Azul: An Accelerator for Sparse Iterative Solvers Leveraging Distributed On-Chip Memory

Axel Feldmann
MIT CSAIL
axelf@csail.mit.edu

Courtney Golden
MIT CSAIL
cgolden@csail.mit.edu

Yifan Yang
MIT CSAIL
yifany@csail.mit.edu

Joel S. Emer
MIT CSAIL/NVIDIA
emer@csail.mit.edu

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00054

***Abstract*—Solving sparse systems of linear equations is a fundamental primitive in many numeric algorithms. Iterative solvers provide an efficient way of solving large, highly sparse systems. However, iterative solvers are inefficient on existing architectures because they perform computations with *(1)* poor short-term reuse, which causes frequent off-chip memory traffic; and *(2)* challenging data dependences, which limit parallelism.**

We present Azul, a hardware accelerator that achieves high arithmetic intensity by keeping data in distributed on-chip SRAM. Azul is organized as a grid of tiles, each with a small memory and a simple processing element (PE). This enables keeping solver data on-chip across iterations, achieving high reuse. We present a novel scheduling algorithm that maps data and computation across PEs to avoid communication bottlenecks while achieving high parallelism, and a specialized PE that achieves high utilization of arithmetic units.

When tested on a representative set of matrices for sparse iterative solvers, Azul is gmean 217× faster than state-of-the art GPU implementations, 159× faster than a previously proposed accelerator for sparse iterative solvers, and 90× faster than a previously proposed distributed-SRAM accelerator.

***Index Terms*—Hardware accelerators, sparse linear algebra, iterative solvers, all-SRAM architectures**

I. Introduction

Solving sparse systems of linear equations is a key computation at the heart of many numeric algorithms. In linear algebra terms, solvers find a vector x such that $Ax = b$, where A is a sparse matrix and b is a known right-hand-side vector. Linear solves dominate the performance of optimization solvers, physics simulations, and engineering tools.

Iterative solvers are a widely used class of methods to solve linear systems (Sec. II). Iterative solvers make some initial guess for x, then iteratively refine it until it converges to the correct solution [9, 10, 24, 31].

Unfortunately, iterative solvers are very inefficient on existing hardware due to a combination of frequent main memory accesses and challenging parallelism (Sec. III). Fig. 1 shows the utilization of an NVIDIA V100 GPU (and percentage of peak throughput in GFLOPS/s), when solving several representative sparse matrices using preconditioned conjugate gradients (PCG), a common iterative solver. Even on the most favorable matrix, the GPU achieves only *0.6%* of its peak throughput!

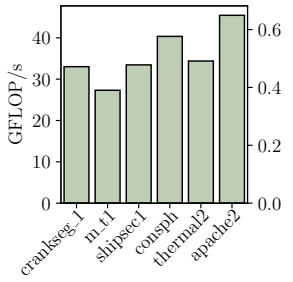

Fig. 1: Performance of a V100 GPU running Ginkgo `Cg` [3], a state-of-the-art iterative solver, on representative matrices.

This is because a few sparse matrices dominate the memory footprint. These matrices commonly take several tens of megabytes. Each iteration traverses these matrices and uses each value *once*, yielding *no intra-iteration reuse*. Conventional architectures like GPUs fetch these matrices from main memory each iteration, making memory bandwidth a key bottleneck and resulting in dismal performance (Fig. 1).

We present *Azul*, a hardware accelerator for iterative solvers that addresses these shortcomings. Azul relies on several key insights:

All-SRAM architecture: We can erase the main-memory bottleneck by building an architecture that fits operands on-chip and can thus exploit *inter-iteration reuse*. Solvers run for many iterations, with each iteration accessing the same matrices. Allocating enough on-chip SRAM to fit these matrices provides a step-function in reuse. In order to provide high-bandwidth, low-latency access to this SRAM, we adopt a *distributed-memory architecture* consisting of many small, spatially distributed *tiles* of memory each with a nearby processing element (PE).

SRAM is large enough: Within a single die, SRAM sizes are limited to moderately sized matrices. However, many numeric applications operate on matrices that fit in single-die SRAM, yet run for hours, as they simulate systems across many (sometimes millions of) timesteps. In Sec. II-C, we describe in detail how solving the system state at each timestep involves at least one linear solve, and these applications' working sets fit into chips using *current* SRAM technology. For example, analog simulation of one read and one write to a 128×32b SRAM generated by OpenRAM [28] takes Xyce [59], a state-of-the-art open-source simulator, 3.5 hours on a 24-core CPU, despite only 1.7 million matrix nonzeros. Similarly, non-linear magnetic simulations [55] take hours with just 3.7 million nonzeros.

There is a class of applications whose system matrices are still too large for current commercial SRAM technology. In Sec. VI, we evaluate Azul assuming SRAM sizes that do not currently fit into single-reticle-sized dies but that *would* apply to wafer-scale chips or multi-chip-modules, to show that Azul's techniques scale gracefully to larger system and problem sizes.

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00054

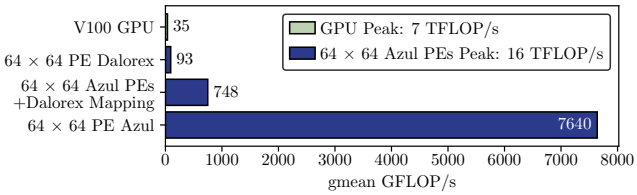

Fig. 2: Azul's performance on iterative solvers compared to baseline systems.

SRAM *alone* is not sufficient: Prior work has proposed distributed all-SRAM architectures for other domains, including machine learning [1, 13] and graph processing [44]. However, these designs are ill-suited for iterative solvers: some are incapable of handling unstructured sparsity, while others have PE designs that limit throughput due to control overheads. Fig. 2 shows that despite having all data on-chip, Dalorex [44], a state-of-the-art distributed SRAM accelerator, only achieves a gmean 2.3× speedup over the GPU baseline on iterative solves.

This low performance stems from two key problems: *(1)* Dalorex's in-order cores limit throughput due to their high control overheads. Most instruction issue slots go to bookkeeping and address calculation, leading to low compute utilization. *(2)* Careful data placement is essential for high performance, but Dalorex neglects it. (This is because Dalorex targets graph processing, not iterative solvers.) The subset of operands present in each tile's memory dictates that tile's need to exchange data with other tiles over the network-on-chip (NoC). Poor data placement results in low arithmetic intensity at a network level (FLOPs/network traffic), making applications network-bound.

We contribute novel hardware and software techniques that erase these bottlenecks. First, we design simple PEs that leverage specialization, dataflow execution, and lightweight multithreading to achieve high utilization of floating-point units. Fig. 2 shows that, while Azul's PEs improve performance by 8× over Dalorex's general-purpose cores, using prior data placement techniques makes the problem communication-bound and results in only $\approx 4.5\%$ of peak compute throughput.

Second, to increase tile-level reuse and remove the NoC bottleneck, Azul adopts a hypergraph-partitioning based data mapping approach. Data values (matrix nonzeros and vector elements) are individually considered for placement at each of Azul's PEs. This fine-grained mapping reduces traffic by 66× gmean, and provides a 10.2× speedup (Fig. 2). While our mapping algorithm is costlier than the simple heuristics of prior work, its large performance gains and the long-running nature of solvers more than compensate this cost (Sec. VI).

We evaluate an Azul implementation with 4096 (64×64) small tiles, each with 96 KB of SRAM (Sec. VI). Tiles are connected with a 2D-torus NoC. This system has 432 MB of SRAM and an aggregate 196 TB/s of SRAM bandwidth, with a modest 6 TB/s network bisection bandwidth. As Fig. 2 shows, Azul achieves high performance and utilization, beyond 8 TFLOP/s of double-precision computations, across a wide range of matrices. Overall, Azul outperforms a GPU by gmean 217×, and prior accelerators for iterative solvers by 159×.

