Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization

Fazle Sadi
fsadi@alumni.cmu.edu
Carnegie Mellon University

Joe Sweeney
jpsety@gmail.com
Carnegie Mellon University

Tze Meng Low
lowt@andrew.cmu.edu
Carnegie Mellon University

James C. Hoe
jhoe@cmu.edu
Carnegie Mellon University

Larry Pileggi
pileggi@andrew.cmu.edu
Carnegie Mellon University

Franz Franchetti
franzf@andrew.cmu.edu
Carnegie Mellon University

ABSTRACT

The importance of Sparse Matrix dense Vector multiplication (SpMV) operation in graph analytics and numerous scientific applications has led to development of custom accelerators that are intended to overcome the difficulties of sparse data operations on general purpose architectures. However, efficient SpMV operation on large problem (i.e. working set exceeds on-chip storage) is severely constrained due to strong dependence on limited amount of fast random access memory to scale. Additionally, unstructured matrix with high sparsity pose difficulties as most solutions rely on exploitation of data locality. This work presents an algorithm co-optimized scalable hardware architecture that can efficiently operate on very large (∼billion nodes) and/or highly sparse (avg. degree <10) graphs with significantly less on-chip fast memory than existing solutions. A novel parallelization methodology for implementing large and high throughput multi-way merge network is the key enabler of this high performance SpMV accelerator. Additionally, a data compression scheme to reduce off-chip traffic and special computation for nodes with exceptionally large number of edges, commonly found in power-law graphs, are presented. This accelerator is demonstrated with 16-nm fabricated ASIC and Stratix® 10 FPGA platforms. Experimental results show more than an order of magnitude improvement over current custom hardware solutions and more than two orders of magnitude improvement over commercial off-the-shelf (COTS) architectures for both performance and energy efficiency.

KEYWORDS

SpMV, custom hardware, sparse matrices, merge parallelization

ACM Reference Format:
Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3352460.3358330

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358330

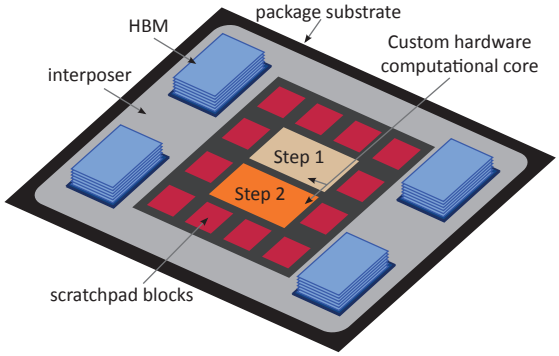
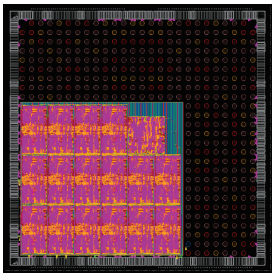


Figure 1: Proposed algorithm co-optimized custom architecture for SpMV on very large and highly sparse graphs.

1 INTRODUCTION

SpMV can be denoted as $y = Ax + y$, where A is the sparse matrix, x and y are respectively the source and resultant dense vectors. While SpMV has wide range of applications including graph analytics, this very kernel often becomes the bottleneck as it renders very low fraction of the peak processor performance (<10%) [4] and poor energy efficiency on COTS architectures. The main reason is poor utilization of off-chip main memory bandwidth due to frequent random access or excessive off-chip traffic overhead. Additionally, for general purpose architectures, 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 [14]. This also incurs a high energy overhead. While an arithmetic operation requires 0.5-50pJ, scheduling instructions in modern core consumes 2000pJ [15, 16]. For these reasons, custom architectures are recently explored for SpMV acceleration [11, 14, 31, 35, 39].

In this work, we refer to large problems/graphs in the sense that the working data set is too large to fit in the on-chip storage. As modern main memory sub-system can store working data set for very large (multiple billion node) graphs, shared memory custom architectures should be able to efficiently handle large problems if compute requirements are met. However, the performance and efficiency of SpMV on shared memory custom architectures are aggravated for large problems (∼billion nodes) due to strong dependence on the on-chip fast memory such as Static Random Access Memory (SRAM), Embedded DRAM (eDRAM), etc. This dependence stems from the fact that most custom accelerators store large portion of sparse



ASIC specifications
Frequency: 1.4 GHz
Occupied area: 7.5 mm^2
Leakage power: 0.10 W
Dynamic power: 3.01 W
Total power: 3.11 W

Figure 2: 16nm FinFET ASIC fabricated as the computation core of proposed accelerator.

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. For example, a Field Programmable Gate Array (FPGA) solution [36] reported maximum graph dimension of 2.3M nodes using 8.4MB on-chip SRAM. The largest dimension efficiently handled by an Application Specific Integrated Circuit (ASIC) base solution [14] is only 8M nodes despite using a large 32MB eDRAM scratchpad. Hence, even tens of million, let alone billion, node graphs are difficult to accelerate with shared memory custom hardware.

Another major difficulty with SpMV operation on large matrices is experienced when the data become very sparse (avg. degree <10) and/or unstructured. SpMV acceleration techniques by somehow exploiting locality in the nonzero patterns of the sparse data, such as sophisticated formats, preconditioning, register-blocking, are widely practiced in the literature for cache-based computation paradigms. However, temporal or spatial locality is difficult to find for highly sparse large matrices rendering these methods to be ineffective.

In this paper, we present an algorithm co-optimized custom shared memory hardware accelerator, as depicted in Figure 1, for high performance and energy efficient SpMV operation on very large and highly sparse graphs for which the working data set far exceeds the on-chip fast storage. The computational core of the accelerator is demonstrated with an ASIC fabricated in 16nm technology, as shown in Figure 2. Intel Stratix® 10 FPGA is also demonstrated as another platform for the proposed custom architecture. The goals of the accelerator are - a) full utilization of main memory bandwidth, b) less dependence on fast on-chip memory to scale, c) reduction of off-chip traffic, and d) data locality unaided computational scheme and avoidance of costly pre-processing. For proper utilization of DRAM bandwidth, we adopt an algorithm, namely Two-Step, that conducts SpMV in two separate phases to ensure 100% DRAM streaming access and incurs less off-chip traffic than other streaming algorithms for large matrices. Hence, this algorithm possesses favorable data access characteristics to efficiently utilize off-chip bandwidth. Furthermore, Two-Step algorithm does not depend on spatial/temporal data locality or costly pre-processing of the matrix. However, there are several challenges in implementing Two-Step algorithm. This work primarily focuses is addressing these challenges. The key contributions of this work are as follows.

- Efficient implementation of Two-Step SpMV requires computationally difficult parallel multi-way merge operation to fully utilize high streaming bandwidth offered by modern DRAM, such as 0.5TB/s of 3D stacked High Bandwidth Memory (HBM). This work proposes a novel radix pre-sorter based scalable multi-way merge parallelization scheme that provides high throughput to saturate extreme bandwidth of 3D stacking technology, thus fully utilize off-chip bandwidth.
- We demonstrate a matrix blocking scheme and buffering method that enable Two-Step SpMV implementation to scale for larger problems without significantly increasing on-chip fast memory requirement. This enables our proposed accelerator to handle very large graphs (4 billion nodes) with reasonably small amount (11MB) of on-chip memory.
- This work proposes a meta-data compression technique, namely Variable Length Delta Index (VLDI), that can significantly reduce off-chip traffic and improve performance.
- We demonstrate an optimization method for iterative SpMV to increase throughput and decrese off-chip traffic.
- This work develops a Bloom Filter based method to enhance SpMV performance for power-law graphs that contain nodes with disproportionately large number of neighbors. This allows to avoid complicated sparse formats and ensures efficient execution without large on-chip storage.

