HPCA 2020 Author's Copy - Nov 2019

ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator

Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, Sudhakar Yalamanchili

ABSTRACT

Sparse problems that dominate a wide range of applications fail to effectively benefit from high memory bandwidth and concurrent computations in modern high-performance computer systems. Therefore, hardware accelerators have proposed to capture a high degree of parallelism in sparse problems. However, the unexplored challenge for sparse problems is the limited opportunity for parallelism because of data dependencies, a common computation pattern in several sparse problems, in which the restricted parallelism and the need for high bandwidth are two contradictory attributes that challenge performance optimization. Our key insight is to extract parallelism by mathematically transforming the computations to equivalent forms. The transforming allows us to break down the sparse kernels into a majority of independent parts and a minority of data-dependent ones, and reorder these parts to gain performance. To implement our approach, we propose a lightweight reconfigurable sparse-computation accelerator (ALRESCHA). To efficiently run the data-dependent and parallel parts and to enable fast switching between them, ALRESCHA makes two contributions. First, it implements a compute engine with a fixed compute unit for the parallel parts and a small reconfigurable engine to facilitate the execution of the data-dependent parts. Second, ALRESCHA benefits from a locally-dense storage format, with the right order of blocks and values to yield the order of computations dictated by the transformations. The combination of the lightweight reconfigurable hardware and the locally-dense storage format enables uninterrupted streaming from memory. Our experimental results of running scientific and graph applications on various data sets show that compared to GPU, ALRESCHA achieves an average speedup of 15.6× for scientific sparse problems, and 8× for graph algorithms. Moreover, compared to GPU, ALRESCHA consumes 14× less energy.

1. INTRODUCTION

Sparse problems have become significantly popular in a wide range of applications, from scientific problems to graph analytics. Since traditional high-performance computing systems fail to *effectively* provide high bandwidth for sparse problems, several studies have advocated software optimizations for CPUs [1, 2, 3], GPUs [4, 5, 6, 7, 8], and CPU-GPU systems [9]. Besides, as the effectiveness issue has coupled with approaching the end of Moore's law, specialized hardware for sparse problems has become more attractive. For instance, hardware accelerators have been proposed for sparse matrix-matrix multiplication [10, 11, 12], matrix-vector multiplication [13, 14], or both [15]. In addition, accelerators for sparse problems have been proposed to reduce memory-access latency or improve energy efficiency [16, 17, 18, 19, 20]. Further, utilizing approaches such as blocking [21, 20] avoids indirect memory accesses.

The aforementioned hardware accelerators and software-optimization techniques for sparse problems often focus on a specific domain of application and take advantages of the specific pattern in computations to improve performance (more details in Table 2). However, flexibility in the range of target applications is an important feature for a hardware accelerator. Such a flexibility is not just for creating more generic accelerators; but, for accelerating all the different kernels in a program to effectively improve the overall performance.

In fact, unlike the assumptions of prior work, sparse problems may involve two groups of kernels with contradictory features: (i) highly parallizable and (ii) data-dependent kernels. In such a case, the challenge is that while the sparse problems require high bandwidth and a high level of concurrency, **the dependent computations prevents benefiting from the available memory bandwidth and the high level of concurrency**. In other words, *the need for high bandwidth* and the *limited opportunity for parallelism* are two contradictory attributes, which challenges performance optimization. Such sparse problems have become a major computation pattern in many fields of science. For instance, a high-performance conjugate gradient (HPCG) [22] benchmark is now a complement to the high-performance Linpack (HPL).

To clarify the challenge, Figure 1a shows the particular data-dependency pattern in scientific sparse problem (details in Section 2.1), which causes a performance bottleneck. As the pseudo code shows, for $col < row$, the computation of $x[row]$ must wait for the previous elements of x to be done. As a result, processing each row of the matrix depends on the result of processing the previous row and the rows cannot be parallelized. Therefore, as Figure 1b shows, typically the rows of the matrix are processed sequentially – even though processing individual rows (i.e., a dot product) can be parallelized. To extract more parallelism, a code optimization such as row reordering or matrix coloring [8] have been proposed to capture and run independent rows in parallel. However, the effectiveness of such high-level methods (i.e., in the granularity of instructions) depends on the distribution of non-zero values in a matrix – e.g., a specific problem may not have independent rows.

The Key Insight: Our observation to resolve the challenge is that the data-dependent operations rely *only on a fraction of the results* from the previous operations. Thus, the key insight is to extract parallelism by mathematically transforming the computation to equivalent forms. Such transformations allows us to **break down the traditionally dependent parts into a**

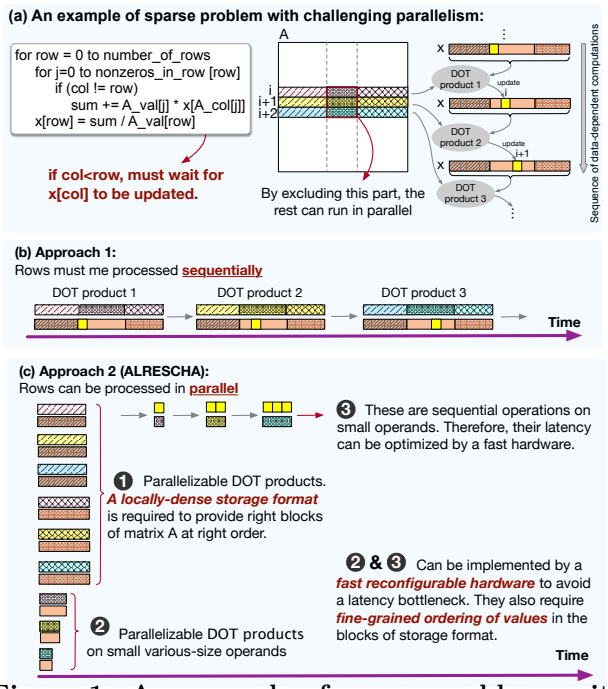


Figure 1: An example of sparse problems with data-dependency patterns in its computations.

***majority* of parallel and a *minority* of dependent operations.** To do so, we exclude the dependency-prone section of the matrix and run the rest in parallel (i.e., we convert Figure 1b to Figure 1c). To implement the key insight, we propose a lightweight reconfigurable sparse-computation accelerator (ALRESCHA[1]), which makes three contributions:

- **Lightweight reconfigurability:** After transforming the computations, while the majority of operations will be highly parallelizable (❶ and ❷), we still need to execute a few data-dependent computations ❸, albeit on small operands. During runtime, the application repetitively switches between the three mentioned groups of operations (i.e., between ❶ and ❷/❸). The switching itself must be fast enough to prevent a bottleneck. To this end, the compute engine of ALRESCHA implements *quick reconfiguration during runtime* by integrating a fixed *data path* for ❶, and a small and simple reconfigurable *data path* for ❷ and ❸.

- **Locally-dense storage format:** To smoothly stream sparse data, ALRESCHA proposes a storage format in which *the order of blocks and their elements match the order of computations*. As ❶ in Figure 1c illustrates, to enable parallel computation, the compute engine requires the ordered blocks of matrix A to be streamed in the right timing. Our proposed locally-dense format captures such an ordering to facilitate ❷ and ❸. ALRESCHA integrates the locally-dense storage format with a data-driven execution model to eliminate the streaming and decoding of meta-data.

[1]A binary star, the two stars of which orbit one another.

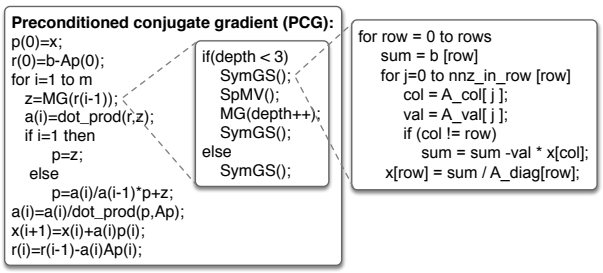


Figure 2: An example of the PCG algorithm [22] for solving a sparse linear system of equations (i.e., $Ax = b$), including SpMV and SymGS.

