CoSPARSE: A Software and Hardware Reconfigurable SpMV Framework for Graph Analytics

Siying Feng*, Jiawen Sun†, Subhankar Pal*, Xin He*, Kuba Kaszyk†, Dong-hyeon Park*, Magnus Morton†,
Trevor Mudge*, Murray Cole†, Michael O'Boyle†, Chaitali Chakrabarti‡, Ronald Dreslinski*
*University of Michigan, USA †University of Edinburgh, UK ‡Arizona State University, USA

***Abstract*—Sparse matrix-vector multiplication (SpMV) is a critical building block for iterative graph analytics algorithms. Typically, such algorithms have a varying active vertex set across iterations. This variability has been used to improve performance by either dynamically switching algorithms between iterations (software) or designing custom accelerators (hardware) for graph analytics algorithms. In this work, we propose a novel framework, CoSPARSE, that employs hardware and software reconfiguration as a synergistic solution to accelerate SpMV-based graph analytics algorithms. Building on previously proposed general-purpose reconfigurable hardware, we implement CoSPARSE as a software layer, abstracting the hardware as a specialized SpMV accelerator. CoSPARSE dynamically selects software and hardware configurations for each iteration and achieves a maximum speedup of 2.0× compared to the naïve implementation with no reconfiguration. Across a suite of graph algorithms, CoSPARSE outperforms a state-of-the-art shared memory framework, Ligra, on a Xeon CPU with up to 3.51× better performance and 877× better energy efficiency.**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is an essential linear algebraic operation which has been widely adopted in many irregular workloads, such as machine learning and data mining [6]. Recent studies have shown that large-scale iterative graph analytics can achieve promising performance on a high-performance backend optimized for SpMV [10]. However, guaranteeing high performance consistently across different input graphs, graph algorithms, or algorithm iterations, is challenging. First, real-world graphs have distinct sizes and distributions. The adjacency matrices used to represent graphs have sizes scaling from hundreds to billions and densities ranging from 10^{-7} to 10^{-1} [4], leading to dramatically different memory footprints. Second, the active vertex set, *i.e.* the frontier vector, varies from iteration to iteration, causing highly optimized solutions for certain use cases to encounter significant performance loss for the other cases. Therefore, it is hard to arrive at a "one-size-fits-all" design for the efficient execution of graph algorithms [1].

To adapt to different scenarios, prior work has followed two distinct routes: (i) software-level optimizations, *e.g.* deciding a suitable sparse storage format based on the density and size of the input matrix and vector, selecting either a dense or sparse dataflow [1], [9], [11]–[14], [16], and (ii) hardware-level optimizations that focus on the efficient use of on-chip memory [3], [15]. Merely relying on software optimizations could fail to fully explore on-chip data reuse due to limitations in hardware. On the other hand, hardware-only optimizations are also likely to achieve suboptimal performance for certain graph algorithms and inputs. For example, a hardware accelerator optimized for graph algorithms based on sparse matrix dense vector computations will consume unnecessary compute cycles for those involving sparse vector computations. The ideal design for SpMV-based graph analytics should run the desired algorithm on hardware that is most efficient for the data access pattern based on the input characteristics. This complex and high-dimensional design space therefore calls for a reconfigurable SpMV framework which provides both flexibility to adapt to different inputs and algorithms and a faithful strategy to speedily traverse the available reconfiguration points to achieve the highest achievable performance.

Our proposed solution, CoSPARSE, explores reconfiguration opportunities in both software and hardware, as shown in Figure 1.

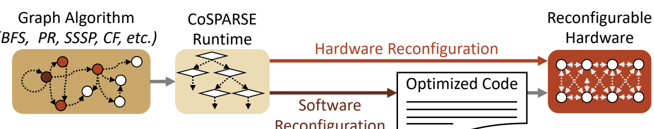

Fig. 1: Overview of the proposed CoSPARSE framework.

In software, it considers two SpMV algorithms based on inner and outer product. The choice of the algorithm directly affects the access pattern of the input/output data, and the load-balancing strategy. In hardware, reconfigurability is manifested in the on-chip memory hierarchy of the underlying hardware, since SpMV is known to be memory-intensive and is bottlenecked by irregular memory accesses. CoSPARSE uses a hardware substrate that supports reconfigurations in both the on-chip memory sharing pattern (shared/private) and on-chip memory type (cache/scratchpad). The software and hardware reconfiguration decisions are made in an integrated dynamic framework, guided by knowledge from extensive experiments and in-depth analysis. In addition, CoSPARSE provides a workload-balancing strategy to harness maximum parallelism for irregular sparse matrices. All of these synergistic benefits are showcased on a suite of common iterative graph analytics algorithms, including Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), PageRank (PR) and Collaborative Filtering (CF), constructed on top of CoSPARSE's SpMV abstraction. Specifically, we make the following contributions:

- On-the-fly, automatic, co-ordinated reconfiguration of the hardware and software based on input data properties, i.e. the dimension/density of matrices and the density of vectors, including:
  - *Software reconfiguration* between inner product and outer product based SpMV implementations, and
  - *Hardware reconfiguration* of the memory subsystem to exploit data-sharing patterns (private/shared) and on-chip memory types (cache/scratchpad).
- Extensive experiments with in-depth analysis to derive the threshold for software and hardware reconfiguration decisions.
- A consistently efficient, high-performance SpMV framework for graph analytics across diverse algorithms and datasets.
- Evaluation of CoSPARSE against competing systems that demonstrates up to 877× better energy efficiency and 3.51× speedup for graph algorithms over Ligra on a Xeon CPU.