In summary, we make the following contributions:

- The observation that sparse iterative solvers are well suited to distributed-SRAM architectures.
- A novel high-quality data mapping algorithm that substantially reduces communication over prior approaches.
- A specialized PE design that achieves high utilization.
- An accelerator, Azul, that combines these techniques to outperform prior accelerators by over two orders of magnitude.

II. Background

Solving systems of linear equations, i.e., solving x such that $Ax = b$ given matrix A and vector b, is a dominant kernel in many scientific applications [21, 25, 30, 54]. There are two classes of solvers: *iterative* solvers work by starting with an initial guess of x, and refining it each iteration, until they converge to a sufficiently accurate value of x. *Direct* solvers, by contrast, work by *factoring* the A matrix, i.e., decomposing A into factors with a structure that makes solving $Ax = b$ easy. For example, LU factorization decomposes A into a product of a lower-triangular matrix L and an upper-triangular matrix U ($A = LU$). LU factorization is itself expensive, but makes solves cheap.

In this work, we focus on iterative solvers because they dominate in important cases [31, 35, 62]. Even discounting the high upfront cost of factorization, a common problem with direct solvers is that factors are much denser than A. For example, in LU factorization, the L and U matrices are *much* denser than A, so $nnz(L+U) >> nnz(A)$. In some cases, $nnz(L+U)$ can be as much as 1000× larger than $nnz(A)$. These large factors cause enormous storage and computation overheads. In contrast, at each iteration, iterative solvers do $O(nnz(A))$ work. Even if the iterative solver takes hundreds of iterations, it is often much faster. Prior work has shown that sparse matrix factorization is amenable to hardware acceleration [22]. Even so, iterative solvers are still vastly more efficient on many input matrices.

Sparsity: Iterative solvers normally process *extremely* sparse matrices ($< 0.001\%$ nonzeros). This is a result of problem structure: many physical systems only have *local* interactions within a large system. For example, when simulating a circuit containing millions of nodes, each node is only connected to a handful of neighbors. Solvers leverage this sparsity by storing and processing only the nonzero values of these matrices. Sparsity causes *irregular* memory access patterns and data dependences, as well as variations in the amount of work, all of which depend on the structure (sparsity pattern) of the input matrix.

Numerical stability and preconditioning: The performance of iterative methods is also a function of the *numeric values* in A, not just its sparsity pattern. With some matrices, the solver converges in relatively few iterations, while others make the iterative solver diverge. To ensure convergence, iterative algorithms rely on *preconditioners*: instead of solving $Ax = b$, they solve $PAx = Pb$, where P (called a preconditioner) is chosen using domain-specific knowledge [5, 15, 36, 49].

A. Preconditioned Conjugate Gradients (PCG) Solvers

To make our discussion of iterative solvers more concrete, we focus on the preconditioned conjugate-gradients (PCG)

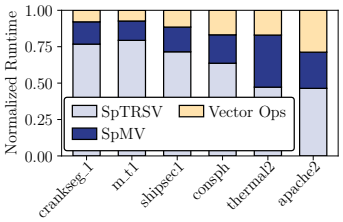

Fig. 3: Runtime breakdown by kernel of PCG (Ginkgo `Cg` [3]), running on one V100 GPU.

Matrix	SpMV	SpTRSV original	SpTRSV permuted
crankseg_1	884517	657	22409
m_t1	1219196	577	36217
shipsec1	1116200	947	37520
consph	858640	2560	52532
thermal2	2145078	1980	490417
apache2	1605956	2086	691630

TABLE I: Maximum available parallelism for SpMV and SpTRSV across representative matrices. Parallelism is total work divided by critical path length.

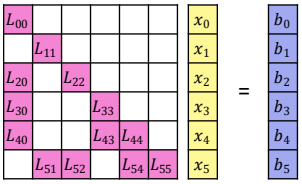

Fig. 4: Given a sparse triangular matrix L and a known right-hand-side vector b, SpTRSV finds x such that $Lx = b$.

```
1  def pcg(A, b, L, tol=1e-10):
2      x = zerovec(), r = b # residual
3      z = p = trisolve(L^T, trisolve(L,r))
4      while (||r|| > tol):
5          Ap = mvmul(A, p)
6          alpha = rz_old / dot(p, Ap)
7          x += alpha * p
8          r -= alpha * Ap
9          y = trisolve(L^T, trisolve(L, r))
10         rz_new = dot(r, z)
11         beta = rz_new / rz_old
12         p = z + beta * p
13         rz_old = rz_new
14     return x
```

Listing 1: Pseudocode for the preconditioned conjugate gradients (PCG) iterative solver (with an ICC preconditioner).

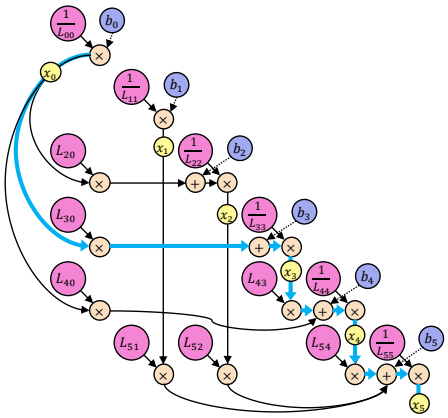

Fig. 5: Data dependences of SpTRSV on the matrix from Fig. 4.

algorithm. Variants of this algorithm are used as a benchmark for supercomputers [31].

Listing 1 shows pseudocode for an implementation of PCG. The algorithm takes as inputs the matrix A, the right-hand-side vector b, a lower-triangular matrix L (i.e., all values above its diagonal are zero) related to the preconditioner,[1] and a small tolerance value *tol* used to determine convergence. A and L are highly sparse matrices stored in a compressed format, while the vectors are dense. Nonetheless, since each row of A and L have several nonzeros, they are by far the largest data structures.

The `for` loop in lines 4–13 progressively refines vector x. Each iteration updates x (line 7), updates a residual vector tracking the value of $Ax - b$ for the current x (line 8), and chooses a new search direction (lines 9–12). Once the residual is small enough, the loop terminates. The mathematical details of how x is refined are not needed to understand PCG's performance.

PCG's performance is dominated by the operations that involve the *large matrices*, A and L: a sparse matrix-vector multiply (SpMV) with A (line 5) and two *sparse triangular solves* (SpTRSVs) using L and its transpose, $L^\top$ (line 9). Fig. 3 shows a breakdown of execution time of PCG on one V100 GPU (using Ginkgo `Cg` [3], a state-of-the-art iterative solver), showing that most time is spent in SpMV and SpTRSV; the other operations are simple vector operations (e.g., dot products).

To effectively accelerate PCG (and iterative solvers in general), we must accelerate these kernels, so we focus on them next. Note that even though GPUs are well suited to element-wise vector operations, reductions (present in vector dot product on lines 6-8, 10, and 12 of Listing 1) consume non-trivial amounts of time. This is because inter-thread communication and synchronization are expensive on GPUs. Furthermore, the all-to-all dependences on the dot products mean that the computation must be broken up across several kernels incurring repeated kernel launch overheads and excess data movement.

SpMV: Sparse-matrix times vector multiplication takes a sparse matrix M and multiplies it by a vector v to create a new vector y. The expression for each element of y is as follows:

$$y_i = \sum_{j \in nz(M_i)} M_{ij} v_j$$

where *nz* refers to nonzeros. Note two interesting properties of SpMV: *(1)* Each matrix element is read only once. This lack of reuse means that SpMV is memory-bound if the matrix comes from main memory. *(2)* Ample parallelism: products of any M_{ij} and v_j are independent and can be computed in parallel.

Since GPUs have more memory bandwidth than CPUs and can effectively parallelize this computation, they achieve sizable speedups on SpMV [26].