- **Generic sparse accelerator:** The ability of ALRESCHA in accelerating distinct kernels, makes it the first generic hardware accelerator for a wide range of sparse applications from scientific to graph applications whether or not they have data-dependent compute patterns. For applications with no distinct kernels, ALRESCHA still offers high performance by utilizing the proposed locally-dense storage format and hence, avoiding meta-data transfer.

ALRESCHA accelerates a wide range of sparse kernels, including sparse matrix-vector multiplication (SpMV), symmetric Guass-Seidel (SymGS) smoother, Page rank (PR), breadth-first search (BFS), and single-source shortest path (SSSP). To evaluate ALRESCHA, we target a wide range of datasets with various sizes. Comparing ALRESCHA with a CPU and a GPU platform shows that ALRESCHA achieves an average speedup of 15.6× for scientific sparse problems and 8× for graph algorithms. Moreover, compared to a GPU, ALRESCHA consumes 14× less energy. We also compare ALRESCHA with the state-of-the-art accelerators for sparse problems, namely, OuterSPACE [15], an accelerator for SpMV, GraphR [19], and a Memristive accelerator for scientific problems [20]. Our experimental results on various data sets show that comparing to state-of-the-art, ALRESCHA achieves an average speed up of 2.1×, 1.87×, 1.7× respectively, for scientific algorithm, graph analytics, and SpMV. The performance gain on data sets with various distributions of non-zero values show s that the gained benefits of ALRESCHA is independent from specific patterns in data.

2. BACKGROUND

This section reviews the background and the characteristics of the two key sparse problems.

2.1 Scientific Problems

A wide variety of physical-world phenomena, such as sound, heat, elasticity, fluid dynamics, and quantum mechanics, is modeled with partial differential equations (PDE). To numerically process and solve them via digital computers, PDEs are discretized to a 3D grid (e.g., using 27-stencil discretization), which is then converted to a linear system of algebraic equation s: $Ax = b$, in which A is the coefficient matrix, often very large for two or higher dimensional problems (e.g., elliptic, parabolic, or hyperbolic PDEs). Such a system of linear equations,

with a symmetric positive-definite matrix, can be solved by iterative algorithms such as conjugate gradient (CG) methods (e.g., preconditioned CG (PCG), which ensures fast convergence by preconditioning). These methods are specifically useful for solving sparse systems that are too large to be solved by direct methods.

An example of the PCG algorithm for solving $Ax = b$ is shown in Figure 2 [22]. The algorithm updates the vector x in m iterations. As Figure 3 shows, the execution time of the algorithm is dominated by two kernels, SpMV and SymGS [23, 8, 24]. The remaining kernels, such as dot product, consume only a tiny fraction of the execution time and are so ubiquitous that they are executed using special hardware in some supercomputers.

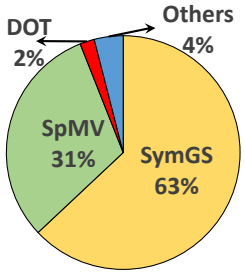


Figure 3: The breakdown of execution time of PCG on NVIDIA K20.

To explore the characteristics of SpMV and SymGS, we use an example of applying them on two operands, a vector ($b_{1\times m}$) and a matrix ($A_{m\times n}$)[1]. Applying **SpMV** on the two operands results in a vector ($x_{1\times n}$), each element of which can be calculated as:

$$x_j = \sum_{i=1}^{k} b[A^T\_ind_i] \times A^T\_val_{ij} \quad (1)$$

in which k, A^T_val and A^T_ind are the number of non-zero values, the non-zero values themselves, and the row indices of the j^{th} column of A^T, respectively. Figure 4a shows a visualization of Equation 1. Since the elements of the output vector can be calculated independently, SpMV has the potential for parallelism. On the other hand, each element of the vector result of applying **SymGS** on the same two operands (i.e. vector $b_{1\times m}$ and a matrix $A_{m\times n}$) can be calculated as follows, based on the Gauss-Seidel method [25]:

$$x_j^t = \frac{1}{A^T_{jj}} - (b_j - \sum_{i=1}^{j-1} A^T_{ij} \times x_i^t - \sum_{i=j+1}^{n} A^T_{ij} \times x_i^{t-1}). \quad (2)$$

Figure 4b illustrates a visualization of Equation 2 (i.e., the blue vectors correspond to $\sum_{i=1}^{j-1} A^T_{ij} \times x_i^t$ and red vectors correspond to $\sum_{i=j+1}^{n} A^T_{ij} \times x_i^{t-1}$). In fact, calculating the j^{th} element of x at iteration t (i.e., the orange element of x^t in Figure 4b) depends not only on the values of x at iteration $t-1$ (i.e., the red elements of x^{t-1}), but also on the values of x^t, which are being calculated in the current iteration (i.e., the blue elements of x^t). Such dependencies in the SymGS kernel limit the parallelism opportunity. Although some optimization strategies have been proposed for employing parallelism [8], the SymGS kernel can still be a performance bottleneck.

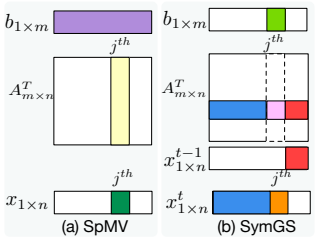


Figure 4: Calculation of j^{th} element of output vector x by: a) SpMV, and b) SymGS.

[1]In the rest of the paper, all matrix As refer to this matrix.

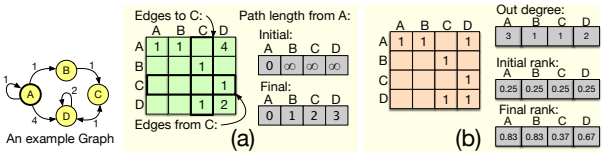


Figure 5: An example graph, its adjacency sparse matrices, and the vector operands of two graph algorithms: (a) SSSP and (b) PR.

2.2 Graph Analytics

A common approach for representing graphs is to use an adjacency matrix, each element of which represents an edge in a graph. Figure 5 illustrates an example of representing a graph as an adjacency matrix. Since in many applications the graph is sparse , the equivalent adjacency matrix includes several zeros. Graph algorithms traverse vertices and edges to compute a set of properties based on the connectivity relationships. Traversing are implemented as a form of dense-vector sparse-matrix operations. Such implementations are suited to the vertex-centric programming model [26], which is preferred to the edge-centric model. The vertex-centric model divides a graph algorithm into three phases. In the first phase, all the edges *from* a vertex (i.e., a row of the adjacency matrix) are processed. This process is a *vector-vector operation* between the row of the matrix and a property vector, varied based on the algorithm. In the second phase, the output vector from the first phase is *reduced* by a reduction operation (e.g., sum). In the final phase, the result is assigned to its destination.

Three widely used graph algorithms are breadth-first search (BFS), single-source shortest path (SSSP), and page rank (PR). Additionally, since graphs are usually sparse, SpMV is frequently used in graph applications. Figure 5 shows example graphs, the adjacency matrices, and the vector operands for the SSSP and PR algorithms. For SSSP (Figure 5a), the vector containing the lengths of nodes from node A is updated iteratively by multiplying a row of the matrix by the path-length vector and then choosing the minimum of the result vector. After traversing all the nodes, the final values of the vector indicate the shortest paths from node A to all the other node s. PR (Figure 5b) iteratively updates the rank vector, initialized by equal values. At each iteration, the elements of the rank vector are divided by the elements of the out-degree vector (i.e., the number of out-going edges for each vertex), chosen by a row of the matrix, and the result vector is reduced to a single rank by adding the elements of the vector.

2.3 Common Features

While the sparse kernels used in both scientific and graph applications are similar in having sparse-matrix operands, some kernels (e.g., SpMV) exhibit more con-

Table 1: The properties of sparse kernels and corresponding dense data paths, implemented in ALRESCHA. Depending on the type of kernel, the *operation* in phase 1 can use the three vector operands at the same time or use just two of them.

