Efficiently Solving Partial Differential Equations in a Partially Reconfigurable Specialized Hardware

Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, *Member, IEEE*, and Sudhakar Yalamanchili, *Fellow, IEEE*

Abstract—Scientific computations with a wide range of applications in domains such as developing vaccines, forecasting the weather, predicting natural disasters, simulating aerodynamics of spacecraft, and exploring oil resources, create the main workloads of supercomputers. The key integration of such scientific computations is modeling physical phenomena that are done with the aid of partial differential equations (PDEs). Solving PDEs on supercomputers, even with those equipped with GPUs, consumes a large amount of power and yet is not as fast as desired. The main reason behind such slow processing is data dependency. The key challenge is that software techniques cannot resolve these dependencies, therefore, such applications cannot benefit from the parallelism provided by processors such as GPUs. Our key insight to address this challenge is that although we cannot resolve the dependencies, we can reduce their negative impacts by using hardware/software co-optimization. To this end, we propose breaking down the data-dependent operations into two groups of operations: a majority of parallelizable and the minority of data-dependent operations. We execute these two groups in the desired order: first, we put together all parallelizable operations and execute them all, subsequently; then, we switch to execute the small data-dependent part. As long as the data-dependent part is small, we can accelerate them by using fast hardware mechanisms. Besides, our proposed hardware mechanisms guarantee quickly switching between the two groups of operations. To follow the same order of execution, dictated by our software mechanism, and implemented in hardware, we also propose a new low-overhead compression format – sparsity is another attribute of PDEs that require compression. Furthermore, the core generic architecture of our proposed hardware allows the execution of other applications including sparse matrix-vector multiplication (SpMV) and graph algorithms. The key feature of the proposed hardware is partial reconfigurability, which on one hand, facilitates the execution of data-dependent computations, and on the other hand, allows executing broad application without changing the entire configuration. Our evaluations show that compared to GPUs, we achieve an average speedup of 15.6× for scientific computations while consuming 14× less energy.

Index Terms—Scientific computation, partial differential equations, data dependency, sparsity, partial reconfigurability

1 Introduction

Scientific computations are the main component in several crucial domains such as exploring biological molecules, forecasting the weather, predicting natural disasters, and simulating aerodynamics that mainly rely on modeling physical phenomena. Such computations remarkably impact human life. Modeling/simulating a vaccine or predicting an earthquake are examples of computations that can save lives if done accurately and in a timely manner. However, modern high-performance computers equipped with CPUs and/or GPUs are poorly suited to these problems, utilizing a tiny fraction of their peak performance (e.g., 0.5 - 3 percent) [1]. Such low performance stems from the slow process of solving partial differential equations (PDEs), a key tool used in mathematical modeling.

- *Bahar Asgari, Tushar Krishna, and Sudhakar Yalamanchili are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA. E-mail: {bahar.asgari, sudha}@gatech.edu, tushar@ece.gatech.edu.*
- *Ramyad Hadidi and Hyesoon Kim are with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332 USA. E-mail: rhadidi@gatech.edu, hyesoon@cc.gatech.edu.*

Manuscript received 1 Aug. 2020; revised 12 Jan. 2021; accepted 31 Jan. 2021. Date of publication 19 Feb. 2021; date of current version 11 Mar. 2021. (Corresponding author: Bahar Asgari.) Recommended for acceptance by J. Yang and Y. Solihin. Digital Object Identifier no. 10.1109/TC.2021.3060700

The *key challenge* of solving PDEs is that the common methods to solve them capture patterns of data dependencies preventing them from utilizing high-level parallelism provided by modern processors such as GPUs even by the aid of software techniques. For instance, our *key observation* suggests that the operations in a PDE solver are only partially dependent. Therefore, we can potentially break down the data-dependent operations into a majority of parallelizable operations and a minority of data-dependent ones. Although such a technique increases the chance of parallelism, it cannot be effective, as long as the dependencies still exist. Thus, software optimization, not only cannot improve the performance but also may cause it to drop by adding extra overheads.

Our *key insight* to address the challenge is that, even though we cannot resolve the data dependencies, we can minimize their negative impact on overall performance by co-optimizing software and hardware. More specifically, we propose (i) extracting the true small data-dependent operations in the *software*, and (ii) accelerating the execution of small data-dependent operations in *hardware*. Besides accelerating the data dependent part, our approach helps improving performance by putting together the none data-dependent operations and executes them concurrently in a non-stop streaming manner before switching to the execution of data-dependant operations. Such a modification in the execution order does not impact the correctness of the

computations, because of the distributive property of inner products.

In summary, this paper makes four *main contributions*:

- *Hardware mechanisms:* By employing simple hardware components such as last-in-first-out (LIFO) and/or first-in-first-out (FIFO) buffers and shift arrays in the proposed architecture, we guarantee smooth concurrent execution of none data-dependent operations, fast switching between the two groups of distinct operations, and fast execution of the data-dependent operations.
- *Compression format:* PDEs are often sparse and require a compression mechanism. In our case, an appropriate compression format that follows the execution order, dictated by the software optimization would help prevent the extra overhead of decompression and guarantee smooth streaming of data from memory into the hardware. To this end, we propose a new compression format similar to the commonly used format, blocked compressed sparse row (BCSR). The difference between BCSR and our proposed format is that is captures the particular order of blocks and order of entries in the blocks. As a result, our proposed compression format incurs the same amount of metadata overhead as BCSR does.
- *Broad applications:* If we exclude the particular hardware mechanism used for handling the execution of data-dependent operations, our core hardware engine is generic enough to be used for accelerating many other applications including sparse matrix-vector multiplication (SpMV) and graph algorithms.
- *Partial reconfiguration:* Not only the distinct operations in scientific computations but also the kernels in several various applications, all require the same core hardware engine. Given this fact our proposed hardware captures the key feature of partial reconfigurability, the benefits of which is two-fold: first, subsequential same-type concurrent operations can be executed without frequent interruptions for instruction decoding or data-path selection; second, executing different applications (or one multi-kernel application including distinct kernels) can be done without needing to change the entire configuration.

We evaluate our proposed work on a wide range of workloads from various domains of scientific computations and graph analytics. Our comparisons with a CPU and a GPU platform show that we achieve an average speedup of 15.6× for scientific problems and 8× for graph algorithms. Besides, our proposed system consumes 14× less energy compared to a GPU. Furthermore, our comparison with the state-of-the-art hardware accelerators for scientific problems [2], for SpMV [3], and graph analytics [4] suggest that we can achieve an average speedup of 1.7×, 2.1×, 1.87×, respectively. We also recommend implementing our proposed hardware on FPGAs to leverage their partial reconfigurability feature, a key feature that has been around for over a decade, but fewer applications have been proposed to use it. For demonstration, we implement our architecture on XC7A200T Xilinx FPGA. Finally, we discuss the further applications of our proposed approach to accelerate a more complex mathematical computation, matrix inversion, another key component in scientific computation.

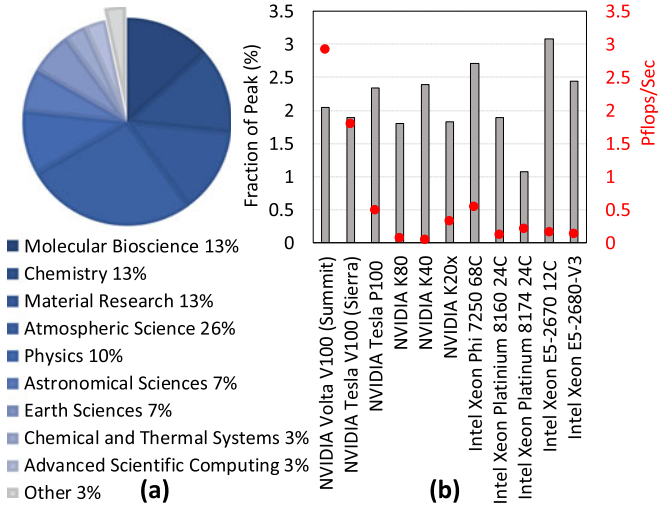


Fig. 1. *Attributes of modeling physical scientific computations:* (a) Different domains of modeling physical create more than 96 percent of workloads on Kraken supercomputer, housed in the Oak Ridge National Lab. (b) The performance of modern computing platforms ranked by the standard metric of HPCG [1] benchmark on GPUs and CPUs.

2 Motivation

By having reached an advanced stage in computing technologies, we expect computers to perform complex tasks quickly and accurately. However, even the performance of most modern computers is not always as fast and accurate as desired. Developing vaccines, forecasting the weather, predicting hurricane path or the exact time of an earthquake, simulating aerodynamics of spacecraft, exploring oil and gas resources, and exploring biological molecules as well as chemical compounds are examples of such time-consuming tasks that can change our lives and future if done accurately. The aforementioned scientific computations that rely on modeling physical phenomena create a significant portion of supercomputer workloads. For instance, as Fig. 1a illustrates, more than 96 percent of Kraken is devoted to various domains of scientific computations from modeling chemical systems to atmospheric sciences [5]. However, not only is modeling physical phenomena costly in terms of dollar and power consumption but also it is slow.

