ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator

Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, Sudhakar Yalamanchili
Georgia Institute of Technology, Atlanta, GA
{bahar.asgari, rhadidi, tushar, hyesoon.kim, sudha}@gatech.edu

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 been 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 scientific sparse problems. Our key insight is to extract parallelism by mathematically transforming the computations into equivalent forms. The transformation breaks down the sparse kernels into a majority of independent parts and a minority of data-dependent ones and reorders these parts to gain performance. To implement the key insight, 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 lightweight reconfigurable engine for the execution of the data-dependent parts. Second, Alrescha benefits from a locally-dense storage format, with the right order of non-zero values to yield the order of computations dictated by the transformation. The combination of the lightweight reconfigurable hardware and the storage format enables uninterrupted streaming from memory. Our simulation results 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 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]. As the effectiveness issue has coupled with approaching the end of Moore's law, specialized hardware for sparse problems has become attractive. For instance, hardware accelerators have been proposed for sparse matrix-matrix multiplication [10, 11, 12, 13], matrix-vector multiplication [14, 15, 16, 17], or both [18, 19, 20]. In addition, accelerators for sparse problems have been proposed to reduce memory-access latency or improve energy efficiency [21, 22, 23, 24, 25]. Further, approaches such as blocking [26, 25, 13] have been used to reduce indirect memory accesses.

The aforementioned hardware accelerators and software-optimization techniques for sparse problems often focus on a specific domain of application and take advantage of a 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 flexibility is not just for creating more generic accelerators, but is 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 comprise two groups of kernels with contradictory features: (i) highly parallizable and (ii) data-dependent kernels. In such a case, the challenge is that while a sparse problem requires high bandwidth and a high level of concurrency, *the dependent computations prevent 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 challenge performance optimization. Such sparse problems have become a major computation pattern in many fields of science. For instance, the high-performance conjugate gradient (HPCG) [27] 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 problems that causes a performance bottleneck (details in §2). 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 processed in parallel. 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, code optimizations 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 allow us to *break down the*

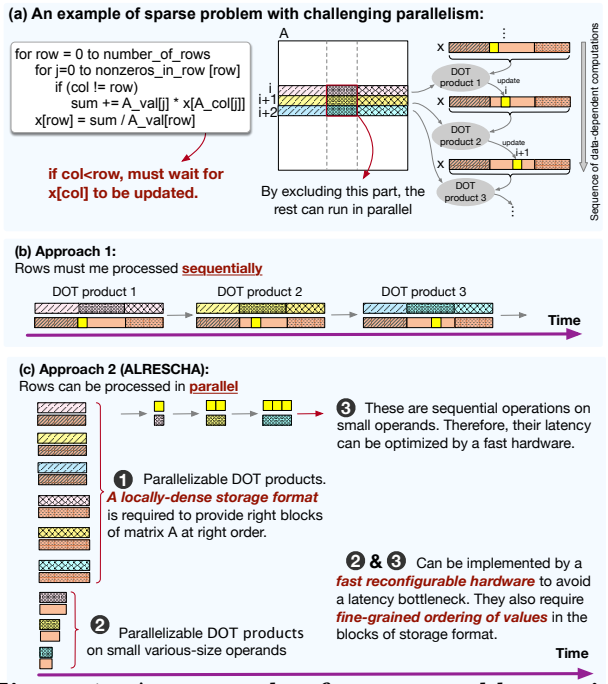


Figure 1: An example of sparse problems with data-dependency patterns in its computations.

traditionally dependent parts into a 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 key 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 computation unit for ❶ and a small reconfigurable computation unit for ❷ and ❸.
- **Locally-dense storage format:** Since Alrescha reorders the computations to process the parallel parts together ❶, we propose a storage format in which *the order of non-zero values matches the order of computations.* The storage format enables streaming the ordered blocks of matrix A in the right timing. Besides, such a storage format facilitates ❷ and ❸ by dictating required element-wise ordering. Alrescha integrates the proposed storage format with a data-driven execution model to eliminate the transferring and decoding of meta-data.
- **Generic sparse accelerator:** The ability of Alrescha to accelerate distinct kernels makes it the first generic hardware accelerator for both scientific and graph applications regardless of whether they have data-dependent compute patterns. For applications with no distinct kernels, Alrescha still sustains high performance by integrating the proposed storage format and the execution model, hence avoiding meta-data transfer.

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

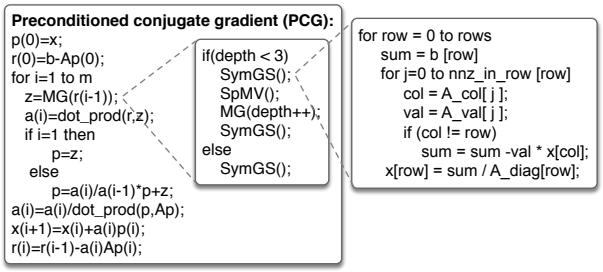


Figure 2: An example of the PCG algorithm [27] for solving a sparse linear system of equations (i.e., $Ax = b$), including SpMV and SymGS.

Alrescha accelerates various sparse kernels such as sparse matrix-vector multiplication (SpMV), symmetric Guass-Seidel (SymGS) smoother, PageRank (PR), breadth-first search (BFS), and single-source shortest path (SSSP). To evaluate Alrescha, we target a wide range of data sets 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 hardware accelerators for sparse problems, namely, OuterSPACE [18], an accelerator for SpMV, GraphR [24], and a Memristive accelerator for scientific problems [25]. Our experiment results on various data sets show that compared to state-of-the-art, Alrescha achieves an average speed up of 2.1×, 1.87×, 1.7× for scientific algorithm, graph analytics, and SpMV, respectively. The performance gain on data sets with various distributions of non-zero values indicates that the gained benefits of Alrescha are independent of specific patterns in sparse data.