II. Background and Related Work

Graph algorithms can be implemented as iterative sparse matrix-vector multiplications (SpMVs) to take advantage of highly optimized SpMV backends [10]. However, the diverse nature of graph processing workloads creates challenges for achieving high performance across a wide range of graph algorithms and inputs.

A. Graph Frameworks using Software Reconfigurations

For graph traversal algorithms such as breadth-first-search (BFS), the size of the active vertex set varies from iteration to iteration [13]. For example, the SSSP algorithm on pokec, a commonly used graph benchmark, shows that during execution the percentage of active vertices increases from <0.1% to 47% and again decreases

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

to <0.1% (Figure 9). To harness this property, switching between dense and sparse representations of the active vertex set and the corresponding dataflows across iterations is widely adopted in recent graph frameworks [1], [9], [11]–[14], [16]. In terms of SpMV, the dense representation is equivalent to the inner product algorithm (IP) and the sparse one corresponds to the outer product algorithm (OP). Graph frameworks usually target existing platforms with no hardware modifications and require user input for accurate reconfiguration. For example, Ligra [9], a lightweight shared-memory based graph framework implementing software reconfiguration, uses an empirical parameter, *i.e.* $|V| = |E|/20$, as the reconfiguration threshold unless set differently, where $|V|$ denotes the active vertex size and $|E|$ is the number of edges. CoSPARSE, instead, automatically analyzes choices at both software and hardware levels within a tightly-coupled framework to achieve best performance at graph iteration granularity.

B. Optimized Hardware Acceleration for Graph Analytics

Many vertex-programming based graph processing accelerators using frontier scheduling have been proposed recently [2]. Graphicionado [3] exploits the on-chip scratchpad memory for random accesses and applies graph slicing to maximize data reuse. TuNao [15] maps the Gather-Apply-Scatter paradigm to ECGRA modules and enhance data reuse by storing high-degree vertices in on-chip buffers. GraphPIM [5] provides efficient processing-in-memory offloading with minor architectural extensions to achieve dramatic memory bandwidth improvement. To obtain the best efficiency with minimum hardware, graph processing accelerators tend to target one dataflow, and often do not consider the characteristics of the input vector. CoSPARSE, instead, is implemented on top of a programmable general-purpose hardware substrate that can be easily extended to support different graph algorithms by providing an SpMV framework abstraction and efficiently executes both IP and OP.

C. Opportunities in Combining Software/Hardware Optimizations

CoSPARSE requires a hardware substrate that is programmable and reconfigurable to orchestrate software and hardware reconfiguration. Recent work has proposed a many-core general-purpose accelerator called Transmuter [7] that supports reconfiguration of the resource sharing pattern (private/shared), and on-chip memory type (cache/scratchpad(SPM)). The architecture features a massive number of lightweight processing elements (PEs) and a reconfigurable memory hierarchy. The PEs are grouped into tiles and are coordinated by a local control processor (LCP). Each PE and LCP are lightweight in-order processors with standard ISA support. The PEs are connected to a two-level memory hierarchy consisting of reconfigurable crossbars (RXBars) and reconfigurable caches (RCaches). Each level of the reconfigurable memory hierarchy (L1/L2) can be configured into shared/private caches/SPMs. The reconfiguration can happen both at compile time or at runtime. The runtime hardware reconfiguration overhead is estimated to be ≤10 clock cycles. We will refer to an $A \times B$ system as a Transmuter design with A tiles and B PEs per tile. The use of programmable cores facilitates dataflow reconfiguration and support for diverse graph algorithms. The hardware reconfigurability of Transmuter also lends a good fit to CoSPARSE, since the hardware is amenable to different data access patterns and flexible in response to properties of the evolving data set.

III. CoSPARSE Reconfiguration Layer Design

Figure 2 gives an overview of the heuristic-driven reconfiguration strategy, which is triggered before each SpMV execution. Based on the density of the input vector, we decide whether to use the IP or OP based SpMV algorithm; this is the software (re)configuration choice. Then, based on the density and size of the matrix and the vector, we decide on the two-level on-chip memory configuration of the hardware; this is the hardware (re)configuration.

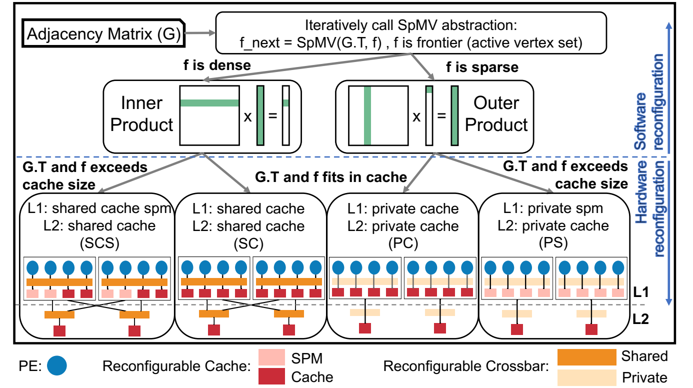

Fig. 2: Structure of CoSPARSE software and hardware reconfiguration framework. For every invocation to CoSPARSE, we select the best software (IP or OP), followed by hardware configurations (SCS or SC for IP, PC or PS for OP), assuming a 2×4 system.