Supercomputers consume large amounts of power, almost all of which is converted into heat, requiring costly cooling. For example, Tianhe-1A, a GPU-based supercomputer, consumes 4.04 megawatts (MW) of electricity [6]. Although it is 3× more power-efficient than a CPU-based supercomputer with same peak performance – such difference in power consumption can provide electricity to over five thousand homes for a year, it indicates approximately $3.5 million per year assuming $0.10/kWh is $400 an hour. Despite all dollar and power-consumption expenses, many scientific problems are not able to utilize the full computation power provided by the CPU- or GPU-based supercomputers. For instance, as the bar charts in Fig. 1b show, no more than 3 percent of the peak performance is utilized by HPCG [1] benchmark, the representative of scientific computations. The secondary axis of Fig. 1b depicts the performance achieved by the fastest implementation based on the HPGC ranking, which implies that executing scientific problems at these speeds takes weeks or months to achieve just an approximate answer – not even the exact accurate solution in many cases. The inefficiently and ineffectiveness

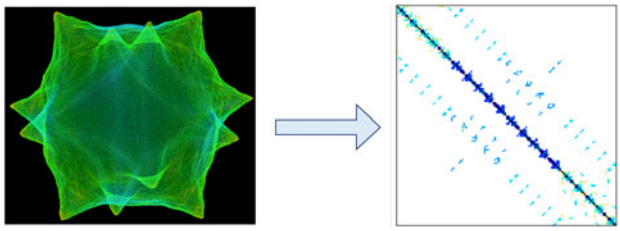


Fig. 2. *Discertization into a 3D grid:* (a) Shallow-water equations (a set of PDEs) discertized into a 3D grid – obtained from Max-Plank Institute of Meteorology and (b) The resulting sparse coefficient matrix A.

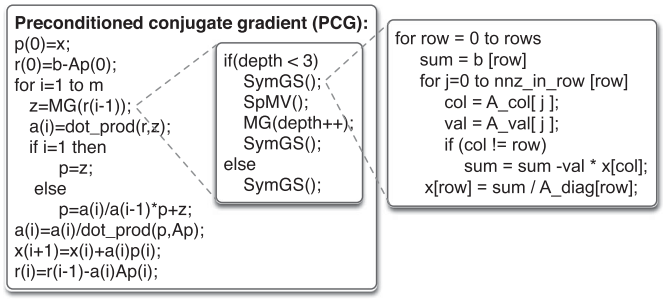


Fig. 3. *An example of the PCG algorithm* [1]**:** solving a sparse linear system of equations (i.e., $Ax = b$), including SpMV and SymGS.

of general-purpose computing in executing many critical problems have motivated us to investigate the sources of issues and explore solutions for them, particularly specialized-hardware solutions, which have initially been aimed for more efficiently utilizing hardware budget for achieving desired performance.

3 Solving PDEs: Attributes & Challenges

A PDE is a differential equation that contains unknown multivariable functions and their partial derivatives. PDEs are used to formulate problems involving functions of several variables, which can be used to create a computer model. Therefore, PDEs are one of the key tools used in scientific computation and modeling of physical phenomena. To use digital computers for solving PDEs, they are often transformed into a system of linear equations, $Ax = b$, in which vector b and matrix A are the coefficients and vector x consists of the unknowns. PDEs have two main attributes that challenge the solving process. First, the methods to solve PDEs include patterns of data-dependent computations. We discuss these dependencies and their consequences in Sections 3.1 and 3.2, respectively.

The second attribute of PDEs is that the coefficient matrix A is often very large and sparse for two or higher dimensional problems (e.g., elliptic, parabolic, or hyperbolic PDEs). This is because, to convert PDEs into a linear system, they must be discretized into a grid. However, since not all the points in a grid are used in the discretization of a phenomenon, the coefficient matrix A is sparse. Fig. 2 shows an example of a set of PDEs – that describe the flow below a pressure surface in a fluid; and the equivalent sparse matrix, A. While the sparseness of data demands high memory bandwidth, the dependencies limits utilizing the available memory bandwidth. Therefore, we cannot simply add more memory bandwidth to gain performance. After proposing our solutions for addressing the challenges risen as a result of dependencies, we show our tailored compression format for envisioning sparseness and dependencies together (Section 4.2.2) and putting together the pieces of our solutions for the sparse matrices (Section 4.2.4).

3.1 Methods for Solving PDEs

To solve PDEs, two categories of methods exist: direct and iterative. Direct methods rely on matrix inversion and attempt to solve the problem by a finite sequence of operations. In the absence of rounding errors, direct methods (e.g., Gaussian elimination) can deliver an exact solution. However, when a linear system involves many variables (sometimes of the order of millions), direct methods would be prohibitively expensive (and in some cases impossible) even with the best available computing power. As a result, in such cases, the second category of solvers, the iterative methods are often the only choice. The iterative methods, gradually update the solution (x) until it converges. The main focus of this paper is also commonly-used iterative methods. After proposing our solution for iterative methods, we discuss the possibility of applying it on direct methods (Section 7).

Iterative methods. When direct methods for solving PDEs are not practical, iterative algorithms such as conjugate gradient (CG) methods (e.g., preconditioned CG (PCG), which ensures fast convergence by preconditioning) are used. An example of the PCG algorithm for solving $Ax = b$ is shown in Fig. 3 [1]. The algorithm updates the vector x in m iterations. The execution time of the algorithm is dominated by two kernels, SpMV and Symmetric Gauss Seidel (SymGS), which for instance, respectively create 31 and 63 percent of the total execution time of PCG on an Nvidia K20 [7], [8], [9]. The remaining kernels, such as the dot product, consume only a tiny fraction of the execution time and are so ubiquitous that they are already executed using special hardware in some supercomputers.

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 ($A_{m\times n}$). Applying SpMV on the two operands results in a vector ($x_{1\times n}$), each element of which can be calculated as

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

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 jth column of A^T, respectively. Fig. 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 [10]

$$x_j^t = \frac{1}{A_{jj}^T} - \left(b_j - \sum_{i=1}^{j-1} A_{ij}^T \times x_i^t - \sum_{i=j+1}^{n} A_{ij}^T \times x_i^{t-1} \right). \tag{2}$$

Fig. 4b illustrates a visualization of Equation 2 (i.e., the blue vectors correspond to $\sum_{i=1}^{j-1} A_{ij}^T \times x_i^t$ and red vectors correspond to $\sum_{i=j+1}^{n} A_{ij}^T \times x_i^{t-1}$). In fact, calculating the jth

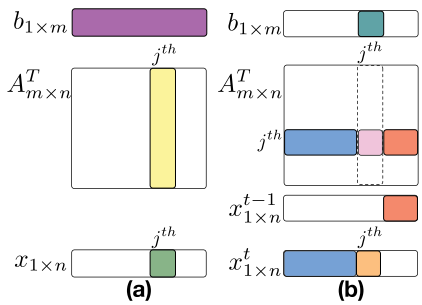


Fig. 4. Calculation of (a) x_j in SpMV and (b) x^t_j in SymGS.

element of x at iteration t (i.e., the orange element of x^t in Fig. 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.

3.2 Key Challenge & Observation

Equation (2) (SymGS) can be written as an extremely simplified expression that still sustains the dependencies:

$$x_i = \sum_{j=0}^{columns} A^T_{ij} \times x_j. \tag{3}$$

The equivalent code for implementing and processing Equation (3) in a computer would include a nested loop: the outer loop over the rows of A^T (i.e., `i=0 to rows`), and the inner loop over the columns of A^T (i.e., `j=0 to columns`). While the iterations of the inner loop can be parallelized, the iterations of the outer loop cannot, because of the data dependencies between them. Fig. 5 demonstrates the dependencies between the iterations of the outer loop. The main cause of such dependencies is that at each iteration of the outer loop, we read the entire vector `x` (Fig. 5, left) and then, we update one element of `x` (Fig. 5, right). Therefore, before reading `x`, we must wait until it is updated. As a result of such dependencies, executing this nested loop cannot benefit from the parallelism provided by GPU by employing common techniques such as loop unrolling. Fig. 6 shows an example, in which we unroll the outer loop three times, and assume that a GPU has nine parallel processing units, three of which are used to process the inner loop. Since the iterations are dependent, three steps are required for processing the three unrolled iterations, at each of which, only is one-third of the GPU utilized.