SpTRSV: Solving a triangular matrix is done by straightforward substitution. Consider the triangular system shown in Fig. 4: lower-triangular matrix L and vector b are known, and we seek x such that $Lx = b$. Finding x_0 is trivial: $x_0 = \frac{b_0}{L_{00}}$. However, consider finding x_2. By multiplying L and x, we obtain:

$$L_{20}x_0 + L_{22}x_2 = b_2$$

Rearranging, we get:

$$x_2 = \frac{b_2 - L_{20}x_0}{L_{22}}$$

[1] L is a matrix such that $LL^\top = P^{-1}$. It is computationally inefficient to store P and directly compute PAx. Instead, we can obtain PAx as follows:

$$v = P(Ax) \rightarrow P^{-1}v = Ax \rightarrow LL^\top v = Ax$$

which allows us to get v by a triangular solve with L, then one with $L^\top$.

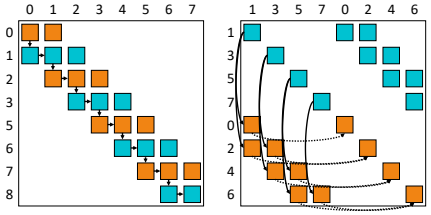

Fig. 6: Example of how graph coloring creates triangular solve parallelism. Note that while non-triangular A matrices are shown, the triangular solves in PCG operate on L and $L^\top$ matrices with the same sparsity pattern as A's lower and upper triangles.

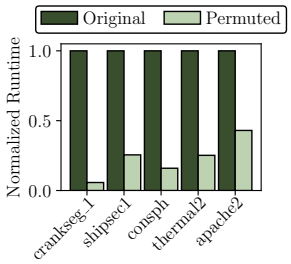

Fig. 7: Improving parallelism using graph coloring significantly improves solver performance on GPUs.

Algorithm	Preconditioner	Kernels
Conjugate	None	SpMV
Gradients	Diagonal/Jacobi	SpMV
	Sym. Gauss-Seidel	SpMV + SpTRSV
	Incomplete Cholesky	SpMV + SpTRSV
Power Iteration		SpMV
SSOR		SpTRSV
BiCGStab	None	SpMV
	Gauss-Seidel	SpMV + SpTRSV
	Incomplete LU	SpMV + SpTRSV

TABLE II: Many iterative solvers use SpMV and SpTRSV as their key kernels. Note that different preconditioners have different numeric properties. Selecting the appropriate solver and preconditioner for a specific problem is an unsolved problem in numerical algorithms.

which implies a *data dependence* on x_0. We can solve for x_2 only after finding x_0. In general, we have:

$$x_i = \frac{1}{L_{ii}}\left(b_i - \sum_{j=0}^{i-1} x_j L_{ij}\right)$$

Observe that like SpMV, each nonzero in L is read only once, causing similarly low arithmetic intensity. But *unlike* SpMV, SpTRSV does *not* have ample parallelism. In fact, the expression indicates that x_i depends on x_j for all j where $L_{ij} \neq 0$. Fig. 5 shows the data dependence graph of the solve from Fig. 4. The data dependence graph captures all dependences, and is directly derived from the matrix's sparsity pattern.

Parallelism-improving preprocessing: SpTRSV's irregular data dependences limit its parallelism. Different sparsity patterns inherently imply different levels of parallelism. Consider for example the tri-diagonal matrix shown on the left of Fig. 6. Naively, executing triangular solve on its lower triangle is a *purely sequential* computation. To solve each row, we must solve the previous row, eliminating any possible parallelism.

Graph coloring is a well-known [2, 43, 51] technique used to boost the parallelism of these inherently sequential computations. This involves treating a matrix as a graph and coloring its rows, then permuting *both its rows and columns* such that same-color rows are adjacent (Fig. 6, right). By definition, rows with the same color are independent, so placing them next to each other eliminates many dependences. Instead of being totally sequential, the triangular solve now has some parallelism.

Permuting a matrix can increase the iterative solver's iteration count, but the parallelism benefits *far* outweigh this cost on parallel architectures. Fig. 7 shows the speedups gained by graph coloring on a GPU, which are at least 2× and often much larger.

Table I demonstrates how graph coloring takes a *parallelism-limited computation* and significantly reduces this bottleneck. We estimate the maximum available parallelism of these computations by dividing the total number of operations by the length of the computation's critical path. Note that these estimates are approximate: they ignore data-movement latency and assume that all operations have single-cycle latency. However, it is clear that as a highly parallel hardware accelerator, it is crucial for Azul to take advantage of such state-of-the-art parallelism improving techniques. Also note that while this preprocessing step *enhances* available parallelism, Table I shows that it is still not boundless. *Unless otherwise specified, all results shown in this paper (including Fig. 1 and Fig. 3) use colored and permuted versions of input matrices.*

B. Other Solvers

Though Sec. II-A discusses PCG, the computations Azul accelerates are very general: other iterative solvers like GMRES and BiCGStab [50, 53] have the same kernels and challenges. Overall, the kernels described in this section form the basis of the vast majority of iterative solvers. Using SpMV and SpTRSV, we can implement all the widely used algorithms shown in Table II. Moreover, these algorithms are only a subset of the vast universe of iterative solvers based around these kernels.

C. Understanding End-to-End Applications of Solvers

Now that we've introduced iterative solvers, we motivate their importance and Azul's focus by looking at an important class of end-to-end applications: simulations of physical systems.

Fig. 8 shows the general structure of a physical system simulator. Simulation proceeds in timesteps. Each timestep begins with a solve of $Ax = b$; x is then used to update the values of b and, optionally, A, for the next timestep.

The meaning of A, x, and b is application-dependent. For example, when simulating the deformation of solid bodies (e.g., a car safety test), x is the vector of velocities of different points; b is the set of instantaneous forces; and A is a stiffness matrix that encodes how forces act upon the system. When simulating heat transfer through an object, x is the set of temperatures at each point; b is the heat stored at each node; and the A matrix encodes how heat transfers across the system.

Various types of simulations update b's and A's values differently. In some cases, for example heat transfer, A is static, and only b changes over time. b_{next} is calculated by a sparse matrix-vector product with the resulting x. Other cases have *simple* updates to A, e.g., in many rigid-body simulations, A_{next}'s nonzero values are a linear function of x. Finally, for more dynamic simulations, updating A is non-trivial (10-20% of

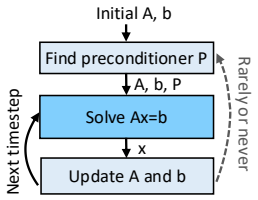

Fig. 8: Structure of a physical system simulator.

total FLOPs [37]). For example, when simulating elastic bodies (like a floppy-eared bunny), the stiffness matrix A changes with the system state. But even then, this computation is point-wise and highly parallelizable [8, 37].

Crucially, while A's values may change across timesteps, *its sparsity structure is static*. For example, a floppy-eared bunny is modeled as a mesh, and its nonzeros denote the connections of adjacent triangles; as the bunny's ears flop around, these connections do not change, only their stiffness values do.

Finally, when A changes over time, the preconditioner P may need to be updated to keep convergence fast. But updating P can be infrequent (A must change substantially to affect convergence), and takes a small fraction of time in most cases [11, 14, 27]. Like A, P's sparsity pattern is constant over time.