2. BACKGROUND

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

Scientific Problems: Many physical-world phenomena, such as sound, heat, elasticity, fluid dynamics, and quantum mechanics, are 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 equations: $Ax = b$, 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 [27]. 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 [28, 8, 29]. The remaining kernels, such as the 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.

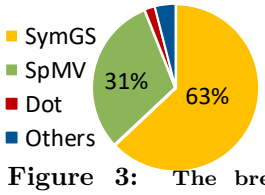


Figure 3: The breakdown of execution time of PCG on NVIDIA K20.

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 $A_{m\times n}$) is calculated as follows, based on the Gauss-Seidel method [30]:

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

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.

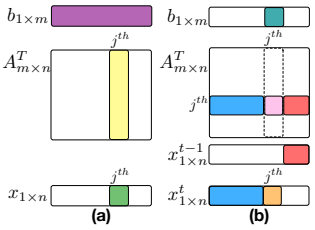


Figure 4: Calculation of (a) x_j in SpMV, and (b) x_j^t in SymGS.

Graph Analytics: A common approach to represent graphs is to use an adjacency matrix, each element of which represents an edge in the graph (Figure 5 illustrates an example of such a representation). 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 is implemented as a form of a dense-vector sparse-matrix operation. Such implementations are suited to the vertex-centric programming model [31], 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.

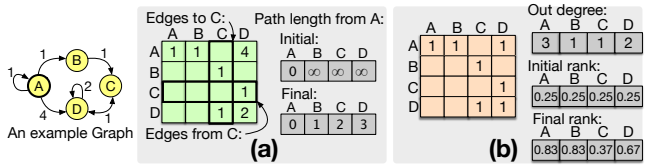


Figure 5: An example graph, its adjacency sparse matrices, and the vector operands of two graph algorithms: (a) SSSP and (b) PR.

Three widely used graph algorithms are BFS, SSSP, PR. 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 nodes. 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.

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 concurrency, 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, and more important, common feature of these kernels is that they follow the three phases of operations iteratively (i.e., vector operation, reduce, and assign). 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

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

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

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 the common features to design an accelerator that is flexible to run all the mentioned sparse kernels without significant overhead. Alrescha converts the sparse kernels into the dense data paths, listed in the second column of Table 1 (details in §4.1).

3. MOTIVATION AND RELATED WORK

Sparse problems in either sparse (i.e., non-compressed) or compressed representation face many performance challenges. The sparse formats are less efficient because they require storing, transferring, and processing of zero values. On the other hand, even the highly preferred compressed representations still rely on transferring meta-data and random memory accesses that limit performance by memory-access latency. Besides, as the ratio of compute-per-data-transfer of many sparse problems (i.e., those including vector-matrix operations) is low, normally, we expect their performance to be directly related to the memory bandwidth. However, gaining higher performance for sparse problems is not as straightforward as adding more memory bandwidth. This section sheds more light on the challenges that sparse problems – specifically those with data-dependent computations – face when being executed on modern CPUs, GPUs, or even on hardware accelerators.

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 [26] have not been effective enough for the aforementioned reason. Figure 6 summarizes the performance of running sparse scientific applications on a range of CPUs and GPUs. As the figure shows, they utilize only a tiny fraction of the peak performance.

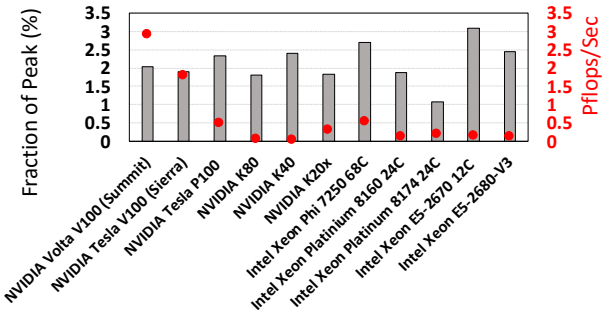


Figure 6: The performance of modern computing platforms ranked by the standard metric of HPCG [27] benchmark on GPUs and CPUs.

Prior Hardware Accelerators: The ineffectiveness of CPUs and GPUs, along with approaching the end of Moore's law, has motivated the migration to specialized hardware for sparse problems. For instance, hardware accelerators have targeted sparse matrix-matrix multiplication [10, 11, 12, 13], matrix-vector multiplication [14, 15, 16, 17], or both [18, 19, 20], which are the main sparse kernels in many sparse problems. A state-of-the-art SpMV accelerator, OuterSPACE [18], employs an outer-product algorithm to minimize the redundant accesses to non-zero values of the sparse matrix. Despite the speedup of OuterSPACE over the traditional SpMV by increasing the data reuse rate and reducing memory accesses, it produces random access to a local cache. To efficiently utilize memory bandwidth, Hegde et al. proposed Extensor [19], a novel fetching mechanism that avoids the memory latency overhead associated with sparse kernels. Song et al. proposed GraphR [24], a graph accelerator, and Feinberg et al. proposed a scientific-problem accelerator [25], both of which process blocks of non-zero values instead of individual ones. Besides, Huang et al. have proposed analog [32] and hybrid (analog-digital) [33] accelerator for solving PDEs.