Key observation. A deeper look at the pattern of dependencies in Fig. 5 and the ineffectiveness of a parallel processor to execute the nested loop quickly, suggests that *not all the operations at each iteration of the outer loop read the newly updated elements by the previous iterations* (Fig. 7a). Based on this observation, we add *blocking* as another optimization on top of the unrolling. As Fig. 7a shows, a block of operations including the iterations of the inner loop (e.g., `j=4,5,6`) that depend on the outcome of the previous iterations of the outer loop (e.g., `i=4,5,6`), can be excluded from the other operations (green part in Fig. 7a). The width of the blocks determines the depth of the unrolling as well as the iterations of *inner* loop that are excluded.

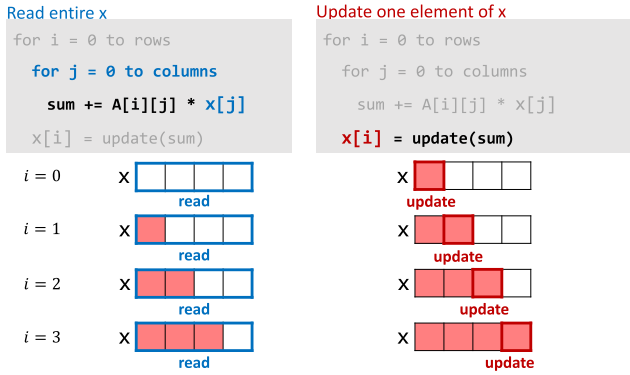


Fig. 5. *Dependencies in SymGS:* Each iteration of the outer loop reads the entire vector x (left) and updates one element of x (right). The iterations of the outer loop are dependent.

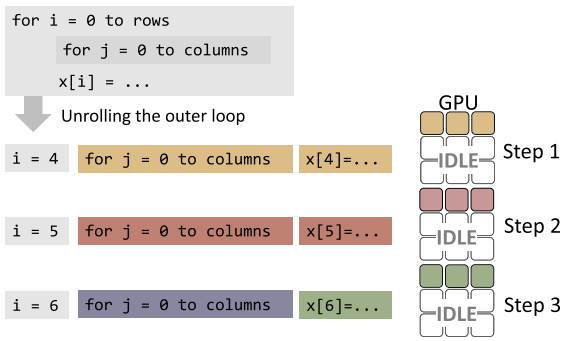


Fig. 6. *Limited parallelism:* Unrolling the iterations of the outer loop (left) and mapping the pararellizable iterations of the inner loop to processing units of a GPU (right).

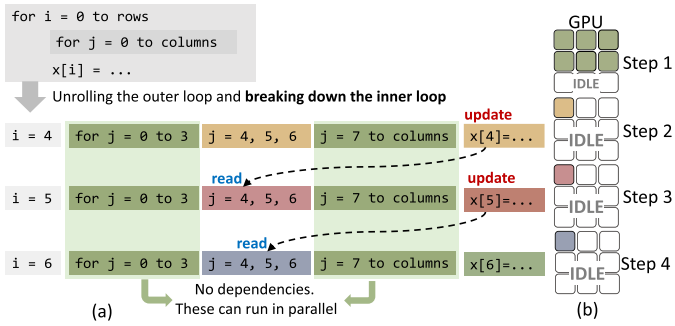


Fig. 7. *(a) Key observation:* the iterations of the outer loop are just *partially* dependent. In fact, only a few iterations of the inner loop read the newly updated elements. Therefore, we can break down the iterations of the inner loop across a few unrolled iterations of the outer loop, into data-dependent part (j = 4, 5, 6), and parallelizable part (green parts); *(b) Key challenge:* ineffectiveness of blocking technique on GPUs. More parallelism at step 1, but dependecies still create the bottleneck through steps 2, 3, and 4.

Key challenge. Even though based on the key observation, more parallel operations can be extracted from the target nested loop, the effort cannot help improve the performance on GPUs. Fig. 7b clarifies this by mapping parallelizable operations into the processing units of the GPU. As illustrated, even though running the green part in parallel increases the level of parallelism at step 1, the rest of the operations still need to wait for the previous steps, which take three additional steps. Therefore, implementing the unrolling and blocking can even worsen the execution time (four steps versus three). This paper seeks to address this

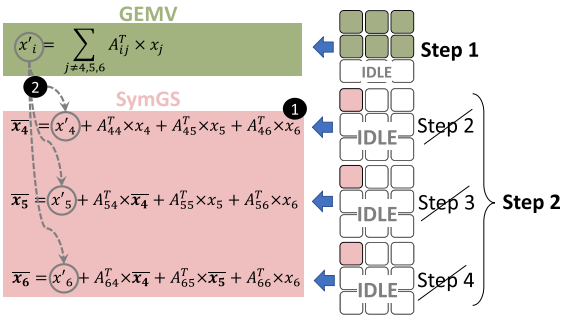


Fig. 8. *Key insight of Alrescha:* We divide a large SymGS into a majority of parallelizable GEMV operations (green) and a minority of *small* data-dependent SymGS (pink). We first run the GEMV and then switch to SymGS. Dependecies still exist in SymGS part but as long as it is small, we can run them in one step in hardware rather than three steps (❶). To be effective, forwarding the outcomes of GEMV to the SyMGS must be fast as well (❷).

challenge and enable benefiting from the key observation and the resulting software optimizations (i.e., unrolling and blocking) by the aid of hardware.

4 Alrescha

4.1 Key Insight

Our key insight to resolve the challenge is to reduce the negative impact of data dependencies on performance by hardware-software co-design, even though we cannot remove the patterns of data dependencies that naturally exist in a program. In our example case, for instance, to allow the unrolling and blocking to be effective, steps 2, 3, and 4 must be executed quickly, preferably in one step as shown in Fig. 8. We propose Alrescha, a hardware-software co-design that divides a SymGS into a large portion of general matrix-vector multiplications (GEMVs) that can be executed in parallel or concurrently and a small data-dependent SymGS. Since the SymGS part is now small, Alrescha [11] can execute it quickly in hardware. To be effective in fast execution of SymGS, Alrescha accelerates (i) the mechanism of immediately using the outcome of operation by the next operations in the SymGS ($\overline{x}_i$ i, ❶); and (ii) the mechanism of using the outcomes of the GEMV in different operations of SymGS (x'_i, ❷).

4.2 Main Contributions

This section overviews main contributions of Alrescha to benefit from the key insight not only in scientific computations but also in broad applications. The contributions include the key hardware mechanisms (Section 4.2.1), our compression format to envision spareness (Section 4.2.2), broad applications (Section 4.2.3), putting the mechanisms together in a systematic manner (Section 4.2.4), and the reconfigurable microarchitecture (Section 4.2.5).

4.2.1 Key Mechanisms in Hardware

After dividing a large SymGS into GEMVs and a small SymGS, the proposed hardware mechanisms of Alrescha help to execute them quickly. To explain the mechanisms, we use a simple example of a SymGS with matrix A operands shown in Fig. 9a. We focus on processing three rows of A (`i = 4,5,6`) to calculate corresponding final elements of x (i.e., $\overline{x}_4, \overline{x}_5$, and $\overline{x}_6$). The green parts of matrix A are the

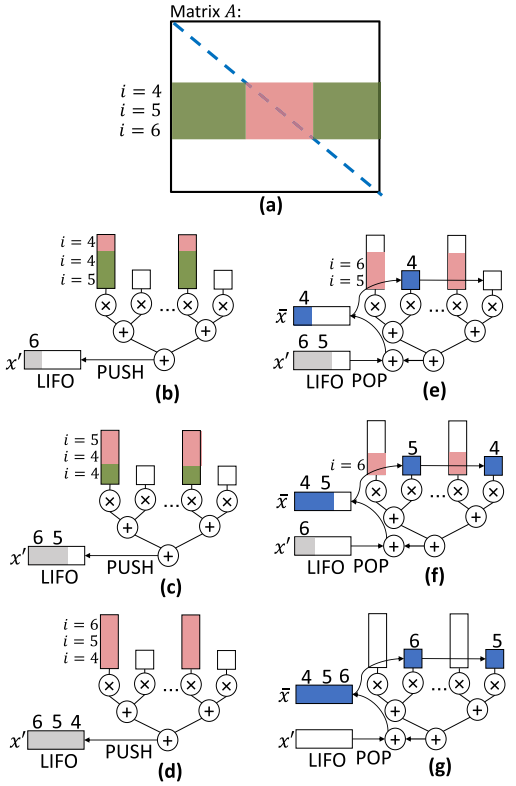


Fig. 9. *Key mechanisms of Alrescha:* (a) Matrix A the operand of the original large SymGS, (b, c, d) Executing GEMV and the mechanism for quickly forwarding the outcome of GEMV to SymGS using a LIFO or FIFO; and (e, f, g) Executing the small SymGS and implementing the scheme of data dependencies through the interconnections between inputs of the tree and the LIFO to quickly execute the small SymGS.

operands to GEMVs, while the pink part is the operand of the small SymGS.