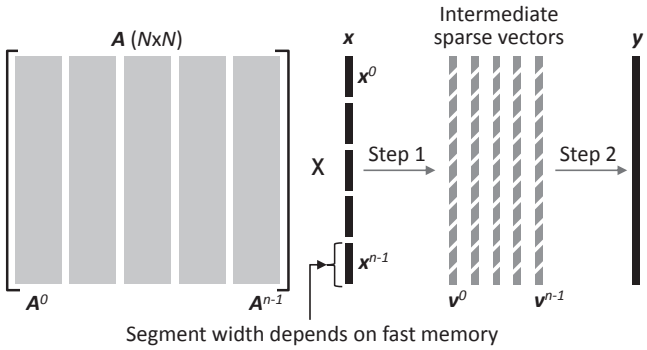


Figure 3: Two-Step SpMV algorithm.

The rest of the paper is organized as following. First we will provide an overview of Two-Step SpMV algorithm. Next the proposed implementation process will be detailed. Three additional optimization methods to reduce traffic and improve performance will be demonstrated afterwards. Later, an on-chip memory comparison against current solutions will be presented. Lastly, experimental results comparing with various benchmarks will be presented followed by conclusion.

2 TWO-STEP SPMV ALGORITHM

As the name suggests, Two-Step SpMV is conducted in two separate steps, which is depicted in Figure 3. In this work, we assume the Disk Access Machine (DAM) model [2] with two levels of memory hierarchy, on-chip storage (fast access) and off-chip main memory (slow access with block transfer). Two-Step SpMV fundamentally depends on matrix partitioning into 1D column-blocks and multi-way merge operation. Initially the source vector x is divided into several segments and matrix A is partitioned into several vertical stripes, i.e. column blocks. The column blocks of A are stored in a row major sparse format, e.g. Row Major Coordinate (RM-COO), Compressed Sparse Row (CSR) [8].

Step 1. In the first step, partial SpMV between a source vector segment (x^k) and the corresponding matrix stripe (A^k) is conducted.

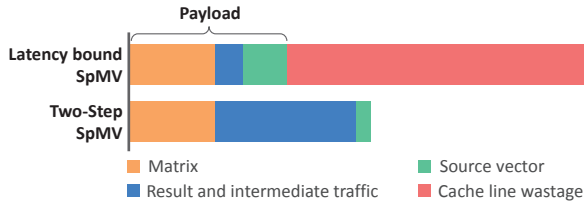


Figure 4: Off-chip traffic for latency bound vs Two-Step SpMV conducted on 1B node graph with average degree 3.

At first, x^k is streamed to the on-chip fast memory from DRAM. Then, A^k is streamed from the main memory to the computation core that results is an intermediate sparse vector v^k. A^k is stored in row major format so that the edges are sequentially traversed in increasing order of row indices. Thus v^k is generated sequentially and streamed back to DRAM. This partial SpMV is conducted for $k = 0, 1, .., n-1$ and there are n intermediate sparse vectors stored in DRAM at the end of step 1.

Step 2. In the second step, all intermediate sparse vectors (v_ks) are streamed back from main memory to the computation core where a multi-way merge operation is conducted to accumulate them into the resultant dense vector y. As these intermediate vectors are sorted in ascending order of their nonzero elements' indices (keys), they are accessed sequentially during multi-way merge operation. Resultant vector y is also generated sequentially and streamed back to DRAM.

2.1 Trade-offs and Advantages

The key insight of Two-Step SpMV is to trade-off random access latency (due to cache miss and DRAM page miss) for more sequential access and more compute. Two-Step algorithm requires the intermediate vectors to be stored in DRAM and causes round trip of these vectors to & from the DRAM. These additional DRAM streaming/sequential accesses and related compute are the overhead of our proposed Two-Step algorithm. One of the fundamental ideas of this work is to eliminate high latency random memory accesses at the cost of more streaming access and more compute. This trade-off is beneficial as more streaming accesses do not necessarily translate to more DRAM traffic. Due to the elimination of cache line wastage (bytes fetched but unused), Two-Step actually incurs less off-chip traffic overall for very large and highly sparse problems. Moreover, DRAM traffic for Two-Step algorithm is transferred at streaming access bandwidth, which is much higher than random access bandwidth. These are the key reasons for the high performance and energy efficiency two-step algorithm, as reported in this work, despite the overhead of more DRAM streaming access.

Figure 4 captures the key insight of this work where the total off-chip traffic for latency bound and Two-Step SpMV are depicted for an example problem. Algorithmically, latency bound SpMV requires least number of total memory accesses. It is named 'latency bound' because the predominant stall reason of this algorithm is due to long latency fetches from main memory resulting from cache miss and DRAM page miss of random accesses. As shown in Figure 4, the overall off-chip traffic is less for Two-Step SpMV despite incurring more payload, i.e. data that takes part in actual computation. As mentioned before, this is because Two-Step eliminates the wastage of cache-level block transfered from DRAM for almost every access to x or y. More importantly, Two-Step SpMV only requires sequential access to DRAM that can be potentially leveraged to fully utilize off-chip bandwidth and completely amortize DRAM row buffer opening cost. Two-Step SpMV also incurs less off-chip than other streaming algorithms for large problems. Additionally, Two-Step SpMV does not require matrix preconditioning and is independent of nonzero locality pattern. Detailed analysis of Two-Step SpMV algorithm is available in [29].

2.2 Challenges

The main challenge of Two-Step algorithm implementation is the multi-way merge operation required in the 2^{nd} step, which is essentially compute-bound [27, 31] and difficult to accomplish with both COTS and custom architectures due to bad scaling behavior. To saturate DRAM streaming bandwidth the multi-way merge implementation needs to have high throughput and scalable parallelization scheme, which is absent in current literature. For example, custom architecture based multi-way merge implementation [13, 18, 21, 22, 27, 30, 32–34] has reported to achieve maximum throughput of 3-10GB/s whereas streaming bandwidth of 3D stacked HBM is in the order of 250-1000GB/s.

Furthermore, thousands of sorted lists with millions of elements have to be merged for large problem. Multi-way merge network of this scale is very resource intensive and grows exponentially with increase in problem size. More importantly, traditional parallelization methods for increasing throughput become unscalable as on-chip memory requirement grows linearly with more number of multi-way merge cores. Difficulties in implementation of multi-way merge operation is one of the key reasons for discarding algorithms similar to Two-Step despite having efficient memory access behavior. One of the key contributions of this work is the development of multi-way merge hardware that can be parallelized to saturate extreme streaming bandwidth without incurring significant increase in on-chip memory requirement.

3 PROPOSED ARCHITECTURE

The proposed architecture for efficient Two-Step SpMV implementation is depicted in Figure 1. As main memory we use multiple HBMs[5]. This state of the art 3D stacked memories can provide extreme bandwidth (in the order of TB with multiple stacks). The entire system sits on a passive interposer to provide wide high speed channel between main memory and computation core. As scratchpad, the random access storage needed for Two-Step algorithm, we use eDRAM due to its high density and low leakage compared to SRAM. Furthermore, eDRAM uses many more banks and small page size that allow low-power operation at modest area penalty [24] and can provide high random access bandwidth [17]. The vector segment width is completely dictated by this available on-chip scratchpad.