The prior specialized hardware designs often have not focused on resolving the challenge of data-dependent computations in sparse problems that prevent benefiting from the available memory bandwidth. Sparse problems may involve a combination of contradictory parallizable and data-dependent kernels. The flexibility to accelerate both types of kernels is necessary not only to improve the overall performance but also to be more generic. Table 2 compares the most relevant hardware approaches and techniques for accelerating sparse problems with Alrescha. As the table lists, *Alrescha is the first reconfigurable sparse problem accelerator for both scientific and graph applications*, which supports multi-kernel execution and resolves the limited parallelism in fine granularity. Besides, unlike prior work, Alrescha reduces the number of accesses to the vector operand of the sparse matrix-vector operations.

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

		GraphR [24]	OuterSPACE [18]	Memristive-Based Accelerator [25]	Row Reordering Matrix Coloring [8]	Alrescha (our work)
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 sparsity pattern)	✓

4. ALRESCHA

Alrescha is a memory-mapped accelerator. The memory dedicated to Alrescha is also accessible by a host for programming. Figure 7 shows an overview of Alrescha, the host, and the connections for programming and transferring data. The programming model of Alrescha is similar to offloading computations from a CPU to a GPU (the programming model is beyond the scope of this paper). To program the accelerator, the host launches the sparse kernels of sparse algorithms (i.e., PCG and graph algorithms) to the accelerator. To do so, the host first converts the sparse kernels into a sequence of *dense data paths* and generates a binary file. Then, the host writes the binary file to a configuration table of the accelerator through the *program interface*. The details of the conversion mechanism and the dense data path implementation are explained in §4.1 and §4.2.

During the execution of an algorithm, repetitive switching between the dense data paths is required. The key feature of Alrescha to enable fast switching among those dense data paths is the runtime reconfigurability. The details of the reconfigurable microarchitecture of Alrescha and the mechanism of real-time reconfiguration are explained in §4.3 and §4.4. Besides switching among the data paths during runtime, Alrescha also reorders the dense data paths to reduce the number of switches. Such a reordering necessitates a new storage format, which is introduced in §4.5. Therefore, the other task of the host is to reformat the sparse matrix operands into a storage format consisting of blocks, each of which corresponds to a dense data path. The formatted data is written into the physical memory space of the accelerator through the *data interface* (Figure 7).

Since the target algorithms are iterative, the preprocessing (i.e., conversion and reformatting) is a one-time overhead. Besides, the complexity and effort of preprocessing depend on the previous format, data source, and host platform. For instance, the conversion complexity from frequently-used storage formats (e.g., CSR and BCSR) is linear in time and requires constant space. Since the preprocessing complexity is linear, it can be done while data streams from the memory. Moreover, if data is generated in the system (e.g., through sensors), it is initially be formatted in the Alrescha format and reformatting is not required.

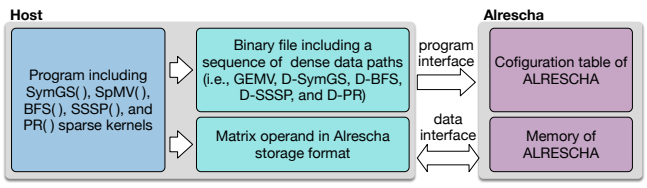


Figure 7: The overview of Alrescha and host.

4.1 Kernel to Data Path Conversion

Algorithm 1 shows the procedure for converting a sparse matrix to dense data paths. Based on the kernel type, the sparse matrix operand ($A_{n\times n}$), and the dimension of the non-zero blocks in the matrix operand (ω), the algorithm generates the configuration table, each row of which specifies the type of data path and information about its operands. More specifically, for each dense data path, besides its type (i.e., DP), illustrated by one bit, the index of the input vector operand (Inx_{in}), the index of the output vector (Inx_{out}), the access order, which can be left to right ($l2r$) or right to left ($r2l$), and the source of the operand (Op) are stored. In other words, the Inx_{in} and Inx_{out} indicate the read and write addresses of a local cache, respectively; Op selects the local cache containing the vector operand. As a result, all mentioned meta data in a row of the configuration table takes $2[\log_2(n/\omega)]+3$ bits. The three bits are for the data path type, access order, and operand source, respectively. The configuration table is used for configuring a *configuration switch* (details in §4.3).

The general procedure of the conversion algorithm is as follows: (i) As lines 8 to 12 show, the sparse kernels with no (or straightforward) data dependencies including 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 kernels do; 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 original sparse kernel. (ii) As lines 13 to 26 show, the sparse kernels with data dependencies (e.g., SymGS kernel, the key computation of PCG algorithm) are broken down into a majority of parallelizable GEMV (lines 14 to 21) and a minority of sequential dense SymGS (D-SymGS) data paths (lines 23 to 26).

The conversion for SymGS is to assign GEMVs to non-diagonal non-zero blocks (line 15) and D-SymGS to diagonal non-zero blocks of the sparse matrix (line 23).

For accelerating SymGS, the key insight of Alrescha is to separate GEMV from D-SymGS data paths to prevent the performance from being limited by the sequential nature of the SymGS kernel. To this end, Alrescha reduces switching between the two data paths (GEMV and D-SymGS) by reordering them so that Alrescha first executes all the GEMVs in a row successively and then switches to a 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. The configuration table for a SymGS example is shown in Figure 8. Based on Equation 2 and as lines 19 and 21 of Algorithm 1 indicate, 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 explains the reasoning for the listed access orders in Figure 8.