Sparse Kernel	Sparse Application	Dense Data Paths	Phase 1 (vector operation)				Phase 2 (reduce)	Phase 3 (assign)
			vector operand1	vector operand2	vector operand3	operation		
SymGS	PDE Solving	D-SymGS/GEMV	a row of coefficient matrix	the vector from iteration (i-1)	the vector at iteration (i)	multiplication	sum	apply operation with A^T and b_j and update vector
SpMV	PDE solving and Graph	GEMV	a row of coefficient matrix	the vector from iteration (i-1)	N/A	multiplication	sum	sum and update the vector
Page Rank	Graph	D-PR	a column of adjacency matrix	the out-degree vector of vertices	the rank vector at iteration (i-1)	AND/division	sum	rank vector update
BFS	Graph	D-BFS	a column of adjacency matrix	the frontier vector	N/A	sum	min	compare and update distance vector
SSSP	Graph	D-SSSP	a column of adjacency matrix	the frontier vector	N/A	sum	min	compare and update distance vector

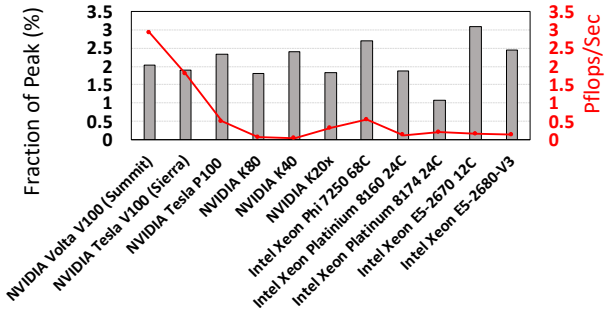


Figure 6: (a) Performance of modern computing platforms ranked by the standard metric of HPCG benchmark on GPUs and CPUs.

currency, whereas others (e.g., SymGS) have several data dependencies in their computations. Regardless of this difference, a common property of sparse kernels is that the reuse distance of accesses to the sparse matrix is high, while the input and output vectors of these kernels are being reused frequently. Moreover, the accesses to at least one of the vectors are often irregular.

The other common feature of these kernels is that they follow the three phases of operations iteratively (see Sections 2.1 and 2.2). Table 1 summarizes these phases for the main sparse kernels, as well as the operands and the operations at each phase. The sparse kernels calculate an element of their result by accessing a row/column of the sparse large matrix only once , and then reuse one or two vector/s for the calculation of all output vector elements. We benefit from these common features to design an accelerator, flexible to run all the mentioned sparse kernels without significant overhead. ALRESCHA converts the sparse kernels to dense data paths , listed in the second column of Table 1 (details in Section 4.2.1).

3. MOTIVATION AND RELATED WORK

Sparse problems, either in compressed or sparse representations, face performance challenges. The compressed formats are less efficient because they require storing, transferring, and processing of zero values . Therefore, sparse representations have been the more preferred approaches, even though they still rely on transferring meta-data and random memory accesses that limit performance by memory-access latency. As the ratio of compute-per-data-transfer of the sparse problem is low, normally, we expect their performance to be directly related to the memory bandwidth.

3.1 Ineffectiveness of CPUs and GPUs

To date, many software-level optimizations for CPUs [1, 2, 3], GPUs, [4, 5, 6, 7, 8], and CPU-GPU systems [9] have been proposed. However, software optimizations alone cannot effectively handle irregular data-dependent memory references, a main characteristic of sparse problems. Irregular data-dependent memory references limit the reach of software schemes and thus lead to poor performance due to degraded bandwidth utilization. Furthermore, optimizations for extracting more parallelism and bandwidth such as matrix coloring [8] and blocking [21] have not been effective enough because of the aforementioned reason. Figure 6 summarizes the performance of running sparse scientific applications on a range of GPUs. As the figure shows, they utilize only a tiny fraction of the peak performance.

3.2 Previous Hardware Accelerators

The mentioned inefficiency challenges, along with approaching the end of Moore's law, have provided the motivation for migration to specialized hardware accelerators for sparse problems. For instance, several hardware accelerators have targeted sparse matrix-matrix multiplication [10, 11, 12], matrix-vector multiplication [13, 14], or both [15], which are key kernels in a wide range of scientific and graph applications. A state-of-the-art SpMV accelerator, OuterSPACE [15], employs an outer-product algorithm to minimize the redundant accesses to non-zero values of the sparse matrix. Despite the significant speedup of OuterSPACE over the traditional SpMV, it is not the most optimal approach when the sparse matrix has spatial locality. Moreover, although OuterSPACE increases the data reuse rate to reduce accesses to memory, it produces random access to a local cache. In another work, to be more efficient in utilizing memory bandwidth, Song et al. proposed GraphR [19], a graph accelerator, and Feinberg et al. proposed a scientific problem accelerator [20], both of which use a mid-point between compressed and sparse representations of sparse problems and process blocks of non-zero values instead of individual ones. Blocked representation reduces randomness in memory accesses and the amount of meta-data to be stored and transferred. In other sparse-problem studies, to reduce memory-access latency, Graphicionado [16], a graph-processing accelerator, substitutes accesses to the memory hierarchy with sequential accesses to scratchpad memory.

To compare the novel hardware approach and techniques of ALRESCHA with the most recent efforts for

Table 2: Comparing state-of-the-art accelerators for sparse kernels.

		GraphR [19]	OuterSPACE [15]	Memristive-Based Accelerator [20]	Row Reordering Matrix Coloring [8]	ALRESCHA
Application Domain		Graph	Graph (Only SpMV)	PDE Solver	PDE Solver	Graph and PDE Solver
Hardware	Multi-Kernel Support	✗	✗	✗	✗	✓
	BW Utilization	Low	Moderate	Low	Moderate	High
	NOT Transferring Meta-data	✗	✗	✗	✗	✓
	Processing Type	ReRAM Crossbar	PEs Connected in a High-Speed Crossbar	Heterogeneous Memristive Crossbar	GPU Instruction	Fixed Vector Processor and a Small Reconfigurable Switch
	Cache Optimizations For Frequently-Used Vectors	N/A	✗	N/A	✗	✓
	Reconfigurability	✗	Only for Cache Hierarchy	✗	N/A	✓
Techniques	Storage Format	4×4 COO	CSR	multi-size blocks (64×64, 128×128, 256×256, 512×512)	ELL	8×8 blocking with fine-grained in-block ordering
	Resolving Limited Parallelism	N/A	N/A	✗	✓ (Instruction-Level Limited by NNZ patterns)	✓

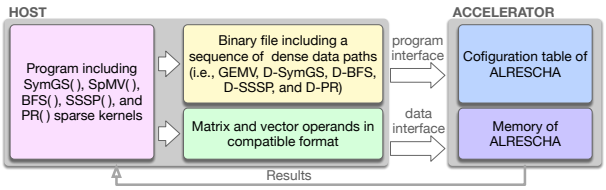


Figure 7: Key components of ALRESCHA.

accelerating sparse problems, Table 2 summarizes their properties. As the table lists, **ALRESCHA is the first reconfigurable sparse problem accelerator for both scientific and graph applications**, which supports multi-kernel execution to resolve the limited parallelism in fine granularity. As we explain in Section 4.2, ALRESCHA also implements optimization such as accesses to the vector operand of the sparse operation.

4. ALRESCHA

Figure 7 shows an overview of ALRESCHA, which is reconfigured at runtime to support fast switching between dense data paths, the building blocks of sparse kernels. Section 4.1 reviews the programming model of ALRESCHA, and Section 4.2 shows how we implement sparse kernels by converting them to a sequence of dense data paths. Section 4.3 provides the details of the microarchitecture and design choices. Section 4.4 explains how reconfigurability works and how switching between the dense data paths occurs. Section 4.5 discusses the proposed locally-dense storage format.