In summary, physical system simulation showcases why Azul's problem is important, and why we focus on linear solves:

1) Simulations with matrices that fit on-chip easily take hours: with 1µs per timestep and 1000 iterations per solve, simulating a system for only 10 seconds takes 10 billion inner-loop iterations—hours even with a few microseconds per iteration.
2) Reuse is very high, as A and b are reused across timesteps.
3) Since the sparsity structure is static, optimizing the placement of nonzeros is the right tradeoff: Azul's placement algorithm spends a few minutes to map each problem, but this cost is quickly recouped when the simulation takes hours.
4) Azul already has the necessary support to run many of these simulations end-to-end, e.g., when A is static or trivially updated, as in heat transfer. Even when the preconditioner needs to be updated, Azul already supports preconditioners like Gauss-Seidel, which simply takes A's lower triangle.
5) Azul does *not* support running certain types simulations end-to-end, e.g., those requiring non-trivial updates to A or recomputing complex preconditioners like incomplete Cholesky. But these comutations are either easy to parallelize or take negligible FLOPs, so simple extensions to Azul would enable end-to-end support at high performance.

III. Sparse Iterative Solvers are Ill-Suited to Prior Architectures

Due to their importance, iterative solvers are the focus of many prior hardware and software techniques.

GPUs: GPUs are the preferred current hardware platform for iterative solvers given their high compute throughput and memory bandwidth. But GPUs still suffer from the bottlenecks identified in Sec. II-A, and suffer poor utilization as shown in Fig. 1. Their small caches do not capture cross-iteration reuse, causing frequent memory accesses, and their programming model struggles with frequent data dependences.

ALRESCHA: Prior work has proposed hardware to accelerate iterative solvers. ALRESCHA [4] is a hardware accelerator that aims to accelerate the SpMV and SpTRSV steps *within one iteration*.[2] Since these steps do not have any intra-iteration reuse, ALRESCHA uses specialized processing elements (PEs) to reduce control overheads resulting in it saturating its main-memory bandwidth (288 GB/s in their implementation). However, this memory bandwidth bound limits ALRESCHA's throughput to 48 GFLOP/s, roughly in line with GPUs.

Distributed-SRAM accelerators: As both GPUs and ALRESCHA are severely bottlenecked by main memory, we now consider distributed-SRAM architectures. These consist of tiles, each with a small SRAM and a core or PE. PEs directly access local memory and communicate asynchronously with other PEs over a network.

There is a rich history of systems of this type, dating back to the J-Machine [7, 20, 42, 58, 61]. There have been recent efforts to develop such architectures in both industry, including Cerebras [13] and Groq [1], and academia, including Dalorex [44] and Tascade [45]. These architecture have been used to accelerate workloads with low arithmetic intensity, like unbatched LLM inference and graph processing. The Cerebras Wafer-Scale Engine has even been used to accelerate sparse iterative solvers [64]. However, this work is limited to the narrow class of *grid-structured problems*, which are completely regular.[3] This limitation is due to Cerebras hardware features, such as a circuit-switched NoC.

Dalorex [44] generalizes the Cerebras architecture for *unstructured* problems like the ones Azul targets. It features a packet-switched 2D-torus, and uses simple in-order cores as its processing elements. Dalorex is evaluated on sparse algorithms (graph processing, SpMV) with large unstructured inputs, where it achieves large speedups. However, Dalorex is not optimized for sparse iterative solvers. Despite its all-SRAM architecture with vastly more memory bandwidth, Fig. 9 shows that a 4096-core 2 GHz Dalorex design running PCG achieves limited speedups over GPUs, with at most 187 GFLOP/s, 1% of its peak throughput (16 TFLOP/s, since each core can do 1 FMAC/cycle, and each FMAC is 2 FLOPs).

We select Dalorex as a baseline because it is the state-of-the-art all-SRAM accelerator that targets unstructured sparse computations. But Dalorex suffers from two performance bottlenecks that make it ill-suited for sparse iterative solvers. First, its data mapping strategy causes excessive network traffic, making solvers *NoC-bound*. Mapping operand values (matrix nonzeros and vector elements) across tiles completely determines the amount of inter-tile data transfer over the NoC.

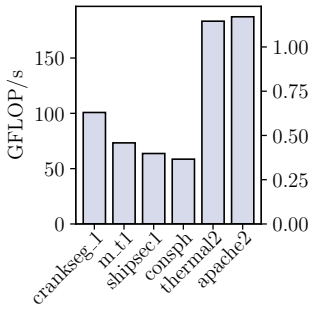

Fig. 9: Dalorex performance running PCG.

[2]The ALRESCHA paper does not directly mention SpTRSV, but instead mentions symmetric Gauss-Seidel (`SymGS`). This is equivalent to two consecutive triangular solves.

[3]In a grid-structured problem, the system is modeled as a regular n-dimensional grid. While the resulting A matrix is sparse, is has a completely regular structure. For example, in a 2D grid, each point (i, j) has four neighbors $(i \pm 1, j \pm 1)$, so each row of A has four nonzeros at fixed offsets. Thus, solvers like Cerebras's map grid points to PEs and do not even materialize the matrix. But in many cases, e.g., when simulating a car or a floppy-eared bunny, the A matrix encodes an unstructured mesh and is irregular.

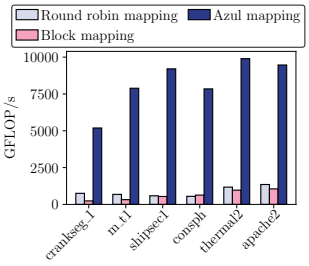

Fig. 10: Performance on PCG of Azul with idealized PEs under different mapping strategies.

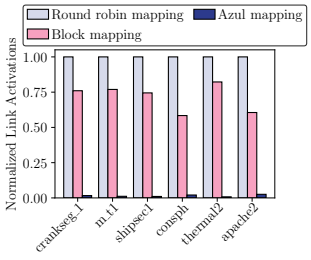

Fig. 11: NoC traffic on PGC under different mapping strategies.

Dalorex's mapping strategy, which we call Round Robin Mapping, partitions a data structure by listing all its nonzeros in row-major order and then assigning each nonzero i to PE i mod P, where P is the number of PEs. Tascade [45] adopts a variation on this approach, which we call Block Mapping, that constructs the same list of nonzeros but instead maps sequential blocks of $\lceil \frac{nnz}{P} \rceil$ nonzeros to each PE. High-performance computing systems use similar techniques (see Sec. IV-E). Both Round Robin and Block Mapping strategies are *sparsity-pattern agnostic*, grouping nonzeros purely based on their position in the row-major enumeration, not their matrix coordinates. As a result, they achieve poor reuse within each tile, forcing PEs to frequently communicate data over the NoC. This severely limits system performance. Fig. 10 and Fig. 11 show the NoC bottleneck in detail. To focus the discussion on network traffic as a driver of performance, we present the results of running PCG with Round Robin, Block, and Azul mappings on hardware that uses *idealized PEs* that run each task as fast as possible. Despite this idealization, the Dalorex and Tascade mappings deliver only a fraction of peak compute throughput (Fig. 10) due to their much higher network traffic (Fig. 11). In Sec. IV, we present Azul's data mapping strategy, which dramatically reduces NoC traffic.

Secondly, Dalorex's in-order cores suffer from high control overhead when executing iterative solvers. Computation in iterative solves features frequent address calculations and branches. Thus, when compiling to a CPU program, the large majority of instructions being executed are *not computing numeric results*. Such high overheads result in limited floating-point instruction throughput. In Sec. V, we present Azul's more specialized PE architecture that is designed to avoid these overheads.

More specialized accelerators: FDMAX [38] is an accelerator to solve PDEs using the finite difference method (FDM). FDMAX uses a custom iterative solver highly specialized to its problem domain. Importantly, FDMAX *only* targets grid-structured sparsity (like Cerebras), a regular problem. FDMAX is built around FDM-specific optimizations and follows a systolic design to exploit problem structure. Despite being more specialized, FDMAX is only 2.9× faster than ALRESCHA, as it suffers the same bottleneck: memory bandwidth.

IV. Azul Data Mapping Algorithm

In this section, we show that *effective data placement is essential to high performance* and present Azul's hypergraph-partitioning-based data mapping algorithm, which greatly reduces NoC traffic and makes most matrices compute-bound.