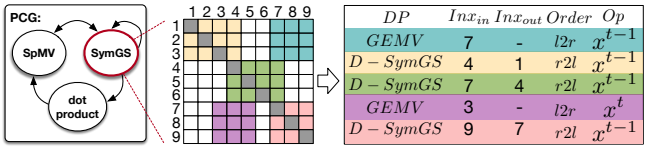
Algorithm 1 Convert Algorithm

```
1:  function CONVERT(KernelType, A_nxn, ω)
        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
17:                         Inx_out = −1 // no write to cache
18:                         Order = l2r
19:                         if (i > j) then
20:                             Op = port2 //which is x^{t−1}
21:                         else
22:                             Op = port1 //which is x^t
23:                     else
24:                         DP = D-SymGS
25:                         Inx_in = j.ω, Inx_out = (i+1).ω
26:                         Order = r2l
27:                         Op = port2 //which is x^{t−1}
28:                 Add2Table(DP, Inx_in, Inx_out, Order, Op)
```

4.2 Dense Data Path Implementation

Alrescha implements two classes of dense 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. This section describes the implementations of the two classes of data paths.

(a) Straightforward data paths: These data paths, such as D-BFS, D-SSSP, D-PR, and GEMV, work on locally dense blocks of A to capture locality in accesses to the dense vector operand. If the sparse matrix operand includes blocks of non-zero values of size ω, Alrescha splits the vector operand into chunks of size ω, and at each cycle, it fetches a chunk 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 ω 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 sparse kernels 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 the vector operand is fetched from the cache only once per n/ω, in which n in the number of columns of A.



DP	Inx_{in}	Inx_{out}	Order	Op
GEMV	7	-	l2r	x^{t-1}
D − SymGS	4	1	r2l	x^{t-1}
D − SymGS	7	4	r2l	x^{t-1}
GEMV	3	-	l2r	x^t
D − SymGS	9	7	r2l	x^{t-1}

Figure 8: An example of the configuration table for a SymGS kernel, in which $n = 9$, $\omega = 3$.

(b) Data-dependent data paths: SymGS, an example of a sparse kernel with such dependencies, includes 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 are 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 element of x^t includes two inner products, the sizes of which change when we move across the rows of A. To enable using a compute engine for both inner products, Alrescha merges them by factoring out the subtraction:

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

which consists of a unified *dot product* of size n, the common operation in both classes of the dense data paths. As a result of the *common operation*, by transforming Equation 2 to Equation 3, the D-SymGS kernel can utilize *the same dot product* engine used by other dense data paths (i.e., the first class of data paths). The only difference between the dot product of D-SymGS and that in other data paths is the sources of the inputs and the destination of the output. In detail, one operand of the dot product of D-SymGS is made by shifting x^t into x^{t-1}. This can be implemented by just rotating the inputs of the multipliers and pushing the x_j^t into the first multiplier to be used in calculating x_{j+1}^t.

To clarify the functionality of Alrescha for D-SymGS, we use a simple example in Figure 10, for which we assume that the size of the problem fits the size of the hardware (i.e., the number of multipliers). As Figure 10 illustrates, Alrescha stores the diagonal of A in a separate vector and does the subtraction with b in parallel with the dot product. Note that separately storing the diagonal of A helps utilize the memory bandwidth only for streaming the operand of the dot product engine. At the first step, one row of the non-diagonal elements of

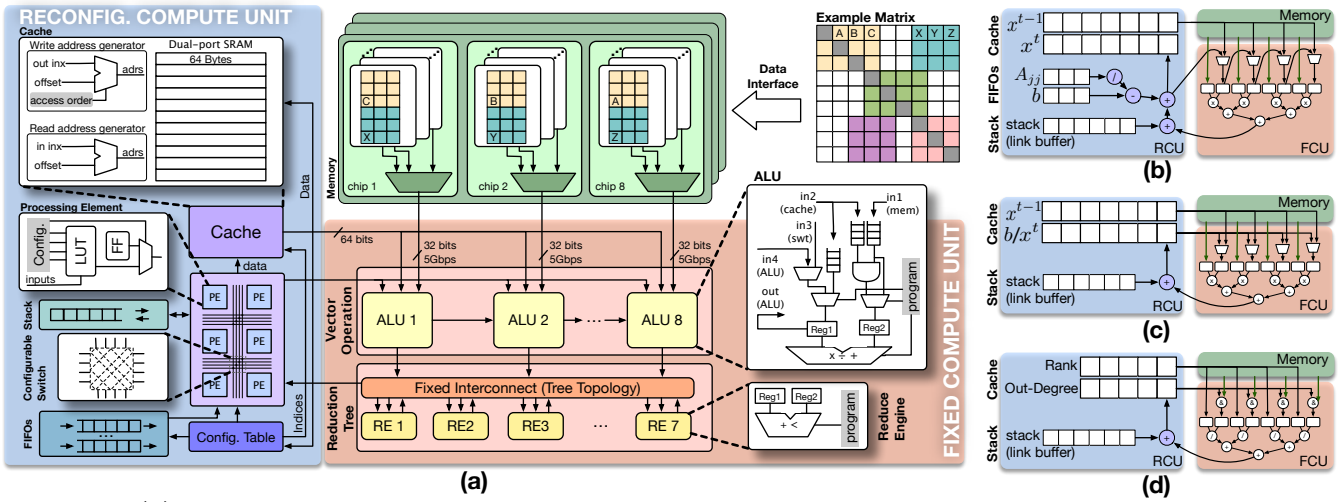


Figure 9: (a) The microarchitecture of Alrescha including the FCU for implementing common computations, and the RCU for providing specific configuration for distinct dense data paths. And, three example configurations for supporting: (b) D-SymGS, (c) GEMV, and (d) PR.

