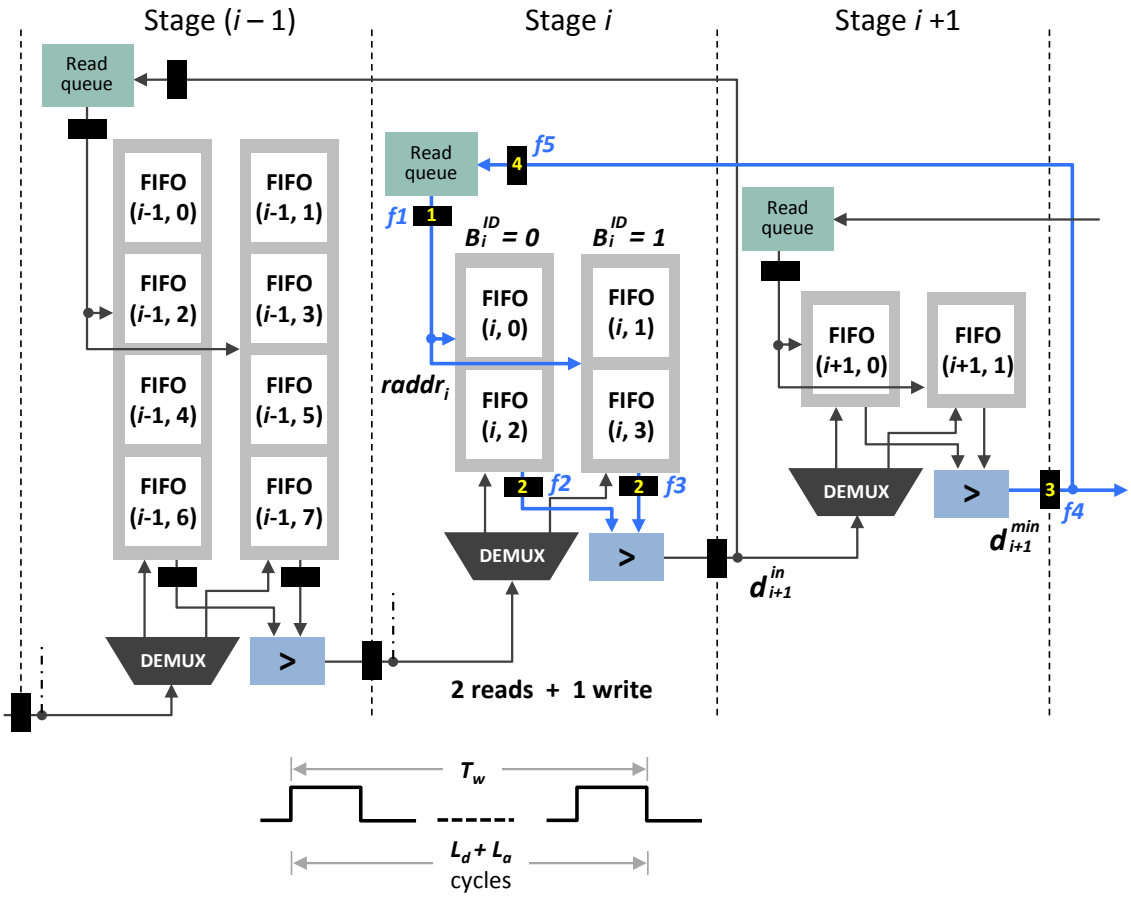


Figure 3.5: Block memory based multi-way merge network (*Scheme-1b*).

$R^{max}_{Scheme\text{-}1b}$ is 1/4 record per clock cycle as shown in Equation 3.2. It is possible to omit register $f4$ and decrease increase the throughput to 1/3 records per cycle. However, it may increase the clock period and decrease the absolute performance.

$$R^{max}_{Scheme\text{-}1b} = \frac{1}{2+2} = \frac{1}{4} \text{ records per clock cycle} \tag{3.2}$$

It is important to note that we can parallelly queue and dequeue records in the block memories as we assume that there are independent read and write ports (one of each). Hence, queuing data can be overlapped with reading data during L_D cycles

in *Scheme-1b*. However, generating read address cannot be overlapped with reading data as read address for stage i in work cycle t_w is directly generated the from the output of stage $i+1$ in working cycle $t_w - 1$, i.e. $raddr_{i,\,tw} = \{raddr_{i+1,\,tw-1},\ b^{ID,\,min}_{i+1,\,tw-1}\}$. Even though $raddr_{i+1,\,tw-1}$ is available during the early phase of the work cycle, i.e. during data read, $b^{ID,\,min}_{i+1,\,tw-1}$ is not achievable until the end of work cycle when $d^{min}_{i+1,\,tw-1}$ is available. A solution similar to *Scheme-1b* is proposed in [58]. The authors of this work used $D_{FIFO} = 4$ to compensate for the logic and pipeline registers used in their implementation.

3.3 Comparison Look Ahead Merge (CLAM)

In this work, we have developed a multi-way merge implementation scheme, namely Comparison Look Ahead Merge (CLAM). CLAM provides better performance (records per cycle) through efficient address generation scheme and is more scalable due to less demand of buffer storage. Furthermore, we propose a method, namely Hybrid Comparison Look Ahead Merge (HCLAM), to hide the block SRAM latency by pragmatically using both SRAM and registers as merge tree buffers. To the best of our knowledge, both CLAM and HCLAM implementations are novel and no similar methods are found in the literature.

Before describing our proposed implementations, we define few terminologies for clarity of explanations.

- In Figure 3.3, Figure 3.4 and Figure 3.5, FIFO $(i,\ j)$ sequentially feeds the records from sorted list $l(i, j)$ to its following stages and, hence, $l(i, j)$ can be thought of the j^{th} input list w.r.t stage i, where $j = 0, ..., k_i$. The leaf level FIFOs at 0^{th} stage feeds the original input data set (K lists) to the entire binary tree.

- The frontier record of list $l(i,j)$ is represented as $r(i,j)$. Frontier record of a list means the top most record that hasn't been dequeued from the list yet.

- All sorted lists pertaining to any pipeline stage are numbered starting with '0'. Comparison between the frontier records of two consecutive lists $l(i,j)$ and $l(i,j+1)$ will always imply that j is an even number. The notation $min\{r(i,j),\ r(i,j+1)\}$ means the record with smaller key between the frontier records of two consecutive lists starting with j at stage i. The notation $max\{r(i,j),\ r(i,j+1)\}$ means the record having larger key with rest being the same.

The main idea of CLAM comes from the observation that in any stage of *Scheme-1* records are read from the FIFOs and the read address in the next cycle is generated form the comparison results of these records. This sequential dependence of the address on data read in previous cycle is inevitable as this is the fundamental operation of multi-way merge. However, without violating this data dependency, we can conduct the following operations.

1. Instead of comparing the keys of records when they are dequeued from the FIFOs, we can compare the keys while being queued to the FIFOs. As we are comparing consecutive lists before we actually need the results of this operation, we term this as 'comparison look ahead'.

2. We can store this comparison information using a single bit, namely 'tag'(g), and use it later to generate address while the pertinent record is actually dequeued.

The above two are the core concept of CLAM. The main benefit of CLAM is that now we do not have to wait for the reading of records from block memory and comparison to be completed before we can start generating the address. As the

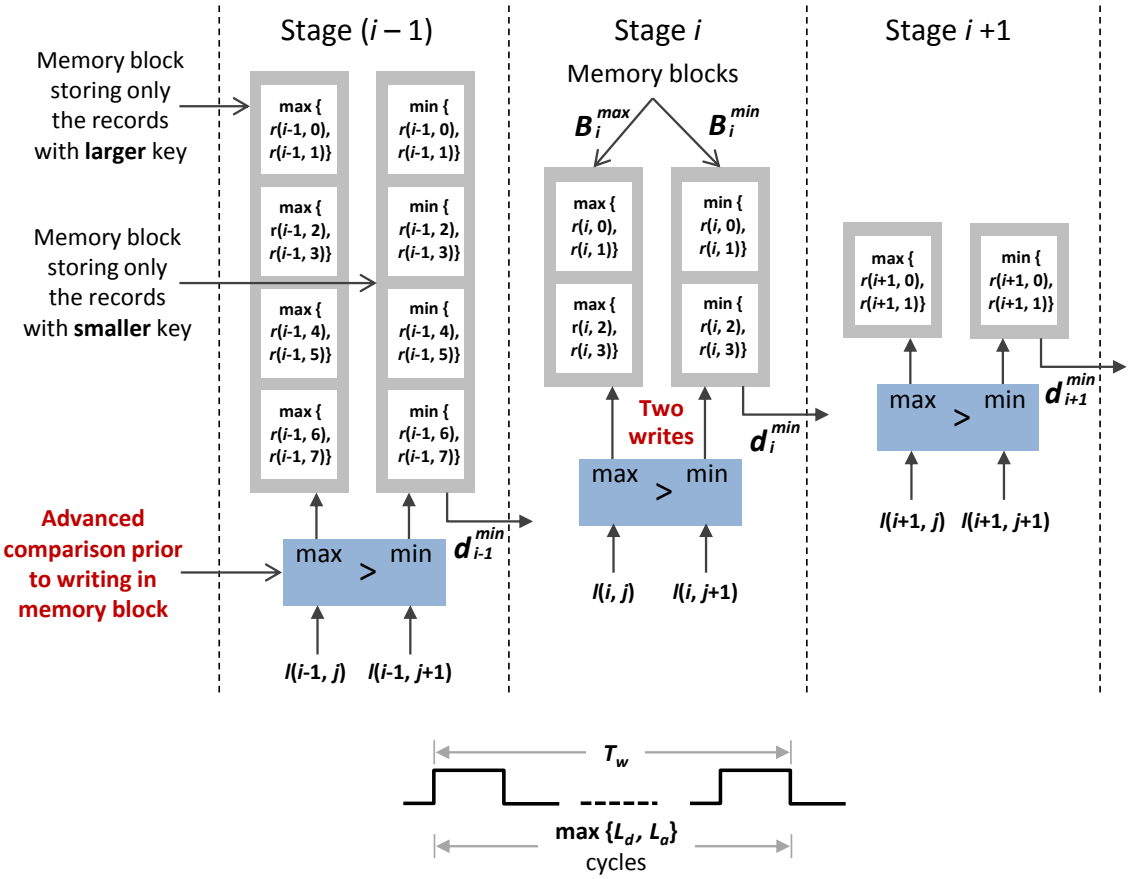


Figure 3.6: Conceptual block diagram of advanced comparison based proposed multi-way merge hardware CLAM.

comparison result is already available, i.e. the tag g is pe-computed, we can start generating the next cycle read address parallelly with initialing the read of current cycle. Hence, the working cycle duration T_w in CLAM is $max(L_a,\ L_d)$ unlike the case of *Scheme-1* (i.e. $L_a + L_d$).

Figure 3.6 shows a simplified and conceptual diagram of CLAM that is derived from the implementation of *Scheme-1*. Instead of directly storing records from the sorted lists, the memory blocks store the output of a comparator. The input to the comparator are the frontier of two consecutive lists. Block B^{min} stores the records

with smaller keys and block B^{max} stores the ones with larger keys. Therefore, when a record is requested from stage i to stage $(i+1)$, it can be directly dequeued from B^{min} without any further comparison.

3.3.1 CLAM Implementation and Operational Details

Figure 3.7 shows a hardware diagram of CLAM implementation only depicting the connections related to stage i. For better clarity, the pipeline registers are not shown. Below we will explain the important aspects of CLAM implementation and operation in detail.

Data and Address Storage

As depicted in Figure 3.7 and Figure 3.8, the data buffer requirements in CLAM are different from *Scheme-1*. It is understandable from the previous conceptual diagram in Figure 3.6 that only one record from B^{min} needs to be transferred to the next stage in a work cycle. Hence, to serve consecutive accesses into the same address without introducing bubble logical FIFOs in B^{min} are needed. On the other hand, we don't need to have logical FIFOs in B^{max} as its records are not directly transferred to the next stage. A record from B^{max} only takes part in the computation for look ahead comparison when a record is queued in B^{min} in any given work cycle. Therefore, B^{max} has single entry words instead of logical FIFOs. It is mentioned earlier that stage i handles $k_i = 2^{(S-i-1)}$ input sorted lists, where S is the total number of stages. Hence, B^{min} is a memory block with $k_i/2$ logical FIFOs and B^{max} is a memory block with simply $k_i/2$ words. If the depth of each logical FIFO is D_{FIFO}, then B^{min} has in total of $(D_{FIFO} \times k_i/2)$ words.

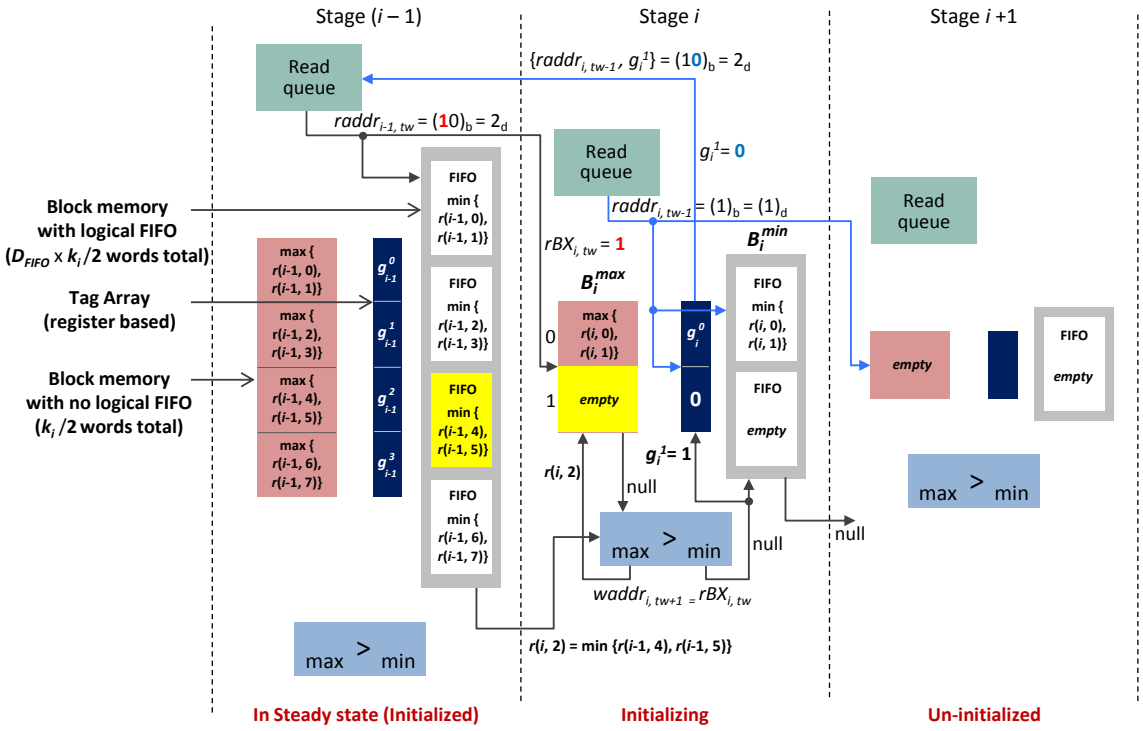


Figure 3.7: CLAM hardware diagram (excluding pipeline registers) and initialization operation at work cycle $t_w - 1$ and t_w. The blue and black paths show active paths during initialization work cycles $t_w - 1$ and t_w respectively.

Another important difference of CLAM from *Scheme-1* is that in every work cycle there are two writes in a stage instead of one. In both CLAM and *Scheme-1*, one record moves from one pipeline stage to the next in a work cycle. In *Scheme-1*, the incoming record is queued to one of the memory blocks to replenish one logical FIFO in one of the memory blocks. Similarly, in CLAM after advanced (look ahead) comparison with the incoming record we need to queue a $min\{r(i,j),\ r(i,j+1)\}$ in B_i^{min} to replenish a FIFO. Additionally, we also need to write $max\{r(i,j),\ r(i,j+1)\}$ in B_i^{max} to conduct the advanced comparison in future. Thus, in CLAM there are two reads and two writes of records in every stage at each work cycle.

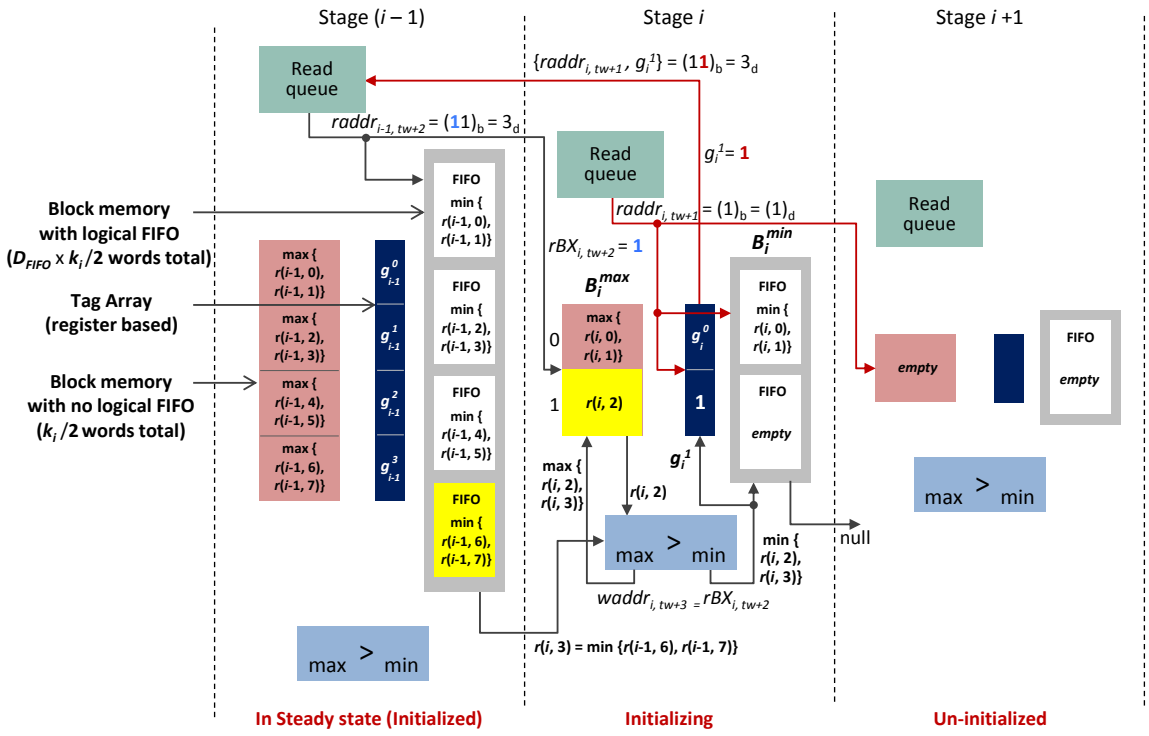


Figure 3.8: CLAM hardware diagram (excluding pipeline registers) and initialization operation at work cycle $t_w + 1$ and $t_w + 2$. The red and black paths show the active paths during initialization work cycles $t_w + 1$ and $t_w + 2$ respectively.

CLAM also has a read queue to store the read address similar to *Scheme-1b*. This read queue serves the same purpose of storing read request addresses from the following stage and helps to avoid bubbles. A read queue in stage $(i - 1)$ provides the read address $raddr_{i-1}$ for B_{i-1}^{min} and B_i^{max}. B_i^{max} handles half of the number of lists than B_{i-1}^{min} does and the LSB of $raddr_{i-1}$ is excluded before using it as read address of B_i^{max}. Hence we define the read address of B_i^{max} as $rBX_i = raddr_{i-1}$[excluding LSB].

Tag Array

Tag (g) is a single bit that stores the result of advanced (look ahead) comparison. An additional buffer is required in CLAM to store the tags that is named as 'Tag Array'.

Only a single tag bit is required per FIFO of B^{min}. Hence, tag array in stage i has $k_i/2$ bits only. As tag array memory requirement is trivial and it can be implemented using registers instead of SRAM for fast access. In every work cycle, one tag bit is updated and utilized to generate address per stage using the following rules.

- **Tag update rule:** We assume $r(i,j)$ and $r(i,j+1)$ are the inputs to the comparator of stage i. If $r(i,j) = max\{r(i,j),\ r(i,j+1)\}$, the tag bit g_i^z is updated to '1'. Otherwise, if $r(i,j+1) = max\{r(i,j),\ r(i,j+1)\}$, the tag bit g_i^z is updated to '0'. Here $z = j/2$ and j is always an even number in this context. The value of z ranges from 0 to $(k_i/2 - 1)$.

- **Tag usage rule:** Tag g_i^z is used to generate the LSB of the read request address whenever a $r(i,j)$ is dequeued from the z^{th} FIFO of B^{min}. This request is stored in the read queue of stage $(i-1)$. The entire request address is formed as $\{z,\ g_i^z\}$, where $z = raddr_i$ is the read address for B_i^{min} at current working cycle.

Initializing Operation

Figure 3.7 and Figure 3.8 depicts three pipeline stages, where stage $(i-1)$ is fully initialized, stage i is being initialized and stage $(i+1)$ is totally un-initialized. Here, we have only shown the connections related to stage i for ease of comprehension. In Figure 3.7, the blue and black lines are the active paths during initialization work cycles $t_w - 1$ and t_w respectively. Similarly in Figure 3.8, the red and black paths shows the active paths during initialization work cycles $t_w + 1$ and $t_w + 2$ respectively. Below the initialization process is described step by step.

1. At the very beginning all the entries in B^{max} and B^{min} are considered empty. However, all the bits in the tag array are initialized with '0's. Figure 3.7 shows the state of all memories at work cycle $(t_w - 1)$.

2. We assume that at a given work cycle $(t_w - 1)$, the read queue in stage i serves as read address $raddr_{i,\,tw-1} = 1_{\mathrm{d}}$. Hence, a record from the last FIFO of B_i^{min} is requested. As this FIFO is empty, a null will be delivered. At the same time, the initial value of $g_i^1 = 0$ will be read from the tag array and used to form the read request address for the read queue in previous stage $(i - 1)$. The tag bit serves as the LSB and the read address in current work cycle $(tw - 1)$ serves as the rest. Hence, a read request of address $\{raddr_{i,\,tw-1},\ g_i\} = (10)_{\mathrm{b}} = 2_{\mathrm{d}}$ is sent to the previous stage $(i - 1)$ to be logged in its read queue at the end of work cycle $(t_w - 1)$.

3. At the beginning of work cycle t_w, read queue of stage $(i - 1)$ serves the address $raddr_{i-1,\,tw} = \{raddr_{i,\,tw-1},\ g_i^1\} = 2_{\mathrm{d}}$. Hence, the record in 2^{nd} FIFO $min\{r(i - 1, 4),\ r(i - 1, 5)\}$ is read from B_{i-1}^{min} and passed to the next stage i as the incoming record $r(i, 2)$ in work cycle t_w. At the same time, $raddr_{i-1,\,tw}$ (excluding LSB) also works the read address $(rBX_{i,\,tw})$ for B_i^{max} in t_w. Hence, a null value from the last entry of B_i^{max} is read.

4. At t_w, the comparator in stage i compares $r(i, 2)$ with null value. We define $min\{r(i, 2),\ null\} = null$ and $max\{r(i, 2),\ null\} = r(i, 2)$. Hence, at the end of work cycle t_w, $r(i, 2)$ is written to B_i^{max} and null value is written to B_i^{min}. It should be noted that write address of both B_i^{max} and B_i^{min} in any work cycle is just a delayed version of the read address of B_i^{max} in previous work cycle excluding the LSB.

Therefore, write address $waddr_{i,\,tw+1} = rBX_{i,\,tw} = raddr_{i-1,\,tw}$[excluding LSB] = 1_{d}.

5. While data is written in the block memories of stage i, the tag bit at address $waddr_{i,\,tw+1}$ is also updated. In this case, $waddr_{i,\,tw+1} = 1_{\mathrm{d}}$. Hence, g_i^1 is updated to '1' following the tag bit update rule state above at the beginning of $(t_w + 1)$.

6. Figure 3.8 shows the state of all memories at work cycle $(t_w + 1)$. As the last FIFO in B_i^{min} is still empty, the read queue in stage i will again serve a read address $raddr_{i,\,tw+1} = 1_{\mathrm{d}}$. Operations as described above from step 2 to step 5 will repeat for work cycle (t_{w+1}) and t_{w+2}. However, this time the tag bit read at t_{w+1} is $g_i^1 = 1$. Hence, the 3^{rd} record from B_{i-1}^{min} and $r(i,2)$ from B_i^{max} is read in $(t_w + 2)$. At the beginning of work cycle $(t_w + 3)$, $max\{r(i,2),\ r(i,3)\}$ is written to B_i^{max} and $min\{r(i,2),\ r(i,3)\}$ is written to B_i^{min}. The tag bit g_i^1 is also updated and, thus, stage i is completely initialized after of $(t_w + 2)$.

Steady State Operation

Figure 3.9 and Figure 3.10 depict detailed diagram of CLAM implementation including all the pipeline registers and all connections. All the memory buffers and tag array are in steady state. We have used pipeline registers at the same depth that we have used for *Scheme-1* in Figure 3.5.

In Figure 3.9, the blue and red paths show the active connections during data read and address generation respectively at t_w. Registers $f1$, $f2$ & $f3$ participate in the data read process and $f4$ & $f5$ participate in the address generation process. The numbers mentioned in the registers represent the clock cycle within a work cycle that triggers them (at rising edge). At the rising edge of clock cycle 1, $f1$ latches the read

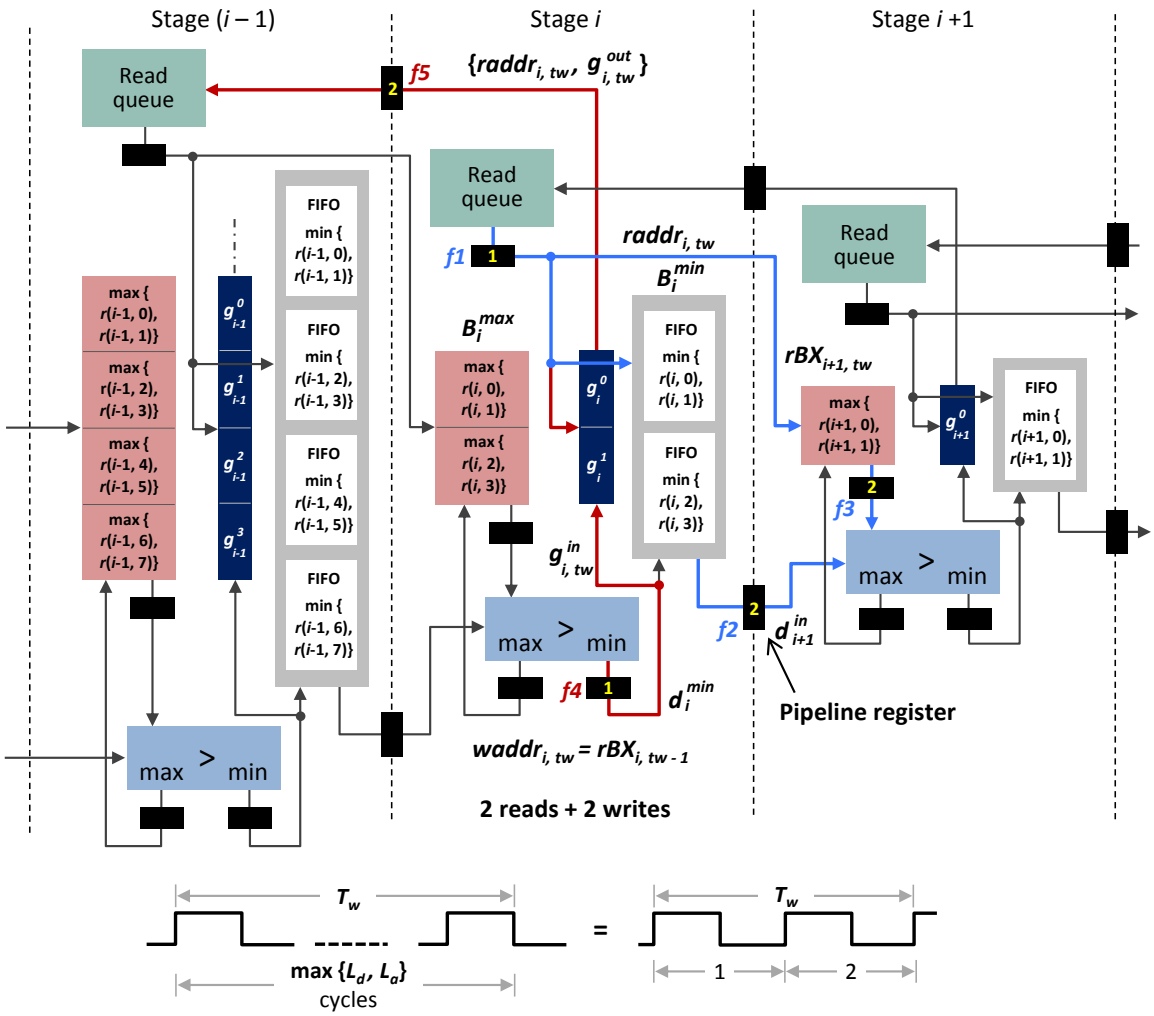


Figure 3.9: CLAM hardware diagram (including all pipeline registers) and steady state read and address generation operations. The blue and red paths show the active paths for read and address generation respectively during t_w.

address $raddr_i$, tw from the read queue. Hence, B_i^{min} and B_{i+1}^{min} is read during clock cycle 1. At the rising edge of clock cycle 2, $f2$ and $f3$ latch these read data and advanced comparison of the keys are conducted during this clock cycle. Thus for data read and look ahead comparison the number of required clock cycles L_d is 2.

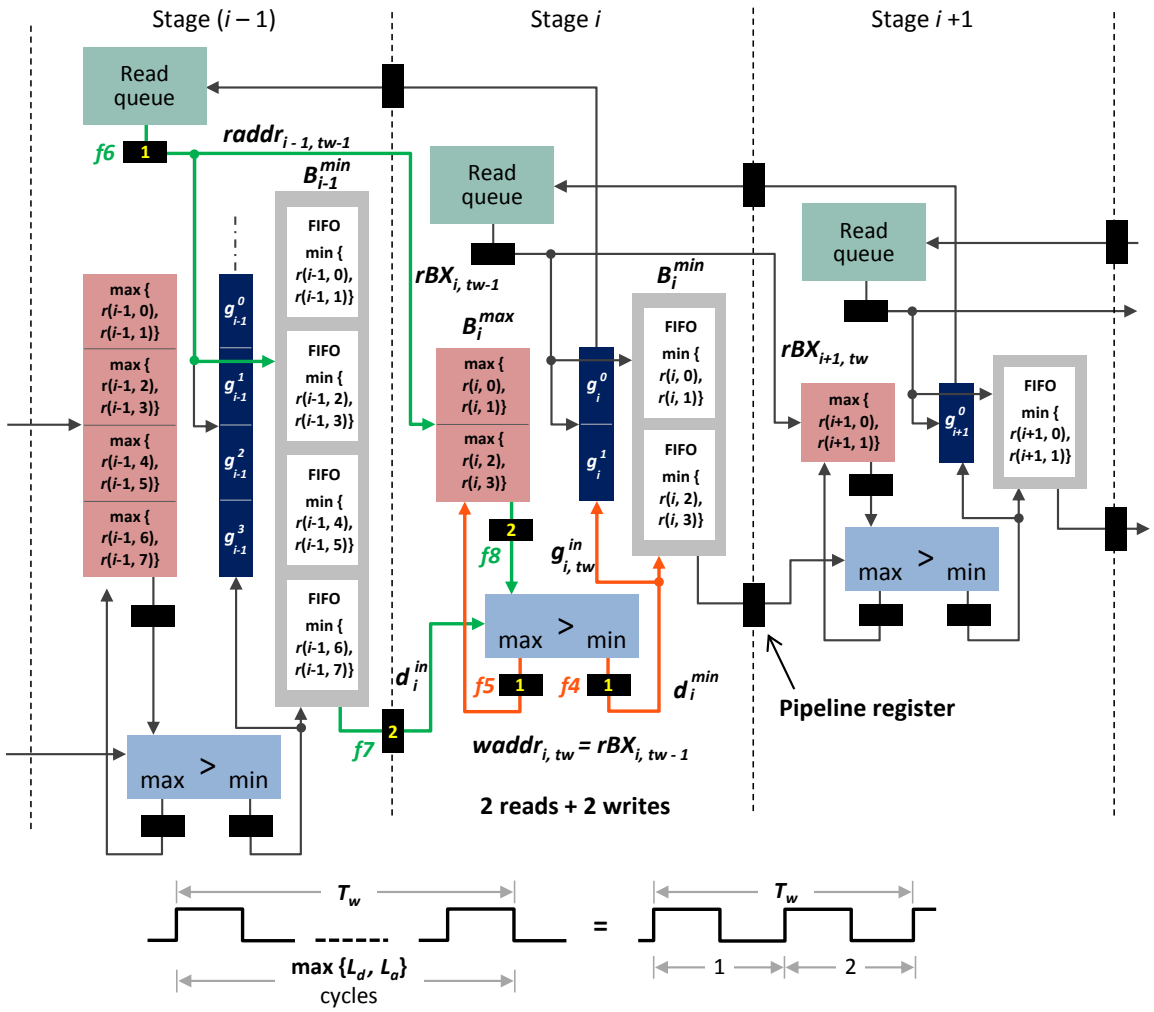


Figure 3.10: CLAM hardware diagram (including all pipeline registers) and steady state read and write operations. The green and orange paths show the active paths for read at $t_w - 1$ and write at t_w respectively.

During clock cycle 1 $raddr_{i,\,tw}$ is also used to read the previously computed comparison result $g^{out}_{i,\,tw}$ from the tag array. At the rising edge of clock cycle 1 $f4$ also latches advanced comparison result $g^{in}_{i,\,tw}$ that is latched into the tag array at the rising edge clock cycle 2 at address $waddr_{i,\,tw} = rBX_{i,\,tw-1}$. During clock cycle 1 if $raddr_{i,\,tw}$ happens to be the same as $waddr_{i,\,tw}$, then $g^{in}_{i,\,tw}$ is used as $g^{out}_{i,\,tw}$ instead of

what is actually read from the tag array. This is because the tag value pertaining to the latest record queued to the FIFOs in B_i^{min} must be used for the read request address generation. In any case, at the rising edge of clock cycle 2, request address $\{raddr_{i,\,tw},\ g_{i,\,tw}^{out}\}$ is stored in the read queue of stage $(i-1)$ through register $f5$. Hence, the overall address generation process also takes 2 clock cycles, i.e. $L_a = 2$.

For efficient implementation of CLAM, data write must be overlapped with read or address generation and finished within $min\{L_d, L_a\}$ clock cycles so that no extra time is spent for write. In Figure 3.10, we have depicted the active paths during data write at t_w (orange paths) besides the ones during data read at $(t_w - 1)$ (blue paths). In fact, the data write process only takes one cycle as the write address generation is trivial and already available from the data read operation in work cycle $(t_w - 1)$. At the rising edge of clock cycle 1 in $(t_w - 1)$, $raddr_{i-1,\,tw-1}$ is latched by $f6$. This address is used to read records from B_{i-1}^{min} and B_i^{max}, which are latched by $f7$ and $f8$ at the rising edge of clock cycle 2 in $(t_w - 1)$. Hence, at the rising edge of clock cycle 1 in t_w both the output records after comparison is latched by $f5$ and $f4$. These stored records at $f5$ and $f4$ are written to B_i^{max} and B_i^{min} memory blocks during the first clock cycle of t_w, which is overlapped with the data read from these memory blocks. For both B_i^{max} and B_i^{min} write address is $waddr_{i,\,tw} = rBX_{i,\,tw-1}$, which is available from the previous work cycle $(t_w - 1)$.

Advantages of CLAM

Performance. Since the data read and address generation is parallelly conducted in CLAM, duration of a work cycle T_w can be derived by $max\{L_d,\ L_a\}$. As both L_d and L_d is 2, duration of T_w is also 2 cycles. Opposed to Equation 3.1, the maximum throughput of CLAM can be calculated as Equation 3.3. We can see that due to the

overlap of data read and address generation R^{max}_{CLAM} is two times faster than $R^{max}_{Scheme\text{-}1b}$, which is, to the best of our knowledge, the highest throughput possible by block memory based multi-way merge implementations available in the literature.

$$R^{max}_{CLAM} = \frac{1}{max\{L_d,\ L_a\}} = \frac{1}{2} \text{ records per clock cycle} \tag{3.3}$$

Scalability. Scarcity in fast on-chip memory is one of main reasons that multi-way merge implementations cannot scale. We have seen that in *Scheme-1b*, both the memory blocks in a pipeline stage comprise of logical FIFOs. However, in CLAM, only one of the memory blocks comprise logical FIFOs. Hence, if the FIFO depth increases only by 50% of the memories in CLAM increase, whereas in *Scheme-1b* 100% of the memories increase in size. Furthermore, as CLAM work cycle is half of the work cycle for *Scheme-1b*, relatively less FIFO depth is required in CLAM to avoid bubbles. The additional resources that CLAM needs is the storage for tag array. However, only single bit per logical FIFO is required, which is trivial.