4.1 Programming Model

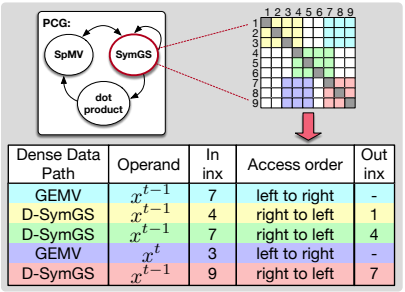
ALRESCHA is a memory-mapped accelerator that works on its memory, which is accessible by the host only for programming. The programming model of ALRESCHA is similar to offloading computations from a CPU to a GPU (the programming model itself is beyond the scope of this paper). To program the accelerator, the host launches sparse kernels of a sparse algorithm to the accelerator. To this end, the host (i) converts a sparse kernel to a binary file (see convert algorithm in Section 4.2.1), consisting of a sequence of dense data paths, and writes them to the configuration table (example in Figure 8) of the accelerator through the *program interface* (Figure 7); it then (ii) converts the sparse-matrix operand to a locally-dense storage format (details explained in Section 4.5) and writes the blocks of non-zero values into the physical memory space of the accelerator through the *data interface* (Figure 7). Each locally-dense block corresponds to a dense data path.

4.2 Implementation of Sparse Kernels

To implement the sparse kernels of sparse algorithms (i.e., PCG and graph algorithms) with large sparse-matrix operands, ALRESCHA breaks down the sparse kernels into a sequence of dense data paths. Among them, the sparse kernels with straightforward data dependencies (i.e., SpMV, BFS, SSSP, and PR) are broken down into a sequence of general matrix-vector multiplication (GEMV), dense BFS (D-BFS), dense SSSP (D-SSSPs), and dense PR (D-PR), respectively. These dense data paths have the same functionality as their corresponding sparse kernel; however, they work on *non-overlapping locally-dense blocks* of the sparse-matrix operand and *overlapping sub-vectors* of the dense-vector operand of the main sparse kernel. On the other hand, the SymGS kernel, the key computation of PCG algorithm with complex data dependencies is broken down into a combination of parallelizable GEMV and sequential dense SymGS (D-SymGS) data paths.The conversion of Sparse SymGS to GEMV and D-SymGS is based on the mathematical transformation, explained in Section 4.2.2. By separating GEMV from D-SymGS data paths, we prevent the performance from being limited by the sequential nature of the SymGS kernel.

4.2.1 *Sparse Kernel to Dense Data Path Conversion*

Algorithm 1 shows the procedure for converting a sparse matrix to dense data paths. Based on the type of kernel, its sparse matrix operand, and the hardware specification (i.e., the number of ALUs in Figure 11a), the host generates the configuration table, each row of which specifies the type of data path, and information about its operands (e.g., read and write address of cache). The bits stored in the configuration table are used for configuring the *configuration switch*. For each dense data path, besides its type (illustrated by one bit), an operand vector and its index (i.e., Inx_{in}), the access order, and the output index (i.e., Inx_{out}) are stored. The Inx_{in} and Inx_{out} indicate the read and write addresses of a cache that contain the vector operands. All information in a row of the table consumes $2\sqrt{n}+3$ bits (as we assume that the size of the vectors is n, indexing the input and output requires $\sqrt{n}$ bits).



Dense Data Path	Operand	In inx	Access order	Out inx
GEMV	x^{t-1}	7	left to right	-
D-SymGS	x^{t-1}	4	right to left	1
D-SymGS	x^{t-1}	7	right to left	4
GEMV	x^{t}	3	left to right	-
D-SymGS	x^{t-1}	9	right to left	7

Figure 8: An example of converting the SymGS kernel to the configuration table, consisting of the sequence of dense data paths.

Algorithm 1 Convert Algorithm

```
1: function CONVERT(A_nxn, ω, KernelType)
     A_nxn: sparse matrix, ω : block width
     DP: Data path type
     l2r: left to right, r2l: right to left
2:   Inx_in := 0, Inx_out := 0
3:   Blocks[] = Split(A, ω) // partitions A to ω × ω blocks
4:   m = n/ω
5:   for (i = 1, i < m, i++) do
6:     for (j = 1, i < m, j++) do
7:       if (nnz(Blocks[i, j]) > 0) then
8:         if KernelType != SymGS then
9:           DP = KernelType.DataPath
10:          Inx_in = i.ω, Inx_out = j.ω
11:          Order = l2r
12:          Op = port1 // the operand vector
13:        else
14:          if (i! = j) then
15:            DP = GEMV
16:            Inx_in = j.ω, Inx_out = −1
17:            Order = l2r
18:            if (i > j) then
19:              Op = port2 //which is x^{t-1}
20:            else
21:              Op = port1 //which is x^t
22:          else
23:            DP = D-SymGS
24:            Inx_in = j.ω, Inx_out = (i+1).ω
25:            Order = r2l
26:            Op = port2 //which is x^{t-1}
27:        Add2Table(DP, Op, Inx_in, Order, Inx_out)
```

The general rule of conversion for SymGS is to assign GEMVs to non-diagonal non-zero blocks, and D-SymGS to diagonal non-zero blocks of the sparse matrix. To decrease the frequency of switching between the two data paths (GEMV and D-SymGS), ALRESCHA reorders them so that it first runs all the GEMVs successively and then switches the data path of D-SymGS. The distributive property of inner products in Equation 2 guarantees the correctness of such reordering. As an example of the outcome of Algorithm 1, Figure 8 shows the state machine of PCG, equivalent to the algorithm in Figure 2, which comprises three sparse kernels, two of which are the focus of this paper and are launched to the accelerator by the host. For each kernel, a configuration table is generated by the host. The configuration table for a SymGS kernel is shown in Figure 8. All the non-zero blocks in the upper triangle of A have to be multiplied by x^t, and all of those in the lower triangle have to be multiplied by x^{t-1}.

4.2.2 *Dense Data Path Implementation*

ALRESCHA implements two classes of data paths. The first class consists of D-BFS, D-SSSP, D-PR, and GEMV with straightforward patterns of data dependencies. The second class (e.g., D-SymGS) captures more complicated sequential patterns.

Dense BFS, SSSP, PR, and GEMV: The dense data paths work on locally dense blocks of A to capture locality in accesses to the dense vector by taking advantage of spatial locality in the non-zero values of the sparse matrix. If the sparse matrix includes blocks of non-zero values of size n, ALRESCHA splits the vector operands into chunks of size n, and at each cycle, it fetches a chunck of the vector from a local cache instead of fetching an individual element of the vector operand. More specifically, a vector operation is applied on n elements of the vector operand and all the non-zero blocks of A in a row. The approach of ALRESCHA for running BFS, SSSP, PR, and SpMV provides two advantages: (i) It guarantees locality of cache accesses (i.e., the values in a cache line are used in succeeding cycles) , and (ii) each element of an operand vector is fetched from the cache only once per $\#cols/n$.

Dense SymGS: SymGS is a matrix-vector operation with a sparse matrix operand A and three vector operands, A_{jj}, b, and a combination of x^{t-1} and x^t (Equation 2). The three vectors respectively include the diagonal of A, the right-hand side coefficient of the $Ax = b$ equations, and a combination of variables of the equations computed in previous iterations (i.e., the initial values) and the variables being computed in the current iteration. Based on Equation 2, calculating an output element of SymGS includes two inner products, the sizes of which change when we move across the rows of A to compute various variables of x. To enable using a compute engine for both inner products, ALRESCHA merges them by factoring out the subtraction as follows:

$$x_j^t = (\frac{1}{A_{jj}^T} - b_j) + (\sum_{i=1}^{j-1} A_{ij}^T \times x_i^t + \sum_{i=j+1}^{n} A_{ij}^T \times x_i^{t-1}). \quad (3)$$

In other words, there is a single inner product of size n, which is a common operation in all of the dense data paths, and hence we can utilize the same computations, used for other dense data paths. One operand of the inner product is made by shifting x^t into x^{t-1}. This can be implemented by just rotating the inputs of multipliers

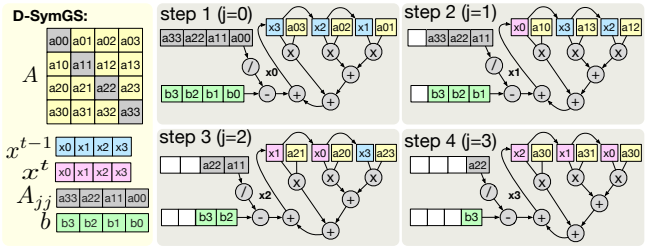


Figure 9: An example of D-SymGS. At each step (i.e., $j = 0$ to 3), one x_j^t is calculated based on Equation 3. The new x_j^t is then used in calculating the next element x_{j+1}^t at the next step. The arrows between x_0^t to x_3^t indicate shifting them one to the right at each step.