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 the kicked-off x_1^{t-1}. As a result, we insert the new variables by shifting the old one to the right. As Figure 10 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.* Such an ordering, which is listed in the configuration table of Figure 8, is also reflected in the storage format (details in Figure 13 in §4.5).

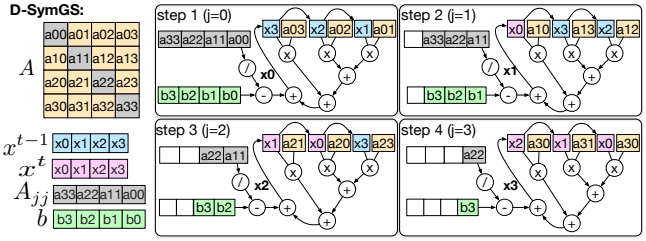


Figure 10: An example of D-SymGS. At each step, one x_j^t is calculated based on Equation 3. The x_j^t is then used in calculating the next element x_{j+1}^t at the next step. The arrows among the operands of the dot product indicate shifting them one to the right at each step.

4.3 Reconfigurable Microarchitecture

This section introduces the microarchitecture of Alrescha to accelerate the sparse kernels by executing their fundamental data paths (i.e., D-SymGS, GEMV, D-BFS, D-SSSP, and D-PR). To deliver close-to-ideal performance, the main idea behind the hardware of Alrescha is to have a separate fixed computation unit (FCU) and a reconfigurable computation unit (RCU) and configuring only the former for switching between data paths (Figure 9). Reconfiguring only a fraction of the entire data path reduces the configuration time.

The FCU streams *only the dense blocks of the sparse matrix* (i.e., no 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 9, 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 input/s (i.e., the vector operands) comes 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 during initializing the D-SymGS does that input come from the cache (i.e., x^{t-1}); after the initialization, that input comes from a processing element in the RCU and a forwarding connection between the inputs of the multipliers. For GEMV, on the other hand, 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 in 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 similarly to those of FPGAs and is directly controlled by the content of a configurable table (§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 vector operands, which require deterministic accesses. For instance, we employ first-in-first-out (FIFO) for A_{jj}^T and b, and use a last-in-first-out (LIFO) stack for the link buffer. The link buffer establishes transmissions between the dense data paths (details in §4.4). For data path transmission, the reduction tree has to be drained, during which the switch is reconfigured to prepare it 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 9b, c, and d illustrate the configuration of the RCU for performing D-SymGS, GEMV, and D-PR data paths.

4.4 Real-Time Reconfiguration

The algorithms with a sequence of distinct sparse kernels 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, meaning that only the configuration of a small fraction of the compute engine is changed frequently. Switching between two data-dependent data paths is performed using the link buffer (i.e., implemented as the stack). During runtime, the intermediate results generated by GEMV are pushed into the stack. Then, the successive D-SymGS 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 the results are pushed into the link stack. At step 3, while the reduction tree (adder tree) is drained, the interconnect between the cache and FIFOs in the RCU is reconfigured to D-SymGS. This does not affect the pushes to the stack, so it can be parallelized and hides the latency. In steps 4 to 6, D-SymGS pops the results of the GEMV from the link stack and consumes them.

4.5 Storage Format

Depending on the distribution of non-zeros in a sparse matrix, various storage formats may suit them. Figure 12 compares four well-known formats based on meta-data per non-zero. The compressed sparse row (CSR), which stores a vector of column indices and a vector of row pointers, locates all the non-zeros independently. Therefore, CSR 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) [35], which stores the non-zeros sequentially, could be the best option. An extension to the DIA format, Ellpack-Itpack (ELL) [36] is more flexible when the matrix has a combination of non-diagonal and diagonal elements. For instance, ELL is used for

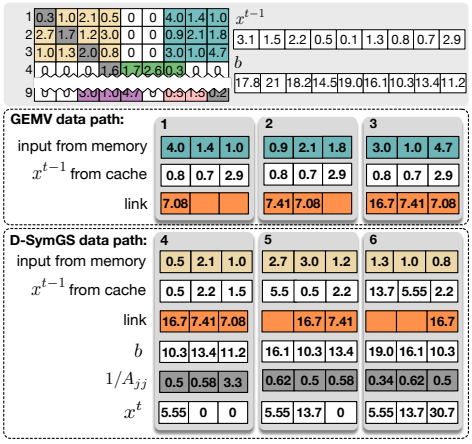


Figure 11: Switching between GEMV and D-SymGS data paths using the link stack.

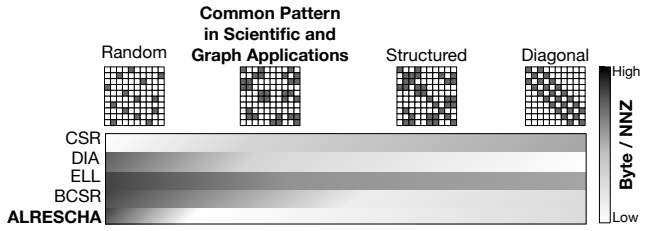


Figure 12: The spectrum of storage formats for various types of sparse matrices (low is better).

implementing SymGS in GPUs. However, such a format does not provide enough flexibility for parallelizing rows because it does not sustain the locality across rows.