From the above discussion, we can see that CLAM provides higher performance and more scalable solution than *Scheme-1* based multi-way merge implementations as problem size increases. However, for practical implementations of K-way merge, as K increases we expect clock frequency to decrease due to additional routing. We have designed a parameterized hardware using Verilog and synthesized using commercial 16nm FinFET library for ASIC for various problem sizes. Figure 3.11 shows the achievable clock frequency of CLAM for increasing number of input lists. We have increased K 64 times, from 128-way to 8192-way. We can see that the clock frequency drops around 30%, which is not prohibitively significant for such a large size multi-way

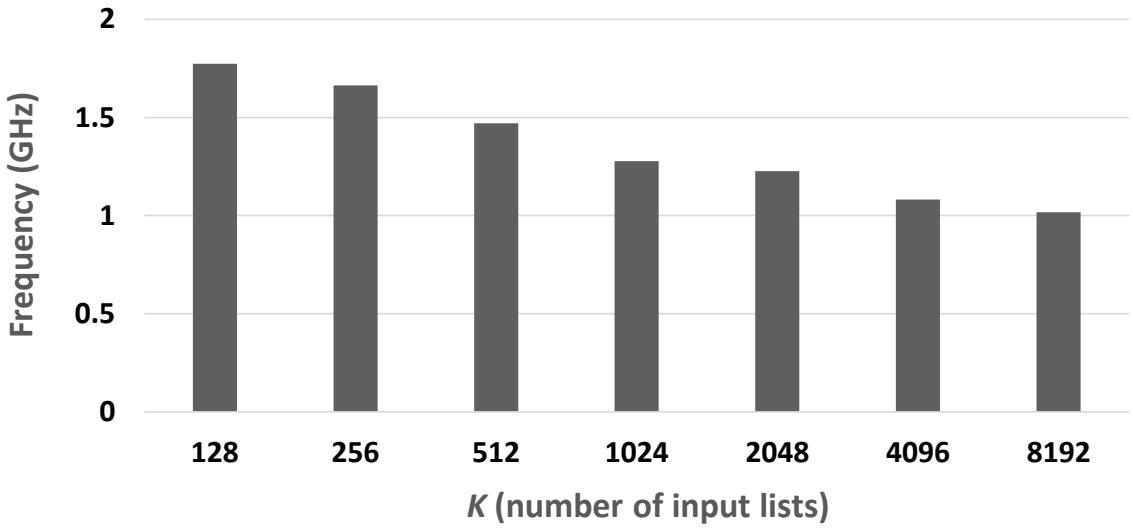


Figure 3.11: Problem size scaling of CLAM vs achievable clock frequency in 16nm FinFET ASIC.

merge network. Nevertheless, in the following sections we will see how we can improve performance further by parallelization and other techniques to address technology scaling.

3.4 Technology Scaling

In our discussion so far in this chapter we have mainly focused on problem scaling of multi-way merge and proposed CLAM as a solution to that. However, from a system performance point of view the throughput of the multi-way merge network should saturate the full off-chip communication streaming bandwidth. With the advent of modern technologies such as 3D stacked main memory, the DRAM streaming bandwidth of a system can be in the order of hundreds of giga bytes. However, for a

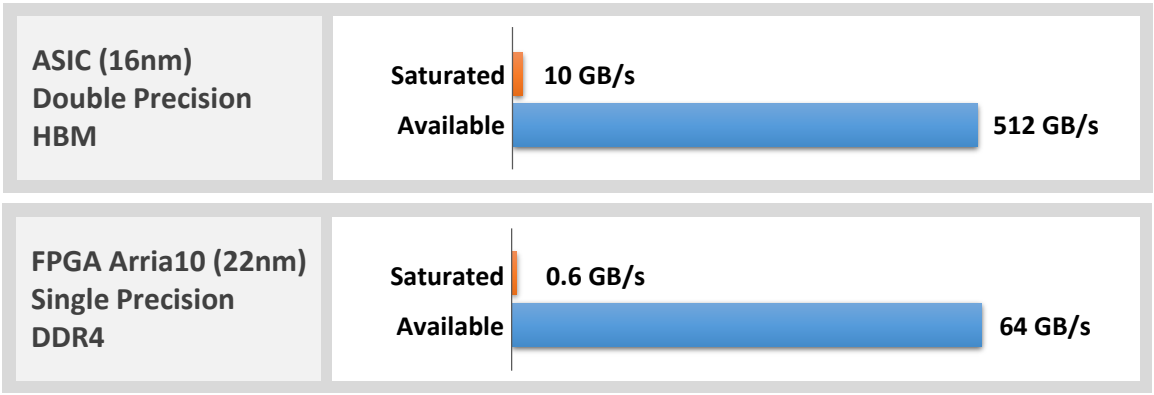


Figure 3.12: Single CLAM multi-way merge network streaming bandwidth consumption on ASIC (2048-way) and FGPA (64-way) platforms.

single multi-way merge implementation on ASIC it is difficult to saturate such extreme bandwidth. Same is true for high end FPGA platforms with DDR4 main memory.

Figure 3.12 shows the streaming bandwidth consumptions of a single multi-way merge network implemented using CLAM on different custom hardware platform. First is an 16nm ASIC architecture with HBM stacks providing an aggregated streaming bandwidth of 512GB/s. Second is a 22nm Arria10 FPGA based acceleration card [59] with four DDR4 channels providing 64GB/s off-chip bandwidth in total. We have implemented a 2018-way and a 64-way CLAM on the ASIC and FPGA platforms respectively. We can see that a single CLAM network is far from utilizing the system bandwidth properly. Hence, real applications using CLAM multi-way merge kernel, such as SpMV, SpGEMM, will not cope up in terms of performance as technology scales. In this section, we will describe a technique to improve multi-way merge performance and a scalable parallelization technique that can effectively address technology scaling.

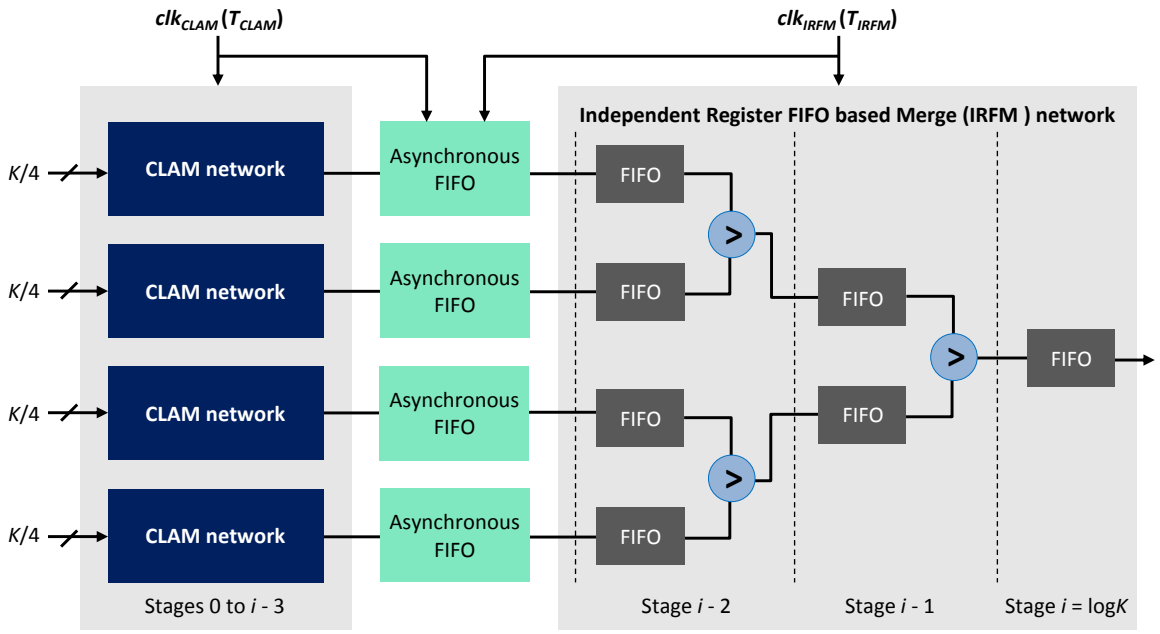


Figure 3.13: Hybrid Comparison Look Ahead Merge (HCLAM).

3.5 Hybrid CLAM (HCLAM)

One of the main drawbacks of using SRAM blocks instead of registers for multi-way merge tree is that SRAM read latency is significantly larger. In both CLAM and *Scheme-1b* we have utilized deeply pipelined design to improve clock frequency. However, the clock period is bounded by the SRAM block read latency. Furthermore, throughput of deeply pipelined CLAM is one element in two cycles, whereas register FIFO based merge, as shown in Figure 3.2, can provide a throughput of one element per cycle. In this work, we propose a pragmatic way of utilizing both SRAM and register based implementation, which is named as HCLAM.

Figure 3.13 elaborates the hardware implementation of HCLAM. Here, the large, in terms of memory usage, stages of the multi-way merge network are implemented using SRAM memory based CLAM. The last few stages, i.e. those close to the final

output, are implemented using independent register FIFO based merge tree, i.e. IRFM as shown in Figure 3.2. We name this as a hybrid implementation because of the mix between two different types of multi-way merge schemes. Previously, we discarded IRFM as a scalable multi-way merge network because it is not feasible to implement for large values of K due its high resource requirement of memory and logic. However, the last few stages of IRFM require only a trivial number of buffers and sorter cells. Hence, for any multi-way merge binary tree implementing the last few, e.g. 3 or 4, stages using IRFM doesn't have any considerable effect on the scalability of overall implementation. However, integrating these few stages with the CLAM network as shown in Figure 3.13 helps to hide the SRAM latency of CLAM. The idea is that multiple CLAM network will queue their output records in multiple asynchronous FIFOs at their peak throughput. At the same time, these asynchronous FIFOs will work as the input lists for a small IRFM network that has higher throughput. If the absolute rates of input and output of these asynchronous FIFOs are matched, then the overall HCLAM implementation will achieve the higher throughput of IRFM, while having the better scalability of CLAM.

The two different multi-way merge schemes in HCLAM are implemented using two independent clocks as shown in Figure 3.13. The clock periods of clk_{CLAM} and clk_{IRFM} are defined as T_{CLAM} and T_{IRFM}. In practical implementations clk_{CLAM} is slower than clk_{IRFM} as the SRAM read latency falls in the critical path of CLAM, i.e. $T_{CLAM} > T_{IRFM}$. Furthermore, the maximum throughput of CLAM R^{max}_{CLAM} is $1/2T_{CLAM}$, which is less than the maximum throughput of the register FIFO based merge $R^{max}_{IRFM} = 1/T_{IRFM}$. The goal of HCLAM is to provide the same throughput as R^{max}_{IRFM} for the entire multi-way merge network. It is easily achievable by integrating multiple CLAM networks with a IRFM network at proper ratio ($Ratio_{HCLAM}$). Equation 3.4

provides a formula to calculated this ratio.

$$Ratio_{HCLAM} = \frac{R_{IRFM}^{max}}{R_{CLAM}^{max}} = \frac{2 \times T_{CLAM}}{T_{IRFM}} \tag{3.4}$$

For our ASIC implementation we have found that clk_{IRFM} is almost twice faster than clk_{CLAM}. In that case, $Ratio_{HCLAM}$ is 4 as computed by Equation 3.4. Furthermore, we can calculate the number of stages required for IRFM as $\log_2(Ratio_{HCLAM}) + 1 = 3$. It is also important to have enough depth in the $Ratio_{HCLAM}$ number of asynchronous FIFOs that interfaces the two different networks. If the data set is not heavily skewed towards any particular set of input lists, FIFO depth of 8 to 32 works reasonably well in our experience. Even if data is heavily skewed, we can afford to increase this depth without considerably affecting overall scalability as $Ratio_{HCLAM}$ is expected to be small (in the order of 2 to 8).

3.6 Parallel Multi-way Merge

As depicted in Figure 3.14, even though HCLAM itself increases the performance of a single multi-way merge network almost 3 times, it still not enough to saturate the system streaming bandwidth properly. Therefore, we need a parallel implementation of the merge network that can output multiple records to match the available streaming bandwidth. In this work, we have developed a scalable parallelization method that can effectively address technology scaling. We name this as Parallelization by Radix Pre-sorter (PRaP). Before describing PRaP, we will discuss a more natural way of parallelization of multi-way merge network by partitioning input lists.

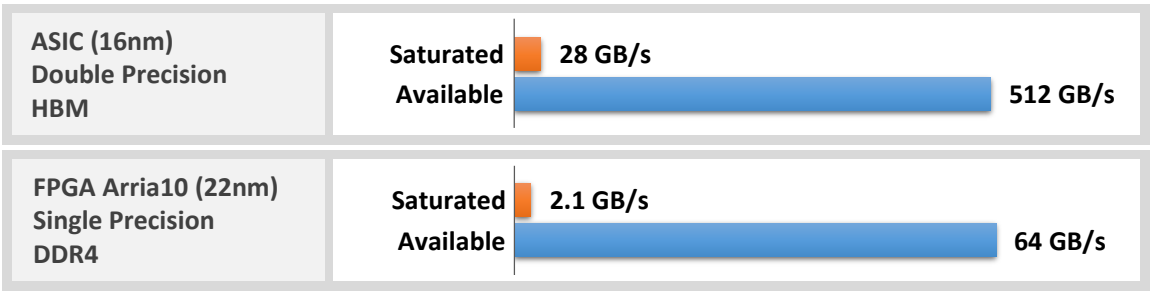


Figure 3.14: Single HCLAM multi-way merge network streaming bandwidth consumption on ASIC (2048-way) and FGPA (64-way) platforms.

3.6.1 Parallelization by Input List Partitioning

We will describe this parallelization method in the context of Two-Step algorithm for SpMV. As depicted in Figure 3.15, the matrix can be 2D blocked that will eventually generate segmented intermediate vectors. The segmented intermediate vectors can be considered as horizontally partitioned input lists for multi-way merge networks (cores). We assume that there are m such partitions. Hence, we can deploy m HCLAM cores that independently merges the lists in a particular partition and ultimately produce a single segment of the resultant vector. In this way, we can achieve a throughput of m records per cycle instead of one. This parallelization method works well when the entire problem set, i.e. all the input lists, fit in the on-chip memory. However, when the problem set is too large to fit in the on-chip storage, this method becomes unscalable. The reason is explained below.

During multi-way merge operation, the records in any particular list is accessed sequentially. However, in every cycle only one list dequeues a record and the selection of that list is practically random. For large problem set the data set will reside in the off-chip main memory and these random accesses will cause poor utilization of off-chip bandwidth. One practical way to ensure full utilization of the off-chip streaming

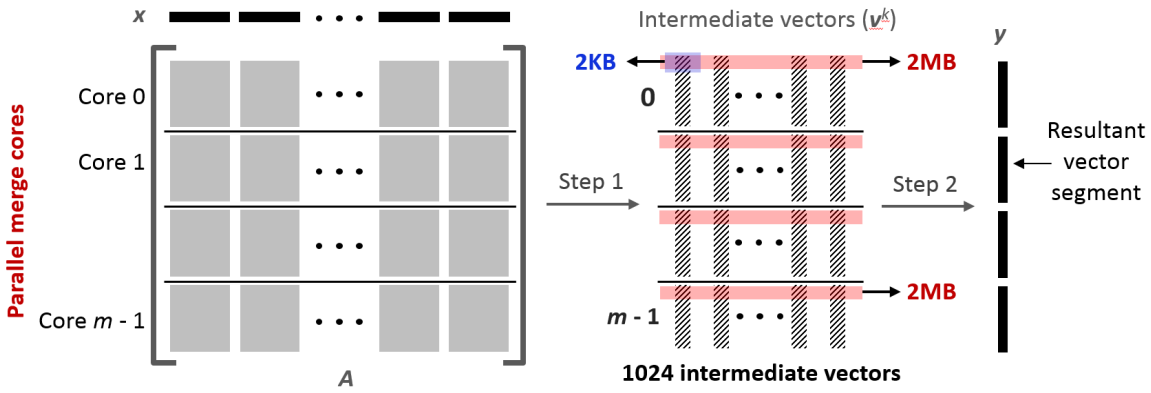


Figure 3.15: multi-way merge parallelization by partitioning input lists for Two-Step SpMV. This method becomes unscalable when the problem is larger than on-chip memory.

bandwidth is to prefetch DRAM page (row buffer) size data block from memory whenever a list is accessed off-chip and store the block on chip for a guaranteed later reuse. We name this as on-chip prefetched page buffer. For K input lists we will need K such page buffers. For example, as shown in Figure 3.15, if we prefetch d^{page} = 2KB page size data block for every list, overall 2MB on-chip buffer for all the lists in a single partition is required. If there are 4 partitions, we require $m \times K \times d^{page}$ = 8MB on-chip memory just for the prefetched page buffers itself. It is noticeable that the on-chip memory requirement grows linearly with increasing number of partitions m. Hence partitioning the input lists for parallel multi-way merge operation is not scalable.

3.6.2 Parallelization by Radix Pre-sorter (PRaP)

From the discussion above it is apparent that we need a parallelization scheme that doesn't require increasing prefetch buffer with more parallel multi-way merge networks. In this work, we propose PRaP as a solution to this problem, which is depicted in Figure 3.16. The idea is to implement p independent multi-way merge networks where

each will only work on records with certain radix within the keys. For that purpose, each record streamed from DRAM is passed through a radix based pre-sorter and directed to its destination merge network. We denote each such merge network as a Merge Core (MC) that is implemented using HCLAM. We also define q as the number of LSBs from the key of a record that is used as the radix for pre-sorting as shown in Figure 3.17. Hence, the number of MCs is $p = 2^q$ and, thus, a multi-way merge network with total output width of p records can be achieved. The main benefit of PRaP is that irrespective of p, the on-chip prefetch buffer size is $K \times d^{page}$, which is only 2MB given the example in previous section. Since p can be incremented without requiring more on-chip storage, PRaP is significantly scalable and effective in addressing technology scaling. It is important to note that PRaP method of parallelization only works when it is guaranteed that the sorted output list is a dense vector, as in the case for output vector y in SpMV. We will elaborate this in detail in later part of this section.

Radix Pre-Sorter Implementation

Without any loss of generality, we assume that the DRAM interface width is also p records. Hence, whenever the i^{th} list $l(i)$ is streamed from DRAM, records $r(i, j)$ to $r(i, j + p)$ is transferred in a single clock cycle as a part of the prefetched data. These p records are then passed through pipelined radix based pre-sorter as shown in Figure 3.18. The pre-soter is implemented using a Bitonic sorting network [60] as p output per cycle is required to match the input rate. In Figure 3.18, we have depicted the Bitonic network in simplistic manner. The horizontal lines show the data path of the record. The downward and upward arrows represent comparison and swap operation in the ascending and descending order respectively. It is important to note that only q bits of the keys take part in the comparison operation of the pre-sorter.

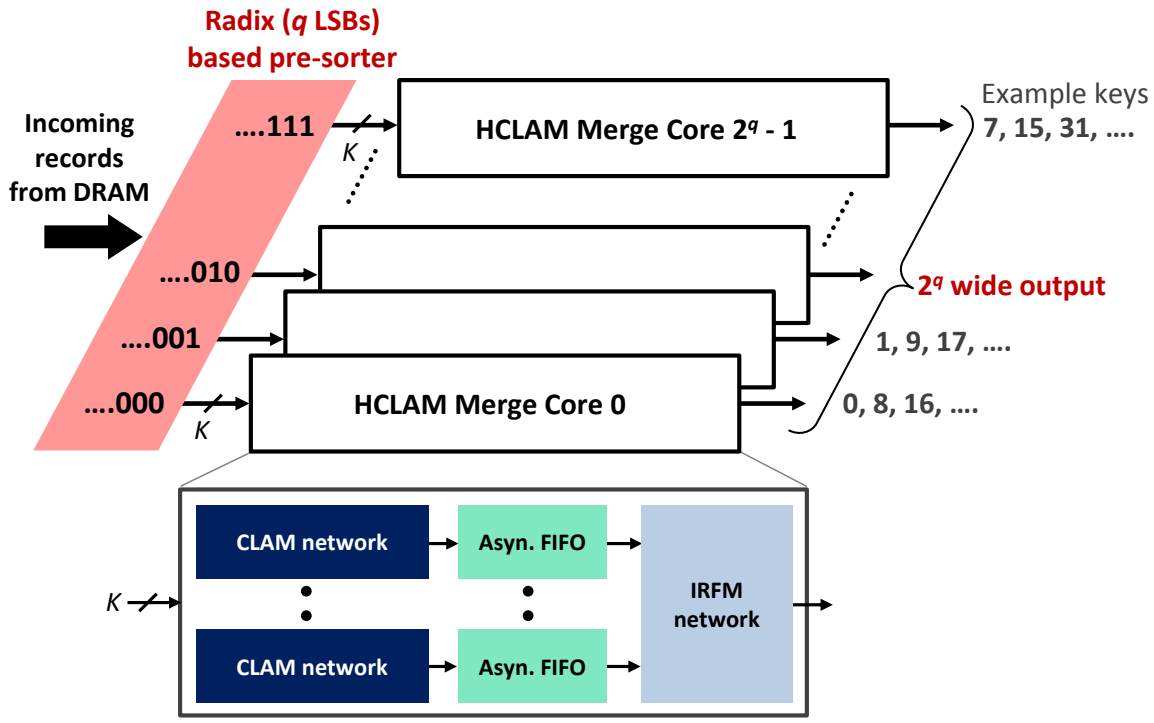


Figure 3.16: HCLAM PRaP unit. Wide output multi-way merge implementation using radix pre-sorter and multiple parallel HCLAM cores.

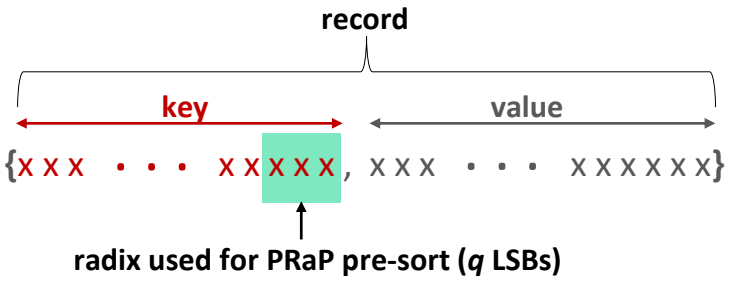


Figure 3.17: Radix selection for pre-sort in PRaP.

Hence the logic resource requirement of PRaP pre-sorter is significantly less than what is required for a one with full key comparison.

During the pre-sort, it is mandatory to maintain the original sequence of the records that possess the same radix. For example, as shown in Figure 3.18, if $r(i, j)$

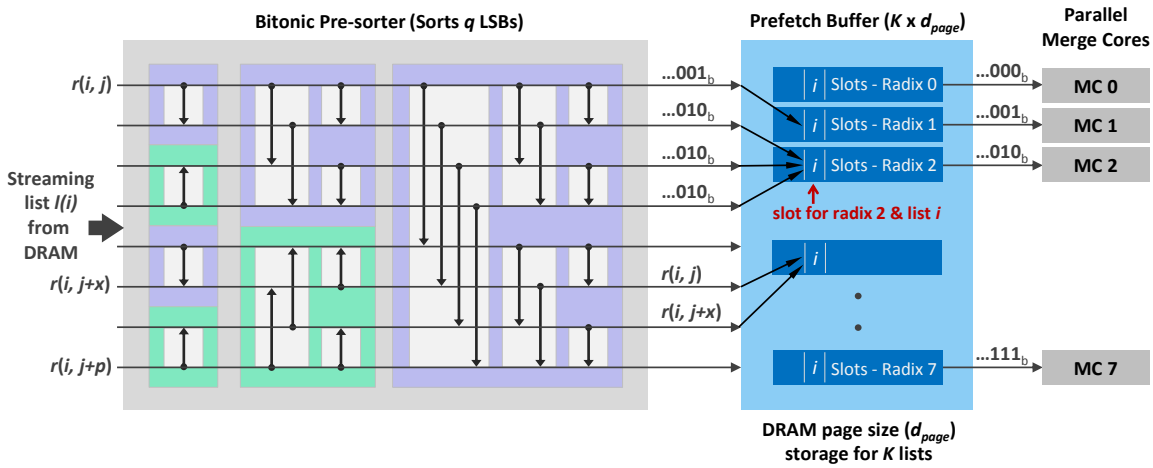


Figure 3.18: Radix pre-sorter implementation using Bitonic sorter and prefetch buffer. We assume that $r(i, j)$ and $r(i, j + x)$ has the same radix.

and $r(i, j + x)$ both have the same radix bits then $r(i, j)$ should precede $r(i, j + x)$. This is imperative because for any given merge core the input records of any list must be sorted w.r.t. the rest of the bits other than the radix within key. After pre-sorting, the outputs are stored in the prefetch buffer at the allocated location for list l_i. The prefetch buffer allocates d_{page} size storage for each list. Internally within the buffer for each list, the radix sorted records are kept in separate slots for the ease of feeding to the appropriate MC. For example, if the radix of record $r(i, j)$, $rad(i, j)$, is $(100)_{\text{b}}$ then record $r(i, j)$ is stored in the page buffer only for consumption of MC 4.

Load Balancing and Synchronization

It is possible for the incoming lists to have keys that are imbalanced in terms of the radices. In such case, the data are unevenly distributed among the MCs and potential load imbalance will occur. More importantly, as the independent MCs work only on a particular radix, further sorting and synchronization among the output of cores should have been required to generate a single sorted final output. Both of these issues can be

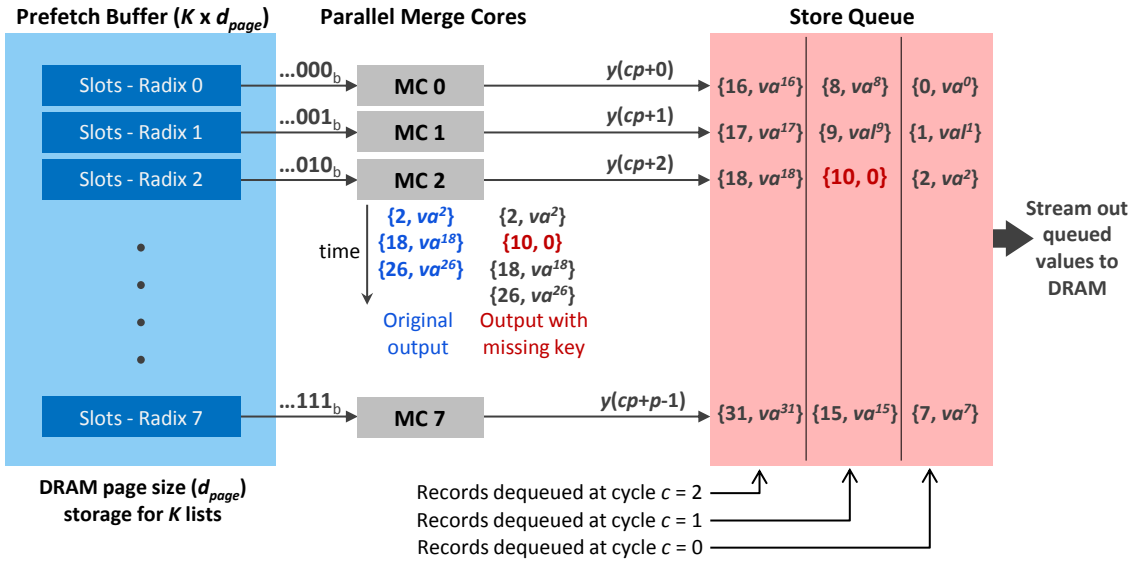


Figure 3.19: Load balancing and synchronization by insertion of missing keys in PRaP when output is dense.

effectively resolved from the observation that final output list is a dense vector. Hence it is guaranteed that each MC will sequentially deliver records with monotonously increasing keys (assuming sort in ascending order). Additionally, is also mandatory that each possible key, which is the row index of the sparse intermediate vector in Two-Step SpMV, is present in the resultant dense vector. For example, as shown in Figure 3.19, we assume that the input data set with radix $(010)_b = 2$ doesn't have any record with key 10. For that reason, the MC 2 sequentially delivers records $\{2, va^2\}$ (key 2 and value va^2) and $\{18, va^{18}\}$. Hence an expected record with key 10 is missing at the output stream of MC 2. To handle this scenario, we have included missing key check logic in MC design. Whenever a missing key is detected at the output, that key is artificially injected in between the original outputs along with a value of '0' and the following records are delayed. Thus, for the given example, an artificial record $\{10, 0\}$ is injected after $\{2, va^2\}$ and $\{18, va^{18}\}$ & $\{26, va^{26}\}$ are delayed.

Insertion of missing keys, necessitated by the dense output vector, solves both the load imbalance and synchronization problem. Firstly, even though data are unevenly distributed among the cores, at the output each MC produces same number of records at similar rate. Hence, effect of load imbalance is practically hidden even if it occurs. Secondly, output from the p cores, $y(cp+0)$ to $y(cp+p-1)$, can be independently queued in a store queue and synchronously streamed out (dequeued) to DRAM. The records $y(cp+0)$ to $y(cp+p-1)$ are consecutive elements of the dense output vector. Furthermore, records dequeued at cycle c and $(c+1)$ are also consecutive segments of the dense vector. Thus, we don't require any more sorting logic to synchronize the outputs from p independent multi-way merge cores. Therefore, in our proposed parallelization method PRaP we can scale the design to multiple cores without increasing on-chip buffer requirement and achieve required throughput to match the streaming main memory bandwidth. We will see later that only $q = 4$ bit radix pre-sorting, i.e. $2^4 = 16$ cores, is enough to saturate the extreme HBM bandwidth that is in the order of hundreds of GBs.

3.7 Summary

In this chapter we have demonstrated a scalable multi-way merge merge scheme that can effectively and practically handle large problem size (thousands of lists) at extremely high throughput (hundreds of GBs or several TBs). The entire proposed method can be termed as HCLAM with PRaP parallelization. HCLAM mainly handles problem scaling and PRaP parallelization handles technology scaling. Our proposed method is scalable because it doesn't prohibitively require more on-chip memory or logic as the problem size grows. Because of PRaP, we can also increase the throughput by

incrementing the number of merge cores without increasing on-chip memory, which is the most critical resource for scaling. Chapter 4 will demonstrate how HCLAM PRaP multi-way merge is integrated with the other parts of the sparse kernel accelerator proposed in this dissertation for SpMV acceleration. Furthermore, Chapter 7 we will elaborate how this developed multi-way merge hardware primitive is used as streaming sparse accumulator to accelerate SpGEMM and SpMSpV operations.

Chapter 4

Implementation of Two-Step SpMV

Contents

This chapter will elaborate implementation details of the separate steps of our proposed SpMV algorithm. We will also describe a meta-data compression technique enabled by our proposed algorithm. Furthermore, this chapter will demonstrate special hardware

techniques to efficiently handle high degree nodes in power-law graphs. Additionally, we will discuss implementation details for systems with multiple DRAM channels.

4.1 Design Goals and Challenges

Scalable, high performance and energy efficient acceleration of memory bound SpMV for very large (~billion nodes) and highly sparse (avg. degree <10) warrants a number of algorithmic and architectural goals to be achieved. These goals are - a) proper utilization of main memory bandwidth, b) reduction of off-chip traffic, c) less dependence on fast memory to scale, and d) data locality unaided computational scheme and avoidance of costly pre-processing. Proper utilization of main memory bandwidth implies full streaming DRAM access and minimization of the off-chip traffic. Two-Step algorithm described in Chapter 2, as shown in Figure 4.1, provides an algorithmic solution to ensure full streaming access to DRAM and reduce off-chip traffic. Furthermore, implementation of Two-Step does not depend on spatial/temporal data locality or costly pre-processing of the matrix. However, there are several challenges related to implementation of Two-Step algorithm implementation and accelerator development for sparse kernels in general.

- **On-chip Memory**: To achieve decent performance and efficiency, many SpMV computations proposed in the literature heavily depend on the availability of large on-chip fast storage. This dependency constraints these solutions to scale effectively as the graph gets larger and sparser. It is because fast on-chip memory is limited and cannot be scaled easily in shared memory architectures. SpMV solutions using custom hardware in the literature have reported only a few million nodes. For example, FPGA solution in [36] reported a maximum graph

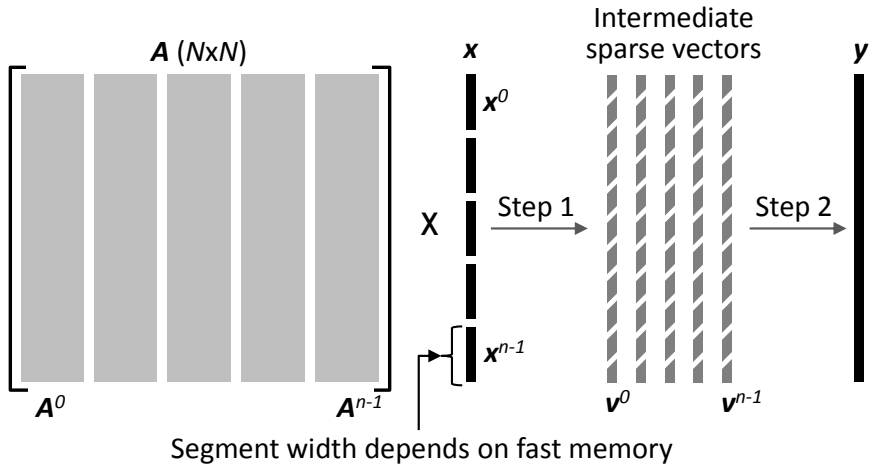


Figure 4.1: Two-Step SpMV algorithm.

dimension of 2.3M nodes using 8.4MB on-chip SRAM. The largest dimension efficiently handled by the ASIC solution in [33] is only 8M nodes despite using a huge 32MB eDRAM scratchpad. Hence, even tens of million, let alone billion, node graphs are difficult to accelerate with shared memory custom hardware architecture.

- **Off-chip Traffic**: Another challenge in accelerating SpMV using streaming algorithm is large amount of off-chip traffic. Reduction of off-chip can lead to significant improvement in performance and efficiency. However, it is difficult to find effective techniques to reduce SpMV off-chip traffic for large graphs in literature.

- **Multi-way Merge**: Two-Step algorithm requires a high throughput and large ($\sim$ several hundred or thousands of ways) multi-way merge network to handle large graphs. Methodology to design such multi-way merge hardware is absent in the literature. High throughput multi-way merge hardware solutions found in

the literature are generally not scalable due to high resource consumption. For example, FPGA based custom merge hardwares in [57] and [61] are only 64-way and 32-way respectively, while saturating only a tiny fraction of the bandwidth that modern 3D main memory can provide. On the other hand, large multi-way merge hardware in literature, such as [62], are generally slower. Hence, the multi-way merge methodology required for Two-Step algorithm that can handle large problems is nonexistent in current literature.

- **Special Nodes**: Power-law graphs comprise special vertices, namely High Degree Node (HDN), that pose separate class of challenges. For example, HDNs have a large number of collisions that render the accumulation to be more difficult than regular nodes.

4.2 Proposed Architecture

In this work, we have developed an algorithm co-optimized custom hardware accelerator to address the above mentioned challenges. As mentioned in previous chapters, the main reason for adopting custom hardware is the requirement of high throughput and scalable multi-way merge merge network by Two-Step SpMV algorithm. A conceptual diagram of the entire accelerator system is given in Figure 4.2. The computational core, is based on custom hardware. We have designed an 16nm FinFET based ASIC chip to serve as the computational core, which is currently under fabrication. An actual image of the ASIC and key specifications are given in Figure 4.3. As the chip is currently being fabricated, these specifications are from post physical synthesis (after place and route) layout of the design. One important aspect of our developed ASIC is that it

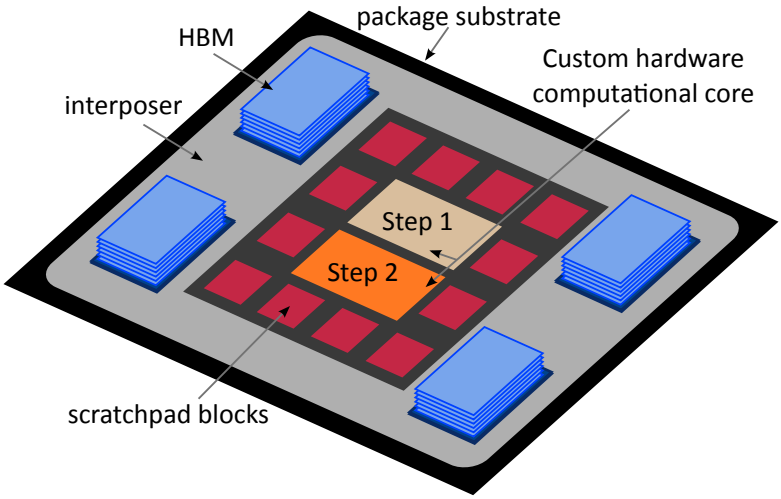


Figure 4.2: Custom hardware based sparse matrix kernel acceleration system including HBM and interposer.

uses synthesized SRAM blocks, also known as Logic in Memory (LiM) technology, instead of compiled SRAM. These synthesized SRAM blocks are distributed all over the chip to facilitate fine grain data access during computation. Details of LiM technology is available in [63–65]. This ASIC is designed to work with two 3D-stacked 2^{nd} generation High Bandwidth Memories (HBM2s) [66,67] as main memory, which are connected through interposer [68] as depicted in Figure 4.2. We also have ported the ASIC accelerator core design to FPGA platform. In this work, we have used Intel® Stratix® 10 [69], shown in Figure 6.3, to implement the FPGA version of our proposed accelerator's computation core.

In this chapter we will demonstrate the implementation details of Two-Step SpMV on our proposed architecture along with a number of additional optimization techniques for better performance, efficiency and scalability. Key contributions of this co-optimized accelerator for SpMV kernel are given below.