Parallel & Concurrent GEMVs. First, Alrescha executes all GEMV operations that result in the partial outputs (i.e., x'_4, x'_5 and x'_6). Figs. 9b, 9c, and 9d show last three steps of the GEMVs that contribute in creating x'_6, x'_5 and x'_4, respectively. Such an order of operations that first performs all the GEMVs corresponding to rows 4 to 6 before performing the SymGS corresponding to the same rows, does not impact the functionality and the correctness of the operations as long as the partial results are correctly aggregated with corresponding values in the next steps, as explained in the following.

Fast Switch From GEMVs to SymGS. Alrescha facilitates the mechanism of aggregating the partial results generated by GEMVs with the partial results of SymGS by using a last-in-first-out – alternatively, a first-in-first-out buffer can also be used if compatible orders also reflected in reading the rows of the matrix. As Figs. 9b, 9c, and 9d show, during GEMV phase, we push the partial results into the LIFO, and POP them out during SymGS, as illustrated in Figs. 9e, 9f, and 9g. This mechanism, which prevents extra accesses to an on-chip cache (with sophisticated addressing requirements) or the main memory, provides a smooth switching between GEMVs and SymGS operations.

Fast Execution of Data-Dependent SymGS. Once all the GEMVs corresponding to rows 4 to 6 are done, Alrescha switches to SymGS (the new step 2 in Fig. 8). The nature of SymGS in step 2 is the same as the GEMVs in step 1, whereas the individual inputs are not available altogether. In fact, step 2 is generating its own inputs. Because of the

similarity between the GEMV and SymGS, SymGS can use the same core mechanism of multiplication followed by the summation-based reduction tree as shown in Figs. 9e, 9f, and 9g. Besides the core mechanism, Alrescha implemented the dependencies using some interconnections between the inputs of the tree and its output. Such interconnection immediately forwards $\overline{x}_i$ to the inputs and shifts the old inputs to the right. this mechanism simply accelerates the three dependent operations in old steps 2, 3, and 4. Note that the smallness of SymGS block is important here since otherwise, the depth of the tree prevents the fast execution.

4.2.2 Compression Format for Sparse PDEs

As explained earlier (Section 3), the matrix A in a linear system is often *sparse*. Therefore, a compression format must be used to efficiently save the matrix A. On the other hand, we saw that the hardware mechanisms demand a unique order of data in matrix A (Section 4.2.1). This section discusses the compression formats suitable for the target applications and explains how we slightly modify an appropriate compression format to sustain the desired order of data, dictated by our proposed mechanism. According to the distribution of non-zeros in a sparse matrix, various compression formats may suit them. For instance, the compressed sparse row (CSR), which stores a vector of column indices and a vector of row offsets, locates all the non-zeros independently 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) [12], which stores the non-zeros in the diagonals sequentially, could be the best option. An extension to the DIA format, Ellpack-Itpack (ELL) [13] is more flexible when the matrix has a combination of non-diagonal and diagonal elements. For instance, ELL is used for implementing SymGS in GPUs. However, such a format does not provide flexibility for parallelizing rows as it does not sustain locality across rows.

Since the choice of compression format should be compatible with the range of sparse applications, blocked CSR [14], an extension of CSR, which assigns the column indices and row offsets to blocks of non-zero values, has been proposed as a more generic 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 the most appropriate match for smoothly streaming data in Alrescha. 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 compression format with the same meta-data overhead but compatible with Alrescha.

Fig. 10 illustrates our proposed compression format for mapping an example sparse matrix to the physical memory addresses of the accelerator. In this compression mechanism, all the non-diagonal non-zero blocks in a row of blocks are stored sub sequentially, followed by a diagonal block. 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 Fig. 10). Accordingly, the difference between the column indices of BSCR and input indices (i.e., Inx_{in}) of our proposed format is shown in Fig. 10. For SymGS, the diagonal of A is excluded and stored separately in a local cache.

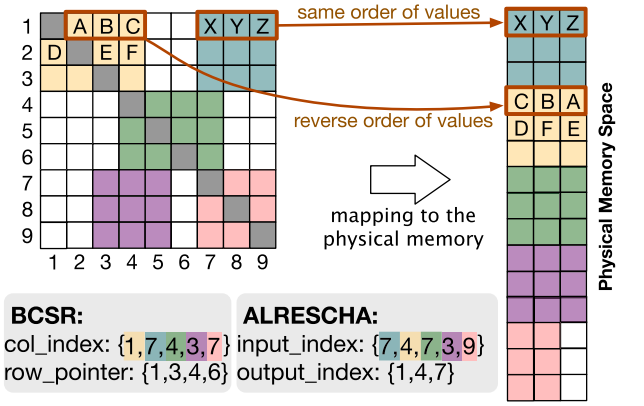


Fig. 10. *Compression format of Alrescha:* the col_index of BCSR and input_index (i.e., Inx_{in}) of Alrescha are color-coded to show their corresponding blocks in the matrix. Alrescha uses the index of the last column for the input index of diagonal blocks.

Therefore, we consider non-square blocks on the diagonal (e.g., 3×4 instead of 3×3) so that the mapping of the non-diagonal element of that block to the physical memory is adjusted. The indices of the input and output (i.e., Inx_{in} and Inx_{out}) are not streamed from memory during run time. Instead, they are stored in a configuration table during a 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.

4.2.3 Broad Applications

Customized hardware has not often been selected as a viable option, except for particular applications such as neural networks. Instead, general-purpose hardware such as CPUs and GPUs have usually been used for executing applications such as scientific computations, even though their performance is dramatically low. A reason for this is the economic. Extensive customization, which includes high design and fabrication costs, have been considered expensive solutions for narrow applications, even if such hardware offers significant performance benefits. To deal with the high cost, we argue that custom hardware solutions must be generic and applicable to a reasonable range of applications. This section elaborates on the applicability of Alrescha for SpMV and graph analytics.

In graph analytics, a common approach to represent graphs is to use an adjacency matrix, each element of which represents an edge in the graph. 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 [15], 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. Since in many applications not all the nodes in a graph are connected, the equivalent adjacency matrix is sparse, too.

TABLE 1
The Properties of Sparse Kernels and Corresponding Dense Data Paths, Implemented in Alrescha

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

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.*

The widely used graph algorithms are SpMV, breadth-first search (BFS), PageRank (PR), and single-source shortest path (SSSP). In SSSP, for instance, the vector containing 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 a source node to all the other nodes. PR 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 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 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 generalize our proposed hardware without significant overhead. Alrescha converts the sparse kernels into the dense data paths, listed in the second column of Table 1 (details in the following).

4.2.4 Putting Them Together for Sparse PDEs

Alrescha is a memory-mapped accelerator, the memory of which is accessible by a host for programming. Fig. 11 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. To program the accelerator, the host launches the sparse kernels of sparse algorithms (e.g., PCG)

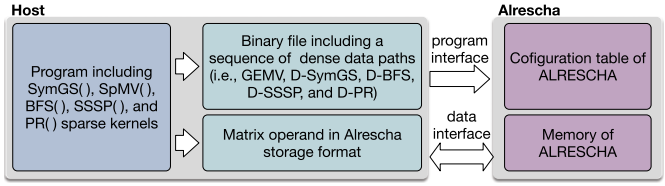


Fig. 11. The overview of Alrescha and host.

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*.

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 real-time partial reconfigurability. The details of the reconfigurable microarchitecture of Alrescha and the mechanism of real-time reconfiguration are explained in Section 4.2.5. 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 the new compression format, introduced in Section 4.2.2. Therefore, the other task of the host is to reformat the sparse matrix operands into the compression 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* (Fig. 11).

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.

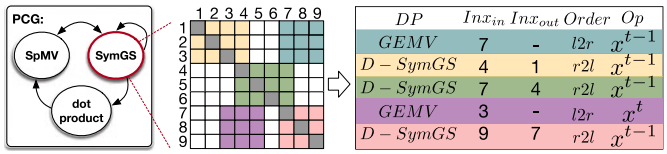
Algorithm 1 shows the procedure for converting a sparse matrix to dense data paths. The general procedure of the conversion algorithm is as follows: (i) As lines 8 to 12 show, the sparse kernels with no (or straightforward) data dependencies including SpMV, BFS, SSSP, and PR are broken down into a sequence of general matrix-vector multiplication

Algorithm 1. Convert Algorithm

1: **function** CONVERT(KernelType, $A_{n\text{x }n}, \omega$)
$A_{n\text{x }n}$: 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: $Blocks[] = \textbf{Split}(A, \omega)$ // partitions A to $\omega \times \omega$ blocks
4: $m = n/\omega$
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: $Inx_{in} = i.\omega$, $Inx_{out} = j.\omega$
11: $Order = l2r$
12: $Op = port1$ // the operand vector
13: **else**
14: **if** $(i! = j)$ **then**
15: $DP =$ GEMV
16: $Inx_{in} = j.\omega$
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: $Inx_{in} = j.\omega$, $Inx_{out} = (i + 1).\omega$
26: $Order = r2l$
27: $Op = port2$ //which is x^{t-1}
28: **Add2Table**$(DP, Inx_{in}, Inx_{out}, Order, Op)$

(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) 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, Fig. 12 shows the state machine of PCG, equivalent to the algorithm in Fig. 3, 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 Fig. 12. Based on Equation (2) and as lines 19 and 21 of



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}

Fig. 12. *The order of operations:* An example of the configuration table for a SymGS kernel, in which $n = 9$, $\omega = 3$.