The choice of storage format *should be compatible with the common pattern in a wide range of sparse applications.* In such a case, blocked CSR (BCSR) [26], an extension of CSR, which assigns the column indices and row pointers to blocks of non-zero values, is the right format. Although BCSR is an appropriate format for scientific applications and graph analytics in terms of storage overhead, the strategy of BCSR for assigning indices and pointers, and the order of values, is not a good match for smoothly streaming data. In other words, the main requirement for fast computation is the order of operations, which in turn dictates the data structures to be streamed in the same order. Thus, we adapt BCSR and propose a new storage format with the same meta-data overhead but compatible with the reconfigurable microarchitecture of Alrescha.

Figure 13, illustrates our proposed locally-dense format for mapping the example sparse matrix to the physical memory addresses of the accelerator. ***The Order of Blocks:*** All the non-diagonal non-zero blocks in a row of blocks are stored together, followed by a diagonal block; ***The Order of Values:*** The non-zero values belonging to the upper triangle of the non-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 column indices of BSCR and input indices (i.e., Inx_{in}) of the Alrescha format is shown in Figure 13; ***The Diagonal Elements:*** For SymGS, the diagonal of A is excluded and stored separately in a local cache. Therefore, we

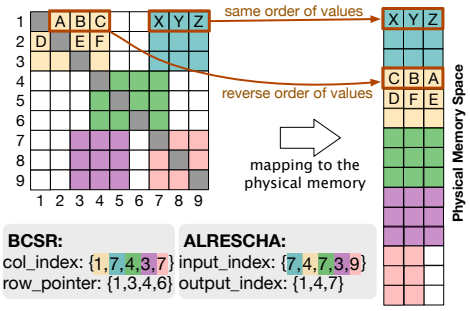


Figure 13: Comparing Alrescha with BCSR: The col_index of BCSR and input_index (i.e., Inx_{in}) of Alrescha are color-coded to show their corresponding blocks in the matrix. Alrescha uses the index of the last column for the input index of diagonal blocks.

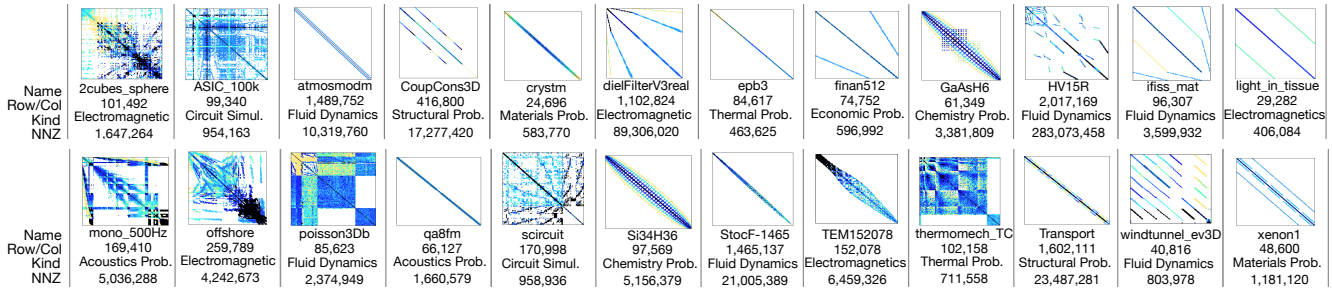


Figure 14: Scientific application data sets attained from University of Florida SuiteSparse [34].

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

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. ***Meta Data:*** the indices of the input and output (i.e., Inx_{in} and Inx_{out}) are not streamed from memory during runtime. Instead, they are stored in the configuration table during the one-time programming phase and are used for reconfiguration purposes. As a result, during the iterative execution of the algorithms, the whole available memory bandwidth is utilized only 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, Algorithms, and Baselines

We pick real-world matrices with applications in scientific and graph problems from the University of Florida SuiteSparse Matrix Collection [34]. 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 the number of non-zeros (NNZ). We run PCG, which includes the SymGS and SpMV kernels, on the matrices listed in Figure 14, and run graph algorithms (i.e., BFS, SSSP, and PR) on the matrices in Table 3. We also run SpMV on both categories of datasets.

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, we exclude disk access time. For fair comparisons, we include necessary optimizations, such as row reordering and suitable storage formats (e.g. ELL) 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 [37] libraries. The graph algorithms running on the CPU are based on the GridGraph [38] and/or CuSha [39] 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 [37] 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 [39], GridGraph [38]

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

Besides the comparison with the CPU and GPU, this section compares Alrescha with the state-of-the-art hardware accelerators, including OuterSPACE [18], an accelerator for SpMV, GraphR [24], a ReRAM-based graph accelerator, and a Memristive accelerator for scientific problems [25]. 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. Moreover, for fair comparison, we assign all the accelerators the same computation and memory-bandwidth budget – this assumption does not harm the performance of our peers.

5.2 Experimental Setup

We convert the raw matrices using Algorithm 1 implemented in Matlab. To do that, we examine block sizes of 8, 16, and 32 for the range of data sets and choose the block size of eight because, unlike the other two, 8 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 28 nm standard cell and the SRAM library at 200 MHz. The reported numbers include programming the accelerator.

5.3 Execution Time

Scientific Problems: The primary axis of Figure 15 (i.e., the bars) illustrates the speedup of running PCG on Alrescha over the GPU implementation optimized by row reordering [8] for extracting a high level of parallelism; the secondary axis of Figure 15 shows the bandwidth utilization. The figure also captures the speedup of the Memoristive-based hardware accelerator [25]. On average, Alrescha provides a 15.6× speedup compared to the optimized implementation 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 all scientific datasets is directly related to memory bandwidth utilization – mainly because of the sparsity 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.