To utilize the extreme off-chip bandwidth in an energy efficient way, we propose custom hardware core for computation. In this work we will demonstrate an ASIC chip fabricated in 16nm FinFET, as shown in Figure 2 with specifications, to serve as the computational core of Two-Step SpMV. We also have ported the ASIC accelerator core design to FPGA platform and will demonstrate Intel® Stratix® 10 [1] as the computation core of the proposed accelerator. The design of custom hardware core for step 1 and 2 of Two-Step SpMV is explained in the following subsections.

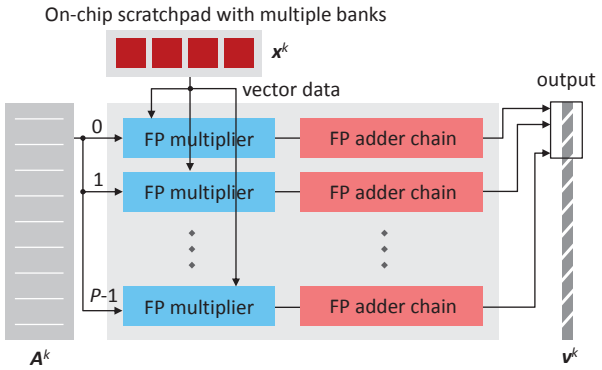


Figure 5: Implementation of step 1 of Two-Step algorithm.

3.1 Implementation of Step 1

The computational hardware to conduct the partial SpMV between A^k and x^k in step 1 is relatively straight-forward than step 2. As shown in Figure 5, this comprises of multiple sets of Floating Point (FP) multiplier and FP adder chain connected in series. The vector segment x_k is streamed from DRAM and stored in the on-chip fast scratchpad memory that consists of multiple banks. Afterwards, matrix stripe is streamed from DRAM for computation. The P sets of FP multiplier and adder chain parallelly work on separate rows of A^k. As the data in matrix stripe is sparse and the scratchpad is separated into several banks, P independent random accesses does not cause significant bank conflict that may introduce stalls in the computation pipeline [28]. We denote output of the multiplier as a record, which is a key-value pair. Here, key is the row index *row* and value is the multiplier output *val*.

The matrix column blocks or stripes are required to be stored in a row major sparse format. However, it is important to note that for large matrices with high sparsity, the matrix stripes might become hyper sparse. A matrix is considered hypersparse if $nnz < N$ [8], 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 array ØN 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 Ønnz, which is more efficient for hypersparsity. A detailed description of this step 1 implementation scheme and an alternative approach can be found in literature [28]. These details are avoided in this work as the methodologies are already available and practiced by researchers.

3.2 Implementation of Step 2

The most critical part of Two-Step SpMV is implementation of step 2 as the multi-way merge network required for it needs to process a large number ($\sim$ multiple thousands) of long sorted lists ($\sim$ hundreds of million or billion elements) at high enough throughput to saturate streaming bandwidth. In this work we implement a binary tree based multi-way merge network, denoted as Merge Core (MC), as depicted in Figure 6. Generally, multi-way merge binary tree uses register based FIFOs in every pipeline state. However, when number of lists, i.e. intermediate vectors (v_ks), grows with larger problems, the storage overhead of FIFOs becomes impractical. Therefore, we use custom sized SRAM blocks in pipeline stages of the tree. A set of consecutive words in the SRAM block logically works as

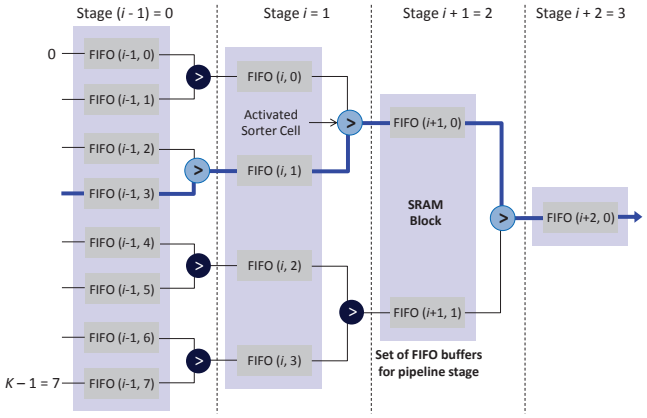


Figure 6: Block memory and binary tree based pipelined K-way MC with single record output per cycle. Highlighted blue line represents activated path in a given clock cycle.

FIFOs and such packed memory based implementation significantly improves scalability of a single MC as the number of lists grows with increasing problem size. Circuit level details of this MC is available in [28] and skipped here for being out of scope.

The output bandwidth of a single MC is inadequate despite running at high frequency using latest technology node. For example, in our 16-nm ASIC implementation a single 2048-way MC saturates 28GB/s bandwidth whereas the HBM based main memory subsystem provides 512GB/s. Hence, approximately an order of magnitude improvement in the multi-way merge throughput is required to utilize full DRAM bandwidth, which can be achieved by parallel multi-way merge. In the next section, general and our proposed method, namely Parallelization by Radix Pre-sorter (PRaP), for parallel multi-way merge implementation for SpMV are discussed.

4 PARALLEL MULTI-WAY MERGE

4.1 Parallelization by Partitioning

A natural way of parallelization is to 2D block the matrix, as depicted in Figure 7, which will eventually generate segmented intermediate vectors. The segmented intermediate vectors can be considered as horizontally partitioned input lists for the MCs. We assume that there are m such partitions. Hence, m MCs are deployed that independently merge the lists in a particular partition and ultimately a single segment of the resultant vector is produced by each MC. Thus a throughput of m records per cycle instead of one can be achieved. 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, as in our case, this method becomes unscalable due to the reasons explained below.

During multi-way merge operation, the records in any particular list are accessed sequentially. However, in every cycle only one list dequeues a record and the selection of that list is practically random. For large problems these lists will reside in the off-chip main memory and random accesses will cause poor utilization of off-chip bandwidth. One practical way to ensure full utilization of the off-chip streaming bandwidth is to prefetch DRAM page (row buffer) size data block (d^{page}) whenever a list is accessed off-chip and store the block in on-chip memory, namely prefetch buffer, for a guaranteed

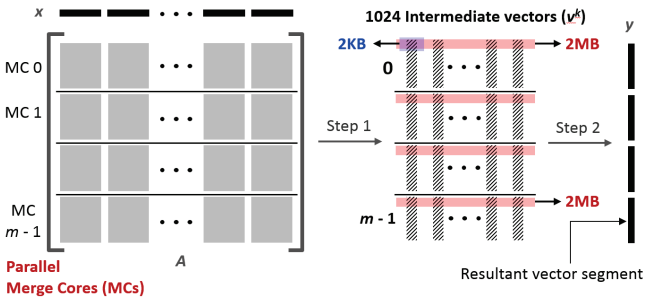


Figure 7: Multi-way merge parallelization by partitioning input lists for Two-Step SpMV. This method becomes unscalable when the problem is larger than on-chip memory.

later reuse. For K input lists K such prefetch buffers are required. For example, as shown in Figure 7, if d^{page} = 2KB page size data block for every list is prefetched, overall 2MB on-chip memory for all the lists in a single partition is required for a 1024-way MC. If there are 16 partitions, we require $m \times K \times d^{page}$ = 32MB on-chip memory just for the prefetch buffers itself. This is significantly large amount to allocate for prefetch buffer as most of the on-chip memory should be dedicated to store segment of x. It should be noted 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 as it strictly depends on limited on-chip memory.