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}.

4.2.5 Reconfigurable Microarchitecture

This section introduces the microarchitecture of Alrescha. The key feature of the proposed microarchitecture is partial reconfigurability. The benefit of this feature is two-fold. First, for SymGS, we have seen that the building blocks of GEMV and SymGS require a common core hardware mechanism. Besides, the GEMVs create a big portion of operations. Therefore, as long as Alrescha is performing subsequential GEMVs, it does not have to select what to do neither by decoding an instruction, nor by selecting a path in the hardware. The second benefit of partial reconfiguration goes to the other applications (e.g., graph kernels) that also share a core hardware mechanism. As a result, Alrescha can simply perform other applications (or one application including distinct kernels) without needing to change the entire hardware. Reconfiguring only a fraction of the entire data path reduces the configuration time. To achieve the goal of partial reconfiguration, Alrescha consists of a separate fixed computation unit (FCU) and a reconfigurable computation unit (RCU) and configuring only the former for switching between data paths (Fig. 13).

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). The seven REs are connected in a tree topology (i.e., the REs are the nodes of a 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 run time, 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 run time.

The responsibility of the RCU is to handle the specific data dependencies in different kernels. The RCU includes a 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. The root of the reduction tree from FCU is one of the inputs to the configurable switch, the other inputs and

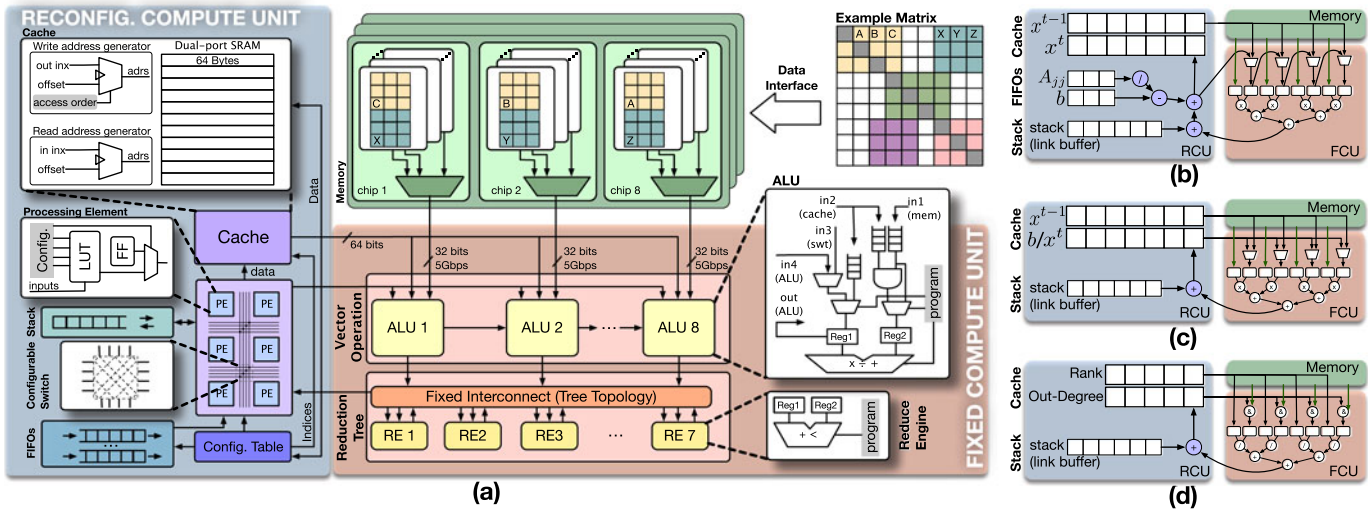


Fig. 13. *The microarchitecture of Alrescha:* (a) the FCU for implementing common computations, and the RCU for providing specific configuration for distinct dense data paths. Three example configurations for supporting: (b) D-SymGS, (c) GEMV, and (d) D-PR.

outputs of which are the PEs, cache, stack, FIFOs, and ALUs. This switch is the only reconfigurable component of Alrescha. The reconfiguration time of this switch is approximately 0.3 ns at 90nm technology based on Xilinx Virtex-4. Therefore, the reconfiguration process is hidden by the draining of the tree in FCU. The 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 for A_{jj}^T and b, and use a last-in-first-out stack for the link buffer. The link buffer establishes transmissions between the dense data paths. 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 is implemented by look-up tables (LUTs) to provide multiplication, division, summation, and subtraction. Figs. 13b, 13c, and 13d illustrate the configuration of the RCU for performing D-SymGS, GEMV, and D-PR.

5 Related Work

Before moving to performance evaluation, we overview the prior proposals including software and hardware optimizations for sparse scientific computations, SpMV, and graph analytics.

Software Optimizations. To date, many software-level optimizations for CPUs [16], [17], [18], GPUs, [8], [19], [20], [21], [22], and CPU-GPU systems [23] have been proposed to deal with the challenges arisen by sparseness such as irregular memory accesses. To relax irregular memory accesses, software optimizations such as partitioning and batching have been proposed. Such approaches batch the accesses and restrict them to a localized region of memory [24]. However, software optimizations alone cannot effectively handle the data-dependent operations, the main source of low performance in scientific computations. Furthermore, optimizations for extracting more parallelism and bandwidth such as matrix coloring [8] and blocking [14] have not been effective enough for the aforementioned reason.

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 [25], [26], [27], [28], matrix-vector multiplication [29], [30], [31], [32], or both [3], [33], [34], which are the main sparse kernels in many sparse problems. A state-of-the-art SpMV accelerator, OuterSPACE [3], 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 [33], a novel fetching mechanism that avoids the memory latency overhead associated with sparse kernels. Song *et al.* proposed GraphR [4], a graph accelerator, and Feinberg *et al.* proposed a scientific-problem accelerator [2], both of which process blocks of non-zero values instead of individual ones. Besides, Huang *et al.* have proposed analog [35] and hybrid (analog-digital) [36] accelerator for solving PDEs. Moreover, many processing-in-memory studies [37], [38], [39], [40], [41] proposed offloading computation to memory to reduce the computation energy of sparse problems. 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. Table 2 compares the most relevant hardware approaches and techniques for accelerating sparse problems with Alrescha.

6 Performance Evaluation

This section explores the performance of Alrescha by comparing it with the CPU, GPU, and state-of-the-art sparse accelerators. We evaluate Alrescha for both scientific applications and graph analytics.

6.1 Datasets, Algorithms, and Baselines

We pick real-world matrices with applications in scientific and graph problems from the SuiteSparse Matrix

TABLE 2
Comparing the State-of-the-Art Accelerators for Sparse Kernels

		GraphR [4]	OuterSPACE [3]	Memristive-Based Accelerator [2]	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	×	✓
	Partial 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)	✓

Collection [42]. The matrices along with their dimensions and the number of non-zeros (NNZ) are shown in Table 3. We run PCG, which includes the SymGS and SpMV kernels, on the matrices with a scientific application, and run graph algorithms (i.e., BFS, SSSP, and PR) on the last eight 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 [43] libraries. The graph algorithms running on the CPU are based on the GridGraph [44] and/or CuSha [45] platforms (whichever achieves better performance).

Besides the comparison with the CPU and GPU, this section compares Alrescha with the state-of-the-art hardware accelerators, including OuterSPACE [3], an accelerator for SpMV, GraphR [4], a ReRAM-based graph accelerator, and a Memristive accelerator for scientific problems [2]. 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. Seeking a fair comparison, we assign all the accelerators the same computation and memory-bandwidth – this assumption does not harm the performance of our peers.