To clarify the impact of resolving dependencies on overall performance, Figure 16 presents the percentage of sequential computations in the GPU implementation,

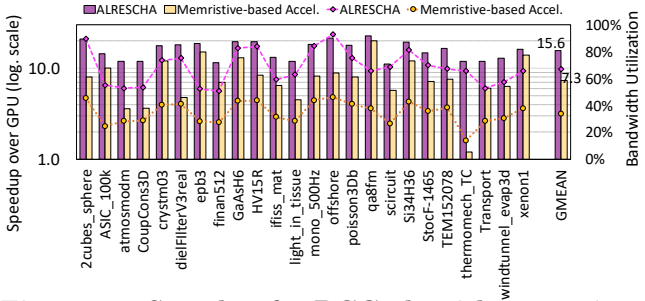


Figure 15: Speedup for PCG algorithm on scientific datasets, normalized to GPU (bar charts), and bandwidth utilization (the lines). The Memristive-based accelerator [25] is the state-of-the-art accelerator for scientific problems.

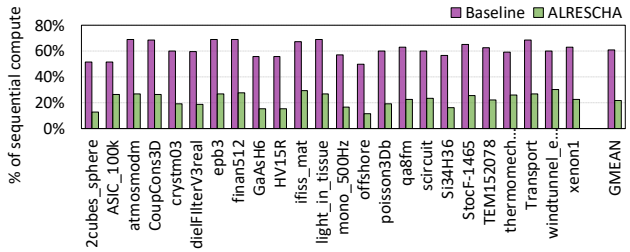


Figure 16: The reduction of the sequential part of the PCG algorithm by applying Alrescha. The baseline shows the percentage of sequential operations by row-reordering optimization.

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 a matrix offers the opportunity for parallelism, the speedup over the GPU is smaller compared to when the non-zeros are mostly distributed in the diagonal.

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

Graph Analytics And SpMV: This section explores the performance of the algorithms consisting of one type of sparse kernel that naturally has fewer data dependency patterns in their computations. Such a study claims that Alrescha is not just optimized for a specific domain and is applicable to accelerating a wide range of sparse applications. First, we analyze the performance of graph applications. Figure 17 illustrates the speedup of running BFS, SSSP, and PR on Alrescha, a recent hardware accelerator for graph (i.e., based on GraphR [24]), and GPU, all normalized to the CPU.

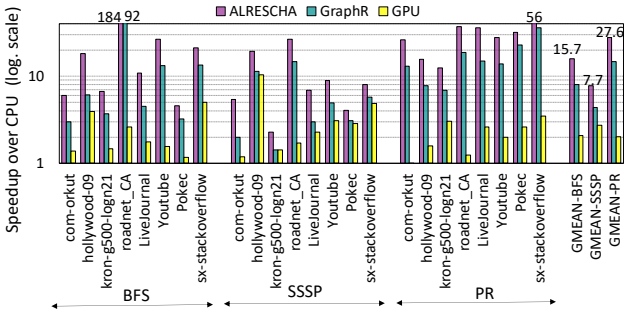


Figure 17: Speedup for graph algorithms on graph data sets over the CPU. GraphR [24] is the state-of-the-art graph accelerator.

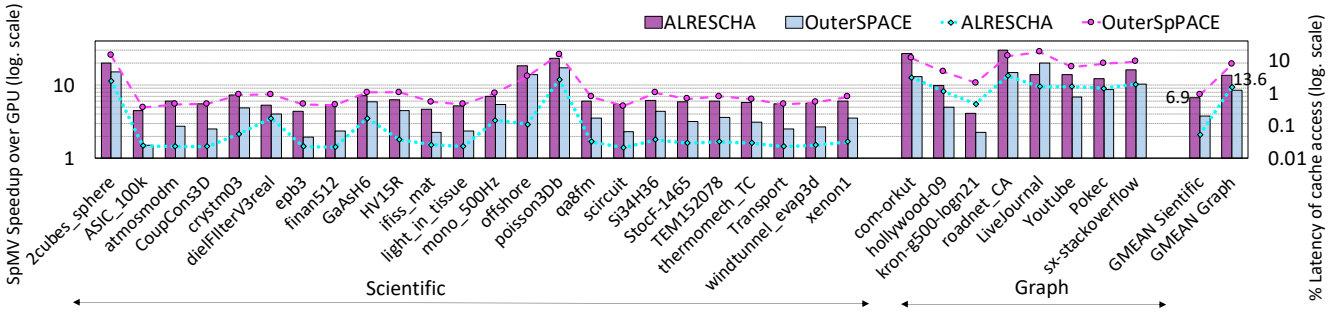


Figure 18: Speedup for running SpMV on scientific and graph datasets normalized to GPU (bar charts), and the percentage of execution time devoted to cache accesses (the lines). OuterSPACE [18] is the state-of-the-art SpMV accelerator.