A. Reconfigurable SpMV Implementation

Figure 2 shows the four hardware configurations that we identified to be most suitable for SpMV, *i.e.* SC (L1: shared cache and L2: shared cache) and SCS (L1: shared cache scratchpad and L2: shared cache) for IP and PC (L1: private cache and L2: private cache) and PS (L1: private scratchpad and L2: private cache) for OP.

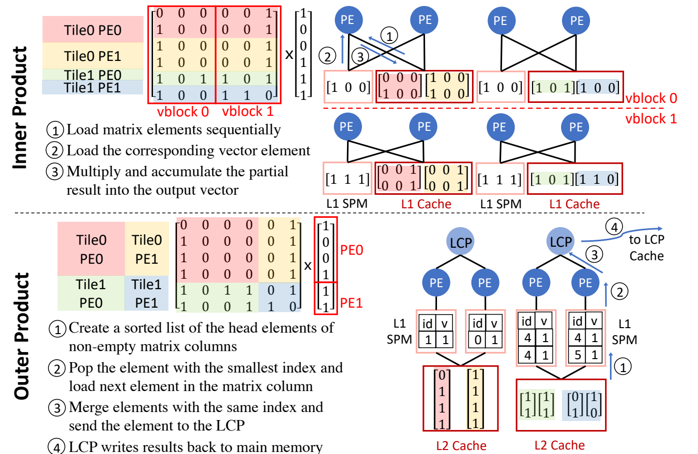

Fig. 3: Matrix partitioning based on non-zero elements and algorithm mapping of IP on SCS and OP on PS that focuses on maximizing data reuse and reducing stalls for random accesses on a 2×2 system.

Inner product (IP) Implementation. To maximize parallelism, the matrix is partitioned into disparate row partitions which are stored in row-major COO format to facilitate spatial locality for accesses. The COO format stores the row index, column index, and the value for each non-zero matrix element. The vector is stored as a dense array. Each tile performs SpMV on one of the matrix row partitions with the vector. Hence, each tile works on different segments of the output vector in parallel without introducing data races, and thus avoids synchronization. In addition, the mapping exploits reuse opportunities of the input vector, which is shared among the tiles and PEs within a tile. Therefore, to maximize data sharing, CoSPARSE selects the SC and SCS modes, which enables the PEs and the tiles to share a large chunk of on-chip memory.

Figure 3(top) illustrates the computation scheme of IP for SCS. The input vector elements are stored in the shared SPM in L1 to curtail the overhead of random accesses to the vector elements due to matrix sparsity. The vector elements in the SPM are shared among all PEs within a tile. For large matrices, the matrix is partitioned vertically to

ensure that the vector segment corresponding to a vertical partition (vblock) fits in the SPM. SC uses the same scheduling except that the vector elements are randomly accessed from the L1 shared caches.

Outer product (OP) Implementation. OP also involves each tile multiplying an exclusive row partition with the vector. The matrix is stored in a column-based sparse format, *i.e.* CSC format, which stores the row index and the value for each non-zero matrix element and an array of pointers to the start row index of each column. The vector is stored in a sparse format, *i.e.* (*index*, *value*) tuples of the vector non-zero elements. The LCP assigns a contiguous chunk of non-zero vector elements to each PE and the PEs perform mergesort with the corresponding matrix columns. Since each PE accesses an exclusive set of columns, there is no data sharing between PEs and between tiles. Therefore, private on-chip memories are used in both L1 and L2 to prevent data thrashing and cache contamination. The PEs can also benefit from higher access bandwidth and shorter latency to L1 since bank conflicts and arbitration are eliminated.

Figure 3(bottom) shows the execution flow of OP in PS. The sorted list maintaining the head elements of the non-empty matrix columns is kept in the private SPM to support fast random accesses from list management. For higher scalability, the sorted list uses a heap structure, *i.e.* a binary tree which guarantees that the parent is smaller than its children. When the sorted list cannot fit in the SPM, it spills over to the shared memory, but the tree nature of heap ensures that the majority of comparisons and swaps still happen in the SPM. The scheduling for PC is the same. However, since PC uses caches in L1 and has no control over the cache replacement policies, the sorted list elements may be evicted to L2 or even the main memory.

B. Workload Balancing Strategies

Many real-world sparse matrices have non-uniform distributions [1], causing imbalanced workload distribution across PEs. To achieve maximum parallelism, both static matrix partitioning (before execution) and dynamic task distribution (during execution) are applied.