TABLE 3
Sparse Matrices From SuiteSparse [42] Matrix Collection

Name	Dim.(M)[1]	NNZ(M)[2]	Kind
2cubes_sphere	0.101	1.647	Electromagnetic Prob.
ASIC_100k	0.099	0.954	Circuit Sim.
atmosmodm	1.489	10.319	Fluid Dynamics
CoupCons3D	0.416	17.277	Structural Prob.
crystm	0.024	0.583	Materials Prob.
dielFilterV3real	1.102	89.306	Electromagnetic Prob.
epb3	0.084	0.463	Thermal Prob.
finan512	0.074	0.596	Economic Prob.
GaAcH6	0.061	3.381	Chemistry Prob.
HV15R	2.017	283.073	Fluid Dynamics
ifiss_mat	0.096	3.599	Fluid Dynamics
light_in_tissue	0.029	0.406	Electromagnetic Prob.
mono_500Hz	0.169	5.036	Electromagnetic Prob.
offshore	0.259	4.242	Electromagnetic Prob.
poission3Db	0.085	2.374	Fluid Dynamics
qa8fm	0.066	1.660	Acoustics Prob.
scircuit	0.170	0.958	Circuit Sim.
Si34H36	0.097	5.156	Chemistry Prob.
StocF-1465	1.465	21.00	Fluid Dynamics
TEM152078	0.152	6.459	Electromagnetic Prob.
thermomech_TC	0.102	0.711	Thermal Prob.
Transport	1.602	23.487	Structural Prob.
windtunnel_en3D	0.040	0.803	Fluid Dynamics
xenon1	0.048	1.181	Materials Prob.
com–orkut	3.072	234.370	Undirected Graph
hollywood–2009	1.139	1.139	Undirected Graph
kron-g500–logn21	2.097	182.082	Undirected Multigraph
roadnet–CA	1.971	5.533	Undirected Graph
LiveJournal	4.847	68.993	Directed Graph
Youtube	1.134	5.975	Undirected Graph
Pokec	1.632	30.622	Directed Graph
sx-stackoverflow	2.601	36.233	Dorected Multigraph

[1]*Dim.: dimension or the number of columns/rows of a square matrix.*
[2]*NNZ: the number of non-zero entries.*

6.2 Experimental Setup

Simulation. 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

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 [43] 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 [45], GridGraph [44]

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

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.

FPGA Implementation. While FPGAs have had the partial reconfiguration feature for over a decade, fewer applications have been proposed to use it. Alrescha proposes a new application for partial reconfiguration. Our goal is to leverage partial reconfigurability to evaluate the switching between the different algorithms, without fully reprogramming the FPGA hence we utilize *static* partial reconfiguration rather than a *dynamic* one – note that dynamic reconfiguration can also be implemented for SymGS. We implement SymGS and graph algorithms, the common function of which is a matrix-vector multiplication. We implement Alrescha using Xilinx Vivado HLS. We use relevant *#pragrma* as hints to describe our desired microarchitectures in C++. We target Xilinx AC701 evaluation kit, including a partially reconfigurable Artix-7 FPGA, XC7A200T. We present the post-implementation resource utilization and power consumption, reported by Vivado. Inputs and outputs of Alrescha are transferred through the AXI stream interface. The clock frequency is set to 200 MHz.

6.3 Execution Time

Scientific Problems. The primary axis of Fig. 14 (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 Fig. 14 shows the bandwidth utilization. The figure also captures the speedup of the Memoristive-based hardware

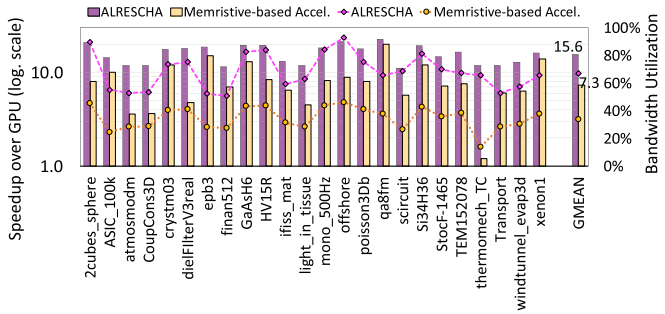


Fig. 14. *Speedup:* PCG algorithm on scientific datasets, normalized to GPU (bar charts), and bandwidth utilization (the lines) compared to the state-of-the-art accelerator for scientific problems [2].

accelerator [2]. 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 Fig. 14. 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, Fig. 15 presents the percentage of data-dependent computations in the GPU implementation, versus that in Alrescha, which has an average of 23.1 percent data-dependent 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 percent of operations are still data-dependent. This is more than 60 percent for highly-diagonal matrices and less than 60 percent 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 than when the matrix is diagonal. Therefore, to conclude, for multi-kernel sparse algorithms with data-dependent computations, Alrescha improves performance by (i) extracting parallelizable data paths, (ii) reordering them and the elements in the blocks to maximize the reuse of data, and (iii) implementing them in lightweight reconfigurable hardware, which results in fast switching not only between the distinct data paths of a single kernel but also among them.

Graph Analytics & SpMV. This section explores the performance of the algorithms consisting of a single type of kernel with 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. Fig. 16 illustrates the speedup of running BFS, SSSP, and PR on Alrescha, a recent hardware accelerator for graph applications (i.e., based on GraphR [4]), and GPU, all

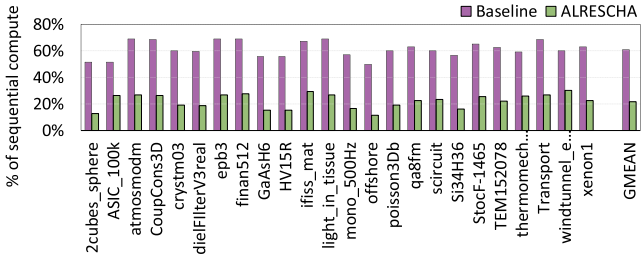


Fig. 15. *Reduction in data-dependent operations:* for the PCG algorithm, after applying Alrescha. The baseline shows the percentage of data-dependent operations by row-reordering optimization.

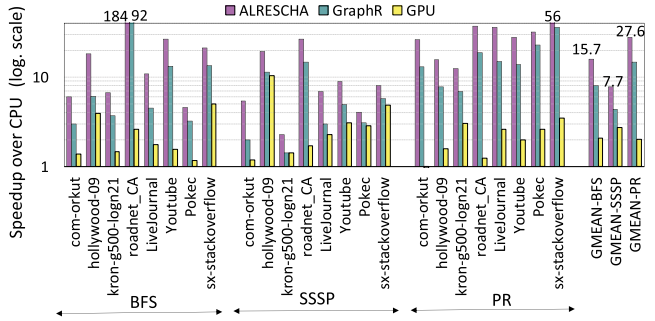


Fig. 16. *Speedup:* graph algorithms on graph datasets over the CPU. GraphR [4] is the state-of-the-art graph accelerator.

normalized to the CPU. 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. Further, to run graph applications, Alrescha performs only subsequential same-type dense data paths that eliminates the need to neither decode instructions (which is the case for GPUs) nor select a data path in the hardware.

The primary axis of Fig. 17 (i.e., the bars) illustrates the speedup of SpMV, a common algorithm of various sparse applications on Alrescha and OuterSPACE [3] (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 Fig. 17 (i.e., the lines) plots the percentage of the whole execution time for accesses to the local cache. Alrescha utilizes locality in cache accesses (i.e., consuming the values in a cache line in succeeding cycles), and increases the data reuse rate of not only the input sparse-matrix operands but also the dense-vector operands and output vector. Although in the outer-product approach, data read from the cache is broadcast to all the ALUs, to be reused as many times as required, before being written back to the cache, an element of the output vector must be fetched several times. During such accesses to the cache, the spatial locality of non-zeros is not captured. On contrary, the approach of Alrescha that applies GEMV to locally dense blocks of the sparse matrix instead of working on individual non-zeros takes advantage of spatial locality in the non-zero values of the sparse matrix. Besides, Alrescha sums up the results of multiplications locally, without redundant accesses to the cache. To do so, Alrescha splits the vector operand into chunks and at each time step, instead of fetching an individual element, it fetches a chunk of vector operand from the cache, and instead of broadcasting, it sends them to individual ALUs. The elements of a chunk are multiplied by all the non-zero blocks of the sparse matrix in a row. As a result, each element of the output vector is fetched from cache only once per $\#cols/n$ (n: chunk size).

6.4 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 with the hardware accelerators have to be efficient. The main resources of energy consumption are instruction decoding and the accesses to SRAM. Therefore, as explained in Section 6.3, not only does Alrescha substitutes the instruction decoding with the ability to switch between the data paths simply when it is required but also it reduces the number of accesses to the SRAM cache. While for the single-kernel application no kernel switches are required, for multi-kernel applications the number of switches depends on the structure of sparsity (i.e., the distribution of non-zero values in the sparse matrix). The patterns of sparsity and thus the number of switches impact the energy consumption of SymGS. The more non-zero blocks in a row of a sparse matrix (e.g., poisson3Db) that results in a long sequence of GEMVs or an extremely diagonal sparse matrix (e.g., atmosmodm) that results in a sequence of D-SymGS data paths reduce the number of kernel switches and consume less energy compared to more irregular sparse workloads. Fig. 18 illustrates the energy consumption of Alrescha for executing SpMV, normalized to that of the CPU and GPU baselines. As Fig. 18 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 computing units, defined by the density of the locally-dense block, impacts energy but not performance. Therefore, to sum up, 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.