A. Dataflow Execution of Azul Kernels

We first show how communication arises in Azul. Azul kernels are structured as a *dataflow graph of tasks*. All memory accesses are local, and inter-tile communication occurs when a task on one tile sends a message to another, which triggers a task on the destination tile.

We illustrate our approach using SpMV as an example. Suppose we want to compute the matrix-vector product shown in Fig. 12 on a toy 2×2-tile Azul system. Operands are *mapped* across the four tiles, so each tile holds only a subset of M, v, and y values. Fig. 14 shows an example mapping.

Fig. 13 shows the corresponding dataflow graph of tasks for SpMV given the illustrated mapping. Each task is a small piece of computation that is triggered by the arrival of a message from the parent task, which may involve sending messages to trigger child tasks. Task execution is local to a specific tile. There are three types of tasks for SpMV: *(1)* Initially, each tile sends the input vector elements that it holds to other tiles that need them. Specifically, each v_j is multicast to tiles holding a nonzero from M's j^{th} column, using `SendV` tasks. *(2)* Receiving each v_j triggers a `ScaleAndAccumCol` task (`SAAC` in Fig. 13), which multiplies v_j with all the local M nonzeros from the j^{th}-column and accumulates these results into local per-row partial sums. Listing 2 shows the pseudocode of the `ScaleAndAccumCol` task. When all local values for the i^{th} row of M have been updated, if the final value y_i is mapped to a different tile, the partial sum t_i is sent to the tile. *(3)* Finally, each received partial sum triggers a `ReduceY` task that accumulates it into the final sum.

As an example, consider the `ScaleAndAccumCol` task executed on Tile_{10}, marked in red in Fig. 13. Since Tile_{10} holds M_{20} and M_{30}, Tile_{00} first sends v_0 to Tile_{10}, as shown in Fig. 14, The arrival of v_0 triggers the `ScaleAndAccumCol` task. The task multiplies v_0 with M_{20} and M_{30} and accumulates their results into local partial sums for rows 2 and 3, respectively, as shown in Fig. 15. Tile_{10} is the home tile for y_2, so it will receive row-2 partial sums from other tiles. However, the home for y_3 is Tile_{11}, so Tile_{10} sends its row-3 partial sum, t_3, to Tile_{11}. Fig. 15 shows these messages, which implement inter-tile reductions. Execution concludes after all reductions finish, with the result vector y stored distributed in its assigned tiles.

To write a kernel in this dataflow manner, a programmer defines a set of tasks and a set of message types, and then specifies the types of messages that can trigger each task and the messages that each task can send. SpTRSV has a task similar to SpMV's `ScaleAndAccumCol` that is triggered by multicasting solved variables, as well as additional tasks to solve the variable and write the final result. Implementing iterative solvers also involves some kernels beyond SpMV and SpTRSV, such as dot-products, but these kernels consume a small fraction of overall execution time and are thus less relevant to performance.

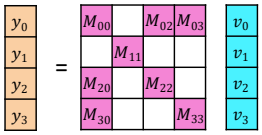

Fig. 12: An example sparse matrix-vector product (SpMV) computing $y = Mv$.

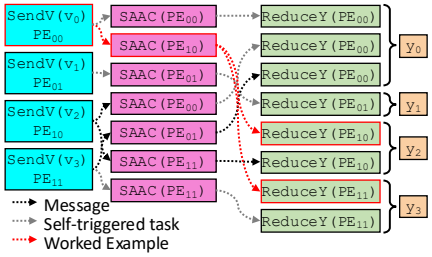

Fig. 13: Dataflow graph of tasks for the SpMV example.

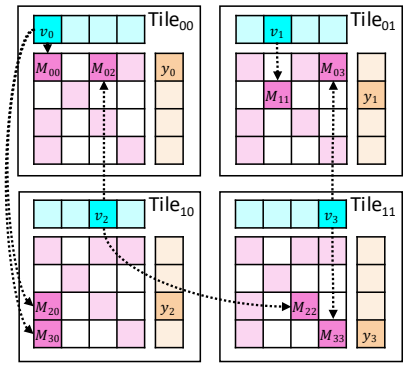

Fig. 14: Executing the SpMV from Fig. 12 on a 4×4 array of Azul PEs requires *multicasting* values v_j to tiles holding *any* nonzeros from M's jth column.

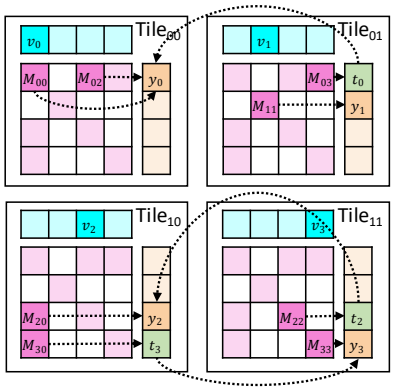

Fig. 15: Each PE keeps a partial sum for each row of nonzeros it stores. If a partial sum t_i is computed on a PE that is not y_i's home PE, an *inter-PE* reduction is needed.

```
void ScaleAndAccumCol(msg_t msg) {
    int n = *msg.col_start;
    for (int idx = 1; idx <= n; idx++) {
        M_val_t Mij = *(msg.col_start + idx);
        ps_t* ps = &partial_sums[Mij.row];
        ps->val += (msg.val * Mij.val);
        if (--ps->updates_remaining == 0)
            send(ps->dest, ps->val);
    }
}
```

Listing 2: The dominant task (`ScaleAndAccumCol`) in SpMV

This example shows that the amount of inter-tile traffic is determined by how data are mapped across tiles. Good mappings improve locality, so that more tasks can trigger local child tasks instead of sending messages to other tiles.

B. Data Mapping Problem Formulation

The objective of our data mapping algorithm is to maximize performance subject to three constraints: *(1)* limited capacity at each tile's small memory, making it important to load-balance data across tiles; *(2)* limited network bandwidth, making it important to minimize communication; and *(3)* data dependences, making it important to avoid unnecessary serialization.

We first discuss our basic formulation of the mapping problem, which optimizes for objectives (1) and (2), and then extend it to handle objective (3). Azul's mapping approach partitions all data structures in the kernel (e.g., the input matrix, input vector, and output vector for SpMV) simultaneously, with the goal of partitioning such that we maximize *locality* among data elements that are involved in the same computation.

In the SpMV example, each element of the input vector v_j forms a *communication set* with all the nonzeros in column j of the input matrix, because v_j must be multiplied with each of them. Similarly, each element of the output vector y_i forms a set with all nonzeros in row i of the matrix. If all data values in a set are co-placed on the same tile, no inter-tile communication is needed. However, if values are spread across multiple tiles, each input vector element must be sent to the other tiles containing data from its set and each partial sum must be reduced into the tile containing the corresponding output vector element. Inter-PE communication thus grows *linearly* with the number of unique tiles containing elements from a communication set. The same locality patterns hold for SpTRSV.

We represent these locality patterns using a *hypergraph*. A hypergraph is a mathematical structure with vertices and edges (like an ordinary graph) except that a hypergraph's edges (hyperedges) are between *sets* of vertices, instead of between just two vertices as in an ordinary graph. In this hypergraph, each *element* of each data structure is represented by a vertex, and each communication set is represented by a hyperedge that connects all of its elements. Fig. 16 (center) shows an example hypergraph for SpMV on a small matrix. Each nonzero data element is a node in the hypergraph, and orange and blue lines represent hyperedges connecting them.

Since each hyperedge connects elements of a communication set, hyperedges encode locality and represent the one-to-many or many-to-one communication that would occur over the network if the vertices in the set were placed on different PEs. Note that placing vertices in a set across N tiles induces $N-1$ messages. Thus, our goal is to keep each communication set restricted to as few tiles as possible—other factors, like the number of vertices mapped to each tile, do not affect communication.