Inner product (IP) treats the vector as dense, so the execution time of a PE is highly dependent on the number of non-zero matrix elements assigned to it. Figure 3(top) illustrates the matrix partitioning method used by IP. The sparse matrix is first statically partitioned into row partitions with the same number of non-zero elements. Each PE is assigned one of the row partitions and thus obtains a similar amount of work. The row partitions are further divided into multiple vertical blocks (vblocks) so that the vector elements corresponding to each vblock can fit in the shared SPM. Ideally PEs work on the same vblock at a time so that each tile can fetch the vector elements for the other tiles into L2 caches. The vertical partition is not required for the SC mode but can still be beneficial because of the improved spatial and temporal locality of vector accesses. Since each tile works on disparate row partitions, no synchronization is needed after each PE finishes processing a vblock. Also, since the matrix is sparse, the imbalance in the number of non-zero elements *within* a vblock is not large enough to visibly impact performance. The proposed partitioning scheme is sufficiently lightweight and effective in that it balances the workload by assigning each PE the same number of matrix non-zeros and fully utilizes the underlying hardware by considering the size of the on-chip storage.

Outer product (OP) is different from IP in that the vector density affects the workload (actual number of non-zero elements) assigned to PEs. If the vector used for SpMV remains the same throughout execution, the matrix partitioning can also take into account vector sparsity. However, this is not the case for our target applications, *i.e.* iterative graph algorithms, and thus dynamic workload-balancing is needed. Similar to IP, the matrix is first divided into row partitions with the same number of non-zero elements and assigned to each tile. Within a tile, the LCP distributes the non-zero elements of the vector evenly to each PE, such that the number of columns assigned to each PE, *i.e.* the storage needed for the sorted list, is roughly the same. The combination of static and dynamic workload-balancing provides an effective solution for irregular matrix distribution and works well for applications with evolving vectors, *e.g.* graph algorithms.

C. Reconfiguration Threshold Analysis

The thresholds used at each level of the reconfiguration decision tree are based on extensive experiments and analysis. The methodology for these experiments is detailed later in Section IV-A. From this point, we denote matrix dimension as N and matrix density as r.

1) Software Reconfiguration Threshold: When the vector is sparse, OP tends to outperform IP because it only considers the matrix columns that have corresponding non-zeros in the input vector, and thus significantly reduces the number of matrix elements fetched during computation. However, the overhead of mergesort grows in a super-linear fashion with the number of matrix columns to merge, which grows with increasing vector density and matrix dimension, and causes the benefits of OP to diminish in comparison to IP.

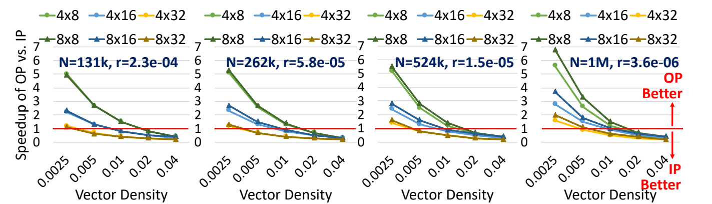

Fig. 4: Speedup of OP (PC) vs. IP (SC). IP performs better for dense vectors and OP performs better for sparse vectors. The crossover vector density decreases when more PEs are present in a tile.

Figure 4 shows the speedup of OP over IP and demonstrates a clear crossover point between the two algorithms for different system sizes and input matrices. We define the *crossover vector density (CVD)* as the density above which the IP algorithm should be used, and below which, the OP algorithm should be used to achieve the best performance. The CVD decreases with an increasing number of PEs per tile because the performance of OP does not scale with the number of PEs as well as it does for IP.

The dimension and density of the matrix also have an impact on the CVD. When the matrix becomes sparser, the total amount of reuse for vector elements becomes smaller for IP, whereas OP is not affected by the matrix sparsity, causing the CVD and the performance benefit of OP to increase slightly.

Takeaways. *There exists a crossover point at which CoSPARSE switches from IP to OP to achieve the best performance as the vector density decreases. The crossover density decreases from ∼2% to ∼0.5% as the number of PEs in a tile increases from 8 to 32.*

2) Hardware Reconfiguration Threshold for Inner Product (IP): The best hardware reconfiguration for IP depends on both the dimension and density of the matrix, as well as the density of the vector. As shown in Figure 5, the performance benefit of the SCS mode is positively correlated to the vector density. In the SC mode, the vector elements are fetched into L1 caches on-demand and could be evicted to L2 caches or even the main memory by the cache replacement policy. The SCS mode stores the vector elements in the L1 SPM to allow fast random accesses. Since SCS eliminates the case where useful vector elements are evicted from L1 and reloaded, SCS encounters a lower number of L2 cache accesses than SC mode and thus fewer memory stalls, especially for high-density vectors.

The matrices evaluated here have the same number of non-zero elements, so the largest matrix is also the sparsest matrix. The performance benefit obtained by SCS is highly dependent on the number of times the vector elements in the SPM are reused

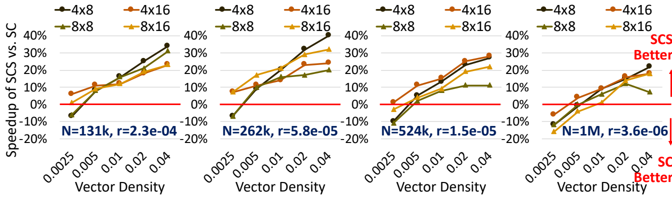

Fig. 5: Speedup of SCS vs. SC for IP. SCS achieves more performance gain for denser vectors or when the reuse of data in SPM is higher.