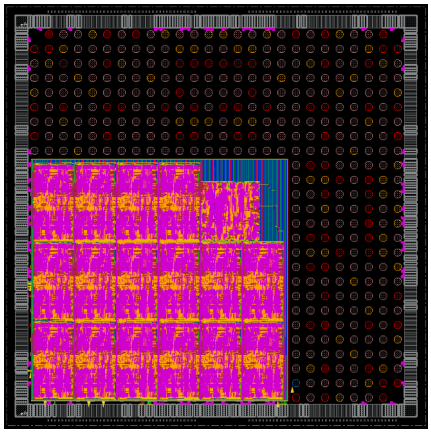


Figure 4.3: 16nm FinFET ASIC (currently under fabrication) for sparse matrix kernel acceleration.

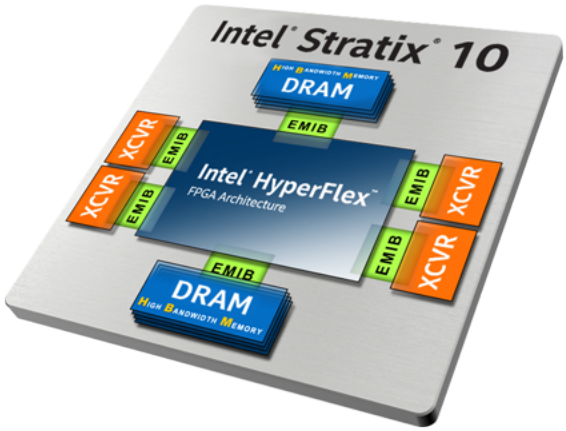


Figure 4.4: Intel® Stratix® 10 FPGA platform [69].

Contributions

1. Our proposed ASIC architecture is able to operate on very large graphs (~2 billion nodes) despite using significantly less amount of fast on-chip memory (11MB). It leaves significant room for possible expansion of on-chip memory and the proposed solution is scalable to handle even larger graphs. This design is portable to FPGA due to reasonable hardware resource requirements.

2. We have developed a novel multi-way merge hardware kernel that is both scalable and provides high throughput. This enables effective implementation of our proposed Two-Step SpMV algorithm.

3. This architecture guarantees 100% DRAM streaming and is capable of utilizing extreme bandwidth delivered by 3D DRAM.

4. A meta-data compression scheme, namely VLDI, that significantly reduces the off-chip traffic is also proposed. This compression scheme strictly is enabled by the adoption of Two-Step algorithm.

5. To properly handle the computation for HDNs in power-law graphs, a Bloom Filter based HDN technique is proposed. This eliminates the cycle delays in accumulation due to numerous conflicts of HDN and helps in identification of HDNs using standard sparse formats.

6. The proposed solution is independent of data (nonzero) locality and only requires basic matrix partitioning.

4.3 Implementation of Step 1

As depicted in Figure 4.1, Two-Step algorithm operates in two distinct steps. In the first step partial SpMV is conducted among the matrix blocks and vector segments. Two-Step requires 1D column blocking of the matrix $\boldsymbol{A}$ and segmentation of $\boldsymbol{x}$ according to the same width of column blocks of $\boldsymbol{A}$. The matrix column blocks or stripes are required to be stored in a row major sparse format. CSR, as depicted in Figure 2.3, is a commonly used row major sparse format. However, it is important to note that

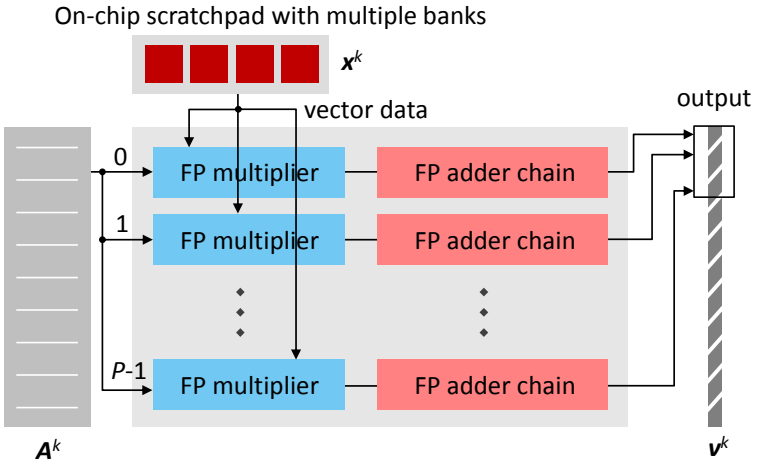


Figure 4.5: Partial SpMV Unit (PSU) to conduct step 1 of Two-Step algorithm by sharing entire scratchpad.

for large matrices with high sparsity, the matrix stripes might become hyper sparse. A matrix is considered hypersparse if $nnz < N$ [70], where nnz is total number of nonzeros and N is the dimension. For hypersparse matrix stripes, CSR might become wasteful as the space complexity of the row pointer JR array, as shown in Figure 2.3, is $O(N)$ due to the repetitions for completely empty rows. In such cases we choose to use RM-COO for matrix blocks as is has space complexity of $O(nnz)$, which is more efficient for hypersparsity.

The computational unit to conduct the partial SpMV between $\boldsymbol{A}^k$ and $\boldsymbol{x}^k$, namely Partial SpMV Unit (PSU), can be implemented in hardware as shown in Figure 4.5. PSU mainly comprises of multiple sets of a FP multiplier and a FP adder chain connected in series. The vector segment $\boldsymbol{x}_k$ is streamed from DRAM and stored in the on-chip fast scratchpad memory that consists of multiple smaller banks. Matrix stripe is streamed from DRAM and directed to the PSU. Each nonzero has a value (val^{mat}), a column index (col) and a row index (row). The multiplier multiplies val^{mat} with the

vector element at location col. We denote output of the multiplier as a record, which is a key-value pair. Here, key is row and value is the multiplier output val.

The P sets of FP multiplier and adder chain can parallelly work on separate rows of $\boldsymbol{A}^k$. As the data is sparse and the scratchpad is separated into banks, it is expected that P independent accesses will not cause significant bank conflict that may introduce stalls in the computation pipeline. However, the number of banks needs to be several times higher than P and a large cross-bar hardware is required for all multiplier-adder set to all bank accesses. This causes implementation inefficiency. A better implementation scheme is to divide $\boldsymbol{A}^k$ into sub-stripes, as shown in Figure 4.6, and $\boldsymbol{x}^k$ into sub-segments. Then each sub-stripe, $\boldsymbol{A}^k_s$,can be processed by a single multiplier-adder set that independently access $\boldsymbol{x}^k_s$ from dedicated banks of the scratchpad. In this way, we completely eliminate any possibility of bank conflicts and stalls in the computation pipeline. However, at the end of processing P sub-stripes we need to merge the results into one final intermediate vector $\boldsymbol{v}^k$. For this merge operation, we have implemented an Insertion based Merge Network (IMN) as depicted in Figure 4.7.

Details of IMN is available in [61]. A P-way IMN is capable of delivering P records, i.e. key-value pairs, per cycle. As shown in Figure 4.7, the main idea of sorting by insertion is to compare keys of P sorted incoming records from a list parallelly with another key of a previously stored record in a single sorter pipeline stage. This stored record is inserted in the appropriate position among the incoming records and a new set of $(P + 1)$ sorted records is constructed. The last among these $(P + 1)$ records replaces the stored record in the following cycle. Such $(P - 1)$ sorter stages can be pipelined to construct a P record 2-way IMN. At the very first stage, P incoming sorted records are sourced from one of two input lists l_0 and l_1. The top records of l_0 and l_1 are compared and the list with smaller key (assuming ascending order) delivers

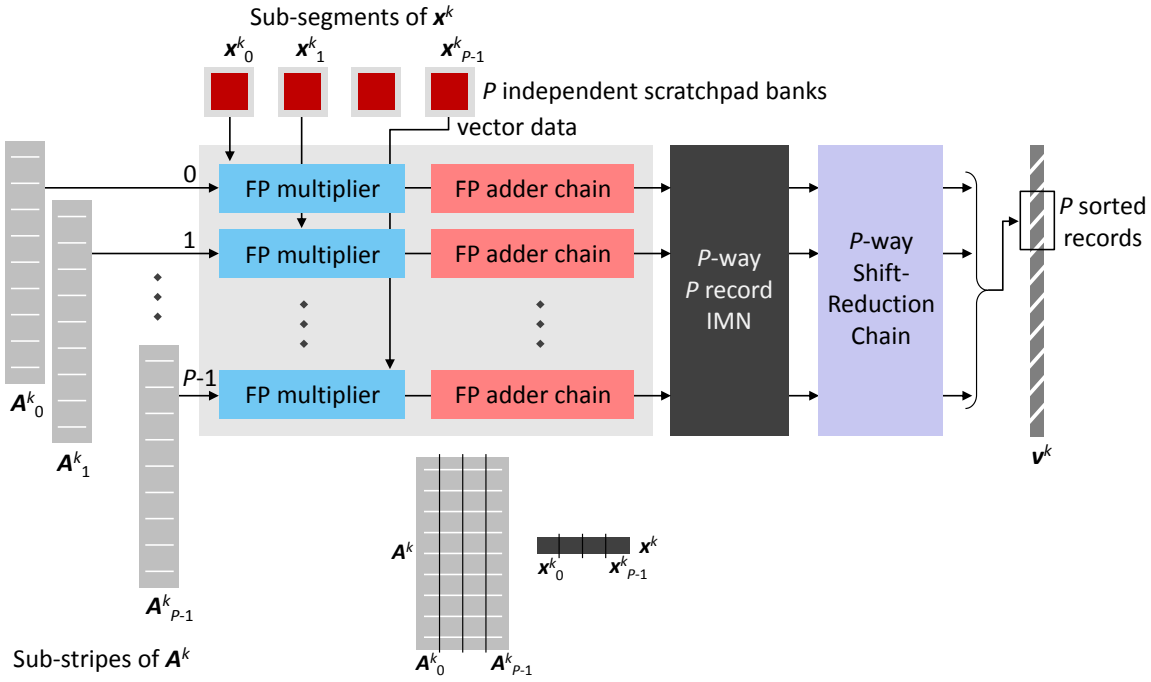


Figure 4.6: Partial SpMV Unit (PSU) to conduct step 1 of Two-Step algorithm using IMN and independent scratchpad bank access.

the P sorted incoming records for the IMN at any given cycle. All stored records at each pipeline stage are initialized with zero.

Figure 4.8 depicts how several smaller 2-way IMNs can be connected together to build a larger IMN with more ways. The coupler consists of FIFOs and simply appends stream of records to double the stream width. IMN is not easily scalable due to high resource usage as P increments. However, for the purpose of PSU, with small P it is possible to maintain high throughput to utilize 3D DRAM streaming bandwidth. For example, $P = 16$ is adequate enough for the HBM2 based ASIC proposed in this work. Hence, we prefer IMN based scheme for PSU rather than the shared scratchpad based one as it provides more computational efficiency with same level of hardware complexity.

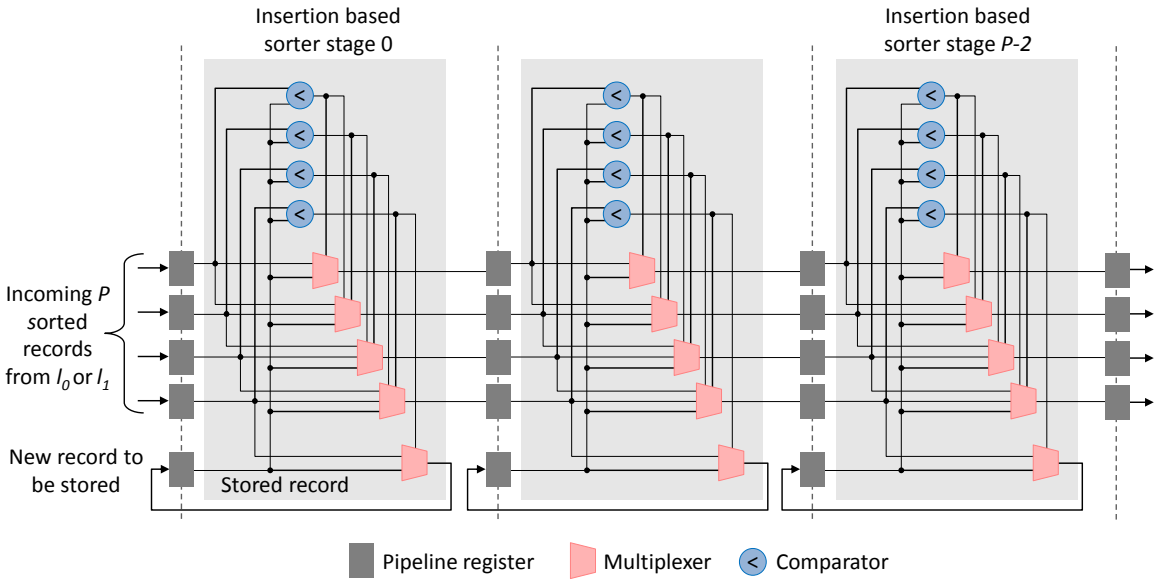


Figure 4.7: Insertion based Merge Network (IMN) sorting stages.

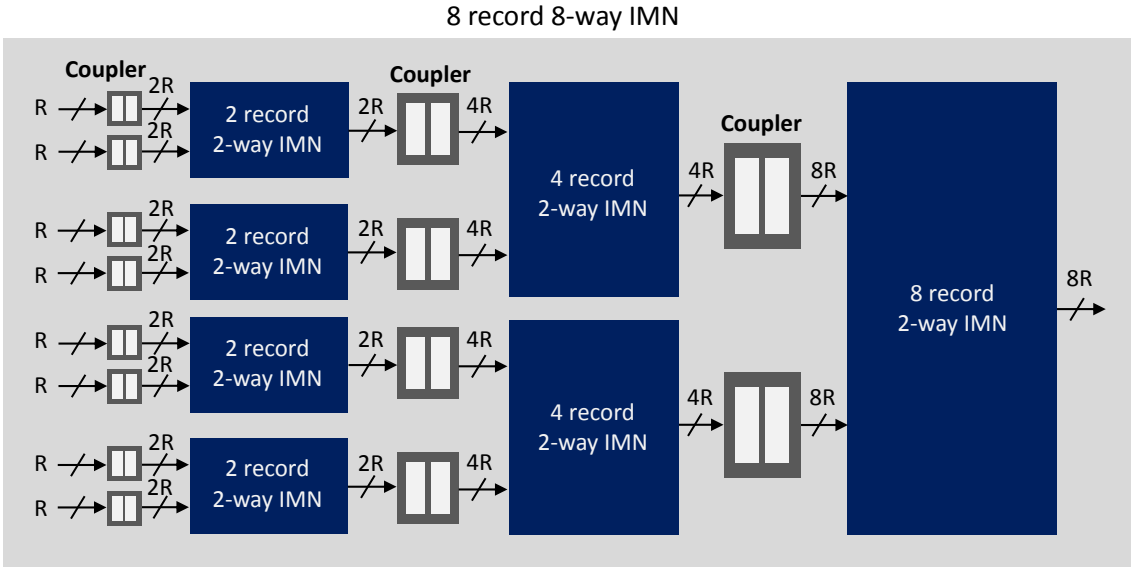


Figure 4.8: Construction of P record P-way IMN from smaller 2-way IMNs.

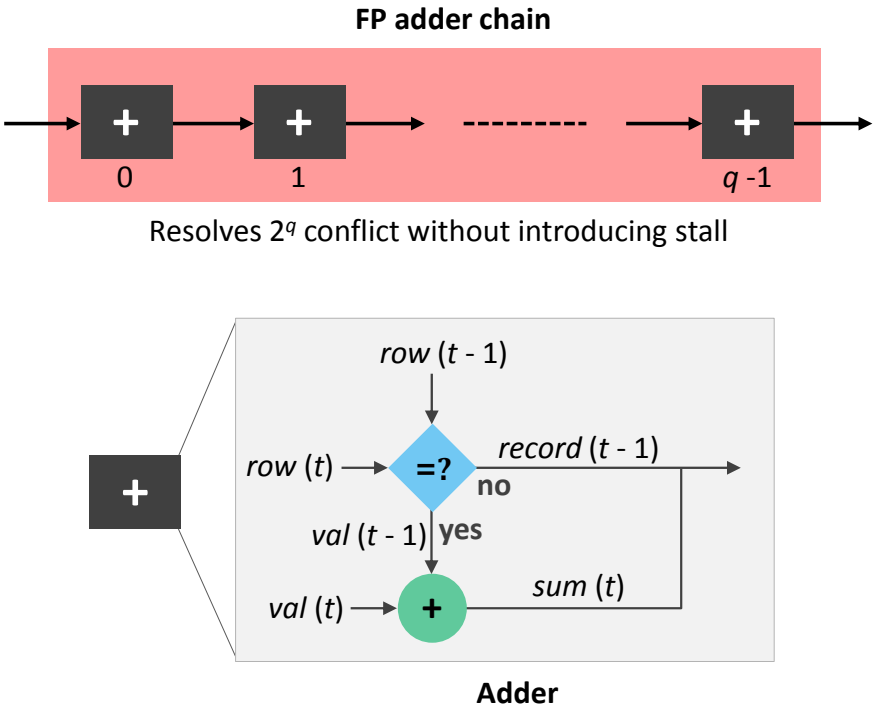


Figure 4.9: FP adder chain structure to avoid stalls due to internal pipelines in the adder.

4.3.1 Adder Chain

One of the important aspect of the PSU computation pipeline is FP adder chain. Accumulation is required for collisions, i.e. when row-indices of two consecutive records match. However, a pipelined FP adder cannot handle more than one collision without introducing stalls in the entire PSU pipeline. These stalls are due to internal pipelines of the adder. We denote F as the number of internal pipelines of an FP adder. An entire addition, which takes F cycles, has to be completed before start resolving the next if another collision is found for the same row index. To overcome this we use a chain of q adders, as shown in Figure 4.9, that can resolve 2^q collisions for any given row index without introducing any stalls. The internal logic of each adder is also

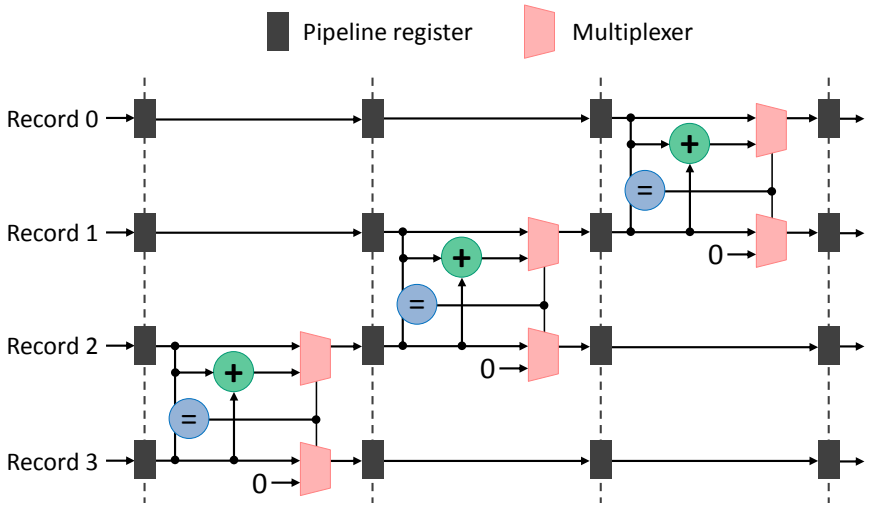


Figure 4.10: Shift-reduction chain to handle collisions among sub-stripes. This diagram illustrates a 4-way shift-reduction chain.

elaborated in Figure 4.9. An adder intakes one record per cycle and compares the row index (row) of current record, $record(t)$, with the row of $record(t-1)$. Here t represents the sequence of records as they appear at the input of the adder. If $row(t)$ and $row(t-1)$ match, the values are added and a new record, $sum(t)$, is delivered that has a key $row(t)$ and a value $val(t) + val(t-1)$. If no collision is found, $record(t-1)$ is delivered and $record(t)$ is temporarily stored to check collision with $record(t+1)$.

4.3.2 Shift-Reduction Chain

While the adder chain resolves all collisions within a sub-stripe of the matrix, there can still be collisions among the P sub-stripes. Therefore, the sorted outputs are required to be checked for collisions and resolved when needed. To achieve this in a pipelined manner, we have implemented a P-way shift-reduction chain as depicted in Figure 4.10. The main idea is to sequentially check two neighboring records for

collision. This process starts from one end of the sorted outputs of the IMN. If starting from bottom, when the indices of the records match the upper output stream delivers the accumulated result and the lower output stream delivers an artificial 0 (or null) value. When this process continues to the next pipeline stage of the chain, the next two recorded are checked for collision as shown in Figure 4.10. We name this process as shift-reduction as in every stage of the chain selection of the records shifts by one. Thus at the end of shift-reduction chain, all collisions are resolved for P-wide sorted output list. For a P-way shift-reduction chain, $P-1$ pipeline stages are required.

4.3.3 Challenges with Power-law Graphs

Power-law graphs, commonly found in social networks, have a relationship between node distribution and degree where former varies as power of the later as shown in Figure 4.11. In such graphs, there are a number of special type nodes that we denote as High Degree Node (HDN). A HDN is a vertex with a disproportionately large number of incident edges. For implementation of PSU HDN poses a separate type of challenge. There will be numerous collisions for each HDN and the adder chain depicted in Figure 4.9 will cause stalls if a large number of adders are not present in the chain. To address this inefficiency, we propose to use a different computation pipeline design to handle HDNs, which is depicted in Figure 4.12. We have introduced an additional FP adder that adds an incoming record with the adder output if they collide. In the case of this special adder for HDN, we don't wait for an addition operation to be completed before another addition is issued to resolve collision for the same row index. As there are F cycles delay between when an addition is issued and the result, sum, is delivered, continuous additions are issued between $record(t+F)$

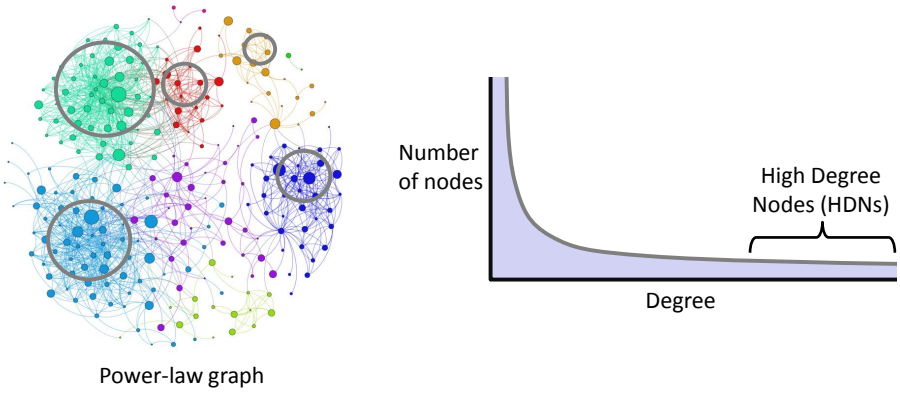


Figure 4.11: Power-law graph and degree distribution.

and $sum(t)$. Here, $sum(t)$ represents the accumulated result for all the collisions at F interval comprising $record(t)$. For any HDN, e.g. with row index row^x when no more collision is found between the incoming record and sum, the output is then released to the FP adder chain. It should be noted that at most F number of records with row^x will be released to adder chain. Hence, we need $log_2(F)$ adders in the chain at most to resolve these remaining collisions without any stalls.

HDN Detection

While we can efficiently handle HDN collisions, this special node is required to be detected first to pass it through the proper computation pipeline. One way of HDN detection is to add one more bit in the standard format, e.g. RM-COO or CSR, and set the bit as a flag for HDN. However, it will require change in widely used standard matrix formats for one specific task only. To avoid this, we propose to use Bloom Filter based detection scheme. Bloom Filter [71, 72] is a compact data structure that enables membership check for large data sets. As shown in Figure 4.13a, Bloom Filter is a bit array that encodes membership information of an element in a set through a

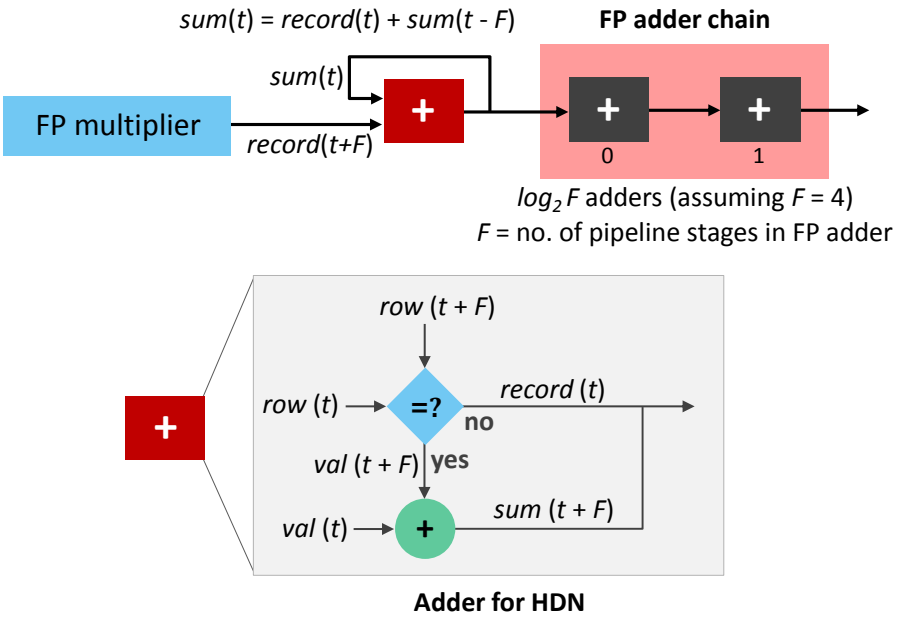


Figure 4.12: Multiplier-adder set for the computation of HDNs in power-law graph.

number of hash functions. Each member from the set is hashed to g random locations in the Bloom Filter array and the corresponding bits are asserted to '1'. To check the membership, the key of an element is similarly hashed to g bits. Only if all of the bits are found to be '1's, the element is deemed to be a member of the set.

The idea of using Bloom Filter for power law graphs is that all the HDNs can be considered as a set. By streaming the meta-data once from DRAM and using a threshold for the number of neighbors (degree) of a node, the bit array of Bloom Filter can be populated with the membership information of the HDNs. Row index of each node is considered as the key for hashing. Later during the computation, each node can be checked if it is considered as a HDN or not. Depending on the result, we can select the proper computation pipeline for partial SpMV during the step 1 of Two-Step algorithm. A block diagram of this scheme is given in Figure 4.13b. It should be noted that, Bloom Filter can provide false positives, but never false negatives. That means,

it is possible that we treat a regular, i.e. not having high degree, node as HDN. This doesn't cause any considerable inefficiency as a regular node will not cause significant stalls in the HDN pipeline. Furthermore, the pipeline for HDN also has FP adder chain that can efficiently handle collisions of regular nodes.

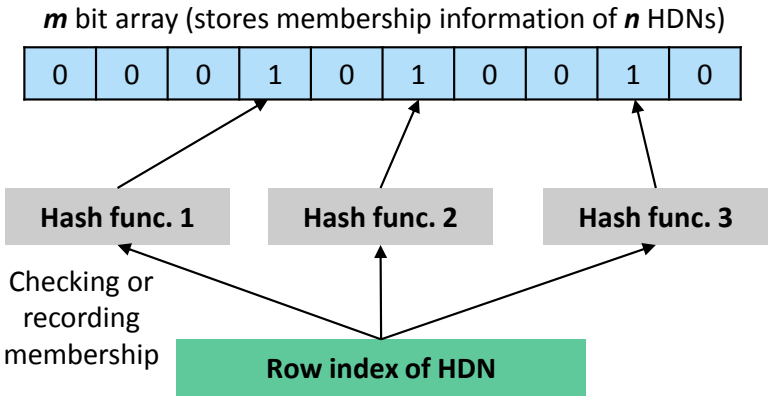
Bloom Filter implementation

There are three important factors for Bloom Filter implementation, which are false positive ratio, processing complexity and space overhead. Let m and q be the number of bits in Bloom Filter array and maximum number of members in the HDN set respectively. The ratio q/m is named as the load factor. We denote g as the number of hash functions. Given these parameters, according to [73], the probability of treating a non-member as a member can be given as the following.

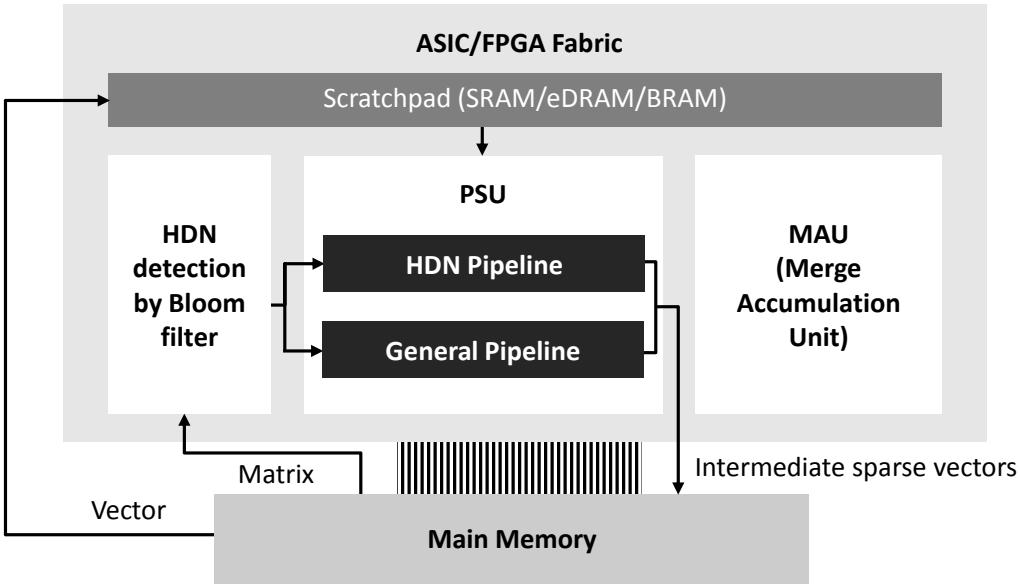
$$f_B = \{1 - (1 - \frac{1}{m})^{qg}\}^g \tag{4.1}$$

To encode and check membership, we need the hash functions to produce $g \log_2 m$ hash bits for the g random locations in the entire array. If a SRAM block is used to store these m bits, this will mean g accesses to the memory block. In this works we have implemented one memory access method proposed in [73]. In this case, the hash functions need to produce $\log_2 d + g \log_2 w$ hash bits, where d and w are the number of words and word width of the SRAM block respectively.

We consider an example graph 'Twitter_www' from KONECT [55] graph collection. Degree distribution of this graph is given in Figure 4.14. This graph has 52 million nodes and 1.9 billion edges with average degree of 74. Maximum degree of this graph, i.e. highest number neighbors of a HDN, is 3 million. As shown in Figure 4.14, we



(a) Bloom Filter membership recording and checking using multiple hash functions.



(b) Separate pipeline for HDN computation in the fisrt step of Two-Step algorithm.

Figure 4.13: Bloom Filter filter based method to process HDNs of power-law graphs efficiently for Two-Step SpMV implementation.

consider any node with more than thousand neighbors as HDN. There are less than 0.1% such nodes. However, to design conservatively we consider a Bloom Filter design to encode 100K HDNs (q), i.e. $\sim 0.2\%$ of the nodes for this example. From the analysis

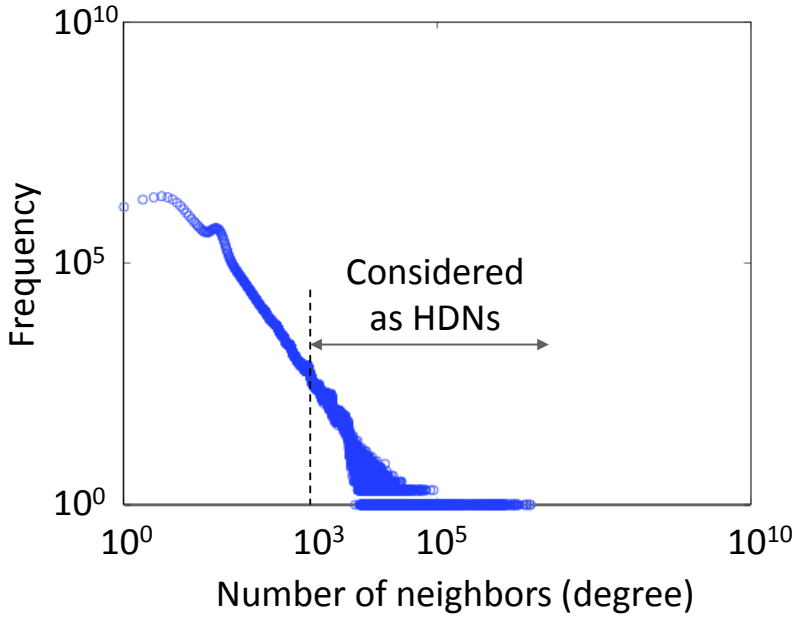


Figure 4.14: Degree distribution of example graph 'Twitter_www' [55].

in [73], for 2% false positive ratio, $g = 4$ and $q = 1e5$ the load factor is 0.1. Hence we can calculate the number of bits required for Bloom Filter as $m = q/0.1$ = 1Mbits = 128KB. If we use a SRAM block with word width of $w = 64$ bits and $d = 16384$ words, the total number of hash bits required is $\log_2 16384 + 3\log_2 64 = 14 + 18 = 32$ bits. In our implementation, we use simple XOR based hardware hash functions to generate these hash bits. Thus, overall the space and processing overhead for the detection of HDNs with a low false positive ratio is is reasonable and does not require significant resources.

4.4 Implementation of Step 2

In the second step Two-Step SpMV the task is to merge all the n number of intermediate sparse vectors, $\boldsymbol{v}^0$ to $\boldsymbol{v}^{n-1}$, into the final resultant vector $\boldsymbol{y}$. This is the most critical

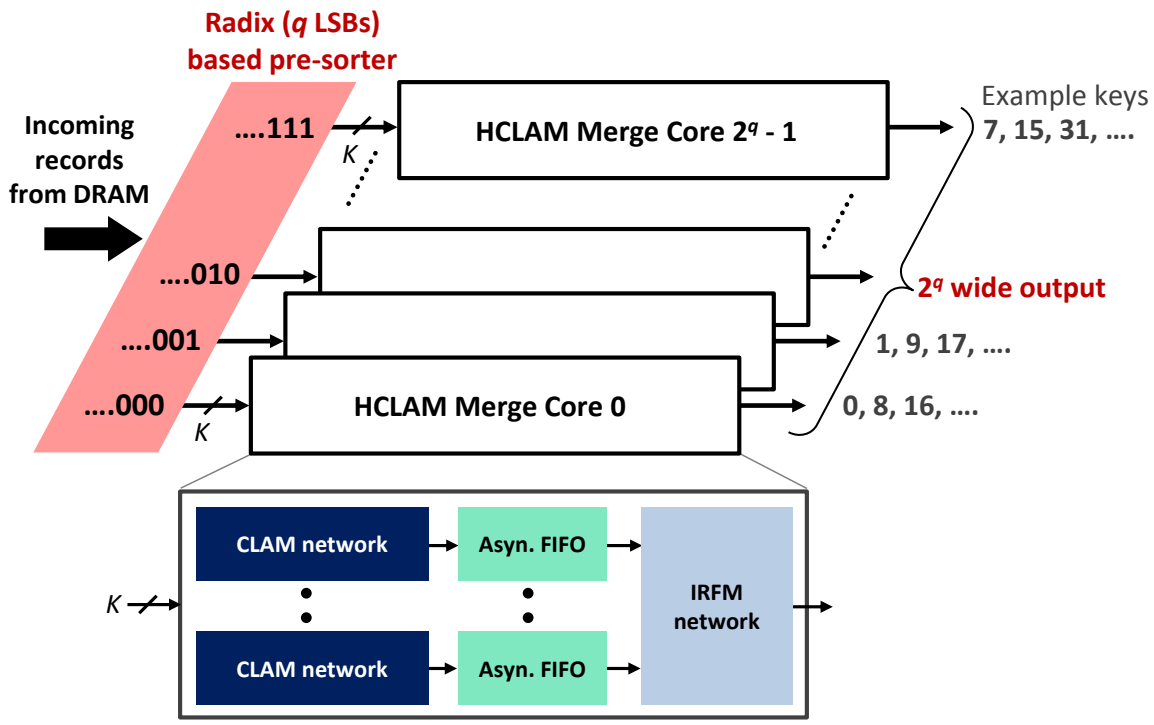


Figure 4.15: HCLAM PRaP unit.

part of Two-Step SpMV implementation as the multi-way merge network required for it needs to handle a large number (~ multiple thousands) of sorted lists that are very long (~ hundreds of million or billion elements). Additionally, the multi-way merge hardware has to be both scalable and high performance at the same time. In this work, we have implemented Hybrid Comparison Look Ahead Merge (HCLAM) based multi-way merge network along with Parallelization by Radix Pre-sorter (PRaP) parallelization, namely HCLAM PRaP unit as shown in Figure 4.15, which are explained in detail in Chapter 3.

Figure 4.16 depicts Merge Accumulation Unit (MAU) that is the computation logic for implementation of the 2^{nd} step of Two-Step algorithm. It is important to note that for shared memory system with multiple DRAM channels, the address

space and main memory ports are divided among the channels. In such case, to fully utilize the streaming bandwidth and memory space of the entire main memory system, the intermediate vectors are equally distributed among all the channels. The custom hardware accelerator developed in this work is designed for four main memory channels, as shown in Figure 4.16. Hence, we have used four HCLAM PRaP units for MAU implementation. Each HCLAM PRaP unit merges K lists that are streamed from one DRAM channel. Thus the total number of intermediate sparse vectors is $n = 4K$.