Armed with this representation of the data structures, the mapping problem reduces to finding a partitioning of this hypergraph that load-balances vertices (matrix and vector values) while minimizing the total number of *cuts* of hyperedges across partitions. Minimizing edge-cut corresponds to maximizing row and column locality, since *cutting* an edge increases the number of unique tiles whose computation involves that row or column.

Fortunately, efficient hypergraph partitioning algorithms have been developed, which we can leverage [12, 34, 52]. Fig. 16 (right) shows an example of partitioning our small example hypergraph into three PEs using this approach. This particular example actually achieves the minimum possible number of hyperedge cuts given balanced partition sizes.

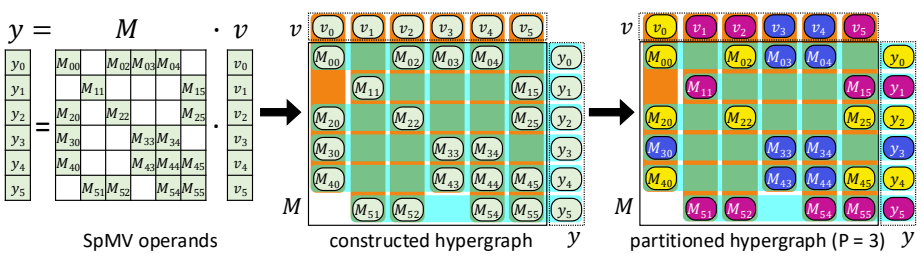

Fig. 16: Hypergraph partitioning formulation for an SpMV. In the eventual partitioned hypergraph, different vertex colors represent placement into different partitions.

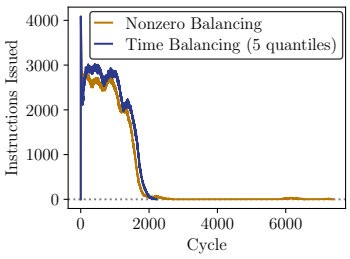

Fig. 17: Effect of time balancing on `consph` lower triangle SpTRSV.

C. Extensions for Limited Parallelism

We now introduce an extension to our hypergraph formulation of the mapping problem that incorporates objective (3), i.e., maximizing parallelism in the face of data dependences, which is important to avoid unnecessary serialization.

Partitioning using only locality and data balance as objectives can cause poor *load balance in time*. For example, hypergraph partitioning can fill specific Azul tiles with computations that are all either *early* or *late* in the dataflow graph's topological order. This results in long tails where a single PE holds up the kernel's completion. Fig. 17 shows a clear example of this when running PCG on the `consph` matrix. Our basic hypergraph partitioning formulation distributes the computation well in terms of *locality*, but poorly in terms of *temporal load balance*.

To integrate temporal load balancing into the hypergraph partitioning's objective, we bucket all nodes into q quantiles based on their associated arithmetic operation's depth (in the dataflow graph's topological order). Existing hypergraph partitioning algorithms allow for multiple balance constraints, so instead of simply balancing all elements across partitions, we apply a constraint to balance the elements *in each quantile* across partitions.

Fig. 17 shows how using this technique with $q = 5$ eliminates a long tail of instructions caused by PEs that are overloaded with tasks that run late in the kernel, yielding a 3.5× speedup on a single SpTRSV. Time-balancing helps when parallelism is limited and dependences are common (e.g., in SpTRSV), but does not help if parallelism is plentiful and the dataflow graph is shallow (e.g., in SpMV).

Finally, non-local reductions are more expensive than multicasts for two reasons: (1) non-local reductions incur an additional standalone `Add` operation that would otherwise be fused into an `FMAC`, and (2) in SpTRSV, reduction messages can be delayed due to queuing, delaying parallelism-revealing variable eliminations. To address this, we assign a larger weight to row hyperedges than column hyperedges, modeling this cost and discouraging breaking up rows instead of columns.

D. Generating Communication Patterns

Once data has been partitioned and placed, inter-tile communication patterns and PE tasks can be generated.

Implementing multicasts and inter-tile reductions naively using point-to-point messages, as shown in Fig. 18 (left) would cause two inefficiencies. First, it would cause redundant network traffic. For example, in Fig. 18, Tile_{33} must send a message to four other tiles, three of which are to its left (Tile_{11}, Tile_{31}, Tile_{61}). Sending separate messages would either use send three identical messages over the same east-west link, or use 3× as many east-west links as needed. Secondly, sending separate point-to-point messages may introduce serialization that lengthens an algorithm's critical path. For example, in both SpMV and SpTRSV, a single PE may be responsible for sending a value to hundreds or even thousands of other tiles. Sending these messages individually adds cycles and hurts performance.

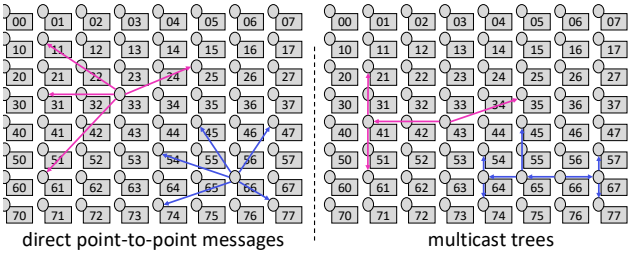

Fig. 18: Azul uses *multicast trees* to distribute values, avoiding redundant traffic over many point-to-point messages.

Instead, Azul's compiler creates *communication trees* to avoid these problems. Multicast trees are shown in Fig. 18 (right). In this example, Tile_{33} send a single east-west message to Tile_{31}, which forwards it north and south. The above also applies to reductions, where Azul implements *reduction trees*.

E. Hypergraph Partitioning in High-Performance Computing

While prior work [66] has used hypergraph partitioning for partitioning sparse matrices across GPU nodes on simple problems (i.e., SpMV), we are the first to apply it to splitting data within a single chip and the first to use it on parallelism-constrained problems like SpTRSV. Furthermore, these methods have achieved limited adoption due to mismatches between the partitioning formulation and the hardware and software constraints. GPUs lack the support for fine-grained communication necessary to make these methods effective. Papers exploring these techniques [46] explicitly state that these communication overheads are the key obstacle to wider adoption. These methods are therefore not supported by major scientific computing packages such as Petsc [41]. Instead, most high-performance computing systems, including distributed-memory systems in MPI environments, partition sparse computations across nodes using variants of Block Mapping [32, 40, 46].

V. Azul Microarchitecture

Azul's mapping strategy removes the *network bottleneck* by minimizing inter-tile communication. In this section, we discuss how Azul's hardware architecture then removes *computation bottlenecks* via specialization and fine-grained multithreading.

As illustrated in Fig. 19, Azul hardware is a tiled architecture with distributed memories. Each tile consists of a tightly integrated *processing element (PE)* and two *scratchpad memories*, as well as a *router* to communicate with other tiles through an on-chip network. The two SRAMs store data and program state, and are small (taking 108 KB per tile). This allows fast and low-energy accesses. Azul achieves high throughput by scaling to *a large number of tiles* (4096 in our implementation). This also provides high aggregate on-chip memory capacity (432 MB), and bandwidth (192 TB/s). Table III details the parameters and performance figures of our evaluated Azul configuration.

Azul's PEs execute tasks when triggered by the arrival of messages over the NoC. This message-driven approach is widely used in prior work [42, 47, 60]. However, the prior system targeting unstructured sparsity, Dalorex, uses *general-purpose in-order cores* as PEs, leading to high control overheads and low floating-point unit utilization.

A. Azul Tile

The Azul tile is designed to support dataflow execution of task graphs. A router connects the tile to the on-chip network and is responsible for sending and receiving messages, which trigger tasks. The multithreaded PE is designed to execute tasks with minimal control overhead and stalls. A Data SRAM (72 KB) holds input operands (matrix and vector values) and an Accumulator SRAM (36 KB) holds partial results. They are both 96-bit wide, allowing a 64-bit floating-point value and 32 bits of metadata to be accessed each cycle.