(N_{reuse}). For uniformly random matrices, N_{reuse} is proportional to the number of non-zero elements in a vblock (Figure 3), i.e. $\frac{N \cdot r \cdot NUM_PES_PER_TILE}{NUM_TILES}$, where N is the matrix dimension and r is the matrix density. Based on the formula, the largest matrix exhibits the least reuse among the four matrices, and thus the least speedups. For the same reason, the performance benefit reduces when the system size changes from 4×8 to 8×8 or from 4×16 to 8×16, since N_{reuse} decreases as the number of tiles increases. When the number of PEs increases, N_{reuse} increases. However, as the SC mode also has a larger cache to fit more vector elements in L1, the performance benefit does not show a clear trend.

Takeways. *The speedup of SCS is positively correlated to vector density as well as the number of times that the vector elements stored in the SPM are reused, i.e. the number of matrix elements corresponding to these vector elements.*

3) Hardware Reconfiguration Threshold for Outer Product (OP): The performance benefit of PS versus PC is reported in Figure 6. As the vector density increases, more matrix columns need to be merged, resulting in an increase in speedup with PS. This is because PS maintains the sorted list of the head elements of the non-empty matrix columns in a heap structure in the SPM, and the majority of random accesses are handled by the SPM. PC, however, does not have control over the locations of the sorted list elements. When the sorted list cannot fit in L1, the list management accesses can span across the memory hierarchy. The situation becomes severe with high vector density since the length of the sorted list grows with the vector density, which is indicated by the lower hit rates of both the L1 and L2 caches. On the other hand, when vector sparsity allows the sorted list to fit in the L1, PC outperforms PS as PC does not have SPM management overhead but has higher access bandwidth to the L2.

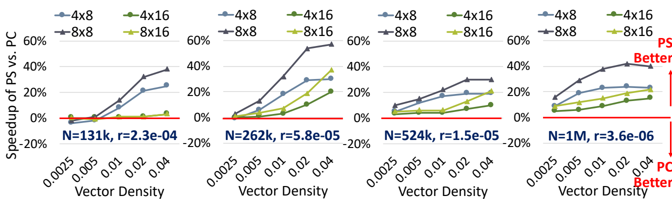

Fig. 6: Speedup of PS over PC for OP. The performance gain of PS grows with increasing vector density, increasing number of tiles, and decreasing number of PEs per tile.

The performance benefit of PS is closely related to the number of columns that need merging, which is determined primarily by the matrix dimension and vector density, and also related to the size of the hardware system. Since the number of PEs and L1 RCache banks are the same, the increased number of PEs in a tile allows PC to have a larger cache to fit the sorted list. As the L1 hit rates increase for PC, the speedup of PS drops when there are more PEs per tile. On the other hand, the performance benefit of PS increases rapidly with the number of tiles. As the number of cores doubles by switching from a 4×8 to an 8×8 system, the PC mode achieves an average speedup of 1.80× and PS mode achieves 1.96×. Increasing the number of tiles keeps the number of matrix columns to merge the same, but reduces the length of the matrix columns, and thus the total number of non-zero elements to merge. In this case, the performance benefit of the PS mode becomes more obvious, because the chances of loading the next elements in the matrix column are reduced, and random accesses to the sorted list become a more significant bottleneck.

Takeways. *PS achieves better performance when there are more columns to merge, or when the length of columns to merge reduces. The speedup of PS decreases for systems with more PEs in a tile.*

D. Graph Analytics Algorithms on CoSPARSE

Hardware-accelerated graph processing solutions often require programmers with in-depth architectural knowledge of the hardware to fully exploit the available performance benefit [3]. Existing graph processing frameworks, on the other hand, enhance user-friendliness by abstracting away scheduling and implementation details, but achieving the best performance still requires expert intervention, *e.g.* to define accurate thresholds [14]. CoSPARSE addresses both performance and programmability with a software and hardware reconfigurable SpMV framework. The software and hardware configurations are automatically determined based on algorithms and input characteristics upon invocation to the decision tree. The runtime hardware reconfigurations are triggered by one of the LCPs and are estimated to take ≤10 cycles. The SpMV scheduling and implementation are embedded in the framework. End users only need to define the key computations to realize a graph algorithm, similar to [9]. Example algorithm implementations are shown below.

1) Graph Analytics Algorithm Mapping: In this work, we implement and evaluate four common graph algorithms which are representative in machine learning and graph traversal, *i.e.* Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), PageRank (PR) and Collaborative Filtering (CF).

To map a graph algorithm to CoSPARSE, two key operations need to be specified. **Matrix_Op** defines the computation between the non-zero elements of the adjacency sparse matrix and the elements of the frontier vector. **Vector_Op** applies computation to the vector elements. Taking SpMV as an example, Matrix_Op denotes the sparse matrix-vector multiplication. Since Matrix_Op already calculates the final result, Vector_Op is not applicable for SpMV. All graph algorithm implementations in CoSPARSE are mapped based on code from the Ligra framework [9]. The definitions of the key operations of the implemented graph algorithms are detailed in Table I.

TABLE I: Definitions of Matrix_Op and Vector_Op of Algorithms mapped to CoSPARSE, where Sp represents the adjacency sparse matrix and V represents the frontier vector.