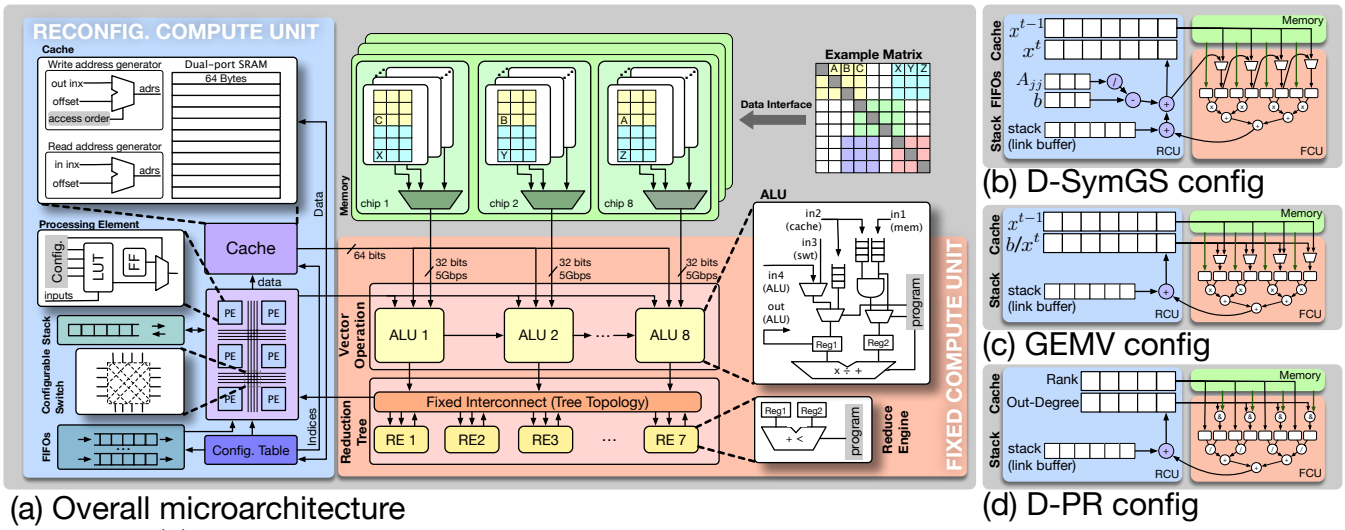


Figure 10: (a) The microarchitecture of ALRESCHA including the FCU for implementing common computations, and the RCU for providing specific configuration for distinct dense data paths. And, three example configurations for supporting: (b) D-SymGS, (c) GEMV, and (d) PR.

and pushing the x_j^t into the first multiplier to be used in calculating $(j+1)^{th}$ element of x.

To clarify the functionality of ALRESCHA for D-SymGS, we use a simple example in Figure 9, for which, we assume that the size of the problem fits the size of the hardware (i.e., the number of multipliers), and that the matrix A is dense. As Figure 9 illustrates, ALRESCHA stores the diagonal of A in a separate vector and does the subtraction with b in parallel with the inner products. Note that separately storing the diagonal of A will help utilize the memory bandwidth only for streaming the operand of the inner product engine. At the first step, one row of non-diagonal elements of A is multiplied by x_1^{t-1} to x_3^{t-1}. In the second step, we need x_0^t instead of x_1^{t-1}. However, the newly calculated x_0^t, will not take the place of kicked off x_1^{t-1}. As a result, we insert the new variables by shifting the old one to the right. As Figure 9 shows, while reading a row from A at each cycle, *the elements, belonging to the upper triangle of A are reordered, while the elements in the lower triangle keep their original orders*. This is correlated with the orders of the values in a dense block of the storage format, shown in Figure 13 in Section 4.5.

4.3 Reconfigurable Microarchitecture

This section introduces the microarchitecture of ALRESCHA for accelerating the sparse kernels by providing their fundamental data paths (i.e., GEMV, D-SymGS, BFS, SSSP, and PR). To deliver close-to-ideal performance, the main idea behind the hardware of ALRESCHA is to separate the fixed computation unit (FCU) from the reconfigurable computation unit (RCU) and configuring only the former for switching between data paths (Figure 10). Configuring only a fraction of the entire data path minimizes the configuration time.

The FCU streams *only the dense blocks of the sparse matrix* (i.e., without any meta-data) from memory and applies the required vector operation (i.e., phase 1 in Table 1). The FCU includes ALUs and reduce engines (REs) connected together in a tree topology , as shown in Figure 10 , and form the reduction tree. The interconnections between the REs of the FCU are fixed for all data paths and do not require reconfiguration. The reduction tree is fully pipelined to yield the speed of the streaming data from memory. One of the inputs of ALUs (i.e., the matrix operand) is always streamed from memory, and the other inputs (i.e., the vector operands) come from the RCU. The former input of the ALU requires a multiplexer because at runtime, its input might need to be changed. For example, only for the initialization of the D-SymGS data path does the input come from the cache (i.e., x^{t-1}), but after initialization, it comes from a processing element of the RCU and a forwarding connection between the multipliers. For GEMV, for example, the ALU requires the multiplexers to choose between x^{t-1} and x^t during runtime.

The responsibility of the RCU is to handle the specific data dependencies of different kernels. The RCU includes a local cache, buffers, processing elements (PE), and a *configurable switch*, which determines the interconnection between the units in the RCU. The configurable switch is not a novel approach here and is implemented similar to those of FPGAs and is directly controlled by the content of a configurable table (Section 4.4). The local cache stores the vector operands, which require addressable accesses (e.g., x^{t-1}, x^t, and b), whereas the buffers handle vectors, which require deterministic accesses. For instance, we employ first-in-first-out (FIFO) for A_{jj}^T and b, and use a first-in-last-out stack for the link buffer. The link buffer establishes transmissions between the dense data paths (Section 4.4). For data path transmission, the reduce tree has to be drained, during which the switch is reconfigured to be prepared for the next data path. Therefore, the latency of configuration is hidden by the latency of draining the adder tree. The PEs of the RCU are implemented by look-up tables (LUTs) to provide multiplication, division, summation, and subtraction. Figure 10b, c, and d illustrate the configuration of the RCU for performing D-SymGS,

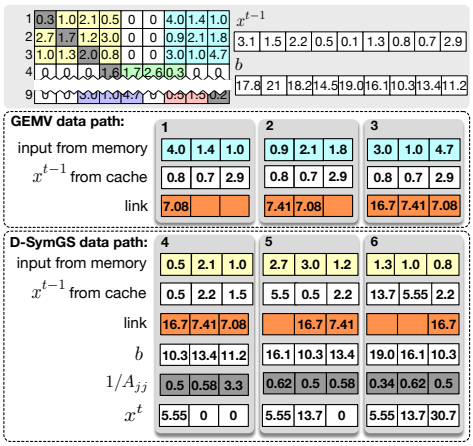


Figure 11: Switching between GEMV and D-SymGS data paths using the link stack.

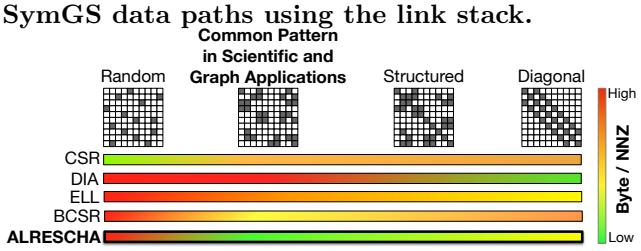


Figure 12: The spectrum of storage formats for various types of sparse matrices (low is better).

GEMV, and PR data paths.

4.4 Reconfigurability

Applications with a sequence of distinct sparse kernels for a portion of their computation benefit from reconfigurability because they frequently switch between the kernels. As a result, to satisfy high-speed and low-cost reconfigurability, ALRESCHA implements a lightweight reconfiguration, which indicates that only the configuration of a small fraction of the compute engine is changed frequently. This section shows how switching between the dense data paths occurs.