Each HCLAM PRaP unit has 2^q wide output, i.e. delivers 2^q records per cycle, where q is the number of radix bits for the pre-sorting. At the output of HCLAM PRaP units, there are 2^q adder chains, as shown in Figure 4.9, per channel to resolve collisions for records with same indices. There are 4×2^q adder chains in total. To merge the outputs for all DRAM channels while keeping the aggregated throughput the same, we have implemented 2^q number of 4 record 4-way IMNs at the output end of the adder chains. Each 4 record 4-way IMN merges the records from all four DRAM channels that have indices with the same radix bits used for pre-sorting. At the output of these IMNs 2^q 4-way shift-reduction chains, as depicted in Figure 4.10, are implemented to resolve collisions among records from all the DRAM channels. Thus, the 4 independent HCLAM PRaP network along with a number of small IMNs construct the Merge Accumulation Unit (MAU) required to fully conduct the multi-way merge for the second step of Two-Step SpMV algorithm. Since there is only a few number of main memory channels, the implementation cost of the IMNs is not significant.

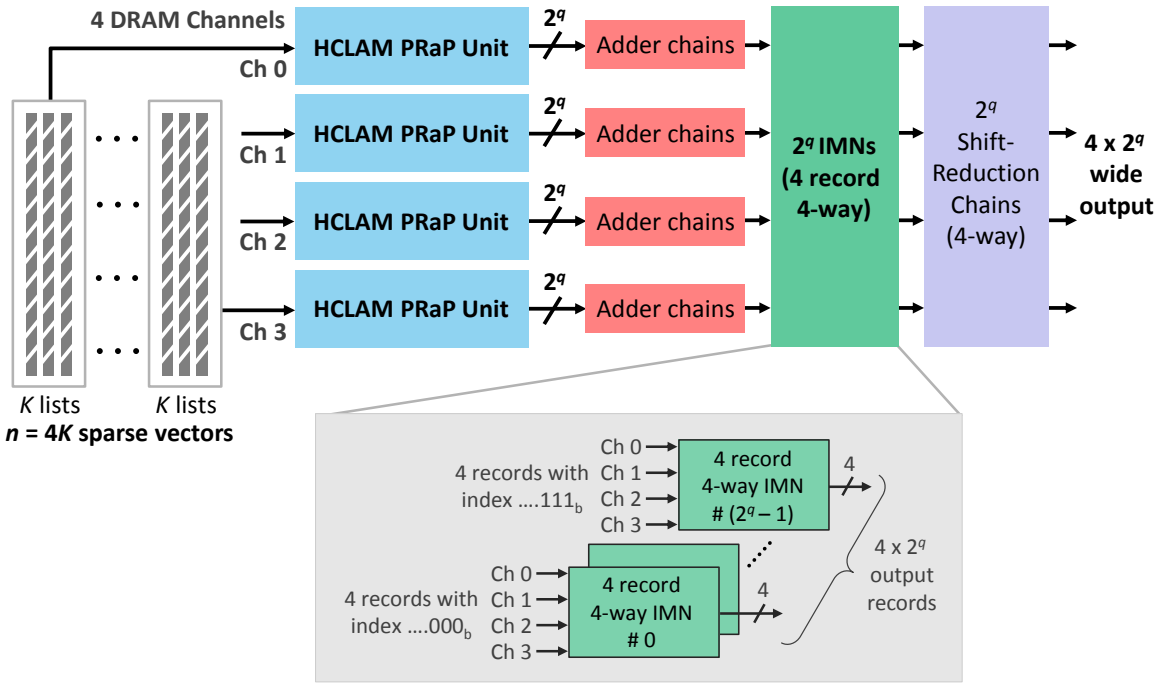


Figure 4.16: Merge Accumulation Unit (MAU) implementing 2^{nd} step of Two-Step SpMV using 4 DRAM channels.

4.5 Meta-Data Compression

Delta Index

In Two-Step SpMV, the round trip of intermediate vectors ($\boldsymbol{v}^k$s) in sparse format to/from DRAM incurs off-chip traffic overhead. Each nonzero of these vectors is represented by a position index, i.e. meta-data, and a value. All these intermediate vectors are both generated and accessed sequentially for Two-Step SpMV implementation and there is an opportunity to exploit this access pattern to achieve off-chip traffic reduction. Generally, meta-data is represented using 32 or 64 bit integers. However, instead of using absolute index within entire vector address space as meta-data for a nonzero, it is possible to only use the distance from its previous nonzero. As shown in

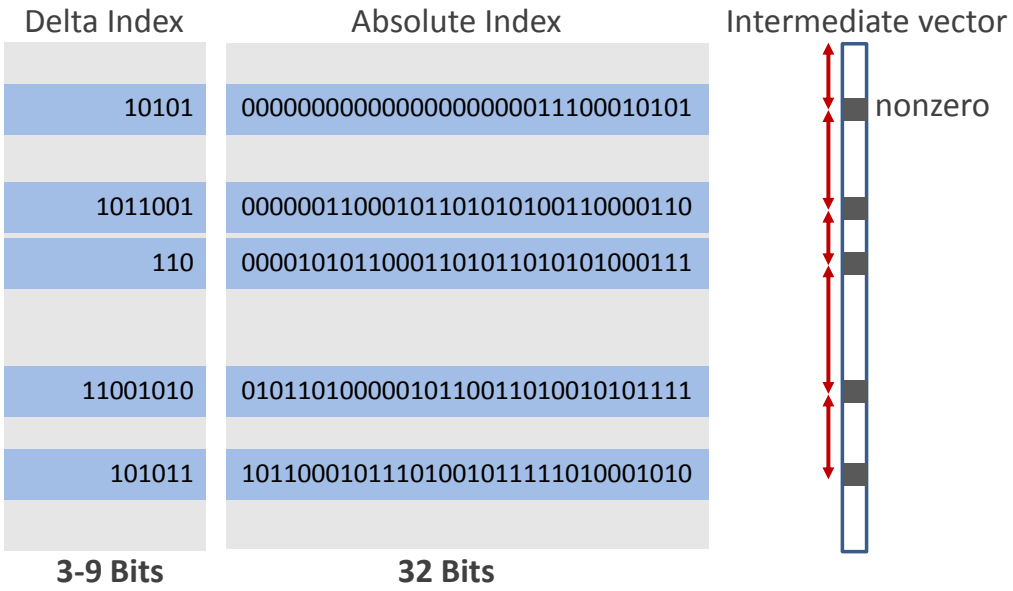


Figure 4.17: Delta index vs absolute index as meta-data.

Figure 4.17, bits required to represent distance between two nonzeros, namely 'delta index', can be significantly smaller than absolute index.

Variable Length Delta Index (VLDI)

Minimum number of bits required to represent delta index varies since the distances among nonzeros vary. To render delta index to be practically usable we propose to use a method, namely Variable Length Delta Index (VLDI), that enables allocation of variable bit width for meta-data. This process is explained in Figure 4.18 using an example. The original delta index of a nonzero requires 17 bits to express the distance from its previous nonzero. We first divide the original delta index into multiple 'VLDI blocks' of predefined width. In this example, the block size is 7 bits. If required, VLDI block comprising the most significant bits is padded with extra zeros to encompass the entire block as shown in Figure 4.18. Afterwards, each block is appended with

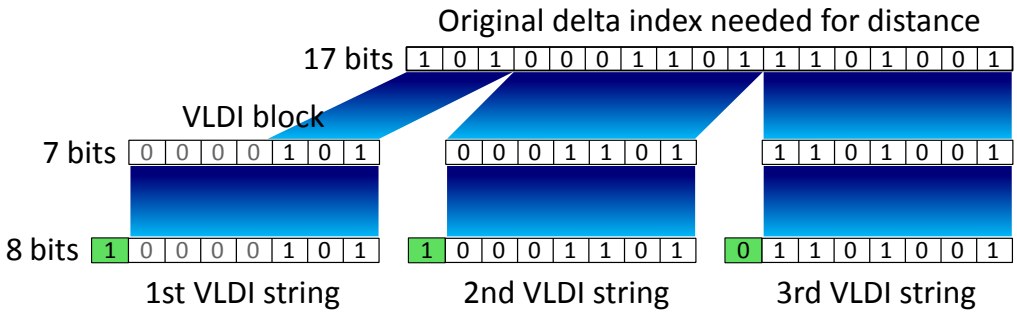


Figure 4.18: Construction of VLDI strings from delta index

an extra leading bit to construct a 'VLDI string'. This extra bit helps in determining propagation of the original delta index. A '1' indicates continuation to the next string while a '0' confirms the termination of the original delta index. This way of segmenting the delta index into multiple VLDI strings provides a practical way of exploiting this data compression opportunity. It should be noted that VLDI is only feasible for sequential generation and access of a stream of elements. As the matrix stripes are stored in RM-COO format and only read from DRAM by streaming access, it is also possible to apply VLDI to compress the matrix stripes. In this case, the meta-data compressed by VLDI is the row index of each nonzero.

VLDI String Length

The optimum VLDI string length, besides the sparsity of the matrix itself, directly correlates to the number of nonzeros in the matrix stripes ($\boldsymbol{A}^k$) and sparse vectors($\boldsymbol{v}^k$) that indirectly depends on the on-chip memory size. With smaller on-chip memory the matrix stripe that can be stored becomes narrow and renders more distance among nonzeros of $\boldsymbol{A}^k$ and $\boldsymbol{v}^k$ on average. Hence, a larger fixed length for the VLDI block is expected to be more efficient as it will reduce the one bit overhead of each VLDI string. However, making the overall string length too wide will cause wastage. Hence,

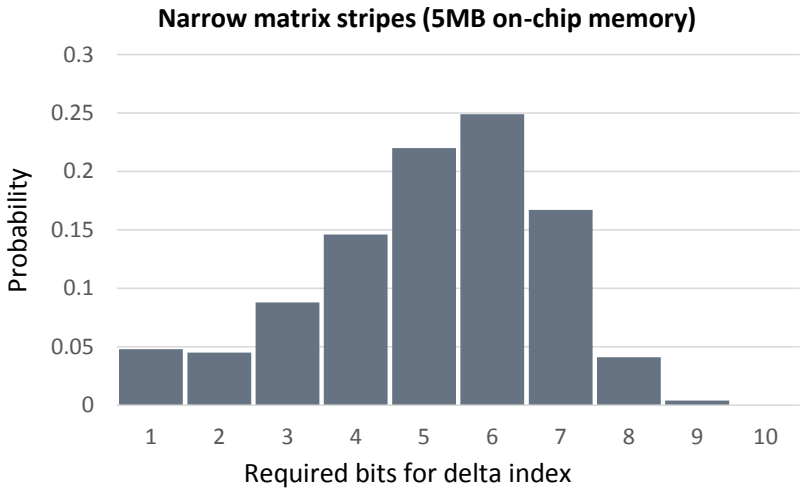
it is important to select the proper VLDI string length given an hardware platform's on-chip fast memory size and sparsity of the problem to achieve efficient compression.

For example, Figure 4.19 shows probability distribution of delta index width for two different on-chip memory sizes, 5MB and 35MB. A randomly generated Erdos Rényi [74] matrix with dimension of 80M×80M and average degree of 3 is used for this example. We have computed the total off-chip traffic for both on-chip storage sizes for a range of VLDI string lengths. Minimum traffic occurs for VLDI string length of 9 bits and 5 bits, i.e. VLDI block length of 8 bits and 4 bits, for 5MB and 35MB on-chip memory sizes accordingly for this problem. Hence, we can tune hardware design parameters for a given memory resource and problem characteristics. FPGA based custom platforms have advantage over ASIC in this regard as it is easier to fine tune VLDI parameters for problems with different sparsity.

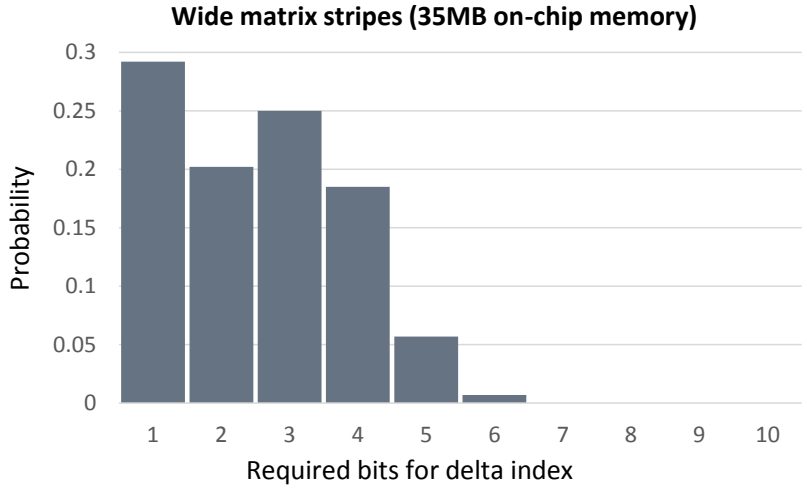
Advantage

To elaborate the VLDI meta-data compression benefit we have shown the total off-chip traffic for the example problem of 80M×80M random sparse matrix given above using 20MB on-chip memory in Figure 4.20. Here we have separately shown the compression capability when only $\boldsymbol{v}^k$ (intermediate sparse vector) is compressed and when both $\boldsymbol{A}^k$ (matrix stripe) and $\boldsymbol{v}^k$ are compressed. Several data precision for the value of nonzeros are used for comparison. Since only meta-data is compressed by VLDI, the compression ratio increases as data precision is decreased. In many real life graphs, the edges of sparse matrix is unweighted, i.e. binary matrix, where we only have meta-data for each nonzero. In such cases VLDI can provide maximum compression benefit.

It should be noted that when the computation throughput, i.e. number of nonzeros processed per cycle, of PSU and MAU is less than the streaming bandwidth of main



(a) VLDI string length of 9 bits (i.e. VLDI block of 8 bits) cause minimum off-chip traffic.



(b) VLDI string length of 5 bits (i.e. VLDI block of 4 bits) cause minimum off-chip traffic.

Figure 4.19: Probability distribution of delta index widths for two different on-chip memory sizes and a randomly generated Erdos Rényi graph with dimension 80M×80M and average degree of 3.

memory VLDI only improves energy efficiency, but does not provide any performance improvement. This is because no matter how much the data is compressed, computation throughput remains unaltered. However, as shown in Chapter 6, when the computation

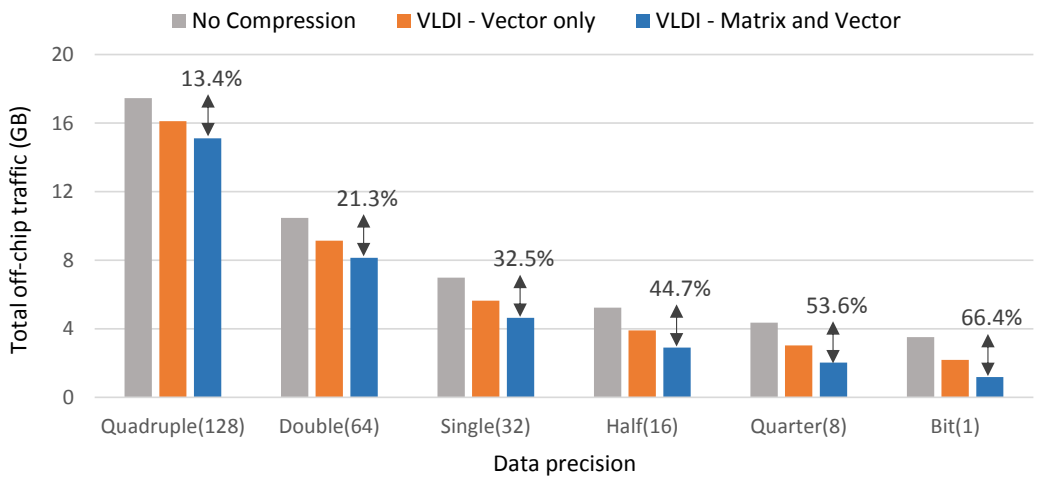


Figure 4.20: off-chip traffic reduction using VLDI meta-data compression.

throughput is higher than the streaming bandwidth of DRAM, VLDI provides both performance and energy efficiency improvement. This is because VLDI can actually provide more data to the computation unit to process and eliminate the stalls due to relatively slower off-chip data transfer speed.

4.6 Summary

In this chapter we have elaborated the implementation of step 1 and 2 of our proposed SpMV algorithm. We have shown that besides main hardware primitive, i.e. HCLAM PRaP merge, we need few small IMN networks for parallelization. Efficient reduction hardware to handle collisions in various scenarios have also been presented. We have demonstrated how Bloom Filter can be used to efficiently process HDNs in power-law graphs. We have presented VLDI meta-data compression technique that is particularly enabled by Two-Step algorithm. These techniques enhances Two-Step

SpMV implementation by improving computation efficiency and reducing off-chip traffic volume.

Chapter 5

Iterative Two-Step SpMV and On-chip Memory Requirement

Contents

Many applications that use SpMV kernel conducts in iterative manner where the resultant vector $\boldsymbol{y}$ of one iteration serves as the source vector $\boldsymbol{x}$ in the following iteration. In this chapter we will demonstrate an optimization for Two-Step algorithm implementation that decreases off-chip traffic and significantly improves computation throughput when SpMV kernel runs for multiple iterations. This optimization technique

overlaps the different computation steps of Two-Step SpMV. In this work, we have chosen widely prevalent PageRank algorithm to demonstrate this SpMV optimization for iterative applications.

In the second part of this chapter, we will elaborate the on-chip memory requirement of our proposed architecture and current state of the art solutions. We will explain in detail how strong dependence on fast on-chip memory adversely affects accelerator scalability and efficiency. As mentioned before, one of the main intents of this work is to handle very large graphs (~ billion nodes) in shared memory environment and efficient on-chip memory management is imperative in achieving this goal.

5.1 PageRank

PageRank is an iterative ranking algorithm that computes relative importances of the nodes in a graph. One application of PageRank is in World Wide Web, where it ranks the web pages for a particular keyword search according to the probability of anyone randomly surfing landing on that specific web page. Mathematical representation for an iteration of PageRank algorithm on a static graph can be given as following.

$$\boldsymbol{x}^T_{(i+1)} = \underbrace{\alpha \boldsymbol{x}^T_i \boldsymbol{A}}_{\text{SpMV}} + \underbrace{(1-\alpha)\boldsymbol{x}^T_i \frac{\boldsymbol{e}\boldsymbol{e}^T}{N}}_{\text{constant addition}} \tag{5.1}$$

In Equation 5.1, $\boldsymbol{x}^T_{(i+1)}$ is the resultant vector at iteration i that is also known as the PageRank vector. $\boldsymbol{A}$ is the sparse matrix of dimension $(N \times N)$ and α is a constant damping factor. The teleportation matrix, $\boldsymbol{e}\boldsymbol{e}^T/N$, models the random probability of a surfer to jump to any page with uniform distribution, where $\boldsymbol{e}$ is a column vector with

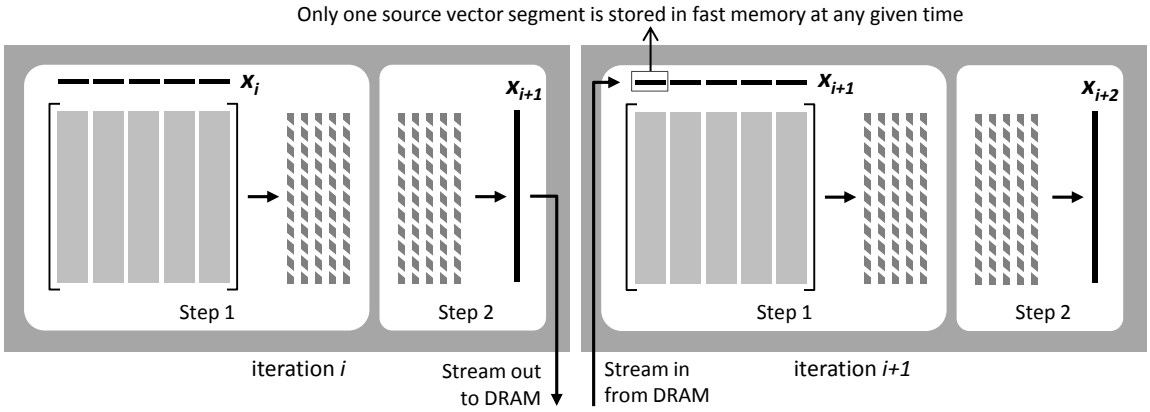


Figure 5.1: Two-Step SpMV driven PageRank with independent iterations (*PR_TS*).

constant 1 for each element. The second term of Equation 5.1 is effectively a constant addition, whereas the first term is SpMV operation that dictates PageRank algorithm's performance and efficiency. Generally, SpMV operation is conducted multiple times in an iterative manner until PageRank vector converges.

5.1.1 SpMV Optimization by Iteration Overlap

PageRank implementation using iterative Two-Step SpMV in a straightforward way is depicted in Figure 5.1. At iteration i the entire Two-Step SpMV operation is conducted and at the end of step 2 the resultant dense PageRank vector, $\boldsymbol{y}_i = \boldsymbol{x}_{i+1}$, is streamed out to DRAM since it is too large to be stored in on-chip memory. In the next iteration $i + 1$, source vector $\boldsymbol{x}_{i+1}$ is streamed back to the computation core from DRAM for step 1 of Two-Step SpMV. We name this implementation of PageRank as *PR_TS*.

In *PR_TS* each iteration is completely independent and the SpMV steps are sequential among the iterations. However, we can parallelize the steps of SpMV among iterations as depicted in Figure 5.2. Effectively, computations of two consecutive iterations are overlapped that eliminates the round trip of $\boldsymbol{x}_{i+1}$ to/from DRAM at the

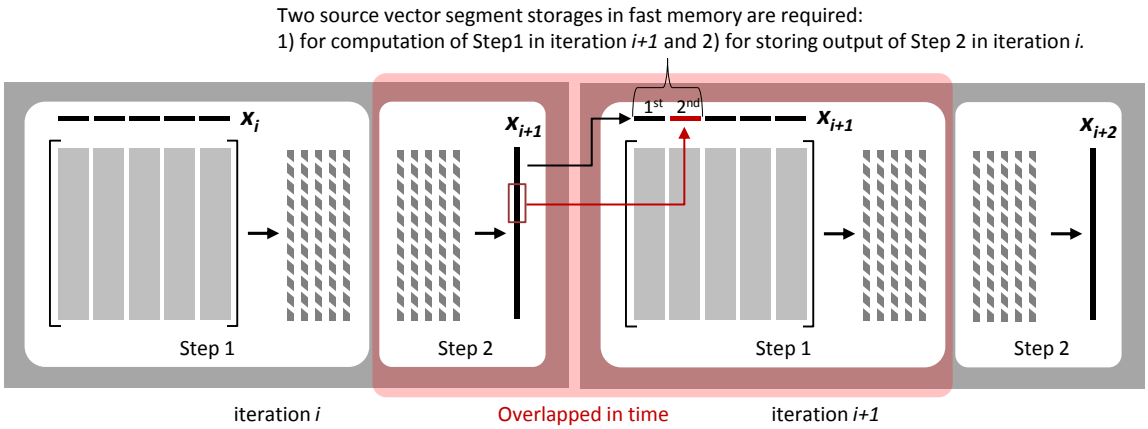


Figure 5.2: Off-chip traffic optimized PageRank with iteration overlap (*PR_ITS*).

transition of iterations. The main enabler of this optimization by iteration overlap is the fact that resultant vector of iteration i, $\boldsymbol{x}_{i+1}$, is generated sequentially in Two-Step algorithm and despite not being able to store the entirety of it on chip, it is possible to store a segment similar to the source vector $\boldsymbol{x}_i$. Once a segment of $\boldsymbol{x}_{i+1}$ is completely generated and stored in the on-chip memory, it is possible to initiate computation of step 1 in the next iteration $i+1$. While step 1 of iteration $i+1$ is conducted using the 1^{st} segment of the source vector $\boldsymbol{x}_{i+1}$, step 2 of iteration i continues concurrently and stores the 2^{nd} segment of the resultant vector $\boldsymbol{x}_{i+1}$ in another on-chip buffer. Thus, computation of step 2 in iteration i and step 1 in iteration $i+1$ are overlapped in time. A pseudocode of iteration overlapped Two-Step SpMV driven PageRank, named as *PR_ITS*, is given in Pseudocode 2.

The cost of iteration overlapped Two-Step SpMV is that now it is required to buffer two source vector segments in on-chip fast memory instead of one. Hence, for a given amount of on-chip memory the maximum matrix dimension that *PR_ITS* can handle is roughly half of what *PR_TS* can handle. Therefore, optimization by iteration

Pseudocode 2: Two-Step SpMV driven PageRank with off-chip communication optimization by iteration overlap.

```
1  T = Total number of iterations
2  for i = 0 to T − 1 do
3  |  STEP 1
4  |  for k = 0 to n − 1 do
5  |  |  Stream in Matrix Column Block A^k
6  |  |  u ← 0
7  |  |  for All rows A^k_{p,:} with nnz > 0 do
8  |  |  |  for Each non-zero A^k_{p,q} in A^k_{p,:} do
9  |  |  |  |  Random access to vector segment x^k_{[i+1]}
10 |  |  |  |  u_p ← α · A^k_{p,q} · x^k_{q[i+1]} + u_p
11 |  |  |  end
12 |  |  end
13 |  |  Sparsify u to v^k_{[i+1]}
14 |  |  Stream out v^k_{[i+1]} to main memory
15 |  end
16 |  STEP 2
17 |  for p = 0 to N − 1 do
18 |  |  for k = 0 to n − 1 do
19 |  |  |  Stream in v^k_{[i]}
20 |  |  |  x_{p[i+1]} ← x_{p[i+1]} + v^k_{p[i]} [Multiway merge]
21 |  |  end
22 |  |  c_{[i+1]} ← c_{[i+1]} + x_{p[i+1]}
23 |  |  x_{p[i+1]} ← x_{p[i+1]} + ((1−α)/N) c_{[i]} [Constant addition]
24 |  end
25 |  Buffer x_{[i+1]} on chip
26 end
```

overlap offers a trade off between maximum problem dimension vs performance and energy efficiency.

5.1.2 Advantages of Iteration Overlapped Two-Step SpMV

Higher Throughput

The most important benefit of iteration overlap is that it significantly increases the computation throughput of the overall SpMV operation. This is because both the computation units for step 1 and 2, i.e. PSU and MAU, operates concurrently, whereas for a single run of Two-Step SpMV one of the computation units remains idle. Thus, *PR_ITS* enables the entire silicon area to be active for all the iterations (except the very first and very last one), which helps to properly utilize extreme off-chip streaming bandwidth offered by 3D stacked HBM. The sustained throughput of *PR_ITS* vs *PR_TS* using our ASIC implementation of Two-Step SpMV is shown in Figure 5.3. For *PR_TS*, loading of source vector segment, partial SpMV and multi-way merge are conducted in sequential manner at sustained throughput pertaining to their computation units, which are less than system's main memory bandwidth. For *PR_ITS*, all operations in step 1 and 2 runs parallelly that almost doubles the maximum possible computation throughput overall. In this way, using the same silicon area and resources performance of the accelerator can be increased significantly and full utilization of HBM bandwidth becomes possible. In fact, the computation throughput of our developed ASIC is well over the system's 512GB/s and can saturate the bandwidth of almost three HBM2s (768GB/s).

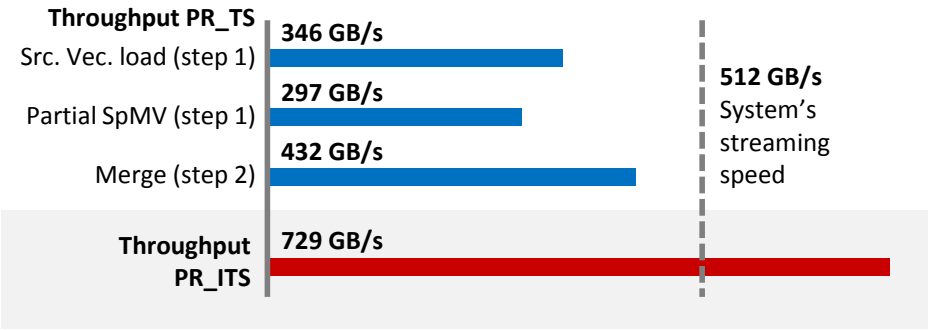


Figure 5.3: Sustained computation throughput/streaming speed of *PR_ITS* vs *PR_TS* implemented in ASIC platform.

Off-Chip Traffic Reduction by Iteration Overlap

There is an additional benefit of reduced off-chip traffic in iteration overlapped Two-Step SpMV. This is due to elimination of the round trip of resultant and source vector to/form main memory. This reduction becomes more impactful when the matrix gets sparser. This is because with more sparsity, i.e. less average degree, the overall off-chip traffic caused by matrix decreases while the off-chip traffic due to source and resultant vector remains constant. To demonstrate this, we generated five Erdos Rényi random graphs of dimension 1B×1B with different sparsity. Figure 5.4 shows the total off-chip traffic for 20 iteration PageRank on these graphs for both unoptimized and optimized Two-Step SpMV, which are termed as *TS* and *ITS* (Iteration-overlapped *TS*) accordingly. We can see that the ratio of reduction in traffic using iteration overlap optimization becomes larger as average degree decreases. For example, with a highly sparse graph of average degree 1.2, 26% more DRAM traffic would incur if iteration overlapped Two-Step SpMV is not implemented.

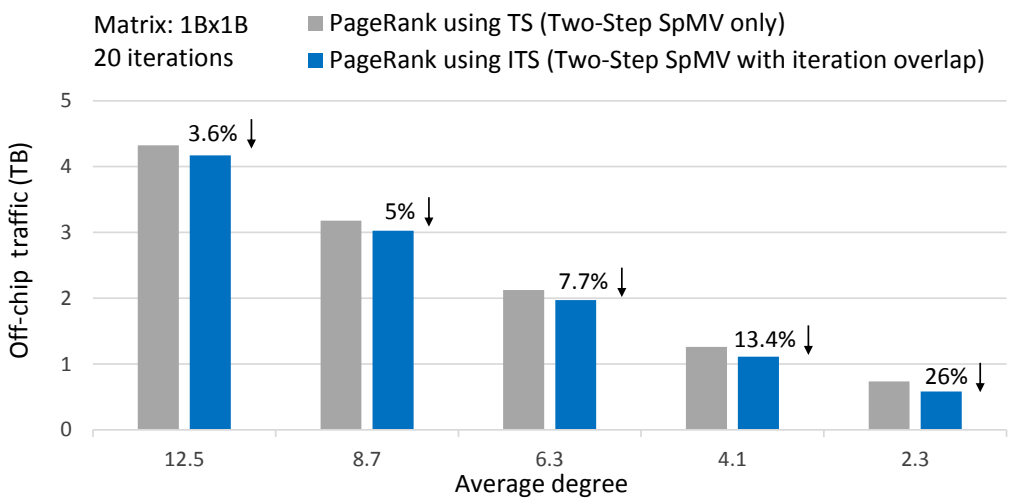


Figure 5.4: Off-chip communication of PageRank using *TS* vs *ITS* (Iteration-overlapped *TS*).

5.2 On-chip Memory Requirement

Dependence on scarce on-chip fast memory is the biggest constraint in handling large problems for most of the hardware accelerators developed for sparse kernels in the literature. One of the key contribution of this work is that our developed accelerator can handle much larger problems while requiring significantly less on-chip fast memory (such as SRAM or eDRAM based cache, scratchpad, etc.) relative to the solutions found in literature. This is due to less dependence on fast memory to scale. Figure 5.5 shows the on-chip memory requirement for Two-Step SpMV implementation. The majority of fast memory is required for the storage of source vector segment. Rest of the on-chip memory is mainly required for the prefetch buffer to store DRAM page size blocks for n intermediate sparse vectors to ensure DRAM streaming. Even though each intermediate sparse vector is accessed sequentially, selection of the vector during the merge process is random. Hence to guarantee full DRAM streaming and total amortization of DRAM page opening cost, we prefetch the entire DRAM page size data

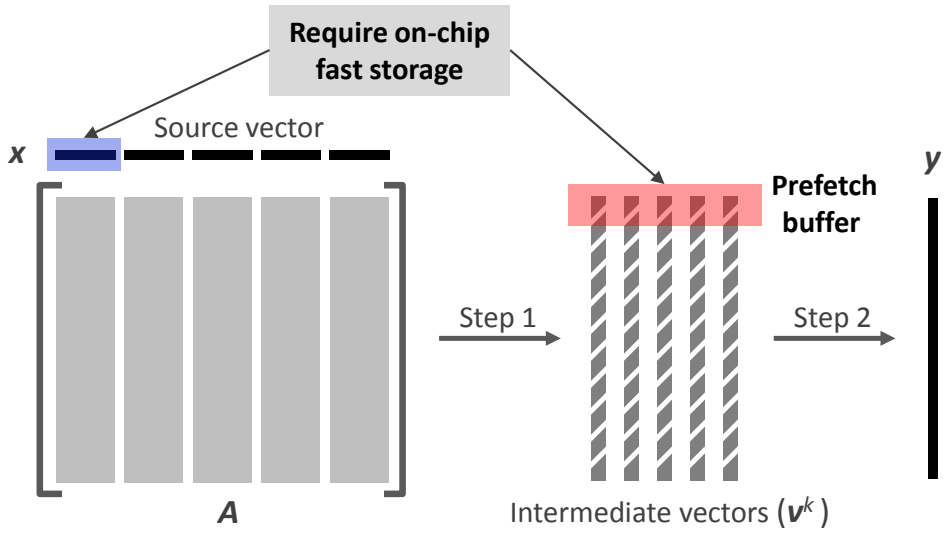


Figure 5.5: On-chip storage requirement for proposed Two-Step SpMV.

block once any sparse vector is accessed in DRAM. This prefetch buffer is explained in more detail in Sec. 3.6 of Chapter 3.

5.2.1 Comparison with Current Solutions

For the developed ASIC in this work, the overall computation core requires 0.5MB of on-chip memory. For the source vector we have allocated 8MB scratchpad. Page (row buffer) size of HBM2 is 1KB and we have allocated 1.25KB for prefetched data (instead of 1KB) for each sparse vector. The extra memory is allocated so that new load request can be issued before completely depleting the prefetched block and the load latency can be hidden. For this given design point, we have implemented a 2048-way multi-way merge network. Hence, n = 2048 and we require $2048 \times 1.25\text{KB} = 2.5\text{MB}$ on-chip fast memory for prefetch buffer overall. The total on-chip fast memory requirement for our proposed ASIC based accelerator is (0.5MB + 8MB + 2.5MB) = 11MB. To

place this into perspective, we have listed the on-chip memory requirement and the largest problem size reported for a number of custom hardware and COTS solutions of current literature in Table 5.1. We compared these shared memory solutions against our proposed Two-Step SpMV without and with optimization by iteration overlap, which are named as *TS* and *ITS* (Iteration-overlapped *TS*) accordingly. As mentioned before, *ITS* causes the maximum problem size to be half of what is possible with *TS*. Nevertheless, we can see our proposed solution requires relatively less on-chip memory while being able to handle graphs with significantly larger dimension. For example, despite using a huge 32MB eDRAM scratchpad the ASIC based solution in [33] can efficiently handle a maximum graph size of 8 million nodes only. On the other hand, our proposed ASIC can handle graph with multiple billion nodes using only 11MB fast memory.

Table 5.1: Fast on-chip memory requirement and largest graph dimension comparison of current and proposed solutions.

Solution	Fast on-chip memory size (MB)	Max. vertices (Million)
FPGA [36]	8.4	2.3
ASIC [33]	32.0	8.0
CPU (single socket) [75]	20.0	95.0
CPU (dual socket) [76]	50.0	118.0
ITS (proposed ASIC)	11.0	2000.0
TS (proposed ASIC)	11.0	4000.0

It is should be noted that, since there is a lot of room to expand on-chip memory using existing technology, our proposed solution can be scaled easily for significantly larger problems. For example, if the source vector buffer is expanded from 8MB to 16MB, we will be able to handle graphs with twice larger dimension, i.e graphs with 4B and 8B vertices using *ITS* and *TS* accordingly. This ability to scale is imperative for

FPGA based implementation in handling large graphs. This is because FPGA has very limited amount on-chip Block RAM (BRAM) and efficiency of FPGA implementation largely depends on the proper utilization of Block RAM (BRAM). FPGA solutions in current literature have reported to handle only small graphs, such 2.3 million nodes in [36]. It should be noted that the total number of edges in a graph is not relevant in determining on-chip memory size for Two-Step SpMV implementation. Total edges only dictates the requirement for main memory storage for our developed accelerator.

5.2.2 Energy Efficiency

One issue which is often overlooked in custom accelerator design using large on-chip memory is the energy usage by large memory blocks. For implementation of large on-chip storage, eDRAM is often used as it is more compact than SRAM. However, eDRAM accounts for significantly large portion of the total energy consumed by the system. For example, Figure 5.6 depicts the normalized energy consumed by a accelerator system using 16MB eDRAM scratchpad and ASIC computation core with HBM main memory system. We have simulated a full SpMV run on the example graph of dimension 80M×80M mentioned in Sec. 4.5. Destiny [77] memory modeling tool is used to calculate eDRAM energy consumption. As shown in Figure 5.6, due to high leakage power, eDRAM consumes almost 70% of the entire energy required for SpMV computation. Leakage power of eDRAM is directly proportional to the memory block size and, hence, smaller eDRAM block achieves higher energy efficiency. Authors of [33] also reported 90% of the entire system energy to be consumed by a 64MB eDRAM scratchpad. Thus less dependence on fast on-chip memory also provides advantage in energy efficiency besides scalability.