4.2 Parallelization by Radix Pre-sorter (PRaP)

From the discussion above it is apparent that a parallelization scheme that doesn't require increasing prefetch buffer with more parallel MCs is needed. In this work, we propose PRaP as a solution to this problem, which is depicted in Figure 8. The idea is to implement p independent MCs 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 MC. We define q as the number of Least Significant Bits (LSBs) from the key of a record that is used as the radix for pre-sorting as shown in Figure 9. Hence, the number of MCs is $p = 2^q$ and, thus, a multi-way merge network with total output width of p records per cycle 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 previously. This is because all parallel MCs are fed data from the same prefetch buffer. Since p can be incremented without requiring more on-chip storage, PRaP is significantly scalable and effective in handling large problems. 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 later part of this section.

4.2.1 Radix Pre-Sorter Implementation. Without any loss of generality, we assume that the DRAM interface width is of p records. Hence, whenever the i^{th} list, li, is streamed from DRAM, records ri, j to $ri, j + p$ is received 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 10. The pre-soter is implemented using a Bitonic sorting network [3] as p output per cycle is

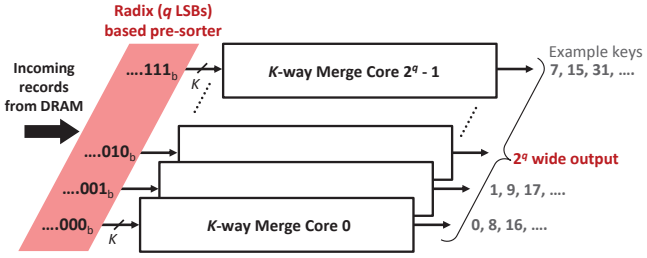


Figure 8: Parallelization by Radix Pre-sorter (PRaP).

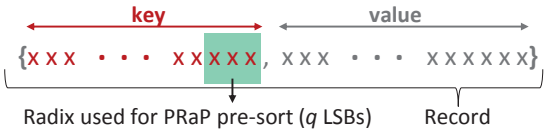


Figure 9: Radix selection for pre-sort in PRaP.

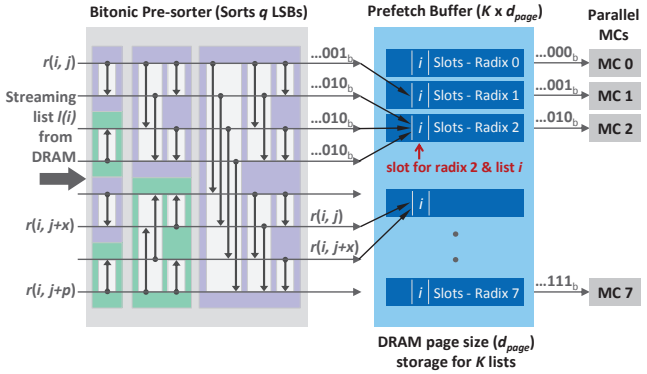


Figure 10: Radix pre-sorter implementation using Bitonic sorter and prefetch buffer. For explanation, we assume that ri, j and $ri, j + x$ have the same radix.

required to match the input rate. In Figure 10, we have depicted the Bitonic network in simplistic manner. The horizontal lines show the data path of the records. 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. 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 10, if ri, j and $ri, j + x$ both have the same radix bits then ri, j should precede $ri, j + x$. This is imperative because for any given MC 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 ri, j, $radi, j$, is 100_b then record ri, j is stored in the page buffer only for consumption by MC 4.

4.2.2 Load Balancing and Synchronization. It is possible for the incoming lists to have keys that are imbalanced in terms of the

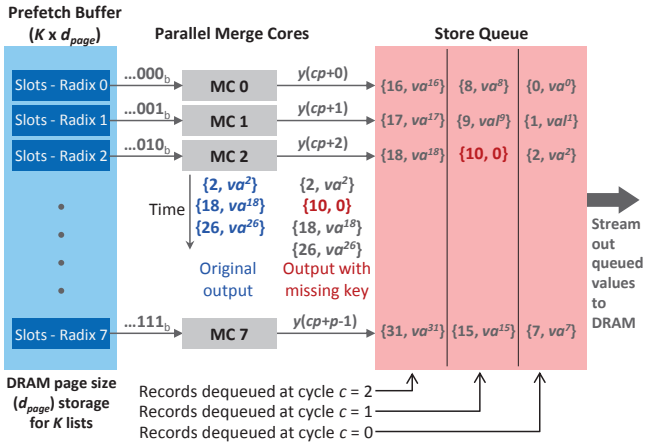


Figure 11: Load balancing and synchronization by insertion of missing keys in PRaP when output is dense.

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 the above issues can be 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, it 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 11, 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, $ycp+0$ to $ycp+p-1$, can be independently queued in a store queue and synchronously streamed out (dequeued) to DRAM. The records $ycp+0$ to $ycp+p-1$ 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 in our fabricated

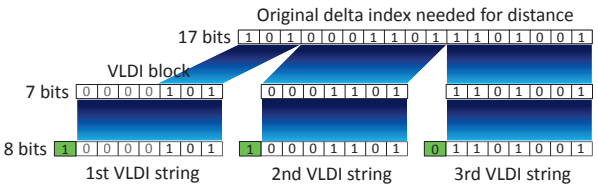


Figure 12: Construction of VLDI strings from delta index

ASIC to saturate the extreme HBM bandwidth that is in the order of hundreds of GBs.

5 ADDITIONAL OPTIMIZATIONS

5.1 Meta-Data Compression

In Two-Step SpMV, the round trip of intermediate vectors (v^ks) in sparse format to/from DRAM incurs off-chip traffic overhead. To reduce this overhead, only the distance between two consecutive elements can be stored instead of the absolute index position in the intermediate vector. However, as the distances can vary within a wide range, we propose a scheme, namely Variable Length Delta Index (VLDI), that practically enables allocation of variable bit width for meta-data.

This process is explained in Figure 12 using an example. The original delta index of a nonzero requires 17 bits to express the distance from its previous nonzero. First, the original delta index is divided 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 12. Afterwards, each block is appended with 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. It should be noted that VLDI is only feasible for sequential generation and access of a stream of elements, which is guaranteed for intermediate vectors by Two-Step SpMV. As the column indices of the matrix stripes are sequential and only read from DRAM by streaming access, it is also possible to apply VLDI to compress the matrix.

5.1.1 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 (A^k) and sparse vectors(v^k) that indirectly depends on the on-chip memory size. With smaller on-chip memory the matrix stripe becomes narrow, due to smaller x^k, and renders more distance among nonzeros of A^k and v^k on average. Hence, a larger fixed length for the VLDI block is more efficient as it reduces the one bit overhead of each string. However, making the VLDI block too wide will cause wastage. Therefore, proper VLDI string length is important given the platform's on-chip fast memory size and sparsity of the problem to achieve efficient compression.