The Azul PE is designed to maximize the throughput of arithmetic operations, obtaining one arithmetic operation per cycle. It achieves this primarily through *specialized control flow*. We recognize that when SpMV and SpTRSV are executed in the dataflow manner described in Sec. IV, the dominant tasks in each kernel share the same control flow pattern and can be implemented using just `FMAC` operations and a custom `Send` operation. We discuss here this dominant control flow pattern and later extend the PE to support the remaining tasks in the SpMV and SpTRSV kernels. In SpMV, the dominant task is the `ScaleAndAccumCol` task (Listing 2). SpTRSV has a similar dominant task with identical control flow. The SpTRSV task is triggered by multicasting solved variables, and involves multiplying the solved variables by local column values, accumulating them to partial sums, and sending partial sums to the tiles that hold corresponding diagonal elements.

The shared control flow pattern is thus as follows: *(1)* at the start of a task, the number of arithmetic operations to be done (i.e., the for-loop count) given the received column index is determined by the number of local M nonzeros in that column (i.e., the result of the first memory access); *(2)* arithmetic operations (`FMAC` in this case) are executed; and *(3)* a branch at the end of each loop iteration (resolved at the output of the ALU) determines whether to send a message. Azul hardens this control flow pattern into the control logic of each PE's simple scalar pipeline. Such specialization significantly improves arithmetic throughput over a general-purpose core, which uses bookkeeping instructions that occupy pipeline slots.

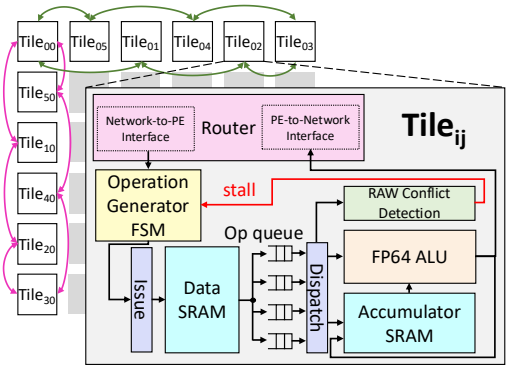

Fig. 19: View of a 6×6 Azul system showing the 2D-torus NoC as well as a more detailed Azul tile diagram.

Clock Frequency	2 GHz
Tiles	64×64 (4096 total)
Scratchpads	(72+36) KB SRAMs/Tile (432 MB total), 2 cycles per memory access, pipelined
Network	2D Torus, 96-bit links, 1 cycle/hop
Aggregate Compute Throughput	16 TFLOP/s (1 FMAC / PE / cycle)
Aggregate Scratchpad Bandwidth	192 TB/s (192 bits / PE / cycle)
NoC Bisection Bandwidth	6 TB/s

7-stage PE pipeline: Decode (1 stage); Data SRAM access (2 stages); Compute and accumulator SRAM read (4 stages: accumulator read and FMAC are partially overlapped, the fist two stages complete the read, and the last two perform the floating-point accumulation); Writeback/send (1 stage).

TABLE III: Parameters of our evaluated Azul configuration. Azul achieves high throughput and on-chip memory capacity by integrating a large number of simple PEs.

We now describe the key features of the Azul PE, which is designed around making this dominant control flow pattern fast. We extend it later in this section to incorporate lightweight multithreading and other tasks. Fig. 19 shows the fully pipelined microarchitecture of the PE. The control flow pattern described above maps to a sequence of `FMAC` operations. Each operation contains two sequentially dependent memory reads, the first to the Data SRAM (to fetch a nonzero and its row coordinate) and the second to the Accumulator SRAM (to fetch the row's partial output). After reading from the Data SRAM, the pipeline checks for data dependences (e.g., an in-flight operation for the same accumulator), then issues the operation, which reads the Accumulator SRAM and performs the `FMAC`. Finally, the `FMAC` result is either written back to the Acumulator SRAM or sent to another tile; to allow this, the final pipeline stage connects directly to the router.

In addition to specialized control flow, Azul also uses *fine-grained multithreading* to achieve high throughput. This is essential because even with a specialized control flow, data dependences can still cause stalls. For example, multiple SpMV `ReduceY` tasks in a row could accumulate partial results to the same y element consecutively, creating a data dependence.

We extend the PE design described above to keep the FMAC unit highly utilized. To support multithreading, we replicate the operation generator context (one per task) and the PE's intermediate operation queue so that operations from multiple tasks can be in-flight. Operations are chosen for execution from the earliest task that has no dependences on other in-flight operations. This ensures forward progress and avoids task starvation. Such a design hides stalls and achieves a throughput of one arithmetic operation per cycle.

The PE described above needs no additional hardware to execute all other tasks in SpMV and SpTRSV. We simply extend the operation generator FSM to produce two new simpler operations, Add and Mul, which flow through the same pipeline and use minimal additional logic to skip unnecessary functionalities (e.g., reading the Accumulator SRAM in Mul). Tasks like SendV and ReduceY map to 1–2 Add/Mul/Send operations.

Each tile contains a small register-based buffer for storing incoming messages. To avoid deadlocks, if the buffer becomes full, additional incoming messages are spilled to the Data SRAM.

B. On-chip Network

Azul uses a 2D-torus network topology. Each PE has its own router. Each cycle, the router is able to receive a message on all input queues and send a message on all output queues.

VI. Evaluation

We evaluate Azul on preconditioned conjugate gradients (PCG) with an incomplete-Cholesky preconditioner. PCG is commonly used and is representative of many other iterative solvers, which consist of SpMV and SpTRSV (Sec. II-B).

A. Experimental Methodology

Simulation infrastructure: We evaluate Azul using a cycle-level simulator with detailed timing models for the PEs and network. We model each hardware component as an object and tick each object for each cycle, thus simulating execution cycle-by-cycle. We faithfully simulate contention in the network and operation interleaving in PEs due to multithreading. We ensure functional correctness by checking the simulator's PCG results against a reference implementation [3].

We use RTL synthesis for Azul's custom PE, and standard modeling tools for the other components, combined with activity factors from simulation, to obtain area and power figures in 7nm technology, as detailed in Sec. VI-E.

Simulated system: By default, we model the 4096-tile Azul configuration in Table III. Sec. VI-G evaluates larger designs.

We implement PCG as shown in Listing 1, with dataflow tasks as detailed in Sec. IV. To remove long-latency floating point divisions from the computation's critical path, we store all diagonal elements d in memory as $\frac{1}{d}$.

Baselines: We compare Azul with three baseline architectures: *1. GPU* is an NVIDIA V100 PCIe GPU running Ginkgo [3], a state-of-the-art linear algebra library, to execute PCG with an incomplete Cholesky preconditioner.