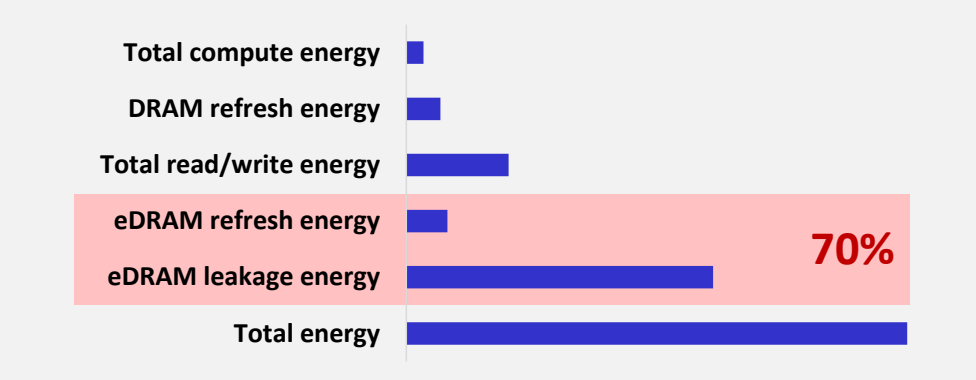


Figure 5.6: Energy consumption by different parts of the system for Two-Step SpMV acceleration.

5.2.3 Fast Storage vs Compute

An interesting observation in the implementation of Two-Step SpMV for large and highly sparse graphs is that the off-chip traffic is not significantly impacted by on-chip storage size. Figure 5.7 elaborates this phenomenon. It shows total off-chip traffic of Two-Step SpMV operation on the random graph mentioned in Sec. 4.5 (80M nodes and average degree 3) for different sizes of on-chip storage. Despite increasing the fast memory from 5MB to 50MB, the overall off-chip traffic reduction is only 5.4%. This is very insignificant given the 10x increase on-chip memory. This is because large sparse matrices have only a few reductions per row and even very wide matrix stripes fail to have any meaningful effect on it. Hence on-chip memory size parameter for our accelerator design is mainly dictated by compute capability of our system.

For any target maximum matrix dimension, there is a trade-off between on-chip storage and merge network size in MAU. If the merge network can handle more lists, it is possible to decrease the on-chip storage. This is because less fast storage renders source vector segment smaller, which eventually causes higher number of matrix stripes

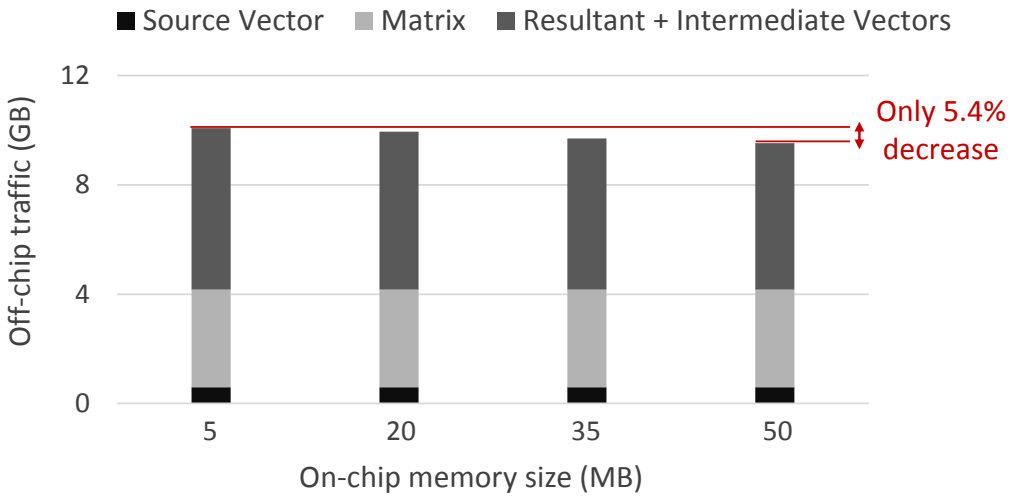


Figure 5.7: Total off-chip traffic of Two-Step SpMV for different on-chip memory size.

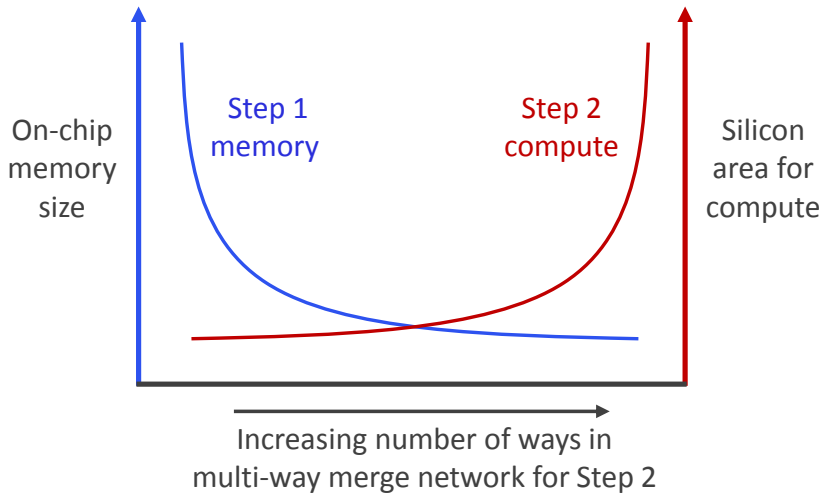


Figure 5.8: Trade-off between on-chip storage and computation in designing Two-Step SpMV accelerator.

and intermediate vectors. If eDRAM is used, smaller on-chip may result in higher energy efficiency. This design trade-off is illustrated in Figure 5.8. Thus for a given silicon area, as in ASIC design, and target problem size design parameters should be

chosen by factoring in this trade-off between memory vs compute while implementing Two-Step SpMV.

5.3 Summary

With the example of PageRank application, we have demonstrated a technique by overlapping the SpMV steps in consecutive iterations that significantly increases the streaming computational throughput and reduces off-chip traffic. Furthermore, we have shown that our proposed solution uses significantly less on-chip memory to handle large graphs in comparison to existing solutions in literature. As on-chip memory is a critical resource for scaling, it is essential to ensure efficient utilization of it in custom hardware design for being able to handle large problems.

Chapter 6

Evaluation of Performance & Energy Efficiency of SpMV

Contents

In this chapter we will evaluate the performance and energy efficiency of our proposed SpMV accelerator. As mentioned in previous chapter, we have designed an ASIC chip for the implementation of Two-Step SpMV that is currently being fabricated. We have also ported this ASIC design to FPGA platform and implemented the proposed accelerator to a relatively smaller design point. To demonstrate the performance and efficiency benefit of our proposed accelerator on both ASIC and FPGA platforms we

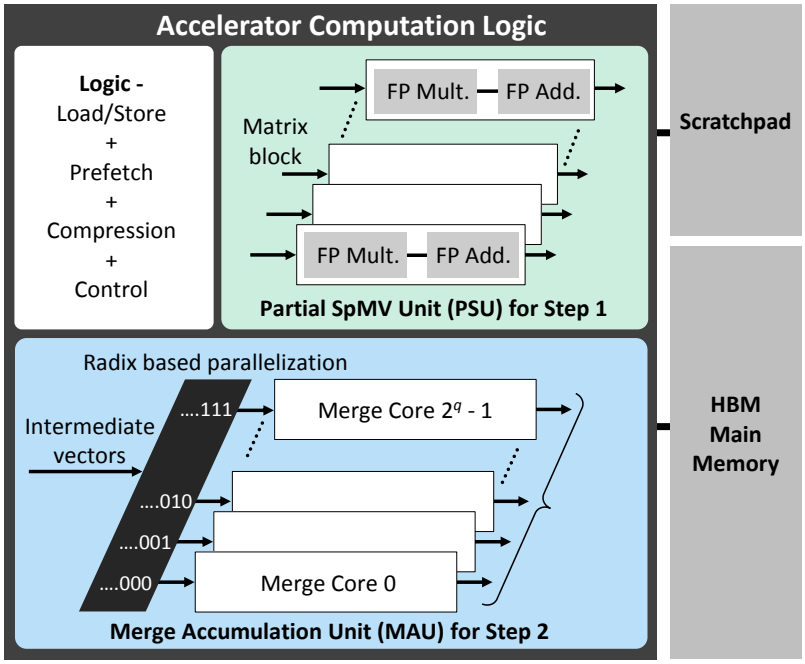


Figure 6.1: 16nm FinFET ASIC (currently under fabrication) for sparse matrix kernel acceleration.

have compared experimental results with various benchmarks. A number of custom hardware and GPU SpMV accelerators from recent literature are used benchmarks. Additionally, we have also used Intel® Math Kernel Library (MKL) SpMV routine on CPU and co-processor (Xeon Phi) as another set of benchmarks.

6.1 Implementation Platform & Design Points

We have implemented the proposed SpMV accelerator and optimizations techniques in multiple design points in several custom hardware platforms. A basic block diagram of the proposed accelerator is depicted in Figure 6.1 and a list of implementations for various design points is Table 6.1 that also includes the maximum problem dimension and maximum computation throughout of each implementation.

Table 6.1: Maximum graph dimension and throughput for different design point and implementation variations of proposed SpMV accelerator.

Platform/ Design point	Implementation ID	Maximum nodes (M)	Sustained computation throughput (GB/s)
ASIC	TS_ASIC	4000	432
	ITS_ASIC	2000	729
	ITS_VC_ASIC	2000	656
FPGA1	TS_FPGA1	134.2	96
	ITS_FPGA1	67.1	178
FPGA2	TS_FPGA2	67.1	190
	ITS_FPGA2	33.6	357

6.1.1 ASIC based Implementation

The accelerator computation unit for ASIC is done using System Verilog and currently being fabricated in 16nm FinFET technology. An actual image (from Cadence® tool) of the ASIC and key specifications are given in Figure 6.2. As the chip is currently being fabricated, these specifications are from post physical synthesis (after place and route) layout of the design. Cadence® Innovus™ is used for area and frequency measurement and Cadence® Voltus™ is used for power measurement. One noteworthy aspect of this chip is that it uses synthesized SRAM blocks, also known as LiM technology [63–65], distributed all over the chip to facilitate fine grain data access during computation.

For Partial SpMV Unit (PSU) implementation in ASIC, sixteen parallel single precision FP multiplier and adder chains are implemented. For implementation of MAU, sixteen parallel HCLAM PRaP 512-way merge cores are used per DRAM channel, which means 4 LSB radix bits are used for pre-sorting. The system is designed to work with four HBM channels and, hence, MAU contains a 2048-way multi-way merge merge network that has a overall maximum throughput of 64 resultant vector elements per

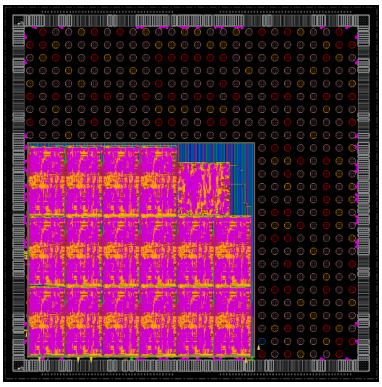


Figure 6.2: 16nm FinFET ASIC (currently under fabrication) for sparse matrix kernel acceleration.

cycle. The two other parts of ASIC based accelerator, i.e. HBM main memory and the eDRAM scratchpad, are emulated using Cacti [78] and Destiny [77] tools. This ASIC is designed to work with two 3D-stacked 2^{nd} generation High Bandwidth Memories (HBM2s) [66, 67] as main memory, which are connected through interposer [68]. One single HBM2 provides 256GB/s aggregated bandwidth. Hence, this chip is designed to saturate the extreme off-chip bandwidth of total 512GB/s offered by state of the art 3D stacked DRAM technology. On the other hand, we have simulated 10.5MB eDRAM scratchpad as fast memory for vector storage (2MB) and prefetch buffer (2.5MB).

For the design point of ASIC described above, we test the accelerator for three different implementation variation as mentioned in Table 6.1. *TS_ASIC* represents straightforward Two-Step SpMV along with optimization techniques described in Chapter 4 except iteration overlap and VLDI compression. *ITS_ASIC* indicates *TS_ASIC* implementation along with the optimization using iteration overlap. Lastly, *ITS_VC_ASIC* means *ITS_ASIC* implementation including VLDI data compression for intermediate sparse vectors. As *TS_ASIC* only stores one source vector segment in fast memory, the maximum matrix dimension it can handle is 4 billion. This is

twice as large what *ITS_ASIC* and *ITS_VC_ASIC* can handle, i.e. 2 billion, since these variations include iteration overlap optimization. However, *ITS_ASIC* and *ITS_VC_ASIC* has relatively higher computational throughput than *ITS_ASIC* as both PSU and MAU is active due to iteration overlap. *ITS_VC_ASIC* has relatively slower sustained throughput than *ITS_ASIC* in terms of DRAM bandwidth saturation, i.e. GB/s, due to meta-data compression. However, *ITS_VC_ASIC* and *ITS_ASIC* both have the same throughput in terms of number of elements processed per unit time.

6.1.2 FPGA based Implementation

To demonstrate portability of the custom hardware design of our proposed accelerator, we have implemented two design points in Intel® Stratix® 10 FPGA (device 1SG280HU1F50E1VGS3), namely *FPGA1* and *FPGA2* as shown in Table 6.1. While the PSU configuration and memory requirement for source vector segment and prefetch buffer are the same as ASIC, *FPGA1* and *FPGA2* have different configuration for the MAU implementation. *FPGA1* is implements an 64-way multi-way merge merge network overall with 4 LSB radix pre-sorter. On the other hand, *FPGA2* implements smaller 32-way merge with 5 LSB radix pre-sorter, i.e. more parallel merge cores. Therefore, *FPGA1* can handle relatively larger problems than *FPGA2*, but at the cost of less computational throughput. For both *FPGA1* and *FPGA2*, scratchpad memory is synthesized using Block RAM (BRAM). The HBM main memory system is simulated in the same way as ASIC considering four channels.

For each design point *FPGA1* and *FPGA2*, there are two different implementations. One version of the design points implement straightforward Two-Step SpMV without iteration overlap optimization, which are named as *TS_FPGA1* and *TS_FPGA2* for

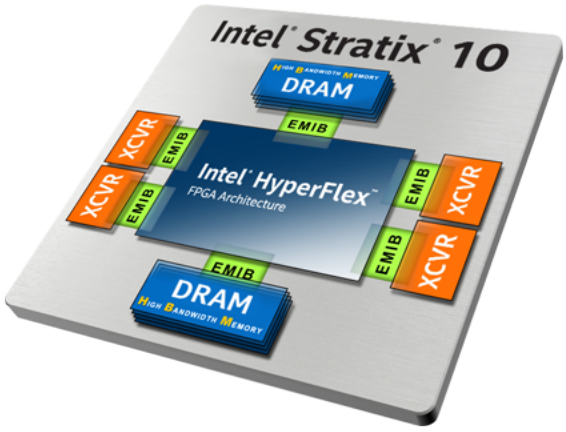


Figure 6.3: Intel® Stratix® 10 FPGA platform [69].

the corresponding design points. The other version of the design points incorporate optimization by iteration overlap that are named as *ITS_FPGA1* and *ITS_FPGA2*. Hence, we have four implementations in total of on Stratix® 10 FPGA as listed in Table 6.1.

6.2 Comparison against Custom Hardware and GPU

We have compared the performance and energy efficiency of our developed accelerator against a number of custom hardware solutions, including both ASIC and FPGA from recent literature. Another set of GPU based SpMV accelerator is also considered from current literature to serve as benchmarks for comparison. A short description of these benchmarks and reference are given in Table 6.2. For the ease of description, we have assigned an ID with each of the benchmarks. It is noteworthy that the ASIC solution in *BM1_ASIC* [33] uses a huge eDRAM scratchpad of 64MB. The set of graphs that are used for comparison against custom hardware and GPU solutions are given in

page 127 and Table 6.4 accordingly. We have considered all relatively large graphs results on which are reported by the related work. It is also noteworthy that most of the reported graphs by these custom hardware and GPU solutions are small (only have few million nodes), whereas our solutions can operate on much larger graphs as shown in Table 6.1. Iterative SpMV application PageRank is run on these graphs using 20 iterations. Reported results for the same operation are collected from pertinent literature for comparison purpose.

Table 6.2: Graph data sets used for comparison with custom hardware (ASIC/FPGA) benchmarks.

Architecture	ID	Description
Custom Hardware	BM1_ASIC	28-nm ASIC, 64 MB eDRAM scratchpad [33]
	BM1_FPGA	Virtex UltraScale+, 25 Mb BRAM & 90 Mb UltraRAM [79]
	BM2_FPGA	Virtex-7, 67 Mb BRAM [36]
GPU	BM1_GPU	8 nodes, Tesla M2050 GPU per node [80]
	BM2_GPU	Single Radeon 5870(RV870) [81]

Table 6.3: Graph data sets used for comparison with custom hardware (ASIC/FPGA) benchmarks.

ID	Description	# Nodes (M)	Avg. Degree	# Edges (M)
FR	Flickr [33]	0.82	12.00	9.84
FB	Facebook [33]	2.93	14.31	41.92
Wiki	Wikipedia [33]	3.56	23.81	84.75
RMAT	RMATScale23 [33]	8.38	16.02	134.22
LJ	LiveJournal [79]	7.80	14.38	69.00
WK	WK [79]	2.40	2.08	5.00
TW	TW [79]	41.6	35.30	1468.40
web-ND	web-NotreDame [36]	0.33	4.61	1.45
web-Go	web-Google [36]	0.88	5.83	5.11
web-Be	web-Berkstan [36]	0.69	11.09	7.60
web-Ta	wiki-Talk [36]	2.39	2.10	5.02

Table 6.4: Graph data sets used for comparison with GPU benchmarks.

ID	Description	# Nodes (M)	Avg. Degree	# Edges (M)
ara-05	arabic-2005 [80]	22.70	28.19	640.00
it-04	it-2004 [80]	41.30	27.85	1150.10
sk-05	sk-2005 [80]	50.60	38.53	1949.40
wiki51	wikipedia-20051105 [81]	1.63	12.08	19.75
wiki60	wikipedia-20060925 [81]	2.98	12.49	37.27
wiki61	wikipedia-20061104 [81]	3.15	15.50	39.38
wiki70	wikipedia-20070206 [81]	3.56	12.62	45.03
edu-01	edu-2001 [81]	9.84	5.81	57.15

6.2.1 Performance against Custom Hardware Solutions

Figure 6.5 shows the speedup of execution time for 20 iteration PageRank using iterative SpMV using our ASIC implementations. In the plot, comparison results for different benchmarks are separated. Our ASIC implementations achieve order of magnitude improvement over the FPGA benchmarks and several times faster than the ASIC benchmark despite significantly less on-chip memory. As expected, solutions with iteration overlap optimization technique, i.e. *ITS_ASIC* and *ITS_VC_ASIC*, has better speedup than *TS_ASIC*. *ITS_VC_ASIC* achieves highest performance as the sustained computation throughput is higher than system's streaming bandwidth of 512GB/s and VLDI compression reduces off-chip traffic.

Figure 6.5 shows speedup with our four FPGA implementations against the custom hardware solutions. These are expected to achieve less performance than our ASIC implementation. However, the overall speedup against these benchmarks are significant. We have used another metric for performance comparison that is Giga Traversed Edges Per Second (GTEPS). Figure 6.6 and Figure 6.7 report GTEPS for

our proposed implementations and the custom hardware benchmarks. We also notice similar significant improvement for GTEPS.

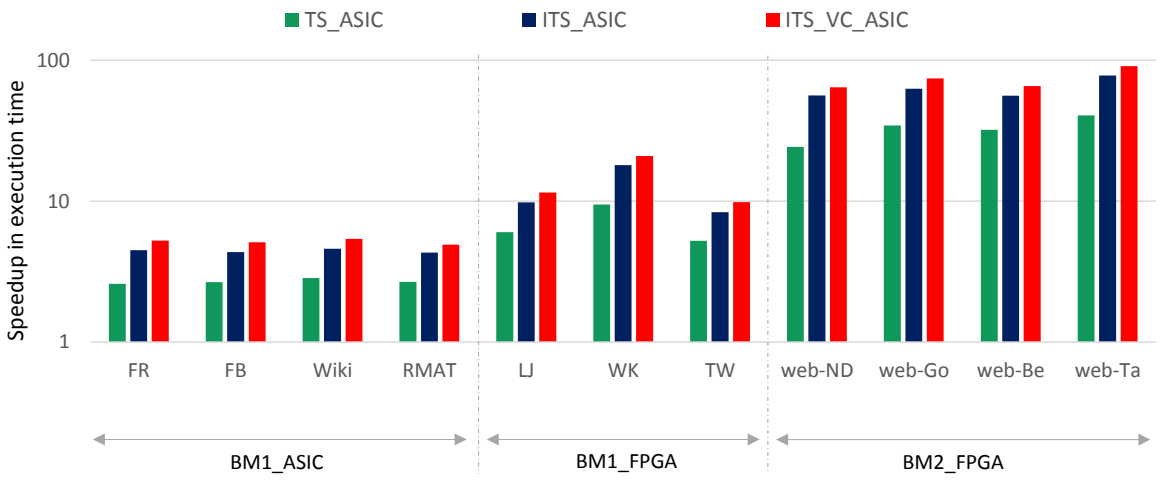


Figure 6.4: Speedup of proposed ASIC over custom hardware benchmarks.

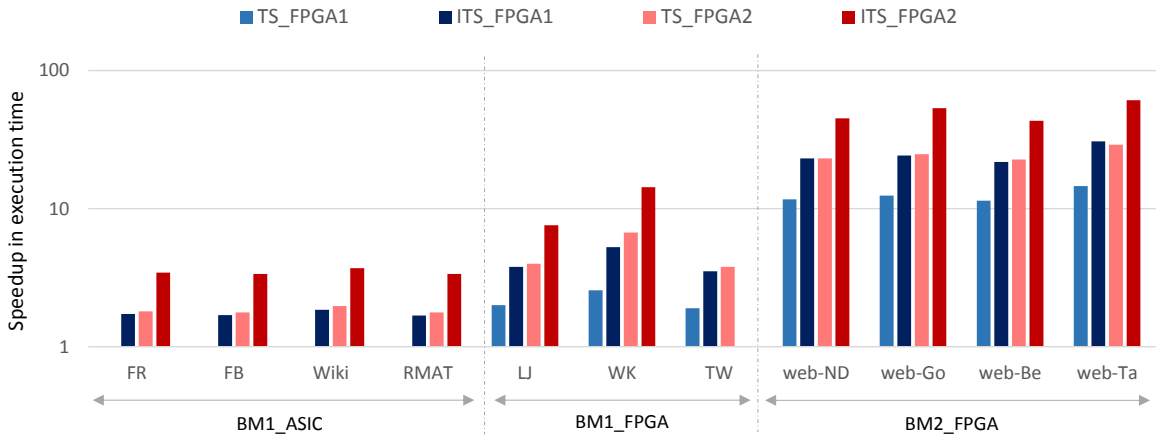


Figure 6.5: Speedup of proposed FPGA implementations over custom hardware benchmarks.

6.2.2 Performance & Efficiency against GPU Solutions

Figure 6.8 and Figure 6.9 show the speedup in execution time for 20 iteration SpMV over GPU benchmarks with our ASIC and FPGA implementations. Using *ITS_VC_ASIC*,

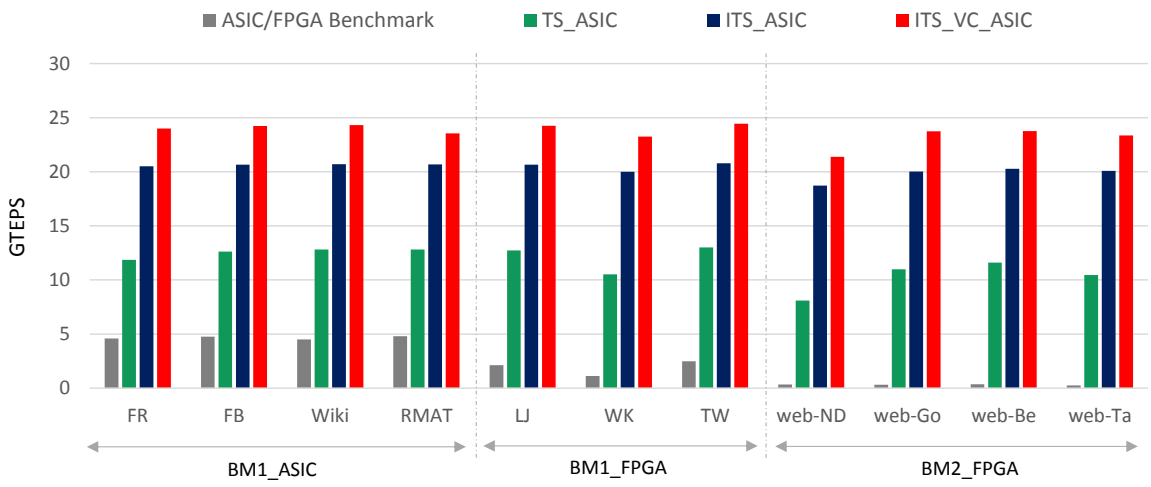


Figure 6.6: Comparison of GTEPS for proposed ASIC against custom hardware benchmarks.

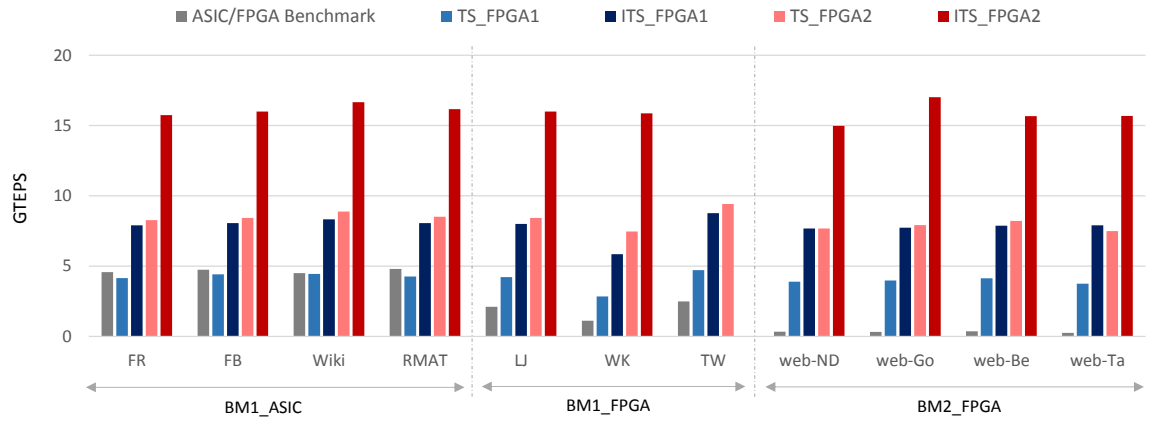


Figure 6.7: Comparison of GTEPS for proposed FPGA implementations against custom hardware benchmarks.

we can achieve multiple orders of magnitude improvement in the execution time against *BM1_GPU*. Comparisons of GTEPS metric are given in Figure 6.10 and Figure 6.11, where we can notice orders of magnitude improvement in performance over GPU benchmarks.

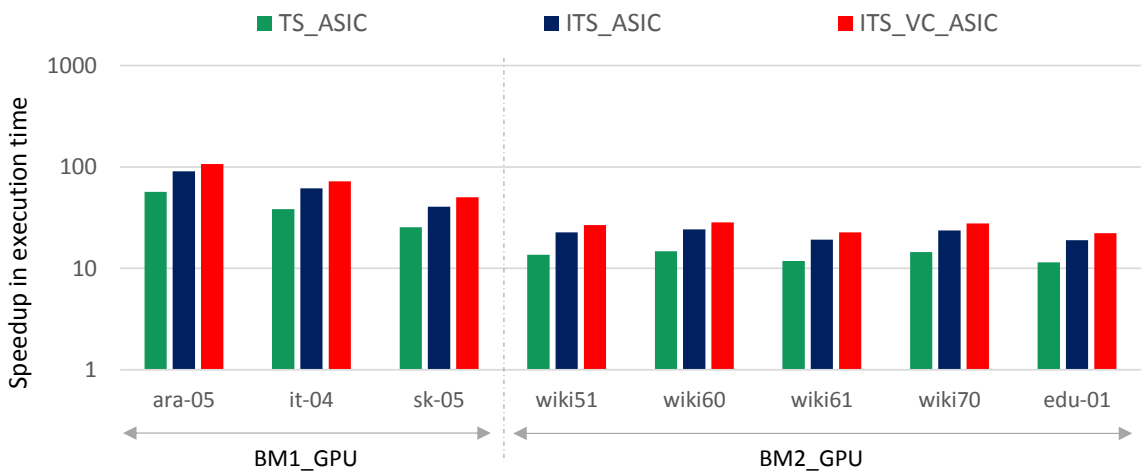


Figure 6.8: Speedup of proposed ASIC over GPU benchmarks.

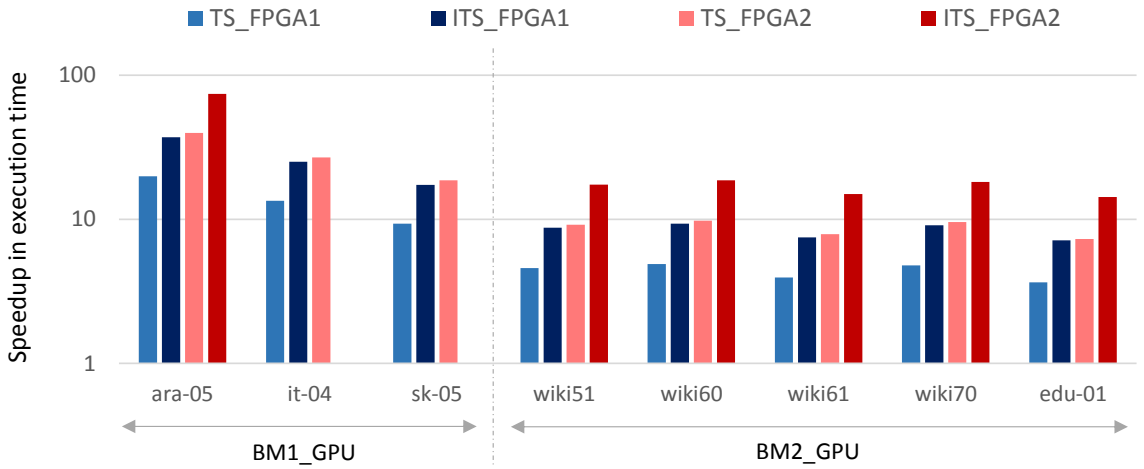


Figure 6.9: Speedup of proposed FPGA implementations over GPU benchmarks.

To demonstrate the improvement in energy efficiency, we have compared the energy per edge traversal for 20 iteration PageRank operation. Comparison results are presented in Figure 6.12 and Figure 6.13. Energy efficiency improvement for proposed ASIC implementations are multiple orders of magnitude for almost every graph. FPGA implementations also achieve significant energy efficiency. This is expected because

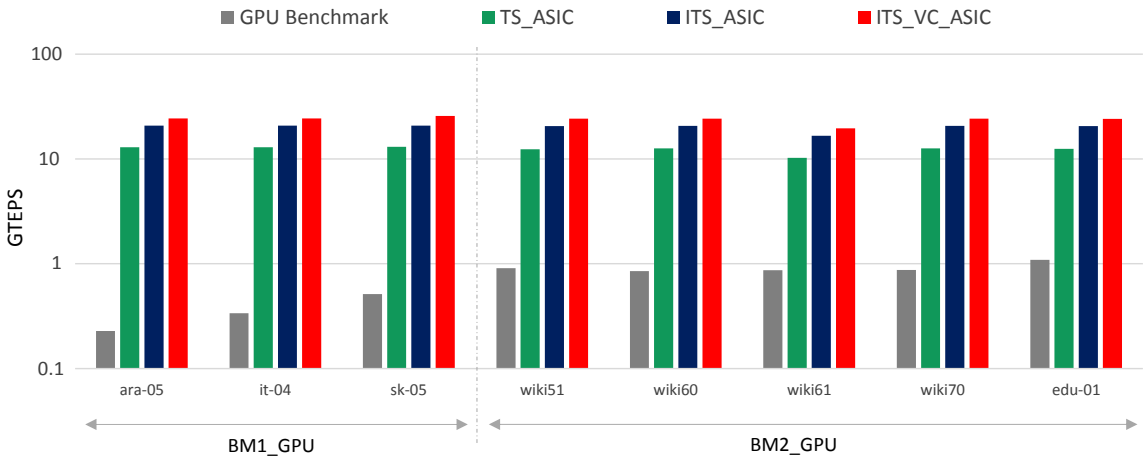


Figure 6.10: Comparison of GTEPS for proposed ASIC against GPU benchmarks.

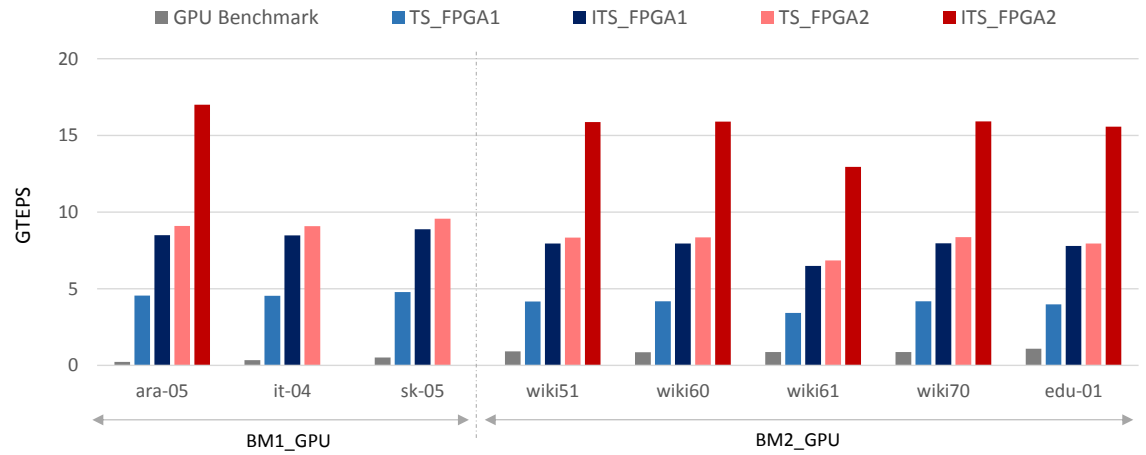


Figure 6.11: Comparison of GTEPS for proposed FPGA implementations against GPU benchmarks for SpMV.

GPUs commonly consume high energy due to large number of parallel cores and arithmetic units.

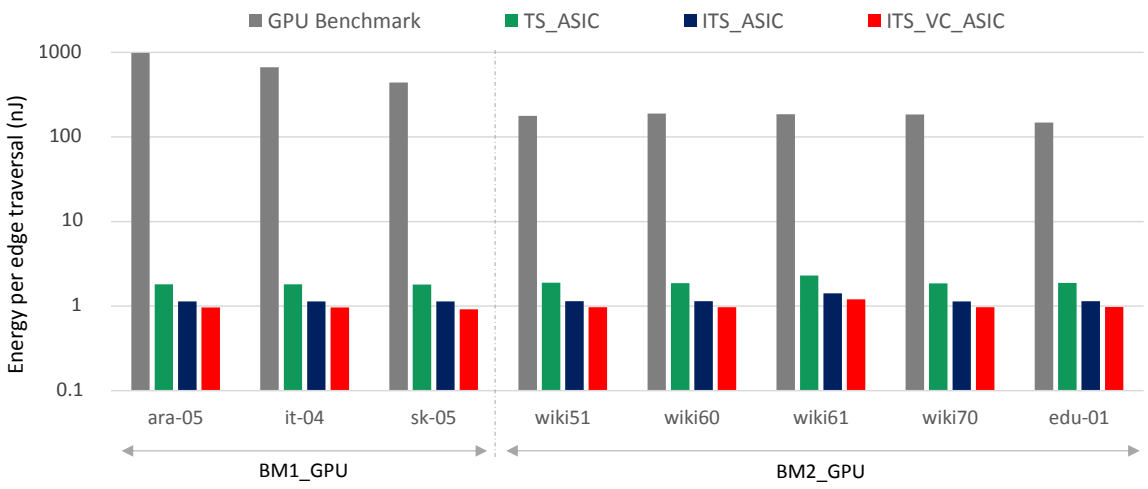


Figure 6.12: Comparison of energy per edge traversal of proposed ASIC against GPU benchmarks.