For example, Figure 13 shows probability distribution of delta index width for two different on-chip memory sizes, 5MB and 35MB.The total off-chip traffic for both on-chip storage sizes for a range of VLDI string lengths are computed. 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, the hardware design parameters for a given memory resource and problem characteristics can be tuned.

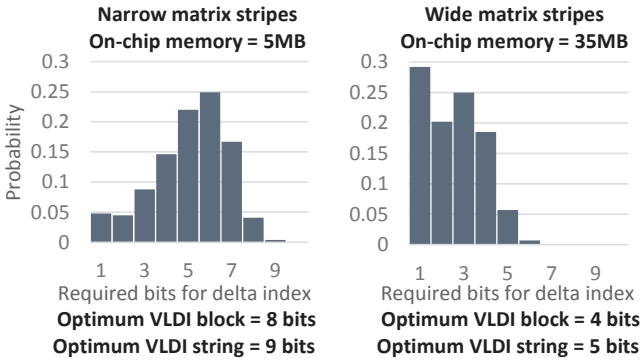


Figure 13: Probability distribution of delta index widths for two different on-chip memory sizes and a randomly generated Erdos Rényi 80M×80M graph with avg. degree 3.

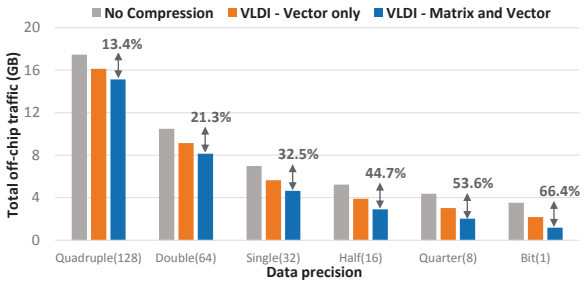


Figure 14: Off-chip traffic reduction using VLDI.

5.1.2 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 14. Here we have separately shown the compression capability when only v^k (intermediate sparse vector) is compressed and when both A^k (matrix stripe) and 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.

5.2 Iteration Overlap Optimization

Many applications, e.g. PageRank, use SpMV kernel iteratively where the resultant vector y of one iteration serves as the source vector x in the following iteration. By overlapping the computations of Two-Step SpMV in consecutive iterations, as depicted in Figure 15, we can decrease off-chip traffic and significantly improve computation throughput. We name this optimization as Iteration overlapped Two-Step (ITS).

As shown in Figure 15, ITS parallelly conducts step 2 of an iteration along with step 1 of the following one instead of running the iterations completely independently. Thus, off-chip traffic due to the round trip of $y_i = x_{i+1}$ to/from DRAM at the transition of iterations is effectively eliminated. The main enabler of this optimization by iteration overlap is the fact that resultant vector of iteration i, x_{i+1}, is generated sequentially in Two-Step algorithm. Despite not

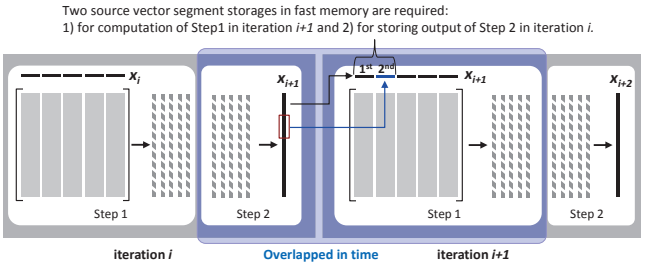


Figure 15: Off-chip traffic optimized using Iteration overlapped Two-Step (ITS) SpMV.

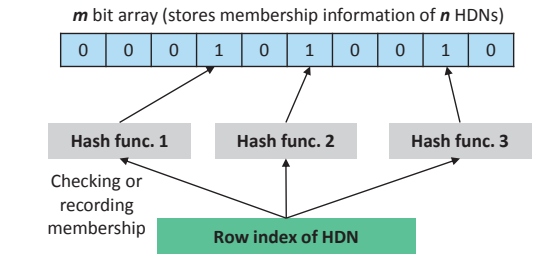
being able to store the entirety of y_i on chip, it is possible to store a segment similar to the source vector x_i. Once a segment of $y_i = x_{i+1}$ is completely generated and stored in the on-chip memory, it is possible to initiate computation of step 1 in the next iteration $i+1$. While step 1 of iteration $i+1$ is conducted using the 1^{st} segment of the source vector x_{i+1}, step 2 of iteration i continues concurrently and stores the 2^{nd} segment of the resultant vector x_{i+1} in another on-chip buffer. Thus, computations of step 2 in iteration i and step 1 in iteration $i+1$ are overlapped in time.

Besides reducing off-chip traffic, a subtle but more important advantage of ITS is that it keeps both the computation resources active for step 1 and 2. This significantly improves overall SpMV throughput and helps in saturating streaming bandwidth. The cost of ITS is that it requires 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 ITS can handle is roughly half of what general Two-Step can process. Therefore, ITS offers a trade off between maximum problem dimension vs performance and energy efficiency.

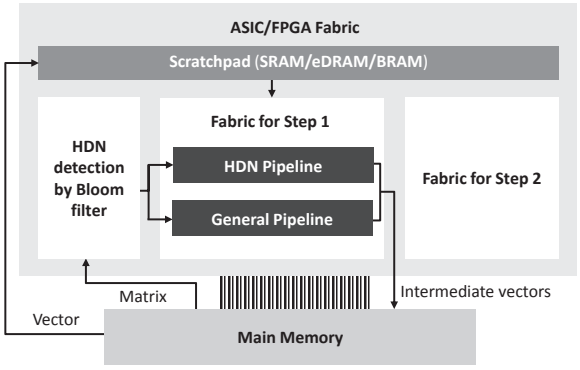
5.3 Optimization for Power-law Graphs

Power-law graphs, commonly found in social networks, possess node distribution that varies with degree in an inverse power relationship. Hence, there are a number of nodes, denoted as High Degree Nodes (HDNs), that have disproportionately large number of neighbors. As HDNs incur numerous collisions during accumulation in step 1 of Two-Step SpMV, it is more efficient to use a separate pipeline with specially tuned accumulator for HDNs. However, detection of HDN efficiently during computation presents a challenge.

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, this requires change in widely used standard matrix formats for one specific task only. On the other hand, prior HDN information is too big to store on-chip for quick access. To resolve this, we propose a Bloom Filter based detection scheme. Bloom Filter [6, 7] is a compact data structure that enables membership check for large data sets. As shown in Figure 16a, Bloom Filter is a bit array that encodes membership information of an element in a set through a 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. If all of the bits are found to be '1's, the element is deemed to be a member of the set.



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



(b) Separate pipeline for HDN in step 1 computation.

Figure 16: Bloom Filter filter based method to efficiently process HDNs without significant on-chip memory overhead.

The idea of using Bloom Filter for power-law graphs is that all the HDNs can be considered as a set. By streaming the metadata 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, proper computation pipeline for partial SpMV during step 1 is selected. A block diagram of this scheme is shown in Figure 16b. 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.

5.3.1 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 qm is named as the load factor. We denote g as the number of hash functions. Given these parameters, the probability of treating a non-member as member can be given as the following [25].