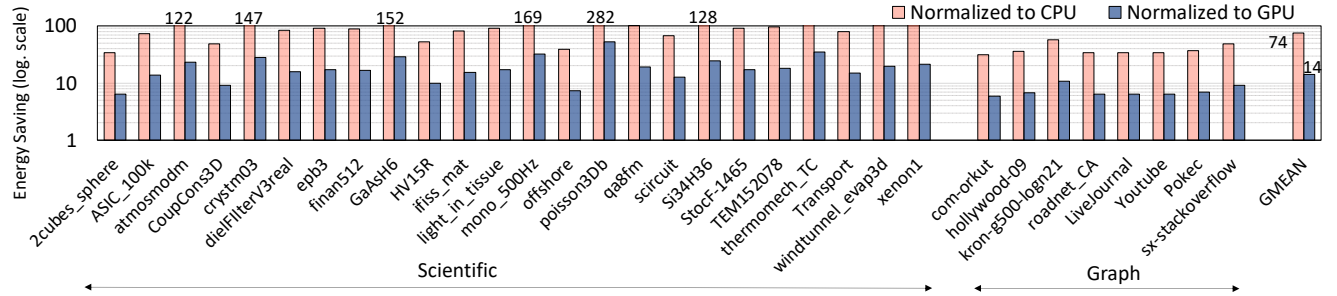


Figure 19: Energy consumption improvement of Alrescha normalized to that of CPU and GPU.

As the figure shows, Alrescha offers average speedups of 15.7×, 7.7×, and 27.6×, for BFS, SSSP, and PR algorithms, respectively. We achieve this speedup by avoiding the transfer of meta-data, reordering the blocks for increasing data reuse, and improving the locality.

The primary axis of Figure 18 (i.e., the bars) illustrates the speedup of SpMV, a common algorithm of various sparse applications on Alrescha and OuterSPACE [18] (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 corresponding element of the output vector. Hence, unlike Alrescha, the computation engine of OuterSPACE has to put the partial products in their right location in the output vector, which may lead 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.

5.4 Energy Consumption

A primary motive for using hardware accelerators rather than software optimizations is to reduce energy consumption. To achieve this goal, the techniques integrated in the hardware accelerators have to be efficient. Since a source of energy consumption is accesses to local SRAM-based buffers or caches, reducing the number of reads and writes from and to local caches by substituting them with computation is beneficial. Figure 19 illustrates the energy consumption of Alrescha for executing SpMV, 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. 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 problems.

Acknowledgment

We would like to thank the anonymous reviewers and gratefully acknowledge the support of NSF CSR 1526798.

7. REFERENCES

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

[2] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "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 and K. Ghose, "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, L. Olson, and N. Bell, "Optimizing sparse matrixâĂŤmatrix multiplication for the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, 2015.

[5] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, 2015.

[6] W. Liu and B. Vinter, "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, S. R. K. B. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *HiPC, 2012 19th International Conference on*. IEEE, 2012.

[8] E. Phillips and M. Fatica, "A cuda implementation of the high performance conjugate gradient benchmark," in *PMBS*. Springer, 2014.

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

[10] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays," in *DAC*. ACM, 2019.

[11] C. Y. Lin, N. Wong, and H. K.-H. So, "Design space exploration for sparse matrix-matrix multiplication on fpgas," *International Journal of Circuit Theory and Applications*, vol. 41, 2013.

[12] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware," in *HPEC, 2013 IEEE*. IEEE, 2013.

[13] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Eridanus: Efficiently running inference of dnns using systolic arrays," *IEEE Micro*, 2019.

[14] B. Asgari, R. Hadidi, and H. Kim, "Ascella: Accelerating sparse computation byenabling stream accesses to memory," in *DATE*. IEEE, 2020.

[15] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, "Fine-grained accelerators for sparse machine learning workloads," in *ASP-DAC*. IEEE, 2017.

[16] E. Nurvitadhi, A. Mishra, and D. Marr, "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.

[17] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G.-Y. Wei, and D. Brooks, "Masr: A modular accelerator for sparse rnns," in *PACT*. IEEE, 2019.

[18] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *HPCA*. IEEE, 2018.

[19] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *MICRO*. ACM, 2019.

[20] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *MICRO*. ACM, 2019.

[21] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*. IEEE, 2016.

[22] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, 2016.

[23] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *HPCA*. IEEE, 2017.

[24] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *HPCA*. IEEE, 2018.

[25] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *ISCA*. IEEE, 2018.

[26] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *HPCC*. Springer, 2005.

[27] J. Dongarra, M. A. Heroux, and P. Luszczek, "Hpcg benchmark: a new metric for ranking high performance computing systems," *Knoxville, Tennessee*, 2015.

[28] D. Ruiz, F. Mantovani, M. Casas, J. Labarta, and F. Spiga, "The hpcg benchmark: analysis, shared memory preliminary improvements and evaluation on an arm-based platform," 2018.

[29] V. Marjanović, J. Gracia, and C. W. Glass, "Performance modeling of the hpcg benchmark," in *PMBS*. Springer, 2014.

[30] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU press, 2012, vol. 3.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010.

[32] Y. Huang, N. Guo, M. Seok, Y. Tsividis, and S. Sethumadhavan, "Analog computing in a modern context: A linear algebra accelerator case study," *IEEE Micro*, vol. 37, 2017.

[33] Y. Huang, N. Guo, M. Seok, Y. Tsividis, K. Mandli, and S. Sethumadhavan, "Hybrid analog-digital solution of nonlinear partial differential equations," in *MICRO*. IEEE, 2017.

[34] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM TOMS*, vol. 38, 2011.

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

[36] D. R. Kincaid, T. C. Oppe, and D. M. Young, "Itpackv 2d user's guide," Texas Univ., Austin, TX (USA). Center for Numerical Analysis, Tech. Rep., 1989.

[37] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *ACM SIGPLAN Notices*, vol. 51, no. 8. ACM, 2016.

[38] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *USENIX ACT*, 2015.

[39] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *HPDC*. ACM, 2014.