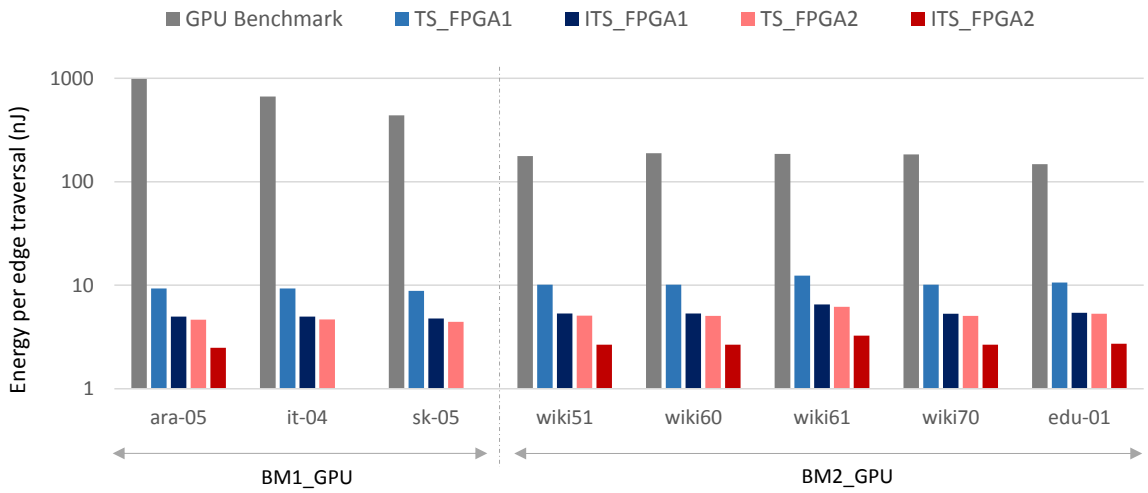


Figure 6.13: Comparison of energy per edge traversal of proposed FPGA implementations against GPU benchmarks.

6.3 Comparison against CPU and Co-Processor

For comparison with CPU and many-core co-processor we have used Intel® MKL routine 'mkl_scoogemv' to run single precision iterative SpMV on dual socket Xeon E5-2620 (22nm, 12 threads) CPU and Xeon Phi 5110P (22nm, 60 cores) co-processor.

Both of the COTS architectures have 30MB LLC. The peak bandwidth is 102GB/s for the CPU and 352GB/s for the co-processor. Data used for comparison with these architectures are listed in Table 6.5. All these graphs (except last six) are collected from University of Floria sparse matrix collection [82]. We also have used a number of random Erdos Rényi [74] graphs for the demonstration purpose of our proposed accelerator's capability in handling large problems. These synthetically generated graphs have names with prefix 'Sy'.

Table 6.5: Graph data sets used for comparison with CPU and many-core co-processor benchmarks.

Name	# Nodes (M)	Avg. Degree	# Edges (M)
patents	3.77	3.97	14.97
venturiLevel3	4.03	2.00	8.05
rajat31	4.69	4.33	20.32
italy_osm	6.69	1.05	7.01
wb-edu	9.85	5.81	57.16
germany_osm	11.55	1.07	12.37
asia_osm	11.95	1.06	12.71
road_central	14.08	1.02	16.93
hugetrace	16.00	1.50	240.00
hugebubbles	19.46	1.50	29.18
europe_osm	50.91	1.06	54.05
Sy-60M	60.00	3.00	180.00
Sy-70M	70.00	3.00	210.00
Sy-130M	130.00	2.23	290.00
Sy-.5B	500.00	1.74	870.00
Sy-1B	1000.00	2.58	2580.00
Sy-2B	2000.00	1.14	2270.00

Figure 6.14 demonstrates the speedup in execution time for our proposed ASIC and FPGA implementations over Intel® MKL on CPU. Figure 6.15 shows the same against Intel® MKL on Xeon Phi architecture. We have only reported the results that we were able to run on these architectures. For example, we couldn't successfully run

graphs over 70M and 30M nodes on Xeon E5 and Xeon Phi respectively. In both of these plots, we can see orders of magnitude improvement in execution time for SpMV operation. Figure 6.16 and Figure 6.17 depict the comparison of GTEPS metric against both the benchmarks for our ASIC and FPGA implementations respectively. These plots also show the performance of our proposed solutions on very large (~ billion nodes) graphs. It should be noted that due to 2048-way multi-way merge network our ASIC solution can handle much larger graphs than FPGA implementations, which is also mentioned in Table 6.1.

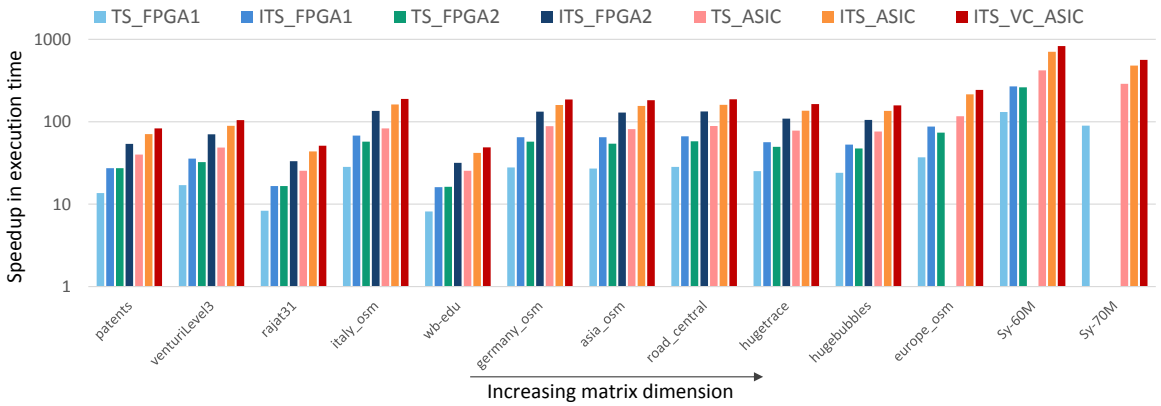


Figure 6.14: Speedup of proposed ASIC and FPGA implementations over Intel MKL on dual socket Xeon E5-2620 (12 threads) CPU for SpMV. For MKL SpMV implementation 'mkl_scoogemv' routine is used.

Furthermore, Figure 6.18 and Figure 6.19 demonstrate the energy efficiency comparison, in terms of energy per edge traversal, for our ASIC and FPGA implementations against the CPU and many-core co-processor. It can be seen that our ASIC solutions achieve multiple orders of magnitude improvement in efficiency against both benchmarks. Our FPGA implementations also achieve multiple orders of magnitude improvement for relatively larger graphs.

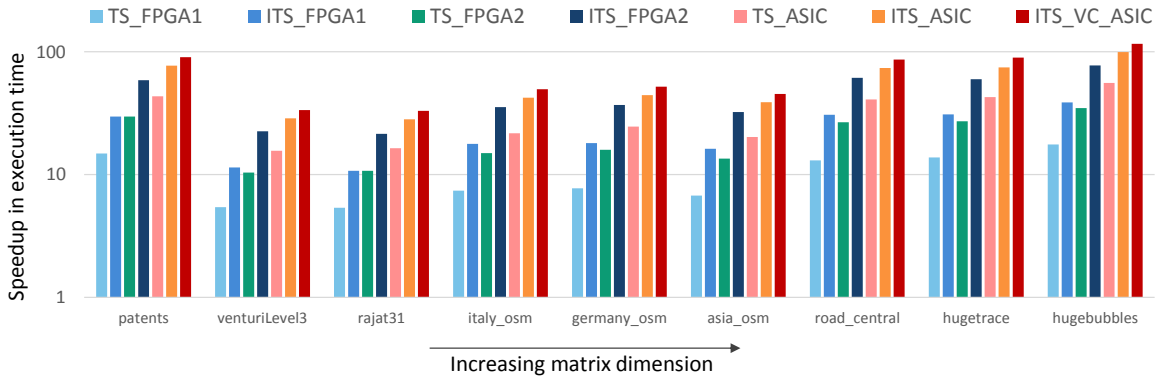


Figure 6.15: Speedup of proposed ASIC and FPGA implementations over Intel MKL on Xeon Phi 5110 co-processor for SpMV. For MKL SpMV implementation 'mkl_scoogemv' routine is used.

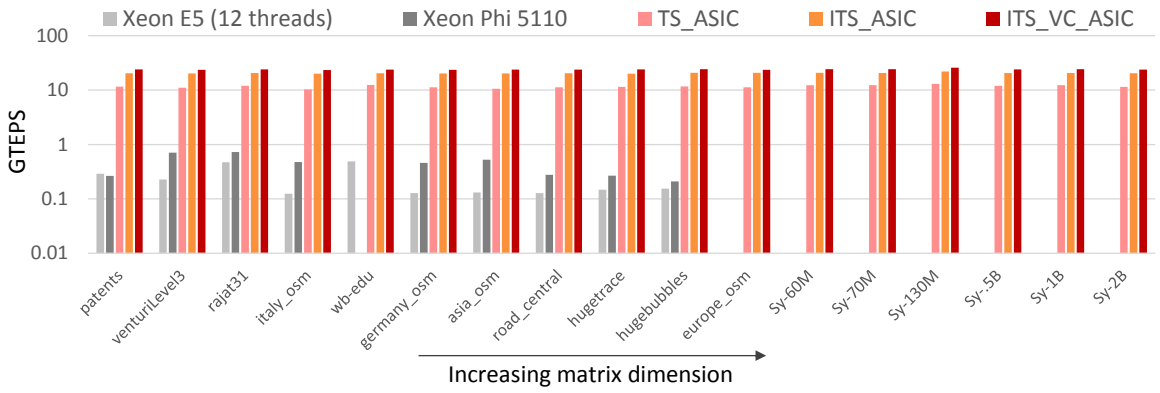


Figure 6.16: Comparison of traversed edges per second in billions (GTEPS) for proposed ASIC against Intel MKL on dual socket Xeon E5-2620 (12 threads) CPU and Xeon Phi 5110 co-processor for SpMV. For MKL SpMV implementation 'mkl_scoogemv' routine is used.

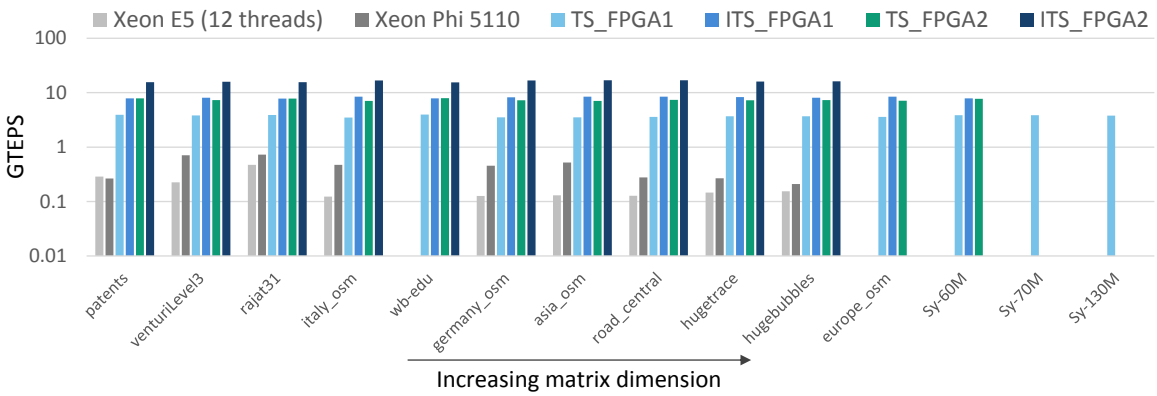


Figure 6.17: Comparison of traversed edges per second in billions (GTEPS) for proposed FPGA implementations against Intel MKL on dual socket Xeon E5-2620 (12 threads) CPU and Xeon Phi 5110 co-processor for SpMV. For MKL SpMV implementation 'mkl_scoogemv' routine is used.

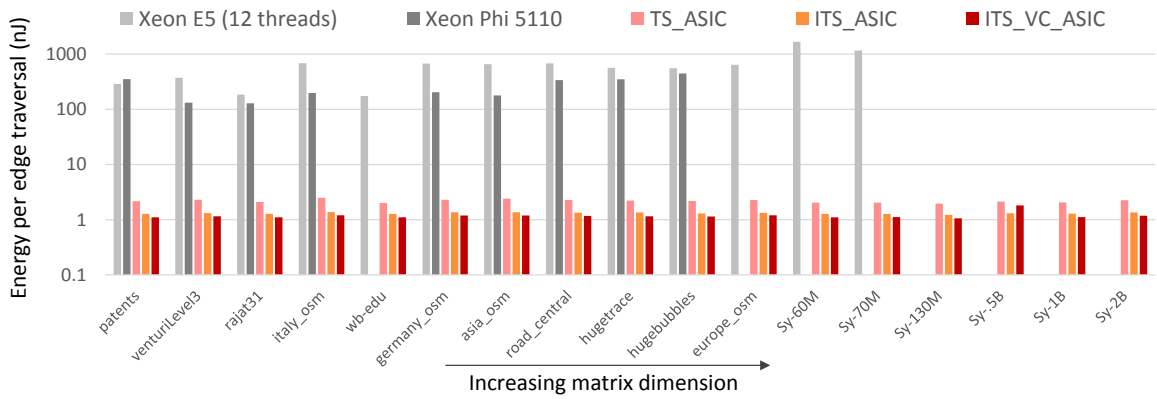


Figure 6.18: Comparison of energy per edge traversal for proposed ASIC against Intel MKL on dual socket Xeon E5-2620 (12 threads) CPU and Xeon Phi 5110 co-processor for SpMV. For MKL SpMV implementation 'mkl_scoogemv' routine is used.

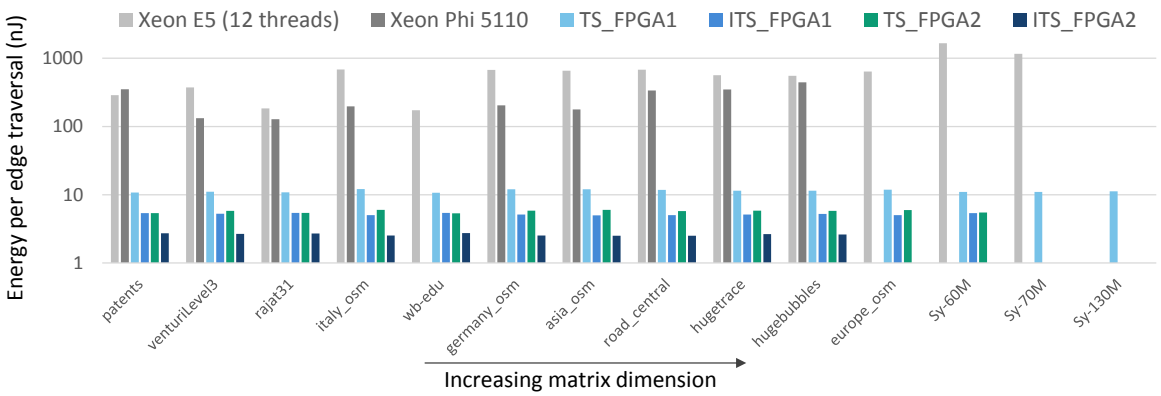


Figure 6.19: Comparison of energy per edge traversal for proposed FPGA implementations against Intel MKL on dual socket Xeon E5-2620 (12 threads) CPU and Xeon Phi 5110 co-processor for SpMV. For MKL SpMV implementation 'mkl_scoogemv' routine is used.

Chapter 7

Accelerating Sparse Matrix-Sparse Matrix Operation

Contents

There is another fundamental sparse operation where a sparse matrix is multiplied with itself or a second sparse matrix, which is termed as Sparse General Matrix-Matrix multiplication (SpGEMM). This can also be construed as sparse matrix multiplication with a set of sparse vectors, namely Sparse Matrix Sparse Vector

multiplication (SpMSpV). SpGEMM is an important core kernel used by many graph analytics applications such as betweenness centrality [19], all pairs shortest path [20] and breadth-first search [21]. Furthermore, it is used in machine learning and high performance computing applications such as peer pressure clustering [83], parsing context-free languages [84], label propagation [85] and multigrid solvers [86].

Similar to SpMV, SpGEMM on COTS architectures has challenges due to low FLOP to memory access ratio, irregular memory access and lack of spatial and temporal locality. Moreover, when the working data set becomes large than the on-chip fast memory, e.g. LLC, scratchpad, rapid increase in off-chip communication traffic takes a significant toll on performance and energy efficiency.

In this chapter, we demonstrate that the same hardware that we have designed for Two-Step SpMV can also be used for SpGEMM acceleration in a shared memory scenario. Similar to SpMV, our SpGEMM implementation utilizes available on-chip memory efficiently to be more scalable. Using the developed multi-way merge network, we propose to conduct SpGEMM accumulation by streaming access that enables better on-chip memory utilization and reduces off-chip traffic. Additionally, we will demonstrate that our proposed accumulation enables a 2D partitioned data traversal scheme that significantly reduces off-chip traffic for large problems.

7.1 Background and Challenges

SpGEMM can be defined as $\boldsymbol{C} = \boldsymbol{AB}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are the input sparse matrices and $\boldsymbol{C}$ is the output sparse matrix. Without any loss of generality, we consider all the matrices to be square such that $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{N \times N}$. The number of nonzero elements in matrix $\boldsymbol{A}$ is denoted by $nnz(\boldsymbol{A})$. We define the number of columns with

at least one nonzero in $\boldsymbol{A}$ as $nzc(\boldsymbol{A})$. Further, the data volume of $\boldsymbol{A}$ is denoted as $vol(\boldsymbol{A}) = nnz(\boldsymbol{A}) \times size^{mat}$, where $size^{mat}$ is the size of every nonzero element of the matrix including meta-data.

In this work we will discuss the shared memory SpGEMM algorithm and implementation challenges from two perspectives. Firstly, SpGEMM causes difficulties regarding the computation of $\boldsymbol{C}$ and access to $\boldsymbol{A}$ & $\boldsymbol{B}$. Secondly, when the matrices become large and partitioning is necessary, the off-traffic and traversal of the partitions pose a separate set of challenges. For our discussions, we assume two levels of memory hierarchy - fast & small on-chip memory (LLC, scratchpad etc.) and slow & big off-chip memory (DRAM).

7.1.1 Computation related Challenges

Inner products based formulation of SpGEMM in Equation 7.1, which usually serves as the definition of matrix multiplication, is probably the most straight-forward way of SpGEMM computation. Figure 7.1 shows this row-by-column operation where $\boldsymbol{C}(m,n)$ is computed as the dot product of sparse row vector $\boldsymbol{A}(m,:)$ and sparse column vector $\boldsymbol{B}(:,n)$. It should be noted in this example that even though these sparse vectors only intersect at the location k, the entire vector from both the matrices have to be read. In worst case, there might as well be no intersection among these vectors. Additionally, logic operations are required to check the existence of the intersections. Hence, regardless of any presence of intersection or not, inner product based computation cause unnecessary operations and redundant accesses to $\boldsymbol{A}$ and $\boldsymbol{B}$. Hence, this method requires $\Omega(N^2)$ operations regardless of the sparsity of the

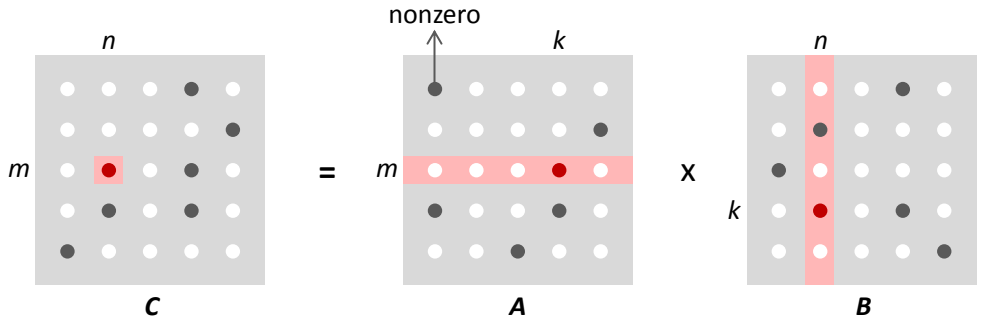


Figure 7.1: SpGEMM computation using inner product (row-by-column).

matrices [87] and is rarely used.

$$\boldsymbol{C}(m,n) = \sum_{q} \boldsymbol{A}(m,q) \cdot \boldsymbol{B}(q,n) \tag{7.1}$$

One of the most commonly used SpGEMM algorithm is by Gustavson [88], which is an outer product based method depicted in Figure 7.2. All the matrices are stored in row-major format. Let $\boldsymbol{A}(m,n)$ be the element in m^{th} row and n^{th} column of $\boldsymbol{A}$. $\boldsymbol{A}(m,:)$ and $\boldsymbol{A}(:,n)$ denote entire m^{th} row and n^{th} column of $\boldsymbol{A}$ respectively. This algorithm traverses $\boldsymbol{A}$ row-by-row. Rows corresponding to the nonzero elements in $\boldsymbol{A}(m,:)$ are read and the nonzero elements of $\boldsymbol{C}(m,:)$ are computed. Hence the entire computation of Gustavson's algorithm can be given by Equation 7.2.

$$\boldsymbol{C}(m,:) = \sum_{q} \boldsymbol{A}(m,q) \cdot \boldsymbol{B}(q,:) \tag{7.2}$$

As each row of $\boldsymbol{C}$ can be computed independently, Gustavson's algorithm is conceptually highly parallel [89]. However, there exists a number of inherent challenges

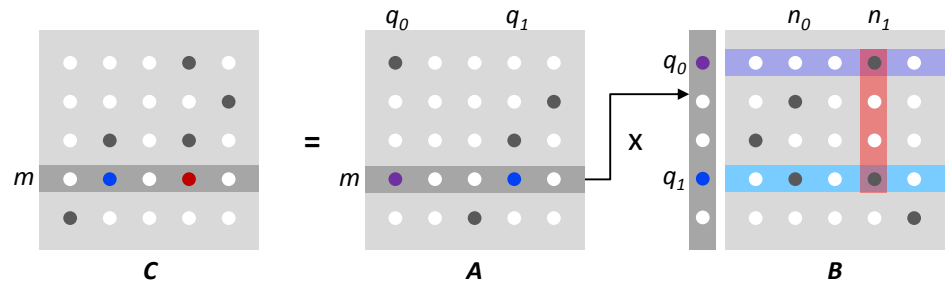


Figure 7.2: SpGEMM computation using Gustavson algorithm (row-by-row).

which are not only relevant to this algorithm, but also to any algorithm that is not entirely inner product based. These challenges are given below.

1. The memory access to $\boldsymbol{B}$ depends on the nonzero elements of $\boldsymbol{A}$. Hence, access to $\boldsymbol{B}$ is indirect and random. This causes high latency accesses if $\boldsymbol{B}$ is not present in the on-chip memory.

2. Any row of $\boldsymbol{C}$, $\boldsymbol{C}(m,:)$ is the sum of various sparse vectors obtained from the product of scaler $\boldsymbol{A}(m,q)$ with the nonzeros of $\boldsymbol{B}(q,:)$. Since these products are in unknown pattern, accumulating the elements of $\boldsymbol{C}$, i.e. $\boldsymbol{A}(m,q0)\boldsymbol{B}(q0,n1) + \boldsymbol{A}(m,q1)\boldsymbol{B}(q1,n1)$ in Figure 7.2, efficiently is difficult.

3. The number of nonzeros in $\boldsymbol{C}$, $nnz\boldsymbol{C}$, is unknown beforehand. Hence, memory allocation of output matrix $\boldsymbol{C}$ is difficult. We need to either pre-calculate the $nnz\boldsymbol{C}$ or allocate large enough on-chip storage for $\boldsymbol{C}$. Both of these methods incur overhead as the former requires extra computation and the later wastes memory by over allocating.

7.1.2 Partition and Off-chip Traffic related Challenges

When the sparse matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ grow significantly larger than the on-chip fast memory, off-chip communication poses another set of challenges. Without partitioned input matrices outer product based methods cause very high latency random access to DRAM. On the other hand, complete inner product based computation as shown in Equation 7.1 can guarantee streaming DRAM access given that $\boldsymbol{A}$ is stored in row-major format and $\boldsymbol{B}$ is stored in column-major format. However, besides $\Omega(N^2)$ operations, full inner product based algorithm streams both input matrices N times, which is prohibitively inefficient. Additionally, storing the input matrices in different formats can be inefficient as $\boldsymbol{A}^2$ is often used in SpGEMM applications.

For large matrices, where the problem set is larger than fast on-chip storage, it is generally beneficial to partition the matrices into blocks and conduct SpGEMM on the individual blocks using outer product based methods such Equation 7.2. In this work we consider 2D data decomposition of the matrices based on Shared and Remote-memory based Universal Matrix Multiplication Algorithm (SRUMMA) [90], which is depicted in Figure 7.3. The overall sequence of block matrix multiplication of this algorithm is similar to Canon's algorithm [91] in principle. However, SRUMMA is developed for clusters and scalable shared memory systems. SRUMMA is popularly used for shared memory implementations such as in [50].

Let N_y and N_z be the number of blocks of $\boldsymbol{A}$ in row-major and column-major direction respectively, as shown in Figure 7.3. Similarly, N_z and N_x are the number of blocks of $\boldsymbol{B}$. Thus $\boldsymbol{C}$ is partitioned into $N_y \times N_x$ blocks. We denote $\boldsymbol{A}_{i,j}$ as the block in i^{th} row and j^{th} column of the 2D partition. Generally, $\boldsymbol{C}_{i,j}$ is computed by block level

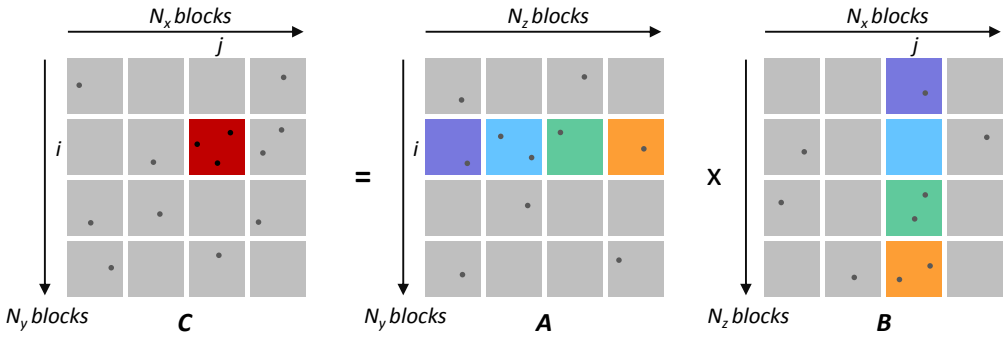


Figure 7.3: 2D partitioned SpGEMM formulation and Block-level Inner Product (BIP) based computation.

inner product as shown below.

$$\boldsymbol{C}_{i,j} = \sum_{k=0}^{N_z-1} \boldsymbol{C}_{i,j}^{temp,k} = \sum_{k=0}^{N_z-1} \boldsymbol{A}_{i,k} \cdot \boldsymbol{B}_{k,j} \tag{7.3}$$

This means that all the blocks of $\boldsymbol{A}$ in row i and all the blocks of $\boldsymbol{B}$ in column j is traversed sequentially to entirely compute $\boldsymbol{C}_{i,j}$. Here, $\boldsymbol{C}_{i,j}^{temp,k}$ denotes the partial result that is computed by the operation $\boldsymbol{A}_{i,k} \cdot \boldsymbol{B}_{k,j}$. Being oblivious to the actual block level computation $\boldsymbol{A}_{i,k} \cdot \boldsymbol{B}_{k,j}$, we name the inner product based the block traversal scheme formulated in Equation 7.3 as Block-level Inner Product (BIP). As described below, there are two key factors that dictate the performance of BIP.

Accumulation

The continuous accumulation of intermediate output matrix $\boldsymbol{C}_{i,j}$ is particularly critical as it is stored in sparse format and requires new element insertion & update. Most solutions in the literature require random access for this accumulation and use specialized data structure or hardware to conduct this accumulation efficiently. One approach of storing and updating the nonzeros of $\boldsymbol{C}_{i,j}$ is to use a hash table [92]. However, as

found in [93], this technique has high overheads due to hash key computation and handling collisions through chaining.

On the other hand, Sparse accumulator (SPA) [94] is popularly used for accumulation. For example, authors of [93] has used blocked SPA to reduce cache miss rate for partitioned Gustavson's algorithm. SPA mainly comprises three components - a) a dense array for storing the running sum, i.e. the real values for the active rows of $\boldsymbol{C}_{i,j}$, b) a dense boolean vector to work as flags indicating the existence of nonzero element and c) a sparse list to store the indices of the nonzero elements. It should be noted that all these data structures have to be stored in the on-chip fast memory for the ease of random access. We will later see that storage of $\boldsymbol{C}_{i,j}$ in the on-chip fast storage may have an indirect but nontrivial effect on the overall performance for certain type of graphs.

While SPA is popularly used for COTS architectures, custom architecture for SpGEMM, as the ASIC in [50], has deployed special hardware for the accumulation of $\boldsymbol{C}_{i,j}$. For example, in [50] the authors have used Content Addressable Memory (CAM) to store $\boldsymbol{C}_{i,j}$ on chip and the fast random access. CAM enables comparison of all the indices in a single cycle and allows direct accumulation when intersection is detected. However, insertion of new element in the sparse format storage cannot be handled only by CAM and additional hardware is required. More importantly, CAM requires several times more area and energy than regular SRAM. Hence, for a given silicon area CAM renders the block dimension of the matrices to be lot smaller than what is possible with regular SRAM based storage. We will next show that how block dimension significantly affects off-chip traffic and overall performance.

Block Dimension

Let D_{BIP} be the total off-chip DRAM traffic for the entire computation of $\boldsymbol{C}$ using BIP as stated in Equation 7.3 and depicted in Figure 7.3. The formulation of D_{BIP} can be given as in Equation 7.4.

$$\begin{aligned} D_{\text{BIP}} &= \sum_{j=0}^{N_x-1} \sum_{i=0}^{N_y-1} \sum_{k=0}^{N_z-1} (vol(\boldsymbol{A}_{i,k}) + vol(\boldsymbol{B}_{k,j})) + \sum_{j=0}^{N_x-1} \sum_{i=0}^{N_y-1} vol(\boldsymbol{C}_{i,j}) \\ &= N_x \cdot vol(\boldsymbol{A}) + N_y \cdot vol(\boldsymbol{B}) + vol(\boldsymbol{C}) \end{aligned} \tag{7.4}$$

Overall, matrix $\boldsymbol{A}$ and $\boldsymbol{B}$ are read from DRAM N_x and N_y times respectively and output matrix $\boldsymbol{C}$ is written to DRAM only once. If $\boldsymbol{C}$ is not significantly denser than the inputs, majority of the off-chip traffic is contributed by multiple transfers of $\boldsymbol{A}$ and $\boldsymbol{B}$. Hence, number of horizontal blocks N_y in $\boldsymbol{A}$ and number of vertical blocks N_x in $\boldsymbol{B}$ play important roles in determining the off-chip traffic. N_y and N_x directly depend on the storage capacity of the on-chip memory. Among the input matrices, either $\boldsymbol{A}_{i,k}$ or $\boldsymbol{B}_{k,j}$ requires random access and needs to be stored the on chip. Hence, with more on-chip storage block dimensions of $\boldsymbol{A}$ and $\boldsymbol{B}$ are larger, which eventually reduces D_{BIP} by decreasing N_y and N_x. Hence, increasing the block size to maximum by efficient usage of on-chip memory is imperative for SpGEMM performance and efficiency.

In most current SpGEMM implementations, e.g. [50, 93, 95, 96], $\boldsymbol{C}_{i,j}$ is also stored in the on-chip fast memory to randomly access intermediate results for accumulation, i.e. update and new element insertion. This causes limited on-chip to be shared among $\boldsymbol{A}$ (or $\boldsymbol{B}$) and $\boldsymbol{C}$ that directly increases N_y and N_x. As we will see later, for large input matrices ($\sim$ millions of nodes) and/or when $\boldsymbol{C}$ is not significantly denser ($\sim$ multiple orders of magnitude) than the input matrices, storing $\boldsymbol{C}_{i,j}$ in on-chip memory adversely

affects off-chip traffic. Additionally, as the data structure is complicated, e.g. SPA [93] and CAM [50], and over-allocation is required due to unknown $nnz(\boldsymbol{C}_{i,j})$, $\boldsymbol{C}_{i,j}$ consumes significantly more on-chip storage than $\boldsymbol{A}$ or $\boldsymbol{B}$ for the same number of nonzeros. Hence, N_y and N_x are further increased to incur additional off-chip traffic.

7.2 Proposed SpGEMM

From the above discussion it is apparent that efficient SpGEMM for large matrices is hindered by both computation difficulties and off-chip traffic. In this work we propose a shared memory SpGEMM hardware accelerator that addresses these issues. The core hardware kernel is essentially the same as what we have developed for SpMV acceleration, which is the scalable and high throughput multi-way merge network. The key contributions of our proposed solution are the following.

1. Random access for $\boldsymbol{C}_{i,j}$ accumulation, i.e. update and insertion, is replaced by streaming access. No complicated data structure or expensive CAM is required. Simpler hardware leads to more scalability and higher energy-efficiency.

2. Entire on-chip memory can be dedicated to store the blocks of input matrix by storing $\boldsymbol{C}_{i,j}$ in off-chip DRAM. Hence, off-chip traffic can be reduced by entire utilization the on-chip memory only for input matrix block and decremented N_y & N_x as a result. This is especially effective for large input matrices ($\sim$ billions of nodes) and/or $nnz(\boldsymbol{C})$ is not significantly higher ($\sim$ multiple orders of magnitude) than $nnz(\boldsymbol{A})$ or $nnz(\boldsymbol{B})$.

3. Off-chip traffic is further reduced by avoiding full block-level inner product based traversal of SRUMMA and adopting a 2D block traversal scheme enabled by streaming accumulation.

We will first discuss the proposed computation technique of input matrix blocks and later demonstrate our proposed method for decreasing the off-chip traffic of partitioned SpGEMM.

7.2.1 SpGEMM Computation with SSPA

In this work we use column-by-column SpGEMM computation of matrix blocks. This is similar to Gustavson's method shown in Figure 7.2, however, instead of row-by-row we use column-by-column operation. We propose to use an accumulator, namely Streaming Sparse Accumulator (SSPA), that doesn't require any random access. A high level depiction of the column-by-column SpGEMM using SSPA is given in Figure 7.4. All the input matrices are stored in column major sparse format. For explanation purpose, we consider the computation of input matrix blocks $\boldsymbol{A}_{i,k}$ and $\boldsymbol{B}_{k,j}$ using the BIP 2D block traversal method depicted in Figure 7.3.

Block $\boldsymbol{B}_{k,j}$ is traversed sequentially in column-major direction. Row indices of the nonzeros in a column of $\boldsymbol{B}_{k,j}$, e.g. q_0 and q_1 of n^{th} column, dictate intersecting columns of $\boldsymbol{A}_{i,k}$. However, for accumulation, instead of using SPA [93] or CAM [50] as discussed before, we propose to use merge hardware that is already developed for SpMV operation. This merge hardware is an integral part of SSPA. The intersecting columns of $\boldsymbol{A}_{i,k}(:,q_0)$ and $\boldsymbol{A}_{i,k}(:,q_1)$ can be considered as sorted lists (according to ascending row indices). The merge hardware produces a final output list that is sorted along the row index of $\boldsymbol{A}_{i,k}$. This output list is further passed through an adder that conducts

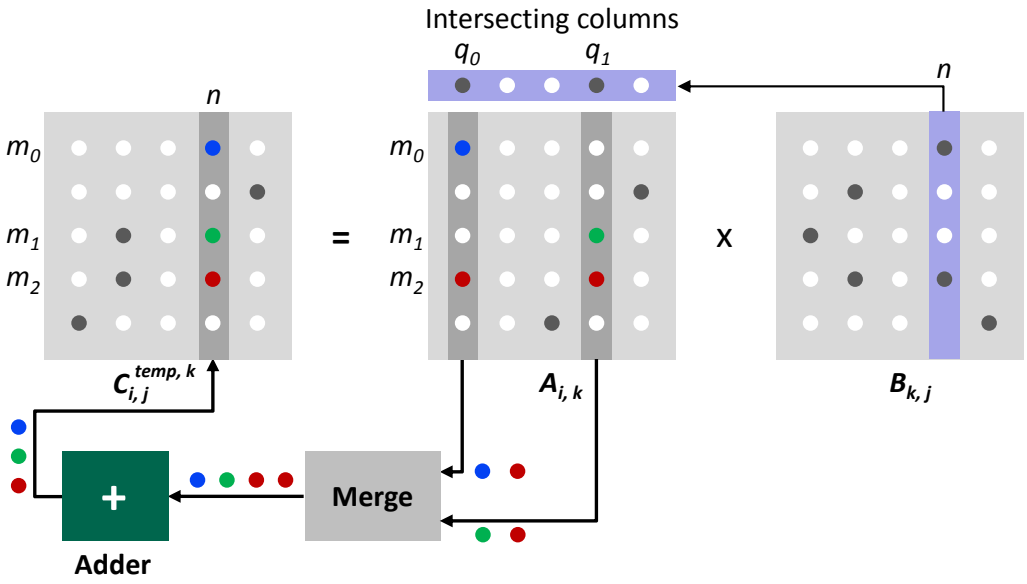


Figure 7.4: Proposed SpGEMM computation method Streaming Sparse Accumulator (SSPA).