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

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 work, we have implemented one memory access method proposed in [25]. 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 [19] graph collection. 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. We consider any node with more than thousand neighbors as HDN. There are less than 0.1% such nodes. However, to be conservative 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 in [25], 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 $m = q0.1 = 1$Mbits = 128KB, which is an insignificant on-chip overhead. Additionally, using a SRAM block with word width of $w = 64$ bits and $d = 16384$ words, the total number of hash bits required is only $\log_2 16384 + 3\log_2 64 = 14 + 18 = 32$ 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 reasonable and does not require significant resources.

6 ON-CHIP MEMORY REQUIREMENT

The on-chip memory requirement and the largest problem reported for a number of custom hardware and COTS solutions of current literature are given in Table 1. We compare these shared memory solutions against our proposed ASIC without and with optimization by iteration overlap, which are denoted as TS and ITS accordingly. Our developed ASIC requires 10.5MB eDRAM scratchpad as fast memory for vector storage (8MB) and prefetch buffer (2.5MB) along with 0.5MB SRAM for computation core, thus 11MB fast storage in total. As mentioned before, ITS causes the maximum problem size to be half of what is possible with TS. Nevertheless, 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 [14] can only handle 8 million nodes without slicing and partitioning larger graphs whereas the proposed solution can handle multiple billion nodes with significantly smaller on-chip storage.

Table 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 [14]	32.0	8.0
CPU (single socket) [38]	20.0	95.0
CPU (dual socket) [20]	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 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.

7 EXPERIMENTAL RESULTS

The proposed SpMV accelerator is implemented in multiple design points in 16-nm FinFET ASIC (Figure 2) and Stratix® 10 FPGA platforms. Table 2 lists the implementations for various design points including the maximum problem dimension and maximum computation throughout of each implementation. The prefixes, namely TS, ITS and ITS_VC, indicate implementation using straight forward Two-Step, Iteration overlapped Two-Step and Iteration overlapped Two-Step with VLDI vector compression respectively.

Table 2: Maximum graph dimension and throughput for different design points 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

7.1 ASIC based Implementation

For the ASIC, only the computation logic of the entire accelerator system is fabricated. We implemented sixteen 2048-way MCs for this ASIC. The two other basic parts of ASIC based accelerator, i.e. HBM main memory and the eDRAM scratchpad, are emulated using Cacti [9] and Destiny [23] tools. *TS_ASIC* only stores one source vector segment in fast memory, the maximum matrix dimension it can handle is 4 billion. This is two times larger than 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 *TS_ASIC*. *ITS_VC_ASIC* has relatively slower sustained throughput than *ITS_ASIC* in terms of DRAM bandwidth saturation as VLDI decreases the traffic while throughput of these implementations are the same.

7.2 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 (1SG280HU1F50E1VGS3), namely *FPGA1* and *FPGA2* as shown in Table 2. *FPGA1* handles relatively larger problems than *FPGA2*, but at the cost of less computational throughput. *FPGA1* utilizes available hardware resources to implement MCs with more ways (64-way), i.e. more inputs, than *FPGA2* (32-way). More number of ways in MCs enables *FPGA1* to handle larger problems. However, *FPGA1* has less parallel MCs than *FPGA2*. On the other hand, *FPGA2* implements MCs with less number of ways while having more parallel cores. With more MCs *FPGA2* can deliver higher throughput than *FPGA1*. For both *FPGA1* and *FPGA2*, scratchpad memory is synthesized using BRAM. The HBM main memory system is simulated in the same way as ASIC assuming four channels.

7.3 Performance and Efficiency Comparison against Existing Solutions

We have compared the performance and energy efficiency of our developed accelerator against a number of exiting custom hardware, GPU, CPU and co-processor based solutions. A short description of used custom hardware and GPU benchmarks along with references are given in Table 3. For ease of description, we have assigned an ID with each benchmark. For comparison with CPU and many-core co-processor we have used Intel® Math Kernel Library (MKL) routine 'mkl_scoogemv' on dual socket Xeon E5-2620 (22nm, 12 threads) CPU and Xeon Phi 5110P (22nm, 60 cores) co-processor. Both of these architectures have 30MB last level cache (LLC). The peak bandwidth is 102GB/s for the CPU and 352GB/s for the co-processor.

Table 3: Custom hardware and GPU based benchmarks.

Architecture	ID	Description
Custom Hardware	BM1_ASIC	28-nm ASIC, 64 MB eDRAM scratchpad[14]
	BM1_FPGA	Virtex , 25 Mb BRAM & 90 Mb UltraRAM[37]
	BM2_FPGA	Virtex-7, 67 Mb BRAM[36]
GPU	BM1_GPU	8 nodes, Tesla M2050 (16GB GDDR5)[26]

7.3.1 Data Sets for Comparison. The graph data sets used for comparison against custom hardware and GPU solutions are given in Table 4 and Table 5 accordingly. We have considered all relatively large graphs, results of which are reported by the related work. It is also noteworthy that most of the reported graphs by the 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 2.

Data used for comparison with CPU and co-processor are listed in Table 6. All these graphs (except last six) are collected from University of Floria sparse matrix collection [10]. We also have used a number of random Erdos Rényi [12] graphs for the demonstration purpose of our proposed accelerator's capability in handling large problems. These synthetically generated graphs have names with prefix 'Sy'.

7.3.2 Performance against Custom Solutions. Figure 17 presents performance comparisons in Giga Traversed Edges Per Second (GTEPS) of our ASIC implementations against custom benchmarks. The comparison results for different benchmarks are separated.

Table 4: Graphs for comparison against custom benchmarks.

ID	Description	# Nodes (M)	Avg. Degree	# Edges (M)
FR	Flickr [14]	0.82	12.00	9.84
FB	Facebook [14]	2.93	14.31	41.92
Wiki	Wikipedia [14]	3.56	23.81	84.75
RMAT	RMATScale23 [14]	8.38	16.02	134.22
LJ	LiveJournal [37]	7.80	14.38	69.00
WK	Wikipedia [37]	2.40	2.08	5.00
TW	Twitter [37]	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 5: Graphs for comparison against GPU benchmark.

ID	Description	# Nodes (M)	Avg. Degree	# Edges (M)
ara-05	arabic-2005 [26]	22.70	28.19	640.00
it-04	it-2004 [26]	41.30	27.85	1150.10
sk-05	sk-2005 [26]	50.60	38.53	1949.40

Table 6: Graphs for comparison with CPU and co-processor.

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	24.00
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

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. It should be noted that all graphs reported by these benchmarks are much smaller than what our proposed accelerator can handle, as detailed in Sec. 6.

Apart from the 3D DRAM bandwidth and ASIC technology, the achieved speed-up can be contributed to mainly two factors. Firstly, the proposed architecture does not incur any cache line wastage due to the adoption of Two-Step algorithm. Secondly, main memory bandwidth saturation is near peak sustained bandwidth for the proposed architecture due to full DRAM streaming access. Thus, the improvement in bandwidth utilization significantly surpasses the overhead of Two-Step algorithm, i.e. main memory round trip of the intermediate vectors.

Figure 18 shows the same for proposed FPGA implementations. These are expected to achieve less performance than our ASIC implementation. However, the overall speedup against custom benchmarks are significant. Energy efficiency comparison was not possible as the related works didn't report comparable metric on energy.