4.4.1 Switching Between Data Paths

Switching between two data-dependent data paths is performed using the link buffer (i.e., implemented as the stack). During the runtime of a data path, the intermediate results are pushed into the stack. Then, the successive data path pops them up. Figure 11 presents a numerical example for switching from GEMV to D-SymGS data paths. During steps 1 to 3, the RCU is configured to GEMV, and results are pushed into the link. At step 3, while the reduce tree (i.e., here the adder tree) is drained, the interconnect between the cache and FIFO buffers in the RCU is reconfigured to D-SymGS. This does not affect the pushes to the stack, so it can be parallelized and its latency hidden. In steps 4 to 6, values are popped from the link and consumed.

4.5 Storage Format

Depending on the distribution of non-zero values in a sparse matrix, various storage formats can suit them. Figure 12 compares four well-known formats based on their meta-data per non-zero values. The compressed sparse row (CSR), which stores a vector of column indices and a vector of row pointers, locates all the non-zero values independently. Therefore, it is the right choice when the non-zeros do not exhibit any spatial localities. On the other hand, when all the non-zeros are located in diagonals, the diagonal format (DIA) [28], which stores the non-zeros sequentially, could be the best option. An extension to the DIA format, Ellpack-Itpack (ELL) [29], is more flexible when the matrix has a combination of non-diagonal and diagonal elements (e.g., it is used for implementation of SymGS in GPUs). However, such a format does not provide enough flexibility for parallelizing rows because it does not sustain the locality across rows. More importantly, the choice of storage format *should be compatible with the common pattern in the majority of sparse applications*. Therefore, blocked CSR (BCSR) [21], an extension of CSR, which assigns the column indices and row pointers to blocks of non-zero values, is the right format.

BCSR is an appropriate format for scientific applications and graph analytics in terms of storage overhead. But, the strategy of BCSR for assigning indices and pointers, and the order of values, are not a good match for smooth data streaming. The main requirement for fast computation is the order of operations, which in turn that dictates the data structures be streamed in the same order. Thus, we adapt BCSR to a format with the same meta-data overhead but that is compatible with the reconfigurable microarchitecture of ALRESCHA.

Figure 13, illustrates our proposed locally-dense format for mapping an example sparse matrix to the physical memory addresses of ALRESCHA. The format follows the following rules: (i) The order of non-zero blocks gives the higher priority to non-diagonal non-zero blocks rather than to diagonal ones; (ii) the non-zero values belonging to the upper triangle of the diagonal blocks are stored in the opposite order of their original locations in the matrix (see the order of A, B, and C in Figure 13). Accordingly, the difference between the indices of BSCR and those of the ALRESCHA format is shown in Figure 13; (iii) for SymGS, as the diagonal of A is excluded

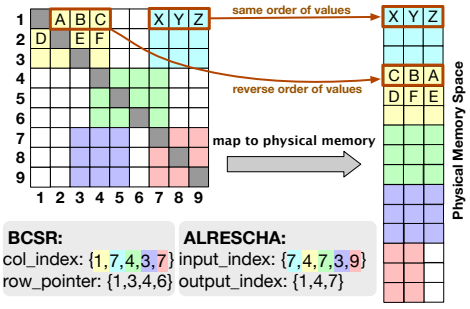


Figure 13: Comparison of the BCSR and the locally-dense format of ALRESCHA. The col_index of BCSR and input_index of ALRESCHA are color-coded to show their corresponding blocks in the matrix. ALRESCHA uses the index of the last column for the input_index of diagonal blocks.

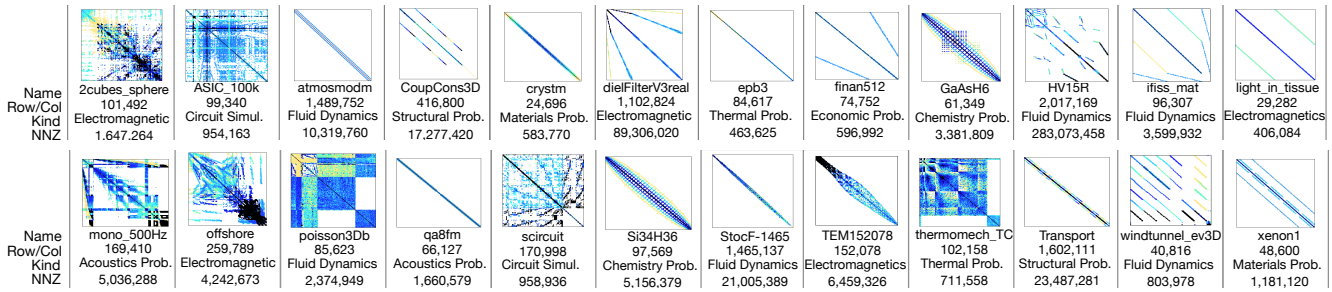


Figure 14: Scientific application datasets attained from University of Florida SuiteSparse (SS) [27].

Table 3: Graph datasets.

Dataset	row/col	NNZ
com–orkut	3,072,441	234,370,166
hollywood–2009	1,139,905	1,139,905
kron-g500–logn21	2,097,152	182,082,942
roadnet–CA	1,971,281	5,533,214
LiveJournal	4,847,571	68,993,773
Youtube	1,134,890	5,975,248
Pokec	1,632,803	30,622,564
sx-stackoverflow	2,601,977	36,233,450

and stored separately in a local cache, we consider non-square blocks on the diagonal (e.g., 3×4 instead of 3×3) so that the the mapping of the non-diagonal element of that block to the physical memory is adjusted; and (iv) the indices of the input and output (i.e., meta data) are not streamed from memory. They are stored only in the configuration table and are used for reconfiguration purposes. Therefore, the whole available memory bandwidth is utilized for streaming payload.

5. PERFORMANCE EVALUATION

This section explores the performance of ALRESCHA by comparing it with the CPU, GPU, and state-of-the-art accelerators for sparse problems. We evaluate ALRESCHA for both scientific applications and graph analytics. This section first introduces the data sets and algorithms, and the experimental setup. Then, we analyze execution time and energy consumption.

5.1 Datasets and Algorithms

We pick real-world matrices with applications in scientific and graph problems from the University of Florida SuiteSparse Matrix Collection [27]. The matrices, shown in Figure 14, are our datasets representing scientific applications, including circuit simulations, electromagnetic, fluid dynamics, structural, material, acoustics, economics, and chemical problems, all of which can be modeled by PDEs. As the figure shows, the non-zero values have various distributions across the matrices. For graph analytics, we choose the eight datasets, listed in Table 3, along with their dimensions and non-zero values. We run PCG, which includes the SymGS and SpMV kernels, on the matrices listed in Figure 14, and run BFS, SSSP, and PR on the matrices in Table 3. We also run SpMV on both categories of datasets.

5.2 Baselines

We compare ALRESCHA with the CPU and GPU platforms. The configurations of the baseline platforms are listed in Table 4. For the CPU and GPU platforms, we exclude disk access time. For fair comparisons , we include the optimizations, such as row reordering and suitable storage formats (e.g. ELL if necessary), proposed for the CPU and GPU implementations. The PCG algorithm and the graph algorithms running on GPU are respectively based on the cuSPARSE and Gunrock [30] libraries. The graph algorithms running on the CPU are based on the GridGraph [31] and/or CuSha [32] platforms, whichever achieves better performance for a specific algorithm.

Table 4: Baseline configurations.

GPU baseline	
Graphics card	NVIDIA Tesla K40c, 2880 CUDA cores
Architecture	Kepler
Clock frequency	745MHz
Memory	12 GB GDDR5, 288 GB/s
Libraries	Gunrock [30] and CUSPARSE
Optimizations	row reordering (coloring) [8], ELL format
CPU baseline	
Processor	Intel Xeon E5-2630 v3 8-core
Clock frequency	2.4 GHz
Cache	64 KB L1, 256 KB L2, 20 MB L3
Memory	128 GB DDR4, 59 GB/s
Platforms	CuSha [32], GridGraph [31]

Besides comparing with CPU and GPU, this section compares ALRESCHA with the state-of-the-art hardware accelerators, including OuterSPACE [15], which is an accelerator for SpMV, GraphR [19], which is a ReRAM-based graph accelerator, and the Memristive accelerator for scientific problem [20]. To reproduce their latency and power consumption numbers, we modeled the behavior of the preceding accelerators based on the information provided in the published papers (e.g., the latency of read and write operations for GraphR and Memristive accelerator). We validate our numbers based on their reported numbers for their configurations to make sure our reproduced numbers are never worse than their reported numbers. For fair comparison with ALRESCHA, we assign all the accelerators the same computation and memory-bandwidth budget – this assumption does not harm the performance of our peers.