6.5 FPGA Resource Utilization & Power Consumption

Here, we evaluate the implementation of Alrescha on a partially reconfigurable FPGA (consisting of total 433K LUTs and 174K FFs), the features of which align with one another. Table 6 compares the resource utilization and dynamic power consumption of a static design with a partially-reconfigurable design including FCU and RCU. As the table suggests, in the static design, each of the dense data paths utilizes the resource as much as required. Therefore, D-PR, which includes division operations, utilizes the most number of LUTs and FFs and the highest power consumption, whereas D-BFS, which requires the simplest design, utilizes the minimum resources. On the other hand, since in the partially reconfigurable design, the FCU must envision the

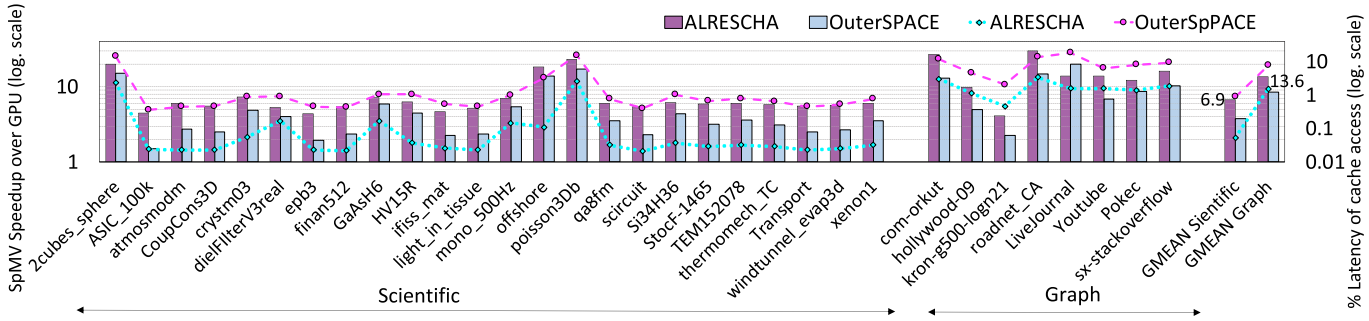


Fig. 17. *Speedup:* executing SpMV on scientific and graph datasets normalized to GPU (bar charts), and the percentage of execution time devoted to cache accesses (the lines). OuterSPACE [3] is the state-of-the-art SpMV accelerator.

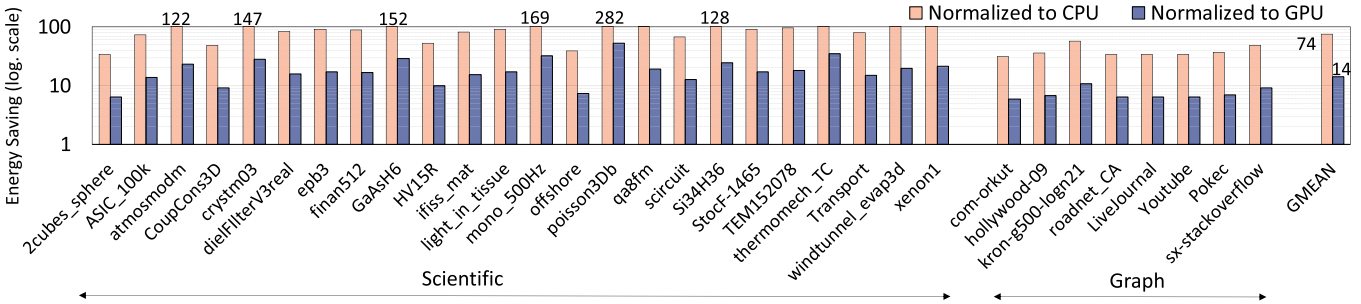


Fig. 18. *Energy consumption:* Energy consumption saving of Alrescha normalized to that of CPU and GPU for scientific and graph workloads.

verity of operations, the overall architecture utilizes more than enough resources for GEMV, D-BFS, and D-SymGS. The RCU, however, is tailored to each design. Thus, D-SymGS, with the most complex RCU utilized more resources that other data paths do. Although in some case, the partially reconfigurable design utilized more resources, the overheads are outweighed by its benefits, especially for multi-kernel applications such as the PCG algorithm that requires switching between distinct kernels during the run time. As the table suggests, the partial reconfigurable implementation of the simple single-kernel workloads (i.e., SpMV and BFS) has an overhead of 2.9× more LUT and 1.9× more FF as well as 1.2× more dynamic power consumption compared to the static implementation. On the contrary, for more complex single-kernel workloads such as PR and the multi-kernel workloads such as PCG (including SymGS and GEMV), the partially reconfigurable implementation is more beneficial as it utilizes 1.2× fewer LUTs and FFs, and consumes 1.15× less dynamic power consumption. Note that here we show only the core computation unit. However, the fixed modules involve many other components of a complete architecture.

TABLE 6
Resource Utilization and the Total Dynamic Power Consumption

	Static Design			
	GEMV	D-PR	D-BFS	D-SymGS
LUT	2386	8632	2246	3467
FF	6489	10233	4439	7845
Power(W)	0.098	0.115	0.065	0.102

	Partially Reconfigurable Design				
	FCU	RCU			
		GEMV	D-PR	D-BFS	D-SymGS
LUT	6594	271	271	123	645
FF	9771	380	380	320	1594
Power(W)	0.086	0.03	0.03	0.01	0.06

7 Discussions & Future Work

This paper showed how Alrescha can run the iterative methods for solving PDEs faster hence running more iterations in a given time and achieving a more accurate solution x for a linear system $Ax = b$. Here, we also explore the applicability of Alrescha on direct methods that are often not practical mainly because they are extremely slow. However, if a certain computation platform allows us to execute them quickly, they would be the preferred methods, which provide the preferred exact solutions. A key matrix algebra and a bottleneck-prone operation in the direct methods is *matrix inversion*. To simplify the time-consuming matrix inversion, decomposition techniques such as lower-upper (LU) decomposition are used. In the following, we explore the mechanism using a decomposed matrix (i.e., the outcome of LU) for calculating the invert of A, the involved challenges, and the applicability of Alrescha for addressing them. LU decomposition factors a matrix as the product of lower and upper triangular matrices: $A = LU$. Therefore, in the following, whenever A is used, it indicates a matrix consisting of L and U. After decomposition, the inverse of A can be calculated as $A^{-1} = U^{-1}L^{-1}$. The decomposition itself is more straightforward and parallelizable compared to the invert algorithm. Therefore, we focus on the invert algorithm to find possible sources of performance bottleneck. Fig. 19 lists a simple pseudo-code for LU invert. As illustrated, it consists of an outer loop for traversing the columns of A with dimensions of N. The outer loop comprises two inner loops for traversing A from up to bottom and bottom to top, respectively. At each of these loops (i.e., phase 1 and phase 2),

```
for j = 0 to N
   for i = 0 to N                                          Phase 1
      for k = 0 to i
         IA[i][j] = IA[i][j] - A[i][k] * IA[k][j]

   for i = N-1 to 0                                        Phase 2
         for k = i+1 to k
            IA[i][j] = IA[i][j] - A[i][k] * IA[k][j]

         IA[i][j] = IA[i][j] / A[i][j]
```

Fig. 19. *LU algorithm:* calculating the inverse of A including two phases and patterns of read and update accesses to matrix A and IA.

an entire column of invert matrix, IA, and a triangular of A are read, and simultaneously the same column of IA is updated. The simultaneous update and read accesses to IA suggests the same type of data-dependencies as SymGS.

To clarify the pattern of data dependency in the LU algorithm for matrix inversion, we focus on four iterations of the outer loop in phase 1 for an example of a 4×4 matrix and the patterns of reads and updates for IA, when $j = 2$. At each iteration, one element of IA is updated, and the very same element is read along with other elements of the same column in the next iteration. This pattern of data dependency is an extension of dependencies in Fig. 5. Therefore, our proposed software techniques (i.e., unrolling and blocking) along with the hardware mechanism for reducing the negative impact of dependencies is applicable here. We apply unrolling and blocking to breakdown the large data-dependent operations into two groups of operations: parallelizable and small data-dependent. If we unroll the iterations of the outer loop in phase 1 for b times (b: block size) and exclude the read operations from IA[i][j] to IA[i+b][j], the rest of operations can be executed in parallel (or concurrently). After such a separation, Alrescha first executes b parallel operations, which result in partial outputs; then, it executes b small dependent operations, which result in the final outputs (i.e., the final elements of IA). Besides these key dependencies, matrix inversion captures other types of dependencies such as dependencies between the two phases. In future work, we are looking for addressing them using more high-level solutions.