Algorithm	Matrix_Op(Sp,V)	Vector_Op(V)
SpMV	$\sum Sp_{src,dst} * V_{src}$	N/A
BFS	$\min(V_{src})$	N/A
SSSP	$\min(V_{src} + Sp_{src,dst}, V_{dst})$	N/A
PR	$\sum(V_{src}/\deg(src))$	$\alpha+(1-\alpha)*V_{updated_dst}$
CF	$\sum(Sp_{src,dst} - V_{src} * V_{dst}) * V_{src} - \lambda * V_{dst}$	$\beta * V_{updated_dst} + V_{dst}$

2) Input and Output Conversion Overhead: Throughout the execution of a graph analytics algorithm, the sparse matrix remains constant, but the sparsity of the vector may vary from iteration to iteration. A new output vector is produced and serves as the input vector for the next iteration. To support the IP and OP algorithms and runtime reconfiguration, two copies of the input compressed sparse matrix (in COO and CSC formats, respectively) are stored in main memory to avoid matrix conversion overhead, similar to [9], whereas the lightweight vector conversion between sparse and dense format is performed for the iterations that require reconfiguration. In most of the graph analytics algorithms in our experiments, switching between IP and OP only happens once or twice during execution, *e.g.* for BFS and SSSP, where the vector changes from sparse to dense and then

TABLE II: Microarchitectural parameters of gem5 model.

Module	Microarchitectural Parameters
PE/LCP	1-issue, 4-stage, in-order (`MinorCPU`) core @ 1.0 GHz
RCache (per bank)	4 kB, 1-ported, word-granular CACHE: 4-way set-associative non-coherent cache with 8 MSHRs and 64 B block size, stride prefetcher SPM: physically-addressed, word-granular
RXBar	$N_{\text{src}} \times N_{\text{dst}}$ non-coherent crossbar with 1-cycle response Arbitrate/Shared: 1-cycle arbitration latency, 0 to (N_{src}-1) serialization latency depending upon number of conflicts Transparent/Private: no arbitration, direct access
Main Memory	1 HBM2 stack: 16 64-bit pseudo-channels, each @ 8000 MB/s, 80-150 ns average access latency

back to sparse. The other algorithms, namely PR and CF, always use dense vectors, and thus no vector format conversion is needed.

IV. Evaluation

A. Experimental Setup

CoSPARSE is modeled using the gem5 simulator. The microarchitectural parameters are listed in Table II. The PEs and LCPs are modeled after an in-order ARM Cortex M4F, and the cache and crossbar latencies are based on prior work [7]. For systems larger than 8×16, the simulation resources required become prohibitive and a trace-based simulation model is used [7]. A power model is built based on the static and dynamic power of each component of the system and cross-verified with a fabricated chip prototype [8]. The crossbar and core power models are based on synthesis reports and cache power is calculated by CACTI 7.0.

The SpMV implementation in CoSPARSE is compared against state-of-the-art SpMV implementations on a CPU (Intel i7-6700K) running MKL 2018.3 and a GPU (NVIDIA Tesla V100) running CUSP v0.5.1. The graph algorithm implementations are evaluated against Ligra [9]. To evaluate the performance and efficiency of CoSPARSE, we use a combination of uniformly random matrices, power-law matrices generated by NetworkX, and real-world graphs from SNAP dataset and SuiteSparse Matrix Collection. The details of the real-world graphs are listed in Table III.

TABLE III: Specifications for real-world graphs.

Graphs	# Vertices	# Edges	Type	Kind	Density
livejournal	4,847,571	68,992,772	Social Network	Directed	2.9×10^{-6}
pokec	1,632,803	30,622,564	Social Network	Directed	1.2×10^{-5}
youtube	1,134,890	2,987,624	Social Network	Undirected	2.3×10^{-6}
twitter	81,306	1,768,149	Social Network	Directed	2.7×10^{-4}
vsp	21,996	2,442,056	Random	Undirected	5.0×10^{-3}

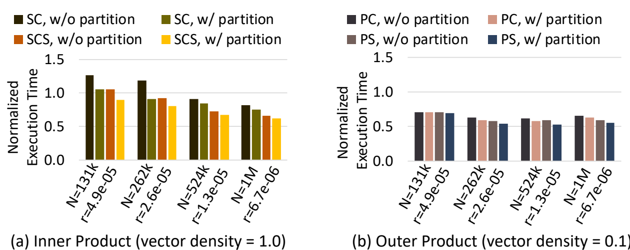

Fig. 7: The SpMV execution time of power-law matrices normalized to uniform matrices on SC (IP) and PC (OP) on an 8×16 system. Workload balancing benefits IP more than OP, especially SC for IP.

B. Workload Balancing Evaluation

The execution time of SpMV for power-law matrices, normalized to that for uniformly random matrices of the same dimension and density on cache-only hardware configurations, is shown in Figure 7. For IP, the workload-balancing technique improves the execution time by 7% to 30% and benefits SC more than SCS. Since SC does not have SPMs, it cannot efficiently handle random accesses. In addition, in SC, the workload imbalance could cause some PEs to finish their assigned work early and remain idle, instead of fetching vector elements that could be reused by other PEs into the shared L1. Therefore, the performance of SC is more sensitive to the irregular matrix distribution, and thus more likely to benefit from the workload-balancing scheme. It is worth noting that in some cases for IP, the execution time of power-law matrices is less than that of uniform matrices. This is because the existence of dense rows/columns in power-law matrices results in fewer non-empty matrix rows/columns. In this case, fewer input vector elements are used for computation and fewer output vector elements are generated, which are more likely to fit in the L1, improving both locality and performance.