5.3 Experimental Setup

In our experiments, we convert the raw matrices to the proposed locally-dense format using the convert algorithm (i.e., the approach explained in Section 4.5) implemented in Matlab. To do that, we examine block sizes of 8, 16, and 32 for the range of the data sets and choose the block size of eight because unlike the other

Table 5: ALRESCHA Configuration

Floating point	double precision (64 bits)
Clock frequency	2.5 GHz
Cache	1KB, 64-Byte lines, 4-cycle access latency
RE latency	3 Cycles (sum: 3, min: 1)
ALU latency	3 Cycles
Memory	12 GB GDDR5, 288 GB/s

two, eight provides a balance between the opportunity for parallelism and the number of non-zero values. We model the hardware of ALRESCHA using a cycle-level simulator with the configurations listed in Table 5. The clock frequency is chosen to enable the compute logic to follow the speed of streaming from memory (i.e., each 64-bit operands of ALU are delivered from memory in 0.4 ns, through the 32-bit 5 Gbps links.) To measure energy consumption, we model all the components of the microarchitecture using a TSMC 28nm standard cell and the SRAM library at 200MHz. The reported numbers include programming the accelerator.

5.4 Execution Time

5.4.1 *Scientific Problems*

Figure 15 illustrates the speedup of running PCG on ALRESCHA over the GPU (i.e., an implantation, optimized by row reordering [8] for extracting a high level of parallelism) on the primary axis, along with the bandwidth utilization on the secondary axis. The figure also captures the speedup of the Memoristive-based hardware accelerator [20]. On average, ALRESCHA provides a 15.6× speedup compared to the best proposed optimizations implemented on the GPU. The speedup of ALRESCHA is approximately twice that of the most recent accelerator for solving PDEs. To investigate the reasons behind this observation, we plot memory bandwidth utilization in Figure 15. As the figure shows, the performance of ALRESCHA and the other hardware accelerator for the 24 scientific datasets is directly related to memory bandwidth utilization – mainly because of the sparse nature. Moreover, none of them fully utilize the available memory bandwidth, because both approaches use blocked storage formats, in which the percentage of non-zero values in a block rarely reaches a hundred percent. Nevertheless, we see that ALRESCHA better utilizes the bandwidth, because it resolves the dependencies in computations, which otherwise limits bandwidth utilization.

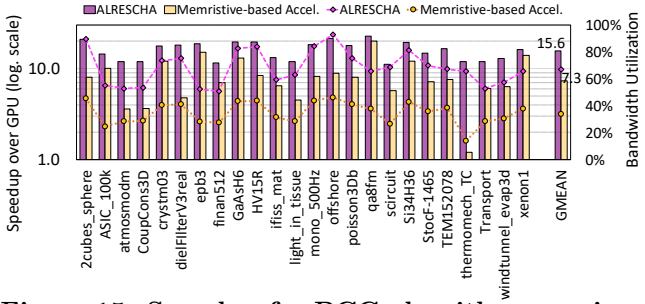


Figure 15: Speedup for PCG algorithm on scientific datasets, normalized to GPU (bar charts), and bandwidth utilization (the lines). The Memristive-based accelerator [20] is the state-of-the-art accelerator for scientific problems.

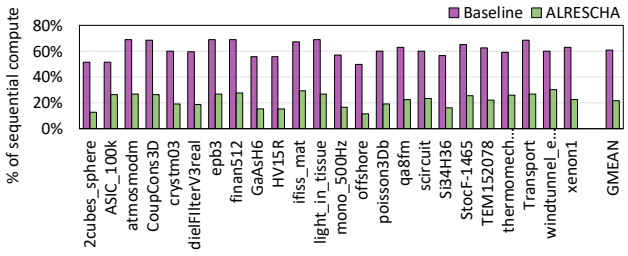


Figure 16: The reduction of the sequential part of the PCG algorithm by applying ALRESCHA. The baseline shows the percentage of sequential operations by row-reordering optimization.

To clarify the impact of resolving dependencies, on overall performance, Figure 16 presents the percentage of sequential computations in the GPU implementation, versus that in ALRESCHA, which has an average of 23.1% sequential operations. As the figure suggests, even in the GPU implementation that extracts the independent parallel operations using row reordering and graph coloring, on average 60.9% of operations are still sequential. This is more than 60% for highly-diagonal matrices and less than 60% for matrices with a greater opportunity for in-row parallelism. Such a trend identifies the distribution of locally-dense blocks as another rationale for determining the speedups. More specifically, when the distribution of non-zero values in rows of the matrix offers the opportunity for parallelism, the speedup over the GPU is smaller to when the non-zeros are mostly distributed in the diagonal.

Insights: *For multi-kernel sparse algorithms (e.g. PCG) with dependent computations, we improve performance by (i) extracting parallelizable dense data paths, correlated with locally-dense blocks, (ii) reordering the dense data paths and the values in the blocks to maximize reuse of data, and (iii) implementing them in a lightweight reconfigurable hardware, all of which result in fast switching not only between the distinct data paths of a single kernel, but also between the sparse kernels.*

5.4.2 *Graph Analytics And SpMV*

In this section, we explore the performance of ALRESCHA for algorithms that consist of one type of sparse kernel and naturally have fewer dependency patterns in their computations. The goal of this section is to show that ALRESCHA is not just optimized for a specific domain and actually is applicable to accelerating a wide range of sparse applications. First , we analyze the performance of graph applications. Figure 17 illus-

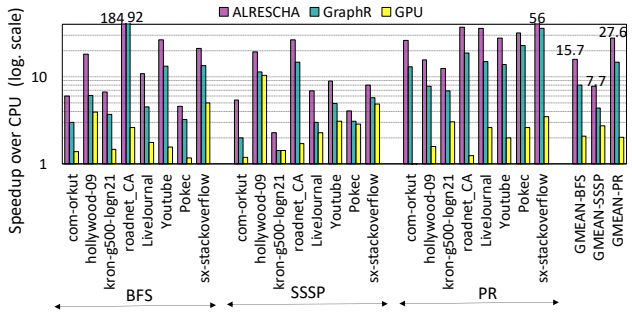


Figure 17: Speedup for graph algorithms on graph datasets over the CPU. GraphR [19] is the state-of-the-art graph accelerator.

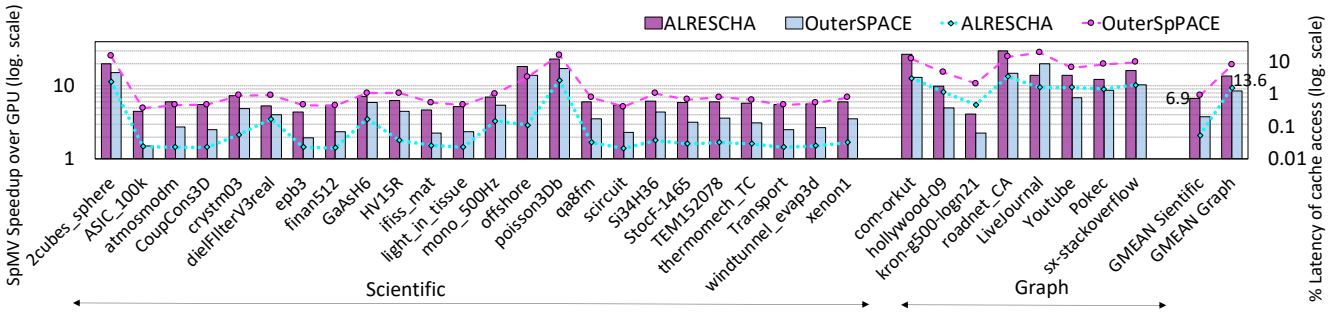


Figure 18: Speedup for running SpMV on scientific and graph datasets normalized to GPU (bar charts), and the percentage of execution time devoted to cache accesses (the lines). OuterSPACE [15] is the state-of-the-art SpMV accelerator.