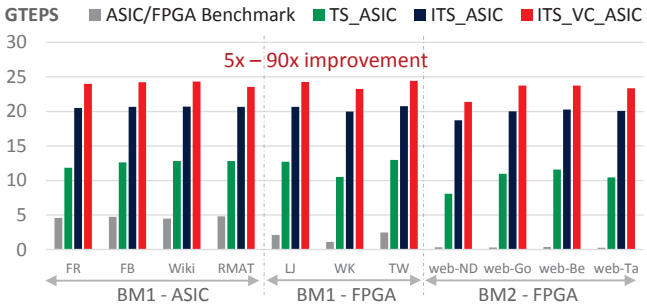


Figure 17: Comparison of GTEPS for proposed ASIC against custom hardware benchmarks.

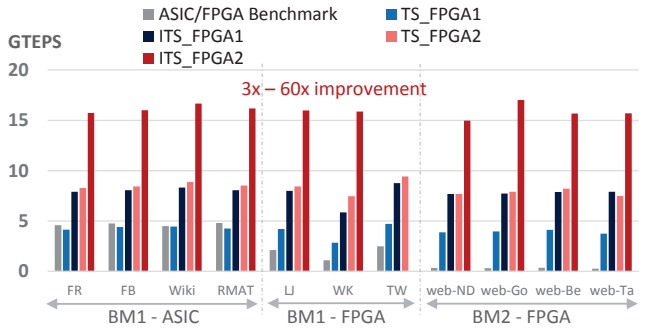


Figure 18: Comparison of GTEPS for proposed FPGA implementations against custom hardware benchmarks.

7.3.3 Performance & Efficiency against GPU. Figure 19 shows the performance and energy efficiency comparison of proposed ASIC against GPU benchmark. Using *ITS_VC_ASIC*, we can achieve up two orders of magnitude improvement in GTEPS. Efficiency improvement in terms of energy per edge traversal are up to three orders of magnitude for almost every graph. This is expected because GPUs commonly consume high energy due to large number of parallel cores and arithmetic units. Figure 20 depicts similar comparison for our FPGA implementations. These plots also show significant improvement in performance and efficiency over GPU solution.

7.4 Comparison against CPU and Co-Processor

Figure 21 demonstrates the speedup and energy consumption of proposed ASIC vs Intel® MKL on CPU and Xeon Phi co-processor. 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

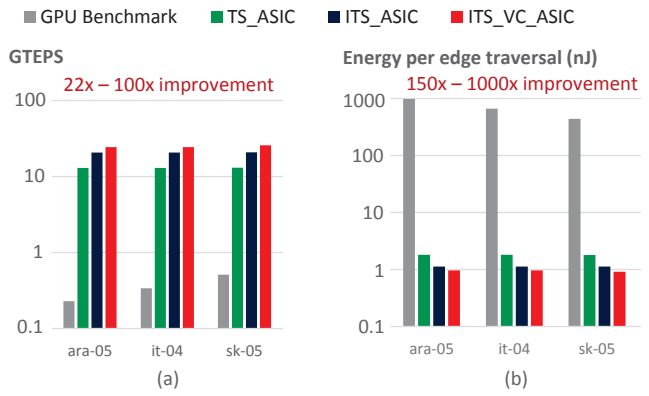


Figure 19: Comparison of a) GTEPS and b) Edge traversal energy for proposed ASIC accelerator with GPU benchmark.

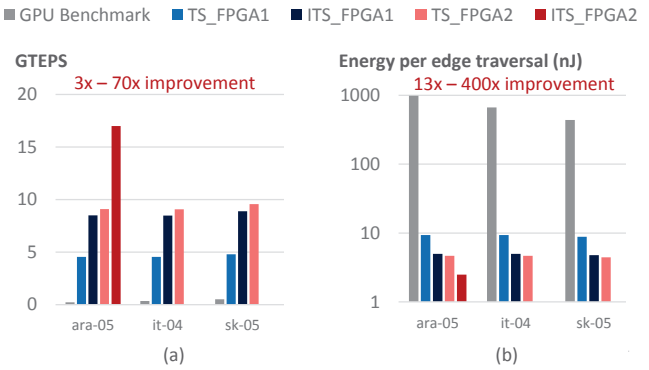


Figure 20: Comparison of a) GTEPS and b) Edge traversal energy for proposed FPGA accelerator with GPU benchmark.

and 30M nodes on Xeon E5 and Xeon Phi respectively. These plots also show the performance of our proposed solutions on very large ($\sim$ billion nodes) graphs. The proposed ASIC achieves up to 800 and 1500 times improvement in GTEPS and energy consumption respectively. Figure 22 depicts similar comparison for proposed FPGA implementations. 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 2. It can be seen that our proposed accelerator on FPGA platform also achieves significant (multiple orders of magnitude) improvements both in terms of performance and efficiency for large graphs.

8 CONCLUSION

Large and highly sparse matrices pose a unique set of challenges for SpMV in terms of scalability, performance and efficiency. This work proposes an custom architecture for implementation of SpMV algorithm that converts random accesses to regular accesses and ensures full main memory streaming. The main contribution of this work is to develop a scalable parallelization of multi-way merge operation to handle large and highly sparse graphs without significantly depending on fast on-chip memory. To the best of our knowledge, such approach in solving SpMV is the first of its kind. As merge-sort and sparse accumulation are fundamental operations in many other applications, this architecture can explored to be utilized beyond SpMV.

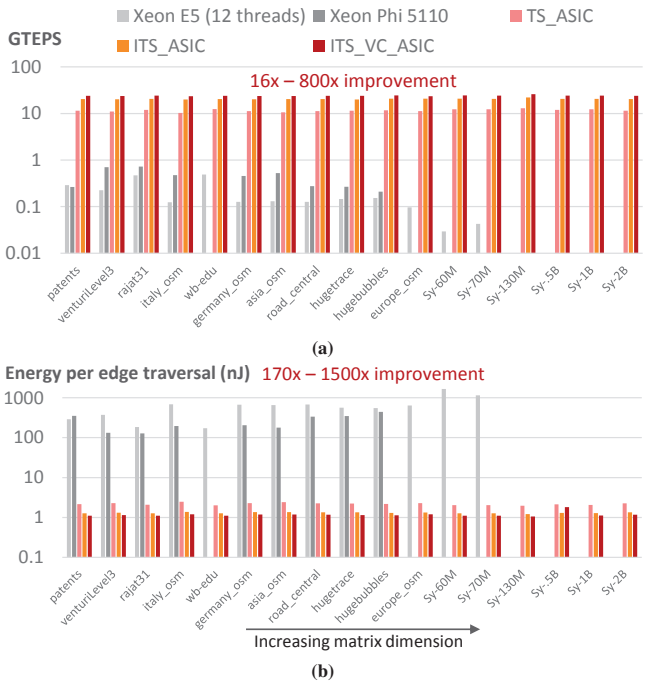


Figure 21: Comparison of a) GTEPS and b) Edge traversal energy for our ASIC accelerator with CPU and co-processor.