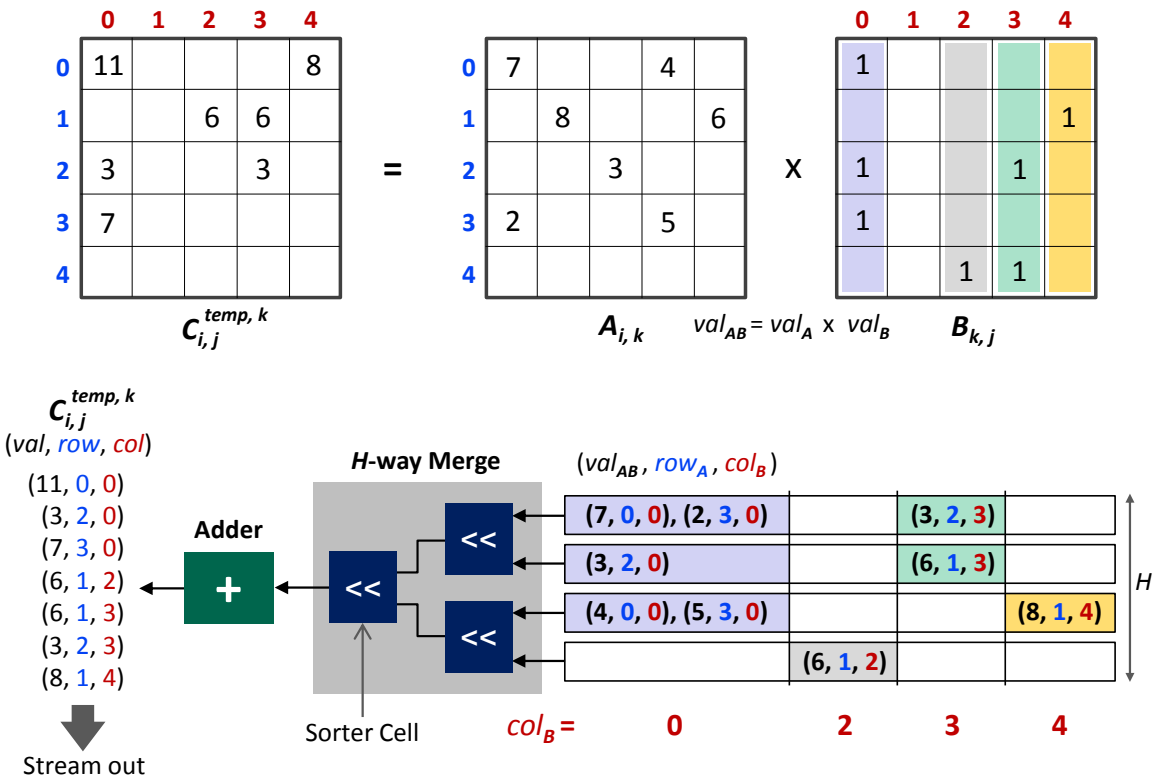
reduction on the elements with same row indices. Eventually the adder delivers the elements of column n of output matrix $\boldsymbol{C}_{i,j}^{temp,k}$ in a sequential order according to row indices. Thus, our proposed SSPA comprises of these two fundamental hardware - a) a multi-way merge network to sort the input columns and b) an adder to conduct reduction.

In Figure 7.4, as $\boldsymbol{B}_{k,j}$ is sequentially traversed in column major direction, the columns of $\boldsymbol{C}_{i,j}^{temp,k}$ are also sequentially generated, where each column is sorted along the row indices. Hence, for the entire computation the output matrix block $\boldsymbol{C}_{i,j}^{temp,k}$ can be updated simply by sequential appending, which is the key enabler of all the advantages of SSPA. The entire set of operations in SSPA is described next using an example problem given in Figure 7.5.

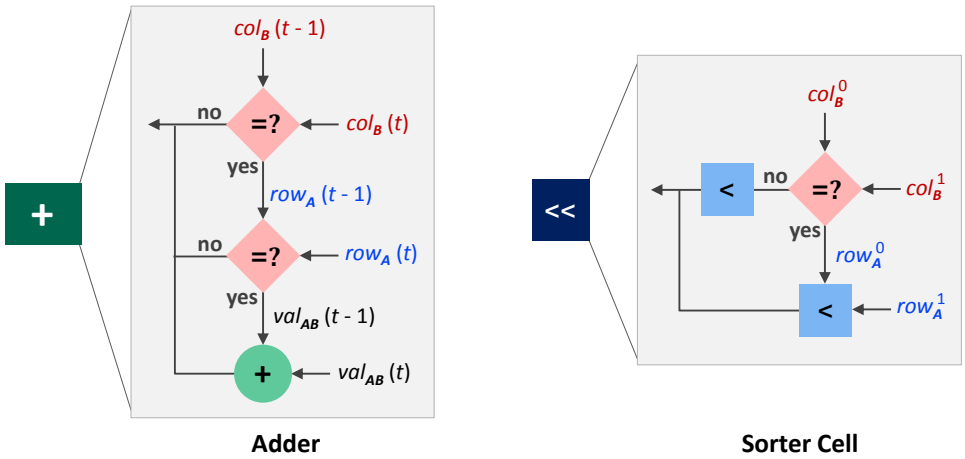
Operational Details of SSPA

For ease of comprehension, let all the nonzeros of $\boldsymbol{B}_{k,j}$ to have a value of 1 as shown in Figure 7.5a. Starting from the 0^{th} column, all the columns of $\boldsymbol{B}_{k,j}$ is sequentially read. For the 0^{th} column $\boldsymbol{B}_{k,j}(:,0)$, there are nonzeros at row indices 0, 2 and 3. Hence the intersecting columns of $\boldsymbol{A}_{i,k}$, i.e. $\boldsymbol{A}_{i,k}(:,0)$, $\boldsymbol{A}_{i,k}(:,2)$ and $\boldsymbol{A}_{i,k}(:,3)$, are read. The values of the nonzero elements of column $\boldsymbol{A}_{i,k}(:,q)$ are multiplied with the value of the corresponding element $\boldsymbol{B}_{k,j}(q,0)$. Let val_{AB} be the result of this multiplication between the values val_A and val_B. We represent each result as a tuple (val_{AB}, row_A, col_B) where row_A is the row index from $\boldsymbol{A}_{i,k}$ and col_B is the column index form $\boldsymbol{B}_{k,j}$. Thus, nonzeros of the three columns of $\boldsymbol{A}_{i,k}$ intersecting with $\boldsymbol{B}_{k,j}(:,0)$ represent three list of tuples, where each list is sorted along ascending row_A. These three lists are sequentially assigned to the inputs of H-way merge network. Similarly, the corresponding lists due to intersection with $\boldsymbol{B}_{k,j}(:,1)$, which is a single list with only one tuple(6, 1, 2), are next sequentially assigned to the rest of the inputs of H-way merge network. Once the last input of the merge network is reached, next assignment starts from the very first input and later assignments are continued in the same manner for rest the columns of $\boldsymbol{B}_{k,j}$.

It should be noted that if the number of columns of block $\boldsymbol{A}_{i,k}$ that intersect with any particular column q of $\boldsymbol{B}_{k,j}$, which is denoted as $nci(\boldsymbol{AB})^q_{i,j,k}$, is greater than H then bubbles will be introduced in the pipelines of the merge network. Generally, $nci(\boldsymbol{AB})^q_{i,j,k}$ is a small number as the matrix blocks are very sparse and H is quite large (in the order of several hundreds). Hence the possibility of bubbles in the H-way merge network is trivial.



(a) Detailed operation example of SSPA.



(b) Internal logic of the Adder and the Sorter Cell of SSPA.

Figure 7.5: SpGEMM operational details of SSPA for an example problem.

The internal logic of the adder and the sorter cell is elaborated in Figure 7.5b. Two parallel comparisons take place for each sorting operation, where the comparison of the column indices gets priority and the tuple with smaller column index passes through first. If the column indices are equal then the tuple with smaller row index passes through. On the other hand, the adder works on two consecutive tuples in cycle $(t-1)$ and t. If either the column indices or the row indices of consecutive tuples are not equal, then no reduction occurs. If both the column indices and row indices match in consecutive tuples, then only the values of tuples are added. Thus the adder delivers all the nonzeros (tuples) of the output matrix block sequentially sorted in ascending column indices (col) and in ascending row indices (row) for any particular column index. This format is also known as Column Major Coordinate (CM-COO) format as depicted in Figure 7.7.

Accumulation of $\boldsymbol{C}_{i,j}$ Across Partitions

As the elements of temporary resultant matrix block $\boldsymbol{C}_{i,j}^{temp,k} = \boldsymbol{A}_{i,k}\boldsymbol{B}_{k,j}$ is generated sequentially, it is possible to stream out to on-chip storage or off-chip DRAM. Let's consider BIP 2D block traversal, as shown in Figure 7.3, along with SSPA computation. As k increments, i.e. $\boldsymbol{A}_{i,:}$ and $\boldsymbol{B}_{:,j}$ are traversed along N_z blocks, we can conduct the entire accumulation for $\boldsymbol{C}_{i,j}$ using merge network of SSPA. This accumulation process is elaborated in Figure 7.6. We denote the resultant matrix block accumulated up to block k as $\boldsymbol{C}_{i,j}^{k} = \boldsymbol{C}_{i,j}^{temp,k} + \boldsymbol{C}_{i,j}^{k-1}$. If the merge network has H inputs, i.e. H-way, then we can use $(H-Q)$ ways only for the computation of $\boldsymbol{C}_{i,j}^{temp,k}$. The rest $(Q-1)$ ways can be used for accumulating with previously computed blocks. As mentioned before, all the previously computed matrix blocks, i.e. $\boldsymbol{C}_{i,j}^{temp,0}$ to $\boldsymbol{C}_{i,j}^{temp,k-1}$, are stored in CM-COO format and treated as a sorted list of tuples. It should be noted that only

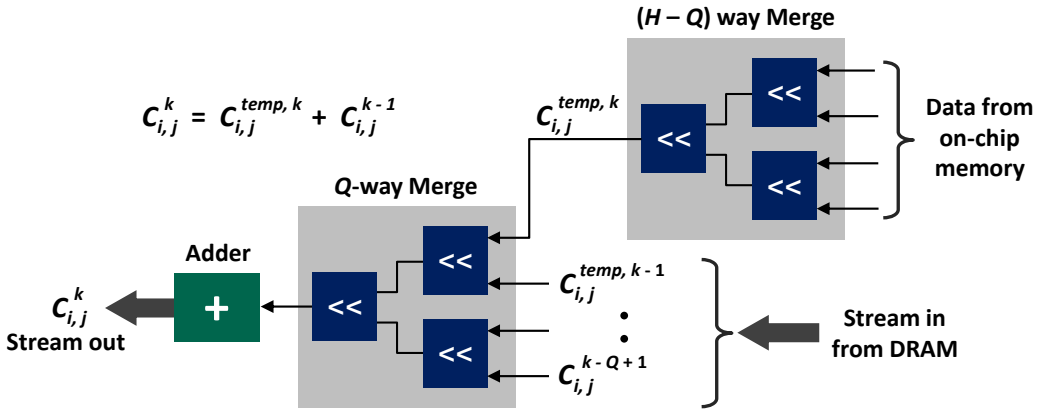


Figure 7.6: Accumulation of $\boldsymbol{C}_{i,j}$ across partitions.

sequential access to the blocks $\boldsymbol{C}_{i,j}^{temp,0}$ to $\boldsymbol{C}_{i,j}^{temp,k-1}$ is needed and they can be read by streaming access at peak DRAM bandwidth if stored off-chip. On the other hand, $\boldsymbol{C}_{i,j}^{temp,k}$ is computed and sequentially. Thus, SSPA can conduct the entire accumulation process using the H-way merge network, while guaranteeing full streaming access to the output matrix $\boldsymbol{C}$.

For the accumulation of $\boldsymbol{C}_{i,j}^{k}$ across all the previously computed blocks, there can be two extremes in the design point. One extreme design point is when $Q = 2$. In this case, every $\boldsymbol{C}_{i,j}^{k}$ is read back from DRAM to be accumulated with $\boldsymbol{C}_{i,j}^{temp,k+1}$. If $\boldsymbol{C}_{i,j}^{k}$ is stored on chip, this is preferable as it maximizes the block size. Other extreme design point is when $Q = H$, which is feasible if $\boldsymbol{C}_{i,j}^{temp,k}$ is stored off-chip. In this case, while being computed, $\boldsymbol{C}_{i,j}^{temp,k}$ is not accumulated with any previously stored temporary results. After the computation of the very last temporary resultant block $\boldsymbol{C}_{i,j}^{temp,N_z-1}$, all the blocks $\boldsymbol{C}_{i,j}^{temp,k}$, where $k = 0$ to $(N_z - 1)$, are streamed back from DRAM to be merged finally to produce $\boldsymbol{C}_{i,j}$. However, in this case the number of inputs of the merge network, i.e. H, has to be at least N_z. Generally, N_z is far less than H. For example, in our proposed ASIC $H = 512$ and $N_z \approx 200$ for one billion node input

graphs with average degree 50. In any case, we can always select a value of Q that can be accommodated properly by any existing merge network.

It should be noted that if $\boldsymbol{C}_{i,j}^{temp,k}$ is stored in DRAM, $Q = N_z$, i.e. inter block accumulation after $\boldsymbol{C}_{i,j}^{temp,k}$ is computed for all k, renders the off-chip traffic to be lowest. This is because the entire inter block level accumulation of $\boldsymbol{C}_{i,j}$ can be conducted in single transfer from DRAM rather than repetitive round trips for incremental streaming accumulation. The total off-chip traffic for $Q = N_z$ can be formulated as the following.

$$D(\boldsymbol{C}_{i,j},\ Q = N_z) = 2 \sum_{k=0}^{N_z-1} vol(\boldsymbol{C}_{i,j}^{temp,k}) + vol(\boldsymbol{C}_{i,j}) \tag{7.5}$$

Here, the traffic due to block computation, $vol(\boldsymbol{C}_{i,j}^{temp,k})$, is multiplied by 2 because of streaming out after computation and streaming in for accumulation. After the final block level accumulation, $vol(\boldsymbol{C}_{i,j})$ is streamed out to DRAM.

Storage Formats for Input and Output Matrices

For column-by-column SpGEMM operation using SSPA, as shown in Figure 7.4, we require all the matrices to be in column major sparse format. Several column major sparse formats are depicted in Figure 7.7. SSPA can similarly be used with Gustavson's row-by row SpGEMM operation in Figure 7.2, where all the matrices would have been required to be stored in row major sparse format. Hence, the following discussion is equally applicable row-by-row SpGEMM computation.

Due to accumulation using SSPA, in our proposed method matrix $\boldsymbol{B}$ and $\boldsymbol{C}$ require only sequential access for both off-chip read/write and on-chip computation. On the other hand, $\boldsymbol{A}$ requires random column access for on-chip computation and sequential access for off-chip read/write. Therefore, Compressed Sarse Column (CSC) format,

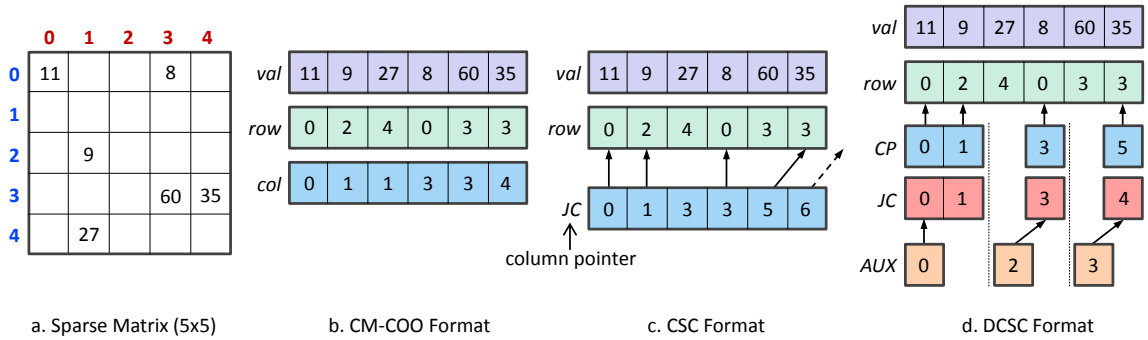


Figure 7.7: CM-COO, CSC and DCSC sparse storage formats.

as depicted in Figure 7.7, is suitable for all the three matrices for both off-chip and on-chip storage. CSC has a space complexity of $O(N + nnz)$, where N is the dimension and nnz is the number of nonzeros in the entire matrix. CSC also provides column-wise random access for A. However, with 2D partitioning it is common to have matrix blocks that are hypersparse. A matrix is considered hypersparse if $nnz < N$ [70]. For hypersparse matrix block CSC becomes wasteful as the space complexity for the column pointer JC array is always $O(N)$ due to the repetitions for completely empty columns. Since the space complexity of CM-COO is $O(nnz)$, it is preferable format for both off-chip and on-chip storage of the hypersparse blocks of $\boldsymbol{B}$ and $\boldsymbol{C}$. CM-COO is also suitable for off-chip storage of the blocks of $\boldsymbol{A}$. However, CM-COO doesn't provide random column access capability and, therefore, it cannot be used for on-chip storage of hypersparse block of $\boldsymbol{A}$.

In [70], the authors have proposed a new sparse matrix storage format, namely Doubly Compressed Sarse Column (DCSC), which provides random column access capability and has space complexity of $O(nnz)$. Therefore for on-chip storage of block $\boldsymbol{A}_{i,k}$ we use DCSC format. As depicted in Figure 7.7, DCSC replaces the column pointer array of CSC with 3 vectors. The JC array is devoid of repetitions in DCSC

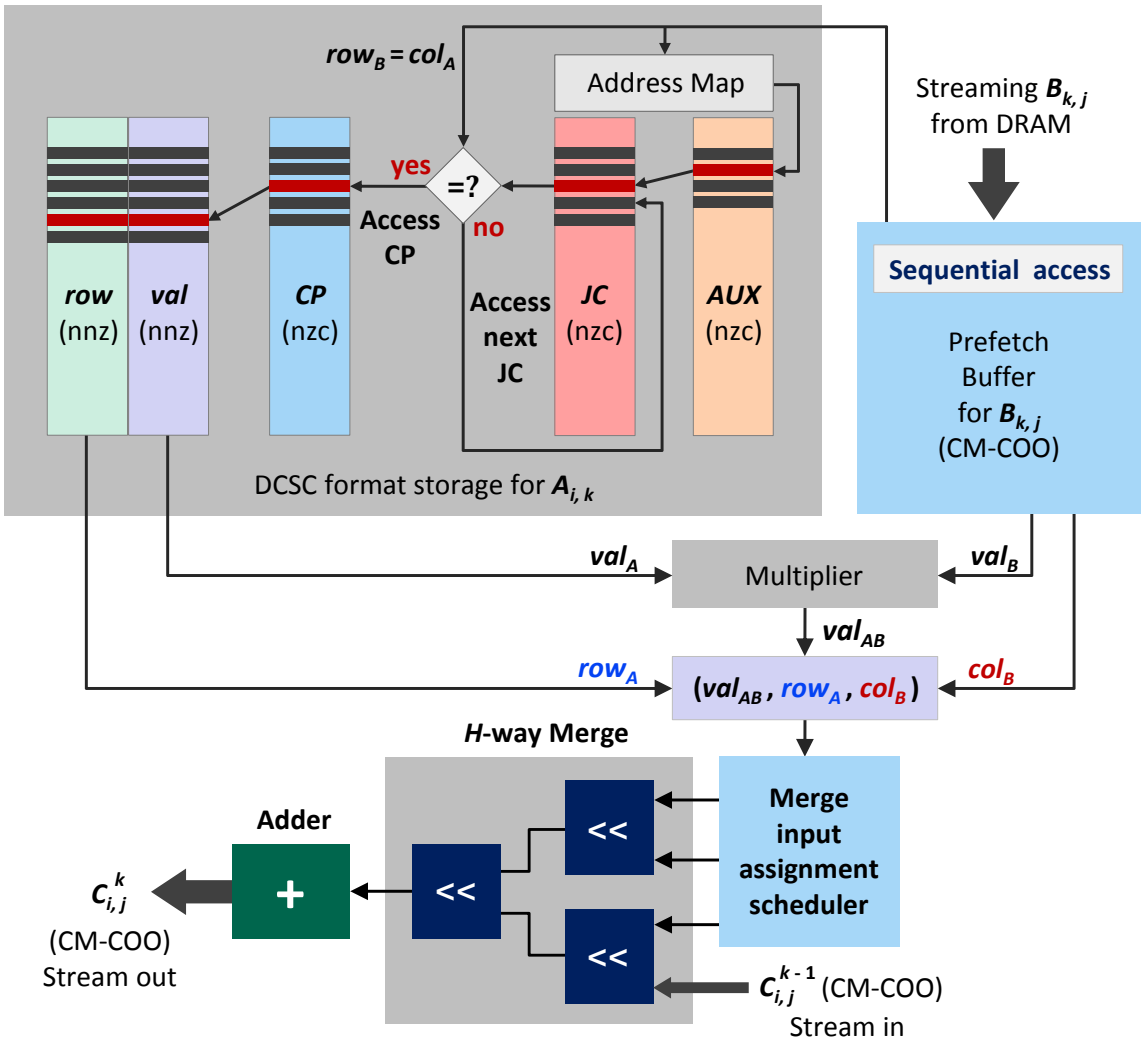


Figure 7.8: SpGEMM computation elaboration using SSPA and DCSC format for $\boldsymbol{A}$. We assume $Q = 2$ for this elaboration.

and is chopped into chunks. The AUX array enables constant time access to the first non-empty column within a chunk. Detailed description of DCSC is available in [70].

Figure 7.8 elaborates the SpGEMM computation using DCSC format for $\boldsymbol{A}_{i,k}$. A prefetch buffer for $\boldsymbol{B}_{k,j}$ amortizes the full cost DRAM page opening while being sequentially accessed. Read data from $\boldsymbol{B}_{k,j}$ initiates random access to $\boldsymbol{A}_{i,k}$ that can be

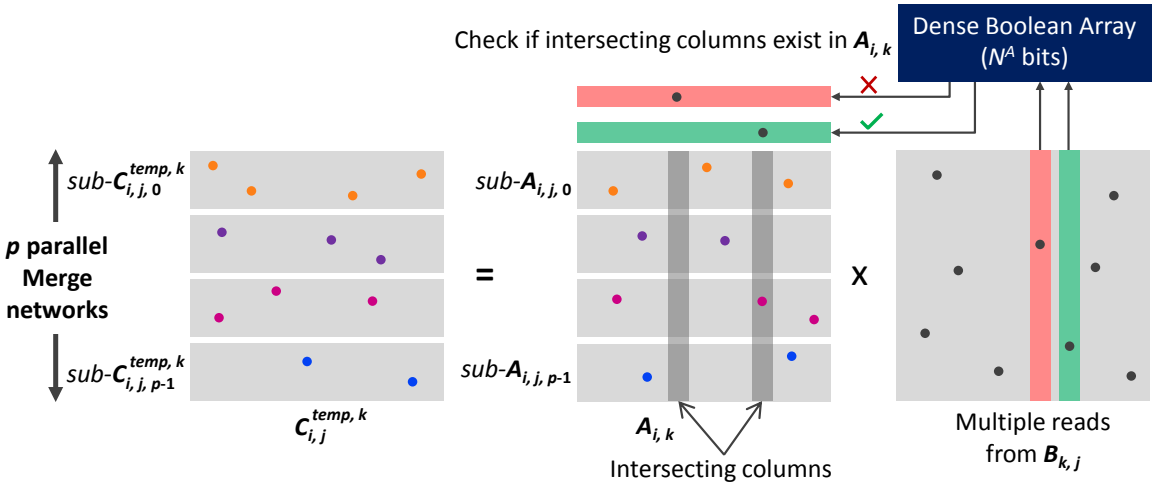


Figure 7.9: Parallel SpGEMM computation using SSPA.

efficiently conducted using DCSC data structure. The five arrays of DCSC are stored in separate memory blocks. A scheduler assigns the tuples corresponding to intersected columns of $\boldsymbol{A}_{i,k}$ at the input of the merge network of SSPA. For this elaboration we assume $Q = 2$. Hence, for every k $\boldsymbol{C}^{temp,k}_{i,j}$ is accumulated with $\boldsymbol{C}^{k-1}_{i,j}$ to produce $\boldsymbol{C}^{k}_{i,j}$, which is streamed out in CM-COO format. We store all the hypersparse matrix blocks of A, B and C in CM-COO format in DRAM as it is often advantageous to store all matrices in same format for operations like $\boldsymbol{A}^2$, $\boldsymbol{A}^3$, etc. When block $\boldsymbol{A}_{i,k}$, which is stored in off-chip DRAM in CM-COO format, is streamed to the computation core, it is converted to DCSC format on the fly and stored in the on-chip memory.

Parallel SpGEMM using SSPA

To achieve higher performance, it is possible to parallelize the computation of the matrix blocks. For this purpose, the input matrix block $\boldsymbol{A}_{i,k}$ and output matrix block $\boldsymbol{C}^{temp,k}_{i,j}$ is further partitioned into p horizontal sub-blocks as depicted in Figure 7.9. Matrix block

$\boldsymbol{B}_{k,j}$ doesn't require any sub-blocking. As the columns of $\boldsymbol{B}_{k,j}$ are sequentially traversed, the intersecting columns of all p sub-blocks of $\boldsymbol{A}_{i,k}$ are fetched. These columns from p sub-blocks can be independently processed using p independent H-way merge cores. In previous chapters, we have seen that our ASIC accelerator has sixty four 512-way HCLAM merge networks. Hence, $p = 64$ and $H = 512$ for SpGEMM acceleration with this ASIC accelerator. After the computation of last temporary resultant sub-blocks of $\boldsymbol{C}_{i,j}$, i.e. $\textit{sub-}\boldsymbol{C}^{temp,N_z}_{i,j,\,0\rightarrow(p-1)}$, a p-way merging process can accumulate the sub-blocks to construct final $\boldsymbol{C}_{i,j}$. As p is generally much smaller than H, this final merging process can be done easily with existing hardware.

One important issue in this parallel SpGEMM is that the computation load of the parallel merge networks is determined by the sequential reading speed of $\boldsymbol{B}_{k,j}$ and total number of intersecting columns in $A_{i,k}$ for entire $\boldsymbol{B}_{k,j}$, which is denoted as $nci(\boldsymbol{AB})_{i,j,k}$. The number of intersecting columns $nci(\boldsymbol{AB})_{i,j,k}$ depends on sparsity of the data. If $nci(\boldsymbol{AB})_{i,j,k}$ is low due to high sparsity, it might often be the case that the entire computation speed is bottlenecked by the sequential read speed of $\boldsymbol{B}_{k,j}$ and not by the aggregated throughput of the parallel merge networks. One way to overcome this is to read multiple elements from $\boldsymbol{B}_{k,j}$ and parallelly check for intersecting columns in all the sub-blocks of $\boldsymbol{A}_{i,k}$. Since the sub-blocks of $\boldsymbol{A}_{i,k}$ is stored in DCSC format, as shown in Figure 7.8, additional hardware to check multiple intersecting columns simultaneously can be demanding. A simpler way is to have dense boolean vector of $N^A = N/N_z$ bits. This vector only stores information of empty columns of $A_{i,k}$. After reading multiple elements from $\boldsymbol{B}_{k,j}$, the row indices ($row_{\boldsymbol{B}}$) are matched against the boolean bits to first check whether the intersecting columns actually exist, i.e. not empty. If any column is found to be empty, it is not considered by the read logic for sub-blocks of $A_{i,k}$. Checking multiple bits of the boolean vector simultaneously is lot

less resource consuming than the multiple access to the p sub-blocks stored in DCSC format. Hence, sequential reading speed of $\boldsymbol{B}_{k,j}$ can be substantially improved by the use of this dense array.

Advantages of SSPA

It is apparent that, while using SSPA, if matrix block $\boldsymbol{C}_{i,j}$ is not stored in the on-chip memory then there will be additional off-chip traffic for SpGEMM accumulation. Hence it is counterintuitive to store $\boldsymbol{C}_{i,j}$ in off-chip DRAM. However, as formulated in Equation 7.3 for BIP block traversal, number of blocks in 2D partitioned matrices also significantly affect the off-chip traffic. The higher the number of blocks, the more is the off-chip traffic and matrix block dimension is inversely related to the number of blocks. For any given system we have a limited amount of on-chip memory and it is important to efficiently utilize it to fit largest possible matrix blocks. As mentioned before, if $\boldsymbol{C}_{i,j}$ is stored in on-chip memory for random access, then $\boldsymbol{A}_{i,k}$ has to share the fast storage with it. On the other hand, SSPA enables us to store $\boldsymbol{C}_{i,j}$ in off-chip DRAM by eliminating random accesses and allows $\boldsymbol{A}_{i,k}$ to be significantly larger as it can consume the entire on-chip fast storage. Thus, storing $\boldsymbol{C}_{i,j}$ in DRAM can reduce the overall off-chip traffic with larger block size despite the additional traffic for streaming accumulation. From now on, whenever SSPA is referred it is implied that $\boldsymbol{C}_{i,j}$ is stored in off-chip memory and entire block level accumulation is conducted after $\boldsymbol{C}_{i,j}^{temp,k}$ is computed for all k, i.e. $Q = N_z$, unless otherwise stated.

For simplicity, let the number of blocks in row and column direction be equal and denoted as $N_y = N_x = N_z = N_{blk}$. We name any accumulator that stores $\boldsymbol{C}_{i,j}$ in the on-chip memory as Random Access Accumulator (RAA). Let N_{blk} for RAA and SSPA be N_{blk}^{RAA} and N_{blk}^{SSPA} respectively. Then the total off-chip traffic for these methods

using BIP block traversal can be formulated as the followings.

$$D_{\text{BIP}}^{\text{RAA}} = N_{blk}^{\text{RAA}} \cdot vol(\boldsymbol{A}) + N_{blk}^{\text{RAA}} \cdot vol(\boldsymbol{B}) + vol(\boldsymbol{C}) \tag{7.6}$$

$$\begin{aligned} D_{\text{BIP}}^{\text{SSPA}} &= N_{blk}^{\text{SSPA}} \cdot vol(\boldsymbol{A}) + N_{blk}^{\text{SSPA}} \cdot vol(\boldsymbol{B}) + \sum_{j=0}^{N_{blk}^{\text{SSPA}}-1} \sum_{i=0}^{N_{blk}^{\text{SSPA}}-1} vol(\boldsymbol{C}_{i,j}) \\ &\quad + 2 \sum_{j=0}^{N_{blk}^{\text{SSPA}}-1} \sum_{i=0}^{N_{blk}^{\text{SSPA}}-1} \sum_{k=0}^{N_{blk}^{\text{SSPA}}-1} vol(\boldsymbol{C}_{i,j}^{temp,k}) \\ &= N_{blk}^{\text{SSPA}} \cdot vol(\boldsymbol{A}) + N_{blk}^{\text{SSPA}} \cdot vol(\boldsymbol{B}) + vol(\boldsymbol{C}) \\ &\quad + 2 \sum_{j=0}^{N_{blk}^{\text{SSPA}}-1} \sum_{i=0}^{N_{blk}^{\text{SSPA}}-1} \sum_{k=0}^{N_{blk}^{\text{SSPA}}-1} vol(\boldsymbol{C}_{i,j}^{temp,k}) \end{aligned} \tag{7.7}$$

Equation 7.6 is identical to the formulation given in Equation 7.4. On the other hand, in Equation 7.7 the off-chip traffic related to $\boldsymbol{C}$ is replaced by the formula given in Equation 7.5. The last term in Equation 7.7 is evidently the additional traffic related to off-chip streaming accumulation with SSPA. However, for any given size of on-chip memory N_{blk}^{SSPA} is smaller than N_{blk}^{RAA} and, thus, can render $D_{\text{BIP}}^{\text{SSPA}}$ to be less than $D_{\text{BIP}}^{\text{RAA}}$. When input matrices become larger in data volume, the difference between N_{blk}^{RAA} and N_{blk}^{SSPA} increases. Additionally, when $\boldsymbol{C}$ becomes sparser the traffic overhead of SSPA, i.e. the last term in Equation 7.7, becomes smaller. Hence, in these two scenarios off-chip streaming accumulation can effectively reduce DRAM traffic for SpGEMM.

7.2.2 Block Traversal using BPOP

In this work, for SpGEMM in shared memory scenario we have only considered BIP block traversal so far for 2D partitioned data that is based on SRUMMA [90] algorithm.

BIP block traversal is useful for accumulation using random access because only one output matrix block is accumulated uninterruptedly and is streamed out to DRAM once after final accumulation. That means, $\boldsymbol{C}_{i,j}$ remains in the on-chip fast memory for the entire accumulation period without being transferred. This maximizes the matrix block size for RAA. On the other hand, for SSPA the temporary resultant blocks of $\boldsymbol{C}_{i,j}$, i.e. $\boldsymbol{C}^{(}_{i,j}temp,k)$, are streamed out to DRAM anyway. Hence, for SSPA it is not mandatory to accumulate $\boldsymbol{C}_{i,j}$ uninterruptedly. Leveraging is opportunity, we propose Block-level Partial Outer Product (BPOP) traversal scheme, which further reduces the off-chip traffic for 2D partitioned SpGEMM. BPOP is elaborated in Figure 7.10. A pseudocode of SSPA using BPOP also given in Pseudocode 3.

The main idea of BPOP is that once $\boldsymbol{A}_{i,k}$ is transferred to on-chip storage, all the required computations related to it can be conducted and results can be stored in DRAM. For the given example in Figure 7.10, for each k, all the temporary resultant blocks of $\boldsymbol{C}$, i.e. $\boldsymbol{C}^{temp,k}_{i,j=0:3}$, can be computed using SSPA by sequentially streaming in $\boldsymbol{B}_{k,j}$ for all j. All temporary resultant blocks $\boldsymbol{C}^{temp,k}_{i,j=0:3}$ are streamed out to DRAM. After k reaches $(N_z - 1)$, all temporary resultant blocks $\boldsymbol{C}^{temp,k=0:3}_{i,j}$ for any particular j are streamed back from DRAM and finally merged using SSPA to construct $\boldsymbol{C}_{i,j}$. This process is continued for all values of j and i to construct all the blocks of $\boldsymbol{C}$. Pseudocode 3 elaborates this process.

Advantage of BPOP

The main benefit of BPOP is that matrix $\boldsymbol{A}$ is transferred from DRAM only once instead of $N_x = N^{\text{SSPA}}_{blk}$ times as formulated in Equation 7.7 for BIP. It should be noted that BPOP is practically feasible only when the accumulation is conducted using sequential access such as SSPA. Furthermore, BPOP does not cause any increase in

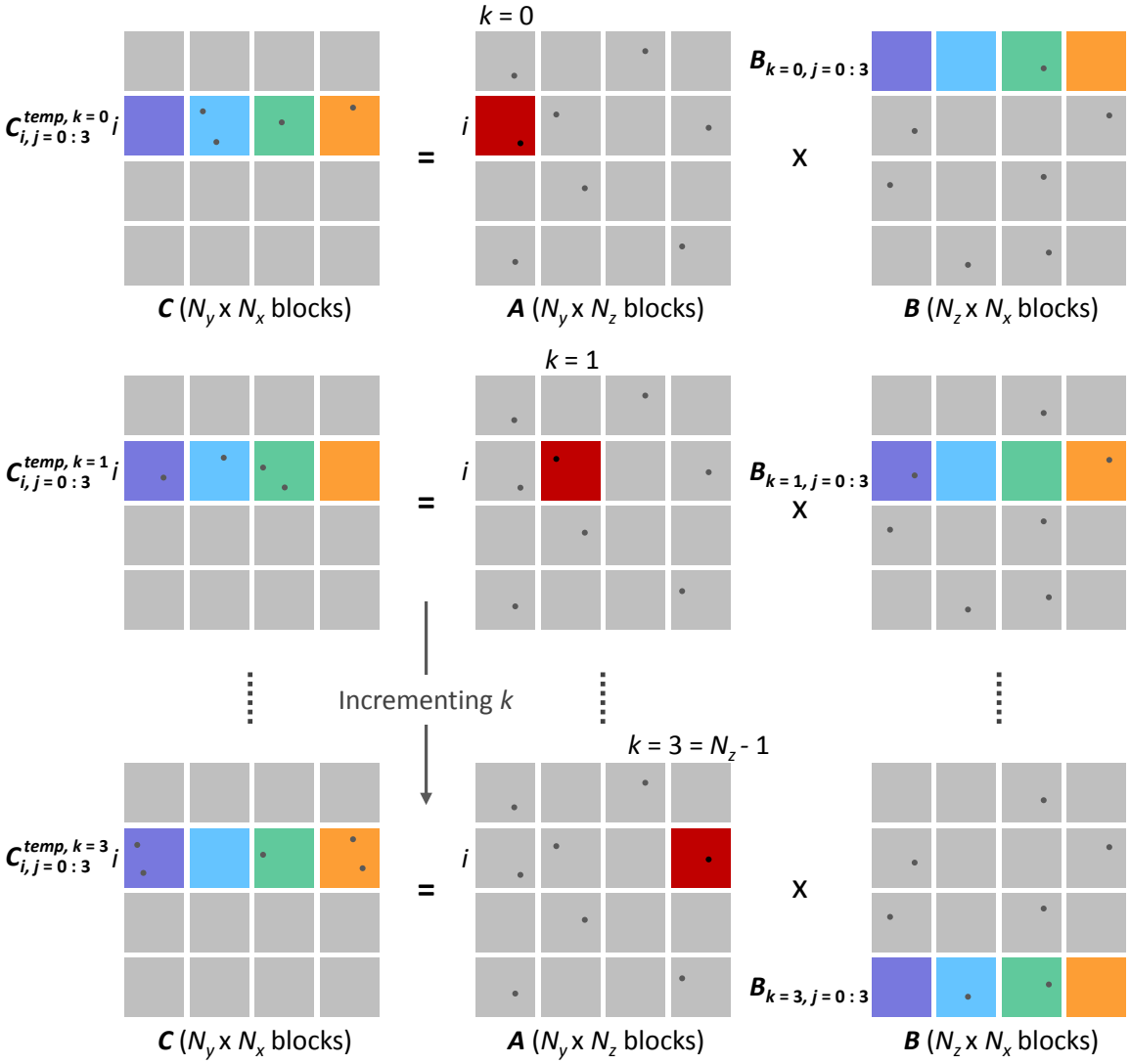


Figure 7.10: 2D partitioned SpGEMM block traversal using Block-level Partial Outer Product (BPOP). We assumed $Q = N_z$ for inter block accumulation.

traffic related to $\boldsymbol{B}$ or $\boldsymbol{C}$ as the overall number of operations for accumulation remains the same as BIP. Hence we can modify the total off-chip traffic formulation of SSPA

Pseudocode 3: Pseudocode for columnbycolumn algorithm using SSPA and BPOP. We assumed $Q = N_z$ for inter block accumulation.

```
1  for i = 0 to N_y − 1 do
2      for k = 0 to N_z − 1 do
3          Stream matrix block A_{i,k} from DRAM
4          for j = 0 to N_x − 1 do
5              Stream matrix block B_{k,j} from DRAM
6              for all non-empty columns B_{k,j}(:,n) do
7                  for each nonzero in B_{k,j}(q,n) in B_{k,j}(:,n) do
8                      Random access column A_{i,k}(:,q)
9                      C^{temp,k}_{i,j}(:,n) += A_{i,k}(:,q) · B_{k,j}(q,n) [intra-block SSPA]
10                 end
11             end
12             Stream out C^{temp,k}_{i,j} to main memory
13         end
14     end
15     for j = 0 to N_x − 1 do
16         for k = 0 to N_z − 1 do
17             C_{i,j} += C^{temp,k}_{i,j} [inter-block SSPA]
18         end
19     end
20 end
```

($Q = N_z$) in Equation 7.7 to the following when using BPOP 2D traversal scheme.

$$D_{\text{BPOP}}^{\text{SSPA}} = vol(\boldsymbol{A}) + N_{blk}^{\text{SSPA}} \cdot vol(\boldsymbol{B}) + vol(\boldsymbol{C}) + 2 \sum_{j=0}^{N_{blk}^{\text{SSPA}}-1} \sum_{i=0}^{N_{blk}^{\text{SSPA}}-1} \sum_{k=0}^{N_{blk}^{\text{SSPA}}-1} vol(\boldsymbol{C}_{i,j}^{temp,k}) \tag{7.8}$$

7.3 Evaluation and Results

To test the effectiveness of our proposed methods and developed accelerator we have conducted off-chip traffic reduction analysis on a number of real world graphs, except

the ones named with prefix 'Syn', listed in Table 7.1 that are collected from the sparse matrix collection of [82]. The last two graphs listed in Table 7.1 have uniformly distributed random nonzeros that are synthetically generated following Erdos Rényi model [74]. We have further conducted experiments to measure the performance and energy efficiency of our developed 16nm FinFET ASIC accelerator using the same specs detailed in Chapter 4. For all experiments, the available on-chip storage for the block of input matrix $\boldsymbol{A}$ is 8MB. For SSPA, $\boldsymbol{C}_{i,j}$ is stored in off-chip DRAM and entire block level accumulation is done after all temporary resultant blocks are computed, i.e. $Q = N_z$.