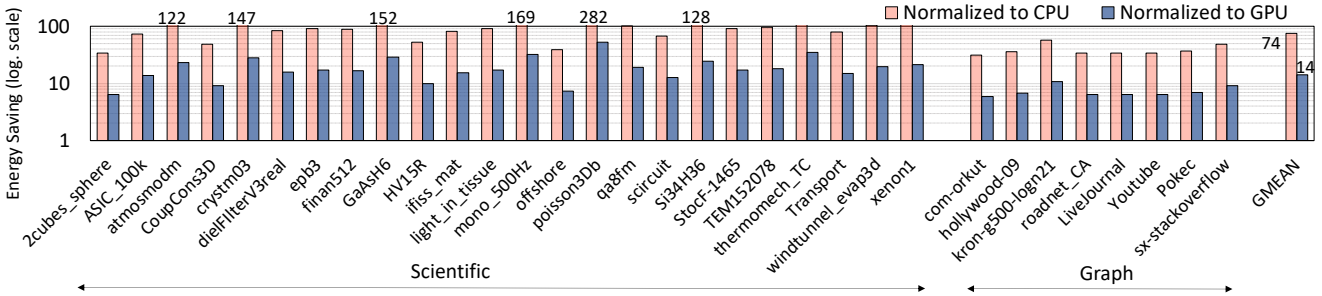


Figure 19: Energy consumption improvement of ALRESCHA normalized to that of CPU and GPU.

trates the speedup of running BFS, SSSP, and PR on ALRESCHA, a recent hardware accelerator for graph (i.e., based on GraphR [19]) and GPU platform s, all normalized to the CPU baseline. As the figure shows, ALRESCHA offers average speedups of 15.7×, 7.7×, and 27.6×, for BFS, SSSP, and PR algorithms over the CPU platform, respectively. We achive this speedup by avoiding the transfer of meta-data, reordering the blocks for increasing data reuse, and improving the locality.

Figure 18 illustrates the speedup of running the SpMV, a common algorithm of various sparse applications on ALRESCHA and OuterSPACE [15] (i.e., the recent hardware accelerator for the SpMV), normalized to the GPU baseline. As the figure shows, ALRESCHA offers average speedups of 6.9× and 13.6× for scientific and graph datasets. When running SpMV, all the data paths are GEMV ; therefore, no transmission between data paths is required. However, optimizations of ALRESCHA help achieve greater performance. The key optimization here is accesses to the cache to obtain frequent accesses to the vector operand of SpMV. To show this, the secondary axis of Figure 18 (i.e., the lines) plots the percentage of the whole execution time for accesses to the local cache. To perform an SpMV, OuterSPACE reads each element of the vector operand, multiples it with all the elements in a row of the matrix and then accumulates each of the partial products, and writes them to their correct corresponding element of the output vector. Hence, unlike ALRESCHA, the compute engine of OuterSPACE has to put the partial products in their right location in the output vector, which may leads to *lack of locality in* accesses to the cache.

Insights: *For single-kernel sparse problems (e.g., SpMV), ALRESCHA gains speedup by (i)improving the locality of cache accesses (i.e., consuming the values in a cache line in succeeding cycles); (ii) increasing the data reuse rate of not only the sparse-matrix operands, but also the dense-vector operands, and (iii) avoiding meta-data transfer and decoding.*

5.5 Energy Consumption

A primary motive for using hardware accelerators rather than using software optimizations is less energy consumption. To achieve this goal, the techniques integrated in the hardware accelerators have to be efficient. A source of energy consumption is accesses to local SRAM-based buffers or caches. That is, reducing the number of reads and writes from and to local memories, by substituting them with computation is beneficial. Figure 19 illustrates the energy consumption of ALRESCHA when running the SpMV kernel, normalized to that of the CPU and GPU baselines. As Figure 19 shows, on average, the total energy consumption improves by 74× compared to the CPU and 14× compared to the GPU platform. Note that the activity of compute units, defined by the density of the locally-dense block, impacts energy but not performance.

Insights: *The main reasons for the low energy consumption are the small reconfigurable hardware of ALRESCHA in combination with utilizing a locally-dense storage format with the right order of blocks and values matched with the order of computation, thus avoiding the decoding of meta-data, and reducing the number of accesses to the cache and the memory.*

6. CONCLUSION

We proposed ALRESCHA, a sparse problem accelerator that quickly reconfigures the data path during runtime to dynamically tune the hardware for distinct computation patterns and enables using high-bandwidth memory at low-cost for fast acceleration of sparse algorithms.

7. REFERENCES

[1] K. Akbudak *et al.*, "Exploiting locality in sparse matrix-matrix multiplication on many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, 2017.

[2] E. Saule *et al.*, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2013.

[3] P. D. Sulatycke *et al.*, "Caching-efficient multithreaded fast multiplication of sparse matrices," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*. IEEE, 1998.

[4] S. Dalton *et al.*, "Optimizing sparse matrixâĂŤmatrix multiplication for the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, 2015.

[5] F. Gremse *et al.*, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, 2015.

[6] W. Liu *et al.*, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014.

[7] K. Matam *et al.*, "Sparse matrix-matrix multiplication on modern architectures," in *High Performance Computing (HiPC), 2012 19th International Conference on*. IEEE, 2012.

[8] E. Phillips *et al.*, "A cuda implementation of the high performance conjugate gradient benchmark," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014.

[9] W. Liu *et al.*, "A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors," *Journal of Parallel and Distributed Computing*, vol. 85, 2015.

[10] C. Y. Lin *et al.*, "Design space exploration for sparse matrix-matrix multiplication on fpgas," *International Journal of Circuit Theory and Applications*, vol. 41, 2013.

[11] L. Yavits *et al.*, "Sparse matrix multiplication on cam based accelerator," *arXiv preprint arXiv:1705.09937*, 2017.

[12] Q. Zhu *et al.*, "Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013.

[13] A. K. Mishra *et al.*, "Fine-grained accelerators for sparse machine learning workloads," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017.

[14] E. Nurvitadhi *et al.*, "A sparse matrix vector multiply accelerator for support vector machine," in *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press, 2015.

[15] S. Pal *et al.*, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018.

[16] T. J. Ham *et al.*, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[17] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, 2016.

[18] L. Nai *et al.*, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *2017 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 2017.

[19] L. Song *et al.*, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018.

[20] B. Feinberg *et al.*, "Enabling scientific computing on memristive accelerators," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[21] R. W. Vuduc *et al.*, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *International Conference on High Performance Computing and Communications*. Springer, 2005.

[22] J. Dongarra *et al.*, "Hpcg benchmark: a new metric for ranking high performance computing systems," *Knoxville, Tennessee*, 2015.

[23] D. Ruiz *et al.*, "The hpcg benchmark: analysis, shared memory preliminary improvements and evaluation on an arm-based platform," 2018.

[24] V. Marjanović *et al.*, "Performance modeling of the hpcg benchmark," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014.

[25] G. H. Golub *et al.*, *Matrix computations*. JHU press, 2012, vol. 3.

[26] G. Malewicz *et al.*, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010.

[27] T. A. Davis *et al.*, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, 2011.

[28] Y. Saad, *Iterative methods for sparse linear systems*. siam, 2003, vol. 82.

[29] D. R. Kincaid *et al.*, "Itpackv 2d user's guide," Texas Univ., Austin, TX (USA). Center for Numerical Analysis, Tech. Rep., 1989.

[30] Y. Wang *et al.*, "Gunrock: A high-performance graph processing library on the gpu," in *ACM SIGPLAN Notices*, vol. 51, no. 8. ACM, 2016.

[31] X. Zhu *et al.*, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015.

[32] F. Khorasani *et al.*, "Cusha: vertex-centric graph processing on gpus," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014.