As shown in Figure 7-b), for OP, the execution time of power-law matrices is also shorter than that of uniformly random matrices. This is because the irregular distribution of the matrices increases the possibility that the matrix column corresponding to a non-zero vector element has no elements, thus reducing both the number of columns and the number of non-zero elements to merge. The matrix partitioning technique further improves the execution time of both hardware configurations by up to 10%.

C. Comparison against Existing Platforms

The hardware substrate used in CoSPARSE is programmable so as to support easy implementation of SpMV-based applications, such as graph algorithms. Therefore, we evaluate SpMV against CPU and GPU and compare the graph algorithm implementations to Ligra [9]. Accelerators are specifically optimized for certain applications by eliminating extraneous hardware overhead for programmability and flexibility, and thus are not considered for performance and energy efficiency analysis for a fair comparison.

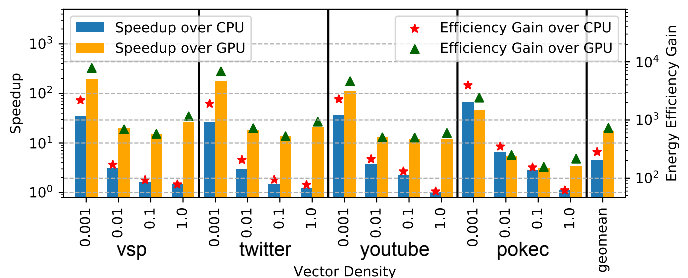

Fig. 8: Speedup and energy efficiency gain of CoSPARSE (16×16) over CPU and GPU. The vector density sweeps from 0.001 to 1.0. CoSPARSE achieves an average speedup (energy efficiency gain) of 4.5×(282.5×) and 17.3×(730.6×) over CPU and GPU, respectively.

1) SpMV: Figure 8 demonstrates the speedup and energy efficiency gain of SpMV, on a suite of real-world graphs, over CPU and GPU implementations. Overall, CoSPARSE achieves an average speedup of 4.5× and 17.3× compared to the CPU and GPU, respectively. Although the GPU has a significantly higher core count and peak memory bandwidth compared to the CPU, the irregular and low-locality memory accesses, coupled with the thread divergence inherent in the SIMT model, bottlenecks the GPU. Memory dependence stalls account for 32% of the GPU stalls (increasing with vector density), and most of the remaining cycles (averaging 35%) are spent in synchronization, instruction fetching, and throttled memory accesses. Despite the high memory bandwidth, the highest average bandwidth utilized by a kernel varies from 12-71%, and the overall performance is <0.006% of the peak performance. The CPU shows better performance than the GPU because the out-of-order cores can hide the overhead of the irregular memory accesses and handle complex execution flow. High power consumption is observed in both the CPU and GPU because of the massive number of threads in the GPU and the high-performance out-of-order cores in the CPU. In contrast, the underlying architecture of CoSPARSE uses lightweight

in-order cores and a flexible memory hierarchy. CoSPARSE improves memory parallelism and locality by determining the best software and hardware configuration, and carefully scheduling and balancing the workload. The average energy efficiency gain over CPU and GPU are 282.5× and 730.6×, respectively.

The performance and energy efficiency gains grow as the vector becomes sparser since CoSPARSE takes advantage of the vector sparsity and skips computation and accesses to the output vector if the vector element is zero. With vector density <0.01, the underlying algorithm switches from IP to OP (except for pokec), and further eliminates accesses to matrix elements that correspond to zero elements in the vector. Since pokec has the largest dimension, it has more columns to merge for the same vector density, and thus, OP only performs better than IP for a vector density of 0.001.

2) Graph Analytics Algorithms: We first conduct a case study illustrating the execution of graph analytics on our CoSPARSE framework. Figure 9 shows the execution time per iteration running SSSP with pokec normalized to IP in the SC mode. From *Iter 4* to *Iter 8*, the IP implementation outperforms OP because of the high vector density (as large as 47% in *Iter 6*). Within these IP iterations, *Iter 6* and *Iter 7* have the highest vector density and achieve the best performance in the SCS mode, whereas *Iters 4, 5,* and 8 favor the SC mode. The rest of the iterations involve vector densities less than 0.5%, and achieve better performance using OP in the PC mode. The synergistic software and hardware reconfiguration amass a net speedup of 1.51×, over the SC-only IP execution, *i.e.* a baseline implementation with no software or hardware reconfiguration. Similar trends are observed with BFS and SSSP for the rest of the graphs. The combined software and hardware reconfiguration achieves a speedup of up to 2.0× across different algorithms and input graphs.