8 Conclusion

This paper proposed, Alrescha, an accelerator for solving PDEs, the important mathematical methods used in modeling physical phenomena in scientific computations. Having the key feature of partial reconfiguration, Alrescha is the first multi-kernel accelerator, which indicates that it can execute a program (e.g., SymGS), consisting of distinct kernels, by quickly switching the hardware data paths. The partial reconfigurability of Alrescha also allows it to accelerate broad applications including graph analytics, SpMV, and any other problems consisting of the same reduction operations, by just partially modifying the interconnections among some of the computation units. As a proof of concept, we emulated Alrescha on an FPGA with a partial reconfigurability feature. Our experimental results suggest that for efficiently performing scientific computation migrating to emerging technologies is not necessary as long as we can find the sources of performance bottleneck and prevent them from negatively impacting performance.

Acknowledgments

The authors would like to gratefully acknowledge the support of NSF CSR 1526798.

References

[1] J. Dongarra, M. A. Heroux, and P. Luszczek, "HPCG benchmark: A new metric for ranking high performance computing systems," *Knoxville, Tennessee*, Tech. Rep. SAND2013-4744, Sandia Nat. Lab., 2013, pp. 1–11. [Online]. Available: https://library.eecs.utk.edu/files/ut-eecs-15-736.pdf

[2] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 367–382.

[3] S. Pal *et al.*, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 724–736.

[4] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 531–543.

[5] H. You and H. Zhang, "Comprehensive workload analysis and modeling of a petascale supercomputer," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2012, pp. 253–271.

[6] Nvidia, "NVIDIA tesla GPUs power world's fastest supercomputer," 2010, Accessed: Jul. 2020. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-tesla-gpus-power-world-s-fastest-supercomputer

[7] 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. [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/116642/1HPCG_shared_mem_implementation_tech_report.pdf

[8] E. Phillips and M. Fatica, "A CUDA implementation of the high performance conjugate gradient benchmark," in *Proc. Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2014, pp. 68–84.

[9] V. Marjanović, J. Gracia, and C. W. Glass, "Performance modeling of the HPCG benchmark," in *Proc. Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2014, pp. 172–192.

[10] G. H. Golub and C. F. Van Loan, *Matrix Computations*, vol. 3. Baltimore, MD, USA: JHU Press, 2012.

[11] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, "ALRESCHA: A lightweight reconfigurable sparse-computation accelerator," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 249–260.

[12] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2003.

[13] T. C. Oppe, "The vectorization of ITPACK 2C," *Int. J. Numerical Methods Eng.*, vol. 27, no. 3, pp. 571–588, 1989.

[14] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *Proc. 1st Int. Conf. High Perform. Comput. Commun.*, 2005, pp. 807–816.

[15] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[16] K. Akbudak and C. Aykanat, "Exploiting locality in sparse matrix-matrix multiplication on many-core architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2258–2271, Aug. 2017.

[17] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon phi," in *Proc. Int. Conf. Parallel Process. Appl. Math.*, 2013, pp. 559–570.

[18] P. D. Sulatycke and K. Ghose, "Caching-efficient multithreaded fast multiplication of sparse matrices," in *Proc. 1st Merged Int. Parallel Process. Symp. and Symp. Parallel Distrib. Process.*, 1998, pp. 117–123.

[19] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix–matrix multiplication for the GPU," *ACM Trans. Math. Softw.*, vol. 41, no. 4, 2015, Art. no. 25.

[20] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, andU. Naumann, "GPU-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM J. Sci. Comput.*, vol. 37, no. 1, pp. C54–C71, 2015.

[21] W. Liu and B. Vinter, "An efficient GPU general sparse matrix-matrix multiplication for irregular data," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 370–381.

[22] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *Proc. 19th Int. Conf. High Perform. Comput.*, 2012, pp. 1–10.

[23] W. Liu and B. Vinter, "A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors," *J. Parallel Distrib. Comput.*, vol. 85, pp. 47–61, 2015.
[24] M. Kumar *et al.*, "Efficient implementation of scatter-gather operations for large scale graph analytics," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2016, pp. 1–7.
[25] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "LODESTAR: Creating locally-dense CNNs for efficient inference on systolic arrays," in *Proc. 56th ACM/IEEE Des. Autom. Conf.*, 2019, Art. no. 233.
[26] C. Y. Lin, N. Wong, and H. K.-H. So, "Design space exploration for sparse matrix-matrix multiplication on FPGAs," *Int. J. Circuit Theory Appl.*, vol. 41, no. 2, pp. 205–219, 2013.
[27] 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 *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2013, pp. 1–6.
[28] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "ERIDANUS: Efficiently running inference of DNNs using systolic arrays," *IEEE Micro*, vol. 39, no. 5, pp. 46–54, Sep./Oct. 2019.
[29] B. Asgari, R. Hadidi, and H. Kim, "ASCELLA: Accelerating sparse computation by enabling stream accesses to memory," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2020, pp. 318–321.
[30] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, "Fine-grained accelerators for sparse machine learning workloads," in *Proc. 22nd Asia South Pacific Des. Autom. Conf.*, 2017, pp. 635–640.
[31] E. Nurvitadhi, A. Mishra, and D. Marr, "A sparse matrix vector multiply accelerator for support vector machine," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2015, pp. 109–116.
[32] U. Gupta *et al.*, "MASR: A modular accelerator for sparse RNNs," in *Proc. 28th Int. Conf. Parallel Architectures Compilation Techn.*, 2019, pp. 1–14.
[33] K. Hegde *et al.*, "ExTensor: An accelerator for sparse tensor algebra," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 319–333.
[34] K. Kanellopoulos *et al.*, "SMASH: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 600–614.
[35] 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, no. 3, pp. 30–38, Jun. 2017.
[36] Y. Huang, N. Guo, M. Seok, Y. Tsividis, K. Mandli, and S. Sethumadhavan, "Hybrid analog-digital solution of nonlinear partial differential equations," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 665–678.
[37] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient SpMV operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 347–358.
[38] R. Nair *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM J. Res. Develop.*, vol. 59, no. 2/3, pp. 17:1–17:14, 2015.
[39] M. Zhang *et al.*, "GraphP: Reducing communication for PIM-based graph processing with efficient data partition," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 544–557.
[40] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 457–468.
[41] J. C. Beard, "The sparse data reduction engine: Chopping sparse data one byte at a time," in *Proc. Int. Symp. Memory Syst.*, 2017, pp. 34–48.
[42] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011, Art. no. 1.
[43] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," *ACM SIGPLAN Notices*, vol. 51, no. 8, 2016, Art. no. 11.
[44] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 375–386.
[45] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 239–252.

Bahar Asgari is currently working toward the PhD degree with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia, and is a member of the Computer Architecture and System Laboratory. As a graduate research assistant under the supervision of Prof. Sudhakar Yalamanchili and Prof. Hyesoon Kim, she conducts research in the field of computer architecture. Her research interests include but are not limited to accelerating sparse problems and deep neural networks, and scalable memory systems.

Ramyad Hadidi received the bachelor's degree in electrical engineering from the Sharif University of Technology, Iran, and the master's degree in computer science from the Georgia Institute of Technology, Atlanta, Georgia. He is currently working toward the PhD degree in computer science under the supervision of Prof. Hyesoon Kim at the Georgia Institute of Technology, Atlanta, Georgia. His research interests include but are not limited to computer architecture, edge computing, and machine learning.

Tushar Krishna received the BTech degree in electrical engineering from the IIT Delhi, India, in 2007, the MSE degree in electrical engineering from Princeton University, Princeton, New Jersey, in 2009, and the PhD degree in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, Massachusetts, in 2014. He is an assistant professor with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia, with an adjunct appointment with the School of Computer Science.

Hyesoon Kim (Member, IEEE) received the PhD degree in electrical and computer engineering from the University of Texas at Austin, Austin, Texas. She is an associate professor with the School of Computer Science, Georgia Institute of Technology, Atlanta, Georgia. Her research areas include the intersection of computer architectures and compilers, with an emphasis on heterogeneous architectures, such as GPUs and accelerators.

Sudhakar Yalamanchili (Fellow, IEEE) received the BE degree in electronics from Bangalore University, India, and the PhD degree in electrical and computer engineering from the University of Texas at Austin, Austin, Texas. He was a Regents professor and Joseph M. Pettit professor of computer engineering with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia. Prior to joining Georgia Tech, Austin, Texas, in 1989, he was a senior and then principal research scientist at the Honeywell Systems and Research Center in Minneapolis. He was a member of ACM.