Table 7.1: Graph data sets used for SpGEMM analysis.

Description	# Nodes (M)	Avg. Degree	# Edges (M)	$nnz(\boldsymbol{C})$ (M)
cs4	0.02	3.90	0.09	0.05
wing_nodal	0.01	13.80	0.15	0.20
delaunay_n18	0.26	6.00	1.57	1.32
coAuthorsCiteseer	0.23	7.16	1.63	2.84
coAuthorsDBLP	0.30	6.54	1.96	4.67
belgium_osm	1.44	2.15	3.10	1.47
delaunay_n20	1.05	6.00	6.29	5.28
NLR	4.16	5.99	24.98	19.89
rgg_n_2_20_s0	1.05	13.14	13.78	17.12
AS365	3.80	5.98	22.74	17.97
venturiLevel3	4.03	4.00	16.11	10.67
road_central	14.08	2.41	33.87	87.80
webbase-1M	1.00	3.00	3.00	51.11
delaunay_n24	16.78	6.00	100.60	347.32
wb-edu	9.84	5.80	57.07	630.08
cage15	5.16	19.24	99.20	929.02
Syn_50_7.2	50.00	7.20	360.00	2109.00
Syn_100_16	100.00	16.00	1600.00	3416.00

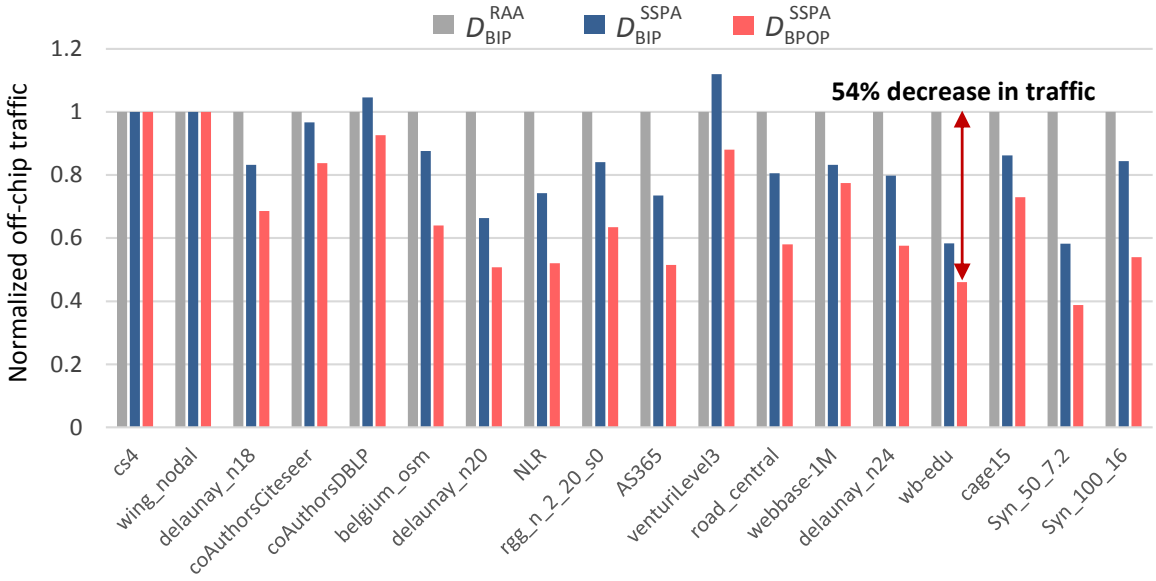


Figure 7.11: Off-chip traffic (normalized against $D_{\text{BIP}}^{\text{RAA}}$) comparison among various combinations of output block storage locations and block traversal schemes. On-chip memory size is 8MB for all. For $D_{\text{BIP}}^{\text{RAA}}$ $\boldsymbol{C}_{i,j}$ is stored in on-chip memory. For $D_{\text{BIP}}^{\text{SSPA}}$ and $D_{\text{BPOP}}^{\text{SSPA}}$ $\boldsymbol{C}_{i,j}$ is stored in off-chip memory.

7.3.1 Off-chip Traffic Reduction

The total off-chip traffic for SpGEMM computation of $\boldsymbol{A}^2$ on the graphs in Table 7.1 using $8MB$ on-chip memory is show in Figure 7.11. $D_{\text{BPOP}}^{\text{SSPA}}$ and $D_{\text{BIP}}^{\text{SSPA}}$ represent off-chip traffic for SSPA using BIP and BPOP 2D block traversal respectively. Both of these are normalized against the off-chip traffic that would be incurred if we have stored $\boldsymbol{C}_{i,j}$ in on-chip memory during accumulation, e.g. using RAA, along with BIP traversal. Hence, any value lower than 1 indicates traffic reduction and vice versa.

From Table 7.1 we can see that $D_{\text{BIP}}^{\text{SSPA}}$ incurs less traffic for most of the graphs. This is completely attributed to the reduction in number of blocks in 2D partitioned data due to off-chip storage of $\boldsymbol{C}_{i,j}$ in SSPA. The number of blocks in one reduction, i.e. either row-wise or column-wise assuming square blocks, for RAA (N_{blk}^{RAA}) and SSPA

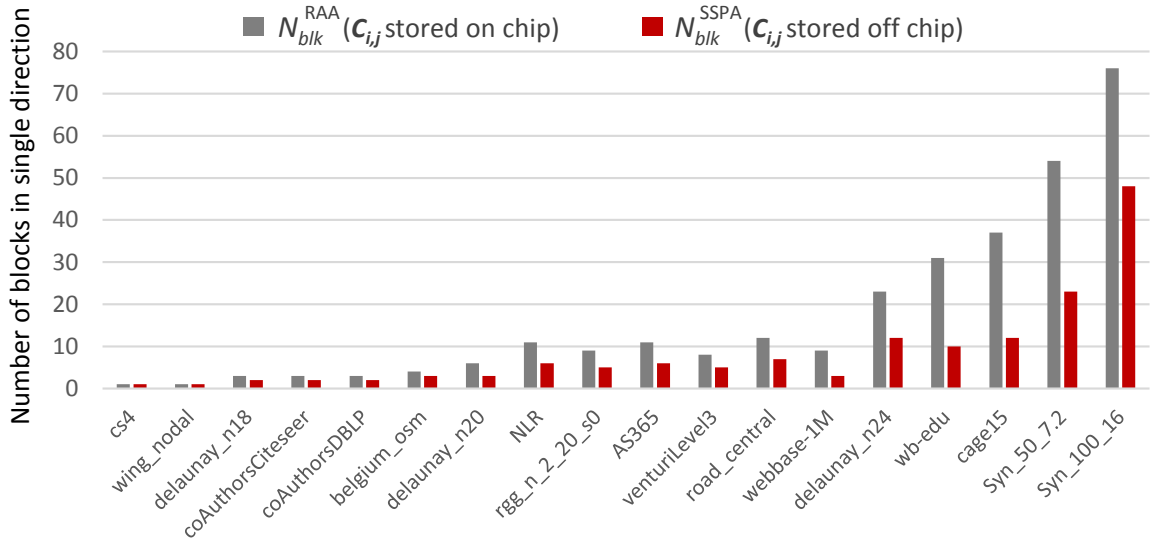


Figure 7.12: Comparison of number blocks (in one direction) for 2D partitioned data between RAA ($\boldsymbol{C}_{i,j}$ stored in on-chip memory) and SSPA ($\boldsymbol{C}_{i,j}$ stored in off-chip memory).

(N_{blk}^{SSPA}) are shown in Figure 7.12. For the first two graphs, there are no reduction in traffic as $\boldsymbol{A}$ entirely fits on chip. For few graphs, $D_{\text{BIP}}^{\text{SSPA}}$ in fact increases the off-chip traffic. This is because the additional overhead of off-chip streaming accumulation is not compensated by the smaller (N_{blk}^{SSPA}). On the other hand, SSPA along with BPOP significantly reduces off-traffic for all the graphs where entire $\boldsymbol{A}$ doesn't fit on chip. The additional reduction of $D_{\text{BPOP}}^{\text{SSPA}}$ is attributed to the fact that $\boldsymbol{A}$ is transfered from DRAM only once instead of N_{blk}^{RAA} or N_{blk}^{SSPA} times as in the case for BIP.

7.3.2 Performance and Energy Efficiency

To demonstrate the performance and energy efficiency of our proposed SpGEMM accelerator, we have used several benchmarks including COTS architectures with commercial sparse libraries and 3D stacked ASIC architecture. We used Intel MKL SpGEMM routine on Xeon E5-2430 (32nm, 15MB cache) with 12 threads. Another

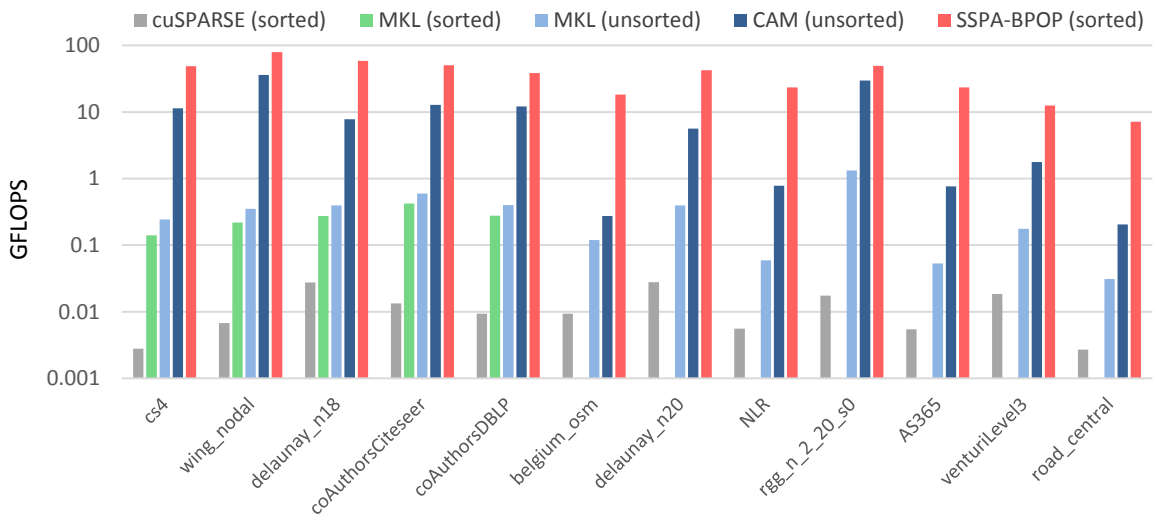


Figure 7.13: Performance comparison of proposed SSPA based SpGEMM using BPOP against COTS and ASIC benchmarks.

COTS benchmark used is CUDA Sparse (cuSPARSE) SpGEMM routine on Nvidia GPU GTX 650 (384 cores). As the custom hardware benchmark we have used the CAM and 3D stacked memory based ASIC (32nm, 668GB/s off-chip bandwidth) in [50]. For one version of COTS MKL implementation and ASIC benchmark, the row indices in any given column of the output matrix are unsorted. As the performance and energy efficiency metric we have used giga floating point operations per second (GFLOPS) and giga floating point operations per second per watt (GFLOPS/W) respectively. Figure 7.13 and Figure 7.14 report the experimental results. We have reported results only for the graphs that we were able to run on or results are available for at least one of the benchmarks.

We can see that our proposed accelerator deliver orders of magnitude better performance and efficiency than the COTS benchmarks. Even though our accelerator uses HBM and smaller technology node, the CPU has twice on-chip storage and the GPU has much higher number of parallel cores. The CAM ASIC benchmark has more

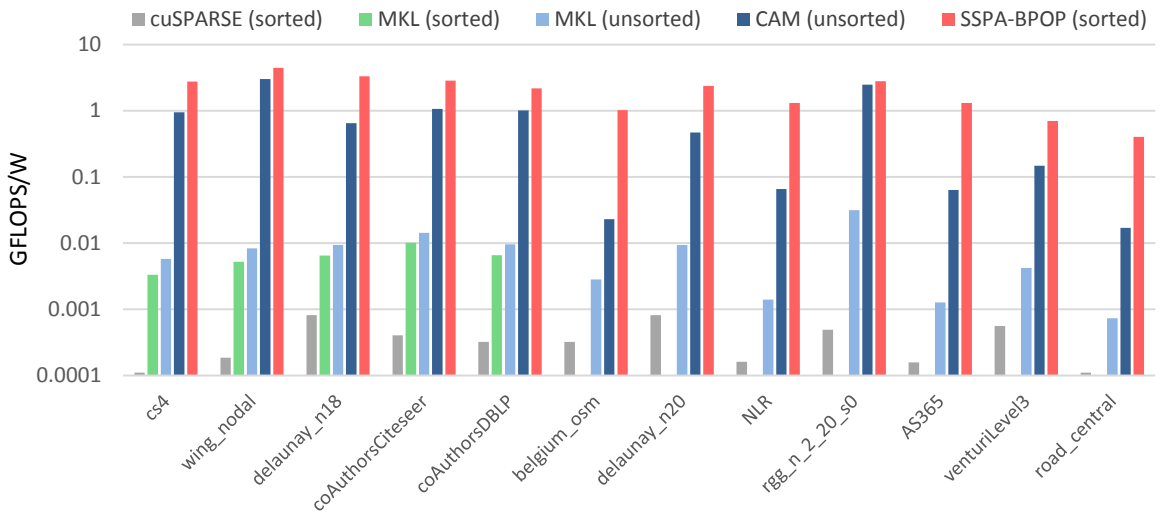


Figure 7.14: Energy efficiency comparison of proposed SSPA based SpGEMM using BPOP against COTS and ASIC benchmarks.

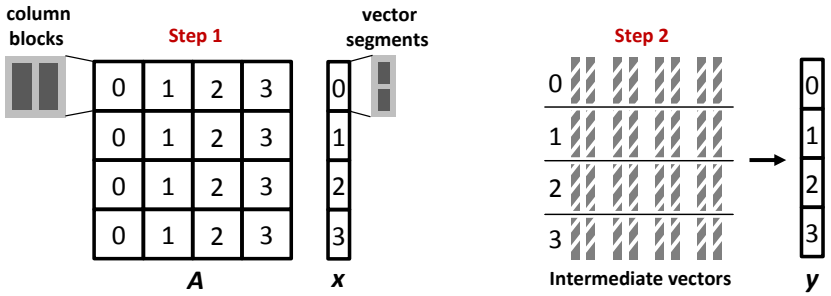
off-chip bandwidth (668GB/s) than our accelerator. However, our proposed solution performs better mainly due to reduced off-chip traffic and larger block size.

Chapter 8

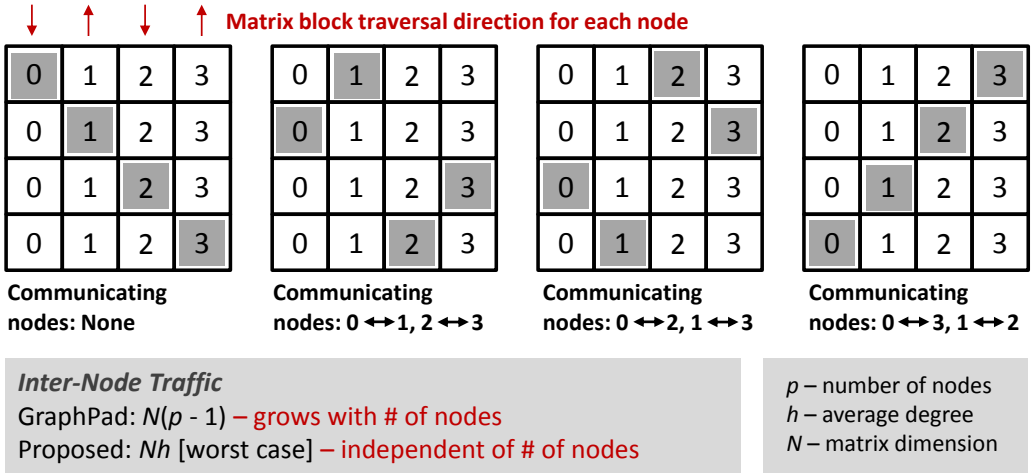
Future Directions and Conclusion

8.1 Future Directions

Distributed Memory System. In this work we have only discussed shared memory architecture for Two-Step SpMV. To address larger graphs, extension of Two-Step algorithm for distributed memory systems can be explored. A potential distributed scheme is depicted in Figure 8.1. The matrix and vectors are distributed al shown in Figure 8.1a. Matrix is two dimensionally partitioned in p^2 partitions, where p is the number of nodes. The source vector is partitioned into p blocks. The matrix partitions are column blocked within each node as described for shared memory scenario. Source vector partition assigned to each node is also segmented into multiple segments according to the on-chip storage limits of each node. Each node possesses multiple column blocks and corresponding source vector segments during 1^{st} step of Two-Step algorithm. Every node computes the partial SpMV on local data in the 1^{st} step. The intermediate sparse vectors generated at the completion of 1^{st} step are transmitted to another node following the scheme depicted in Figure 8.1b. To make



(a) Data ownership among distributed nodes.



(b) Inter node communication scheme during 1^{st} step of Two-Step algorithm. Grey box indicates the block on which the corresponding node is conducting 1^{st} step of Step2 spmv.

Figure 8.1: Implementation of Two-Step algorithm in distributed memory scenario.

the inter-node communication effective, intermediate sparse vectors are segmented and the same segments from all the intermediate vectors are merged in one node. At any given moment, each node only communicates with one another node and utilizes the inter-node link bandwidth entirely. To ensure balanced time sharing of the inter-node links, we horizontally partition the column blocks and deploy a scheme of sequence to operate on the column block partitions.

Specialized Sparse Operation Combination. There are graph analytic applications which can often benefited by special combination sparse matrix operations.

For example, Triangle Counting is an application for understanding graph structure, which can be mathematically computed using SpGEMM. However, efficient algorithm for Triangle Counting boils down to a integrated vector-matrix-vector [97] operation followed by accumulation, where all the vectors and matrix are sparse. This vector-matrix-vector multiplication is essentially set of parallel 2-way merge operations that can efficiently implemented by the multi-way merge network developed in this work. Therefore, important specialized combination of fundamental kernels can be explored as a future extension of this research. Furthermore, as multi-way merge merge is an fundamental primitive for numerous operations, future research should consider the impact of this accelerator in other fields than graph analytics.

Heterogeneous Architecture. Heterogeneous architectures, such as shared memory CPU-FPGA, provide a new dimension in accelerating data intensive operations as it can deliver flexibility and usability of general purpose architecture along with the capability of custom hardware. This kind of architecture actually poses an interesting platform for our proposed Two-Step algorithm. It is because, only the multi-way merge operation in second step requires custom hardware for efficient implementation. On the other hand, partial SpMV operation in the first step can be implemented on general purpose architecture relatively more efficiently. Hence, on a CPU-FPGA heterogeneous system, technically we should be able to run step 1 on CPU and step 2 on FPGA. Exploration in this field deserves attention because such implementation can potentially save silicon real estate on custom hardware, which can be put to better use for accelerating multi-way merge operation for larger problems.

New Memory Technology. As fast random access memory is the most crucial resource for sparse matrix operations, advancements in memory technologies will directly impact acceleration methodologies. There are a number of emerging fast

random access memory technologies, such as magnetoresistive RAM (MRAM), spin transfer torque RAM (STT-RAM), resistive RAM (ReRAM) and phase change memory (PCM), that present wide range of possibilities in improving sparse matrix kernel scalability, performance and efficiency. These technologies possess characteristics that can be exploited to our advantage. For example, MRAM is a non-volatile memory technology that has higher density and lower leakage power than SRAM. However, MRAM requires higher dynamic power for writing than to read, which is also higher than what it is for SRAM write. Additionally, MRAM read speed is higher than its write speed. Hence, MRAM has more efficient read access than write access of itself and read access of SRAM [98]. Therefore, combination of MRAM-SRAM memory technologies can be explored where read and write accesses are prioritized to MRAM and SRAM respectively. Thus, these new memory technologies open a vast new research space for sparse matrix acceleration.

8.2 Concluding Remarks

Research on architectural and circuit level opportunities in accelerating sparse matrix algebra hasn't received much deserved attention. While software frameworks and libraries take the burden of writing efficient graph analytic code away from wide range of users, incoherence between traditional memory hierarchy based compute engine, where locality is highly expected, and sparsity severely limits performance and energy efficiency. This work makes an effort to overcome this barrier by developing a data sparsity aware custom architecture. This is considered as algorithm/hardware co-optimized approach as algorithm is developed keeping technological limits in mind and later implemented using finely tuned custom hardware.

This co-optimization process is guided by a key observation. Since CMOS has abundant compute capability relative to off-chip bandwidth and DRAM technology provides significantly better bandwidth characteristics than latency, for data intensive applications we should trade bandwidth for latency elimination to extract maximum possible system capabilities. To trade bandwidth for latency elimination, all off-chip random accesses should be converted to sequential access. As long as the overhead of this conversion doesn't surpass the cost of latency bound operation, this approach is should potentially provide better performance and efficiency. In this work we have demonstrated a practical way of implementing this conversion mechanism through algorithm/hardware co-optimization. We have shown that off-chip traffic aware streaming algorithms for sparse linear algebra are required at software level. As general purpose architecture are not efficiently capable of implementing these kind of algorithms due to data locality dependence, we have resorted to custom hardware and circuit level implementation techniques.

One of the main contributions of this work is to identify and develop an custom architecture that can implement off-chip traffic aware streaming algorithms for fundamentally important sparse kernels without diminishing the benefits of random to sequential access conversion. The core hardware primitive for this architecture is a scalable and high performance multi-way merge network. This work devises novel ways such as HCLAM and PRaP that facilitates scalability and fine grained parallelism. We have demonstrated how SpMV, PageRank (iterative SpMV) and SpGEMM operations can be considerably benefited by this common architecture. As these operations are work horses for a wide array of graph analytic applications, our proposed hardware promises broad impact. Moreover, as multi-way merge is a fundamental primitive

for numerous mathematical operations, the proposed methodologies in this work can potentially be useful beyond graph analytics.

In this work, we have demonstrated off-chip traffic aware algorithms, such as Two-Step SpMV, SpGEMM by streaming accumulation and BPOP, that significantly reduce main memory data accesses for large problems. Furthermore, we have implemented traffic and throughput optimization techniques such as VLDI meta-data compression and iteration overlap. A Bloom Filter filter based technique has been proposed to efficiently resolve collisions for HDNs in power-law graphs. Additional hardware techniques for efficient accumulation and multi-channel DRAM data distribution have also been demonstrated.

Another important goal of this work is to handle large problems ($\sim$ billion nodes) in shared memory architecture as modern DRAM main memory sub-system is capable of providing the storage for required working data set. We have found that custom architecture solutions in current literature is severely constrained by on-chip fast memory storage. Hence, in this work we have emphasized in reducing dependency on fast memory for scaling both problem size and computation throughput. For example, Two-Step SpMV is implemented using PRaP and 1D partitioned matrix instead 2D to eliminate prefetch buffer storage increment during scaling. We also have elaborated a trade-off technique between compute resource and on-chip fast memory. Therefore, this architecture has demonstrated large problem handling capability using significantly less on-chip memory, which is unprecedented in current literature.

Our proposed architecture for sparse matrix algebra acceleration makes a research effort in exploring the architectural and lower level circuit domain. We envision the overall solution to contribute in a ecosystem provided by GraphBLAS-like standards, where versatile user base can have easy access to efficient graph analytic capability

through a well defined set mathematical operations. We expect that promising results demonstrated in work will evoke research efforts in this arena to further the impact by integrating unexplored and emerging technologies.

Bibliography

[1] J. Kepner, D. Bade, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the graphblas: Seven good reasons," *arXiv preprint arXiv:1504.01039*, 2015.

[2] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[3] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015. [Online]. Available: http://dx.doi.org/10.14778/2809974.2809983

[4] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," vol. 31, no. 3, pp. 302–325, September 2005.

[5] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[6] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[7] C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg, *Practical graph analytics with apache giraph*. Springer, 2015.

[8] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings*

of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010, pp. 135–146.

[10] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: a datalog-based language for large-scale graph analysis," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1906–1917, 2013.

[11] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 2012, pp. 440–451.

[12] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, Citeseer, 2003.

[13] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[14] J. D. Davis and E. S. Chung, "Spmv: A memory-bound application on the GPU stuck between a rock and a hard place," *Microsoft Research Silicon Valley, Technical Report14 September*, 2012.

[15] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*. IEEE, 2008, pp. 283–292.

[16] E. Nurvitadhi, A. Mishra, Y. Wang, G. Venkatesh, and D. Marr, "Hardware accelerator for analytics of sparse data," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 1616–1621.

[17] D. A. Patterson, "Latency lags bandwidth," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[18] E. Nurvitadhi, A. Mishra, and D. Marr, "A sparse matrix vector multiply accelerator for support vector machine," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Oct 2015, pp. 109–116.

[19] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342011403516

[20] P. D'Alberto and A. Nicolau, "R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," *Algorithmica*, vol. 47, no. 2, pp. 203–213, Feb. 2007. [Online]. Available: http://dx.doi.org/10.1007/s00453-006-1224-z

[21] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices," in *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, ser. PARA'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 260–269. [Online]. Available: http://dl.acm.org/citation.cfm?id=1775059.1775097

[22] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[23] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 979–990.

[24] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[25] M. Bohr, "A 30 year retrospective on Dennard's MOSFET scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, Winter 2007.

[26] A. Madry, "From graphs to matrices, and back: new techniques for graph algorithms," Ph.D. dissertation, Massachusetts Institute of Technology, 2011.

[27] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[28] M. D. Hill, S. Adve, L. Ceze, M. J. Irwin, D. Kaeli, M. Martonosi, J. Torrellas, T. F. Wenisch, D. Wood, and K. Yelick, "21st century computer architecture," *arXiv preprint arXiv:1609.06756*, 2016.

[29] N. Hardavellas, "The rise and fall of dark silicon," *The advanced computing systems association*, pp. 7–17, 2012.

[30] W. Dally, "Power efficient supercomputing," in *Accelerator-based Computing and Manycore Workshop (presentation)*, vol. 1, 2009.

[31] S. W. Keckler, "Echelon: Nvidia & team's uhpc project," in *DARPA Ubiquitous High Performance Computing Program Meeting, Houston, TX*, 2011.

[32] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[33] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–13.

[34] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 37–47.

[35] L. Wilson, "International technology roadmap for semiconductors (itrs)," *Semiconductor Industry Association*, 2013.

[36] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing memory performance for FPGA implementation of PageRank," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015, pp. 1–6.

[37] J. B. White III and P. Sadayappan, "On improving the performance of sparse matrix-vector multiplication," in *High-Performance Computing, 1997. Proceedings. Fourth International Conference on.* IEEE, 1997, pp. 66–71.

[38] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* ACM, 2009, p. 18.

[39] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," *SIGPLAN Not.*, vol. 45, no. 5, pp. 115–126, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693471

[40] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 231–242, 2011.

[41] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[42] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies," *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[43] M. Garland, "Sparse matrix computations on manycore GPU's," in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 2–6.

[44] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific computing kernels on the cell processor," *International Journal of Parallel Programming*, vol. 35, no. 3, pp. 263–298, 2007.

[45] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "Fpga and gpu implementation of large scale spmv," in *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*, June 2010, pp. 64–70.

[46] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos, "Fpga vs. gpu for sparse matrix vector multiply," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, Dec 2009, pp. 255–262.

[47] D. Zou, Y. Dou, S. Guo, and S. Ni, "High performance sparse matrix-vector multiplication on fpga," *IEICE Electronics Express*, vol. 10, no. 17, pp. 20 130 529–20 130 529, 2013.

[48] Y. El-Kurdi, W. J. Gross, and D. Giannacopoulos, "Sparse matrix-vector multiplication for finite element method matrices on fpgas," in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, April 2006, pp. 293–294.

[49] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 111–117.

[50] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–6.

[51] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988. [Online]. Available: http://doi.acm.org/10.1145/48529.48535

[52] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3D Systems Integration Conference (3DIC), 2013 IEEE International*, Oct 2013, pp. 1–7.

[53] Q. Wu, K. Rose, J. Q. Lu, and T. Zhang, "Impacts of though-dram vias in 3d processor-dram integrated systems," in *3D System Integration, 2009. 3DIC 2009. IEEE International Conference on*, Sept 2009, pp. 1–6.

[54] T. Haveliwala, "Efficient computation of pagerank," Stanford InfoLab, Technical Report 1999-31, 1999. [Online]. Available: http://ilpubs.stanford.edu:8090/386/

[55] J. Kunegis, "Konect: The koblenz network collection," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion. New York, NY, USA: ACM, 2013, pp. 1343–1350. [Online]. Available: http://doi.acm.org/10.1145/2487788.2488173

[56] L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, and J. M. Zurada, *Artificial Intelligence and Soft Computing: 12th International Conference, ICAISC 2013, Zakopane, Poland, June 9-13, 2013, Proceedings, Part I ... / Lecture Notes in Artificial Intelligence)*. Springer Publishing Company, Incorporated, 2013.

[57] K. Fleming, M. King, M. C. Ng, A. Khan, and M. Vijayaraghavan, "High-throughput pipelined mergesort," in *Proceedings of the Sixth ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, ser. MEMOCODE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 155–158. [Online]. Available: https://doi.org/10.1109/MEMCOD.2008.4547704

[58] T. Usui, T. V. Chu, and K. Kise, "A cost-effective and scalable merge sorter tree on fpgas," in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, Nov 2016, pp. 47–56.

[59] "Nallatech 510t compute acceleration card," https://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-510t-fpga-computing-acceleration-card/.

[60] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[61] S. Mashimo, T. V. Chu, and K. Kise, "High-performance hardware merge sorter," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 1–8.

[62] D. Koch and J. Torresen, "Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting," in *Proceed-*

ings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2011, pp. 45–54.

[63] D. Morris, K. Vaidyanathan, N. Lafferty, K. Lai, L. Liebmann, and L. Pileggi, "Design of embedded memory and logic based on pattern constructs," in *Symposium on VLSI Technology (VLSIT)*, 2011, pp. 104–105.

[64] Q. Zhu, K. Vaidyanathan, O. Shachamy, M. Horowitzy, L. Pileggi, and F. Franchetti, "Design automation framework for application-specific logic-in-memory blocks," in *IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, July 2012, pp. 125–132.

[65] D. Morris, V. Rovner, L. Pileggi, A. Strojwas, and K. Vaidyanathan, "Enabling application-specific integrated circuits on limited pattern constructs," in *Symposium on VLSI Technology (VLSIT)*, 2010, pp. 139–140.

[66] J. Standard, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[67] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," *2017 IEEE International Memory Workshop (IMW)*, pp. 1–4, 2017.

[68] K. Cho, H. Lee, H. Kim, S. Choi, Y. Kim, J. Lim, J. Kim, H. Kim, Y. Kim, and Y. Kim, "Design optimization of high bandwidth memory (hbm) interposer considering signal integrity," in *2015 IEEE Electrical Design of Advanced Packaging and Systems Symposium (EDAPS)*, Dec 2015, pp. 15–18.

[69] "Intel® Stratix10® FPGA platform," https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html.

[70] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–11.

[71] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[72] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[73] Y. Qiao, T. Li, and S. Chen, "One memory access bloom filters and their generalization," in *2011 Proceedings IEEE INFOCOM*, April 2011, pp. 1745–1753.

[74] P. Erdos and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[75] S. Zhou, K. Lakhotia, S. G. Singapura, H. Zeng, R. Kannan, V. K. Prasanna, J. Fox, E. Kim, O. Green, and D. A. Bader, "Design and implementation of parallel PageRank on multicore platforms," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2017, pp. 1–6.

[76] K. Lakhotia, R. Kannan, and V. K. Prasanna, "Accelerating pagerank using partition-centric processing," *CoRR*, vol. abs/1709.07122, 2017. [Online]. Available: http://arxiv.org/abs/1709.07122

[77] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, March 2015, pp. 1543–1546.

[78] K. Chen, S. Li, N. Muralimanohar, J.-H. Ahn, J. Brockman, and N. Jouppi, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Design, Automation Test in Europe (DATE)*, 2012, pp. 33–38.

[79] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An fpga framework for edge-centric graph processing," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. New York, NY, USA: ACM, 2018, pp. 69–77. [Online]. Available: http://doi.acm.org/10.1145/3203217.3203233

[80] A. Rungsawang and B. Manaskasemsak, "Fast pagerank computation on a gpu cluster," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Feb 2012, pp. 450–456.

[81] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient pagerank and spmv computation on amd gpus," in *2010 39th International Conference on Parallel Processing*, Sept 2010, pp. 81–89.

[82] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[83] V. B. Shah, "An interactive system for combinatorial scientific computing with an emphasis on programmer productivity," Ph.D. dissertation, Santa Barbara, CA, USA, 2007, aAI3274428.

[84] G. Penn, "Efficient transitive closure of sparse matrices over closed semirings," *Theoretical Computer Science*, vol. 354, no. 1, pp. 72 – 81, 2006, algebraic Methods in Language Processing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397505008546

[85] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks." *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 76 3 Pt 2, p. 036106, 2007.

[86] N. Bell, S. Dalton, and L. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012. [Online]. Available: https://doi.org/10.1137/110838844

[87] A. Buluç and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," *CoRR*, vol. abs/1006.2183, 2010. [Online]. Available: http://arxiv.org/abs/1006.2183

[88] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, Sep. 1978. [Online]. Available: http://doi.acm.org/10.1145/355791.355796

[89] G. H. Loh, "3D-Stacked memory architectures for multi-core processors," in *35th International Symposium on Computer Architecture (ISCA)*, June 2008, pp. 453–464.

[90] M. Krishnan and J. Nieplocha, "Srumma: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pp. 70–.

[91] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Bozeman, MT, USA, 1969, aAI7010025.

[92] P. N. Q. Anh, R. Fan, and Y. Wen, "Balanced hashing and efficient gpu sparse general matrix-matrix multiplication," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 36:1–36:12. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926273

[93] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer International Publishing, 2015, pp. 48–57.

[94] J. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992. [Online]. Available: https://doi.org/10.1137/0613024

[95] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "High-performance sparse matrix-matrix products on intel KNL and multicore architectures," *CoRR*, vol. abs/1804.01698, 2018. [Online]. Available: http://arxiv.org/abs/1804.01698

[96] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 370–381.

[97] T. M. Low, V. N. Rao, M. Lee, D. Popovici, F. Franchetti, and S. McMillan, "First look: Linear algebra-based triangle counting without matrix multiplication," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2017, pp. 1–6.

[98] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie, "Power and performance of read-write aware hybrid caches with non-volatile memories," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 737–742. [Online]. Available: http://dl.acm.org/citation.cfm?id=1874620.1874803