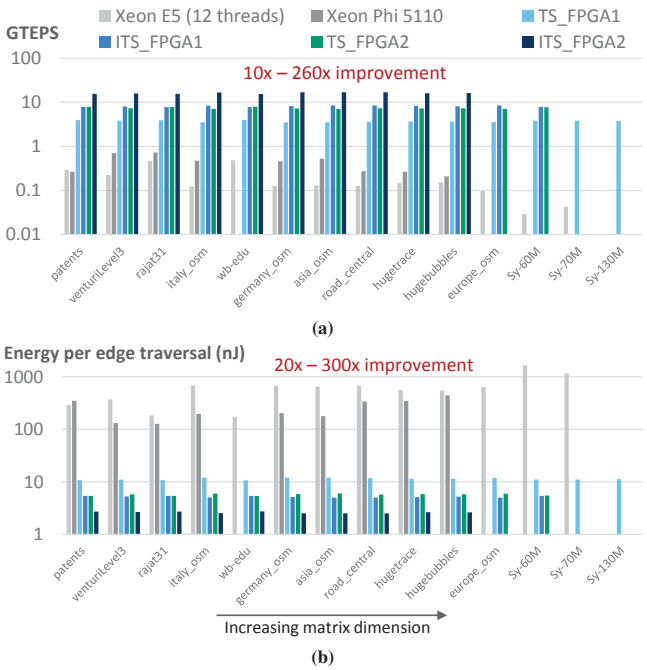


Figure 22: Comparison of a) GTEPS and b) Edge traversal energy for our FPGA accelerator with CPU and co-processor.

ACKNOWLEDGMENTS

This work was supported in part by DARPA contract HR0011-16-C-0038 (CRAFT) and FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM18-0845]. Additional sponsorship was provided by DARPA contract HR0011-13-2-0007 (PERFECT). The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

REFERENCES

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

[2] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (Sept. 1988), 1116–1127. https://doi.org/10.1145/48529.48535

[3] K. E. Batcher. 1968. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference (AFIPS '68 (Spring))*. ACM, New York, NY, USA, 307–314. https://doi.org/10.1145/1468075.1468121

[4] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 18.

[5] Bryan Black. 2014. Die Stacking is Happening in Mainstream Computing. *Additional Conferences (Device Packaging, HiTEC, HiTEN, & CICMT)* 2014, DPC (2014), 001183–001206.

[6] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[7] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[8] A. Buluc and J. R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. https://doi.org/10.1109/IPDPS.2008.4536313

[9] Ke Chen, Sheng Li, N. Muralimanohar, Jung-Ho Ahn, J.B. Brockman, and N.P. Jouppi. 2012. CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory. In *Design, Automation Test in Europe (DATE)*. 33–38.

[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[11] Y. El-Kurdi, W. J. Gross, and D. Giannacopoulos. 2006. 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*. 293–294. https://doi.org/10.1109/FCCM.2006.65

[12] Paul Erdos and Alfréd Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci* 5, 1 (1960), 17–60.

[13] Kermin Fleming, Myron King, Man Cheuk Ng, Asif Khan, and Muralidaran Vijayaraghavan. 2008. High-throughput Pipelined Mergesort. In *Proceedings of the Sixth ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '08)*. IEEE Computer Society, Washington, DC, USA, 155–158. https://doi.org/10.1109/MEMCOD.2008.4547704

[14] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[15] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 37–47.

[16] Nikos Hardavellas. 2012. The rise and fall of dark silicon. *The advanced computing systems association* (2012), 7–17.

[17] M. Jacunski, D. Anand, R. Busch, J. Fifield, M. Lanahan, P. Lane, A. Paparelli, G. Pomichter, D. Pontius, M. Roberge, and S. Sliva. 2010. A 45nm SOI compiled embedded DRAM with random cycle times down to 1.3ns. In *Custom Integrated Circuits Conference (CICC), 2010 IEEE*. 1–4. https://doi.org/10.1109/CICC.2010.5617634

[18] Dirk Koch and Jim Torresen. 2011. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 45–54.

[19] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13 Companion)*. ACM, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173

[20] Kartik Lakhotia, Rajgopal Kannan, and Viktor K. Prasanna. 2017. Accelerating PageRank using Partition-Centric Processing. *CoRR* abs/1709.07122 (2017). arXiv:1709.07122 http://arxiv.org/abs/1709.07122

[21] S. Mashimo, T. V. Chu, and K. Kise. 2017. High-Performance Hardware Merge Sorter. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–8. https://doi.org/10.1109/FCCM.2017.19

[22] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. 2017. Cost-Effective and High-Throughput Merge Network: Architecture for the Fastest FPGA Sorting Accelerator. *ACM SIGARCH Computer Architecture News* 44, 4 (2017), 8–13.

[23] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie. 2015. DESTINY: A tool for modeling emerging 3D NVM and eDRAM caches. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*. 1543–1546.

[24] John Poulton. 1997. An embedded DRAM for CMOS ASICs. In *Advanced Research in VLSI, 1997. Proceedings., Seventeenth Conference on*. IEEE, 288–302.

[25] Y. Qiao, T. Li, and S. Chen. 2011. One memory access bloom filters and their generalization. In *2011 Proceedings IEEE INFOCOM*. 1745–1753. https://doi.org/10.1109/INFCOM.2011.5934972

[26] A. Rungsawang and B. Manaskasemsak. 2012. Fast PageRank Computation on a GPU Cluster. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 450–456. https://doi.org/10.1109/PDP.2012.78

[27] Leszek Rutkowski, Marcin Korytkowski, Rafal Scherer, Ryszard Tadeusiewicz, Lotfi A. Zadeh, and Jacek M. Zurada. 2013. *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.

[28] Fazle Sadi. 2018. *Accelerating Sparse Matrix Kernels with Co-Optimized Architecture*. Ph.D. Dissertation. Carnegie Mellon University.

[29] F. Sadi, L. Fileggi, and F. Franchetti. 2017. Algorithm and hardware co-optimized solution for large SpMV problems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091096

[30] Makoto Saitoh, Elsayed A Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. 2018. A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 197–204.

[31] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang. 2010. FPGA and GPU implementation of large scale SpMV. In *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*. 64–70. https://doi.org/10.1109/SASP.2010.5521144

[32] Wei Song, Dirk Koch, Mikel Luján, and Jim Garside. 2016. Parallel hardware merge sorter. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 95–102.

[33] T. Usui, T. V. Chu, and K. Kise. 2016. A Cost-Effective and Scalable Merge Sorter Tree on FPGAs. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. 47–56. https://doi.org/10.1109/CANDAR.2016.0023

[34] Stephan Wong, Stamatis Vassiliadis, and Jae Young Hur. 2005. Parallel merge sort on a binary tree on-chip network. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*. Citeseer, 365–368.

[35] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos. 2009. FPGA vs. GPU for sparse matrix vector multiply. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. 255–262. https://doi.org/10.1109/FPT.2009.5377620

[36] S. Zhou, C. Chelmis, and V. K. Prasanna. 2015. Optimizing memory performance for FPGA implementation of PageRank. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6. https://doi.org/10.1109/ReConFig.2015.7393332

[37] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. 2018. An FPGA Framework for Edge-centric Graph Processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers (CF '18)*. ACM, New York, NY, USA, 69–77. https://doi.org/10.1145/3203217.3203233

[38] S. Zhou, K. Lakhotia, S. G. Singapura, H. Zeng, R. Kannan, V. K. Prasanna, J. Fox, E. Kim, O. Green, and D. A. Bader. 2017. Design and implementation of parallel PageRank on multicore platforms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2017.8091048

[39] Dan Zou, Yong Dou, Song Guo, and Shice Ni. 2013. High performance sparse matrix-vector multiplication on FPGA. *IEICE Electronics Express* 10, 17 (2013), 20130529–20130529. https://doi.org/10.1587/elex.10.20130529