Iteration	Vector Density	Normalized Execution Time					Best Configuration	
		Inner Product		Outer Product				
		SC	SCS	SC	PC	PS	SW	HW
0	<1%	1.0	1.0	<0.1	<0.1*	<0.1	OP	PC
1	<1%	1.0	1.1	<0.1	<0.1*	<0.1	OP	PC
2	<1%	1.0	1.2	0.1	0.1*	0.1	OP	PC
3	<1%	1.0	1.2	0.6	0.5*	0.6	OP	PC
4	1%	1.0*	1.2	7.5	6.7	6.8	IP	SC
5	12%	1.0*	1.1	>10	>10	>10	IP	SC
6	47%	1.0	0.8*	>10	>10	>10	IP	SCS
7	27%	1.0	0.9*	>10	>10	>10	IP	SCS
8	5%	1.0*	1.0	4.1	3.7	3.8	IP	SC
9	<1%	1.0	1.1	0.5	0.4*	0.4	OP	PC
10	<1%	1.0	1.0	0.1	0.1*	0.1	OP	PC
11	<1%	1.0	1.0	<0.1	<0.1*	<0.1	OP	PC
12	<1%	1.0	1.1	<0.1	<0.1*	<0.1	OP	PC
13	<1%	1.0	1.1	<0.1	<0.1*	<0.1	OP	PC

Reconfiguration   * The execution time of the best configuration

Fig. 9: Vector density, execution time normalized to IP in SC, and the best software/hardware configuration for each iteration of CoSPARSE (16×16) for SSSP on pokec. Each iteration is color coded with the best configuration. The best configuration changes with the active vertex set, which conforms to the analysis in Section III-C.

The performance and energy efficiency gain of CoSPARSE on a 16×16 system over Ligra on a Xeon CPU is shown in Figure 10. In terms of performance, CoSPARSE outperforms Ligra in most cases and achieves a maximum speedup of 3.5×. Ligra outperforms CoSPARSE for pokec on BFS and SSSP slightly because the CPU has much more hardware resources, *e.g.* on-chip memory, to handle the large memory footprint of pokec. However, the CPU consumes at least 200× more power and 40× more area than CoSPARSE. Upon normalizing the performance by the power consumption, we obtain an efficiency gain of 84× for BFS and 129× for SSSP. Overall, CoSPARSE achieves an average energy efficiency gain of 404.4× across all evaluated algorithms and graphs, compared to Ligra.

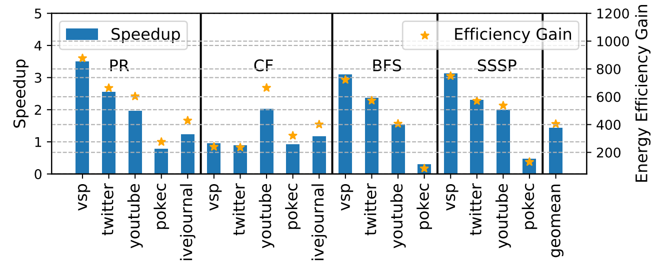

Fig. 10: Speedup and efficiency gain of CoSPARSE (16×16) over Ligra (Intel Xeon E7-4860 at 2.6 GHz, 48 cores with 256GB DRAM).

V. Conclusion

This work proposed CoSPARSE as a novel solution that combines software and hardware reconfiguration strategies to optimize the performance and efficiency of SpMV, and thereby SpMV-based graph analytics algorithms. We mapped different SpMV algorithms with custom scheduling and workload balancing onto an architecture with fast reconfiguration of the on-chip memory hierarchy. As a fully automated system, CoSPARSE judiciously decides the best-performing software/hardware configuration. The parameters that guide the reconfiguration decision-making engine are obtained by evaluating SpMV on a wide range of matrices and system sizes. For SpMV, CoSPARSE showed significant speedups (4.5× and 17.3× on average) and energy efficiency gains (282.5× and 730.6× on average) compared to the CPU and GPU, respectively. CoSPARSE also provides an energy-efficient platform for graph analytics; compared to Ligra on CPU, CoSPARSE achieved an average speedup and energy efficiency improvement of 1.5× and 404.4×, respectively.

Acknowledgment

This work was supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864.

References

[1] R. Chen et al. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *TOPC*, 2019.
[2] C.-Y. Gui et al. A survey on graph processing accelerators: Challenges and opportunities. *JCST*, 2019.
[3] T. J. Ham et al., Graphicionado: A high-performance and energy-efficient accelerator for graph analytics, In *MICRO*. 2016.
[4] H. Kwak et al., What is twitter, a social network or a news media?, In *WWW*, 2010.
[5] L. Nai et al., Graphpim: Enabling instruction-level pim offloading in graph computing frameworks, In *HPCA*, 2017.
[6] S. Pal et al., Outerspace: An outer product based sparse matrix multiplication accelerator, In *HPCA*. 2018.
[7] S. Pal et al., Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration, In *PACT*. 2020.
[8] S. Pal et al., A 7.3 M output non-zeros/j sparse matrix-matrix multiplication accelerator using memory reconfiguration in 40 nm, In *VLSI*. 2019.
[9] J. Shun and G. E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, In *PPoPP*, 2013.
[10] N. Sundaram et al. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 2015.
[11] H. Wang et al., Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu, In *PPoPP*. 2019.
[12] Y. Wang et al., Gunrock: A high-performance graph processing library on the gpu, In *PPoPP*, 2016.
[13] C. Yang et al., Implementing push-pull efficiently in graphblas, In *ICPP*, 2018.
[14] K. Zhang et al., Numa-aware graph-structured analytics, In *PPoPP*, 2015.
[15] J. Zhou et al., Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing, In *CCGRID*. 2017.
[16] X. Zhu et al., Gemini: A computation-centric distributed graph processing system., In *OSDI*, 2016.