Matrix	n	nnz	A	b	Matrix	n	nnz	A	b
s3dkt3m2	9.04e4	3.75e6	29	1	G3_circuit	1.59e6	7.66e6	59	13
cant	6.25e4	4.01e6	31	1	shipsec1	1.41e5	7.81e6	60	2
offshore	2.60e5	4.24e6	33	2	thermal2	1.23e6	8.58e6	66	10
pdb1HYS	3.64e4	4.34e6	34	1	m_t1	9.76e4	9.75e6	75	1
thread	2.97e4	4.47e6	35	1	crankseg_1	5.28e4	1.06e7	81	1
apache2	7.15e5	4.82e6	37	6	bmwcra_1	1.49e5	1.06e7	82	2
ecology2	1.00e6	5.00e6	39	8	hood	2.21e5	1.08e7	83	2
tmt_sym	7.27e5	5.08e6	39	6	pwtk	2.18e5	1.16e7	89	2
consph	8.33e4	6.01e6	46	1	BenElechi1	2.46e5	1.32e7	101	2
boneS01	1.27e5	6.72e6	52	1	nd12k	3.60e4	1.42e7	109	1
af_1_k101	5.04e5	1.76e7	134	4	Emilia_923	9.23e5	4.10e7	313	8
af_shell8	5.05e5	1.76e7	135	4	ldoor	9.52e5	4.65e7	355	8
bundle_adj	5.13e5	2.02e7	155	4	Hook_1498	1.50e6	6.09e7	465	12
msdoor	4.16e5	2.02e7	155	4	Geo_1438	1.44e6	6.32e7	482	11
StocF-1465	1.47e6	2.10e7	161	12	Serena	1.39e6	6.45e7	493	11
Fault_639	6.39e5	2.86e7	219	5	bone010	9.87e5	7.17e7	547	8
inline_1	5.04e5	3.68e7	281	4	audikw_1	9.44e5	7.77e7	593	8
PFlow_742	7.43e5	3.71e7	284	6					
Flan_1565	1.56e6	1.17e8	896	12	Queen_4147	4.15e6	3.29e8	2514	32
Bump_2911	2.91e6	1.28e8	975	23					

TABLE IV: Benchmark matrices used in the evaluation. Matrices in the first section fit in 4K tiles, matrices in the mid section fit in 16K tiles, and matrices in the bottom section fit in 64K tiles. The A and b columns report matrix and vector SRAM footprints, respectively, in MB.

2. ALRESCHA [4] is a prior accelerator for iterative solvers (Sec. III). We model it as a *full-utilization* accelerator that *completely saturates* its 288 GB/s main-memory bandwidth, and achieves perfect reuse on all vectors, so that the only memory traffic is from the sparse matrices in SpMV and SpTRSV. This generously overestimates ALRESCHA's actual performance.

3. Dalorex is modeled using the same configuration as Azul (Table III), except that each PE is a scalar RISC-V core. The core has a fully pipelined FPU that can do FMACs, ensuring the same peak throughput as Azul. Sends take a single instruction. We compile each task using gcc with -O3.

Data Mapping algorithms: We implement Azul's data mapping algorithms using PaToH v3.3 [12] to perform hypergraph partitioning. Sec. VI-C compares with prior works' mapping algorithms.

Input matrices: Because PCG works on symmetric positive-definite (SPD) matrices, we select large SPD matrices from SuiteSparse [19]. For most of the evaluation, we use the 20 largest SPD matrices that fit in 4096-tile Azul's memory, shown in Table IV.[4] They come from diverse domains such as circuit simulation, finite-element modeling, and computer vision.

In Sec. VI-G, we evaluate scaled-up Azul designs with 4× and 16× more memory. These designs fit all the largest SuiteSparse matrices (Table IV), which we use in these experiments.

We color and permute matrices with networkx.greedy_coloring [29] to increase available SpTRSV parallelism.

B. Performance Analysis

Speedup comparison: Fig. 20 shows the speedups of Azul, ALRESCHA, and Dalorex over the GPU baseline when running PCG. Matrices in this and later figures are sorted by their available parallelism (parallelism grows from left to right). As seen

[4]We remove near-duplicate matrices, e.g., we only take one (af_shell).

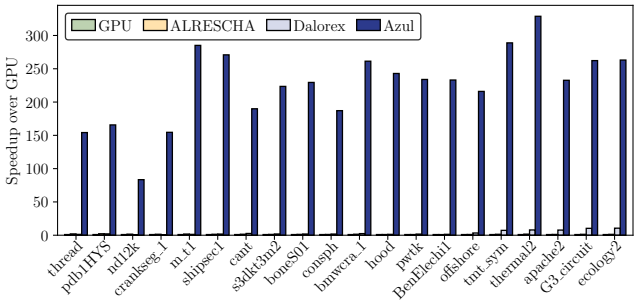

Fig. 20: End-to-end speedup comparison with baselines on PCG.

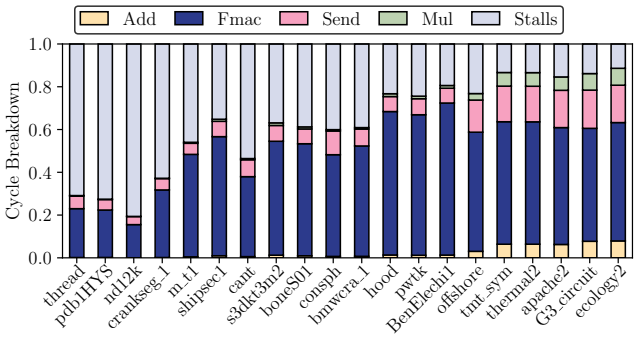

Fig. 21: End-to-end cycle breakdown of Azul PE.

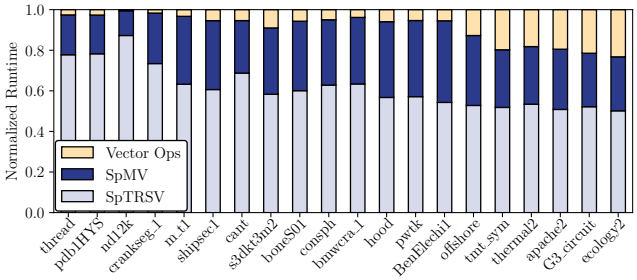

Fig. 22: End-to-end runtime breakdown by kernel.

in the figure, Azul significantly outperforms all baselines, with a gmean speedup of 217× over GPU, 159× over ALRESCHA and 90× over Dalorex. Azul achieves gmean 7,640 GFLOP/s, ranging from 2,541 GFLOP/s (`nd12k`) to 11,755 GFLOP/s (`BenElechi1`).

Each matrix's performance depends largely on its sparsity pattern. The sparsity pattern determines both the total available parallelism as well as the available row and column locality. All figures have matrices sorted left to right in order of increasing parallelism. Some matrices, specifically `thread`, `nd12k`, and `crankseg_1` are parallelism-bound (even with parallelism-improving preprocessing), which limits their compute throughput. Nonetheless, Azul's architecture and mapping strategy provide large speedups over all baselines. On the other end of the spectrum, many high-parallelism matrices such as `G3_circuit` and `ecology2` have only ≈ 5 nonzeros per row, so per-row fixed costs (message sending) become noticeable.

Note that achieving these throughputs without inter-iteration reuse of the matrices would require around 6 GB/s of off-chip memory bandwidth for every 1 GFLOP/s achieved—a totally infeasible 71 TB/s for `BenElechi1`.

PE cycle breakdown: Fig. 21 shows the breakdown of cycles spent in the Azul PEs. Overall, the PE achieves high throughput arithmetic operations (over 40% of cycles are spent on `FMAC` operations on almost all inputs). Stalls can be primarily attributed to limited parallelism in the SpTRSV kernel. Due to having few nonzeros per row, `thermal2` and `apache2` spend a higher fraction of time on reductions (i.e., sends and adds).

Breakdown by kernel: Fig. 22 shows the breakdown of cycles spent on SpMV, SpTRSV, and other vector operations for PCG. With Azul's acceleration, SpMV and SpTRSV are still the dominant phases of the computation. Because SpMVs have ample parallelism, they achieve consistently high performance across our test matrices. On the other hand, parallelism-limited SpTRSVs are slower and thus take a larger fraction of runtime.

C. Azul Data Mappings

To understand the impact of Azul's data mapping strategy, we compare against several baselines. Recall that Dalorex uses the **Round Robin** mapping and both Tascade and many distributed MPI-based systems use the **Block** mapping. Both of these approaches are examples of *position-based* [57] partitioning: data values are placed into partitions based on their *position* in the data's enumerated representation. These approaches are fast to compute and balance data per-partition by construction. However, they do not account for the target algorithm's communication patterns.

Prior work has separately proposed *coordinate-based* [57] partitioning approaches. These approaches place data values into partitions based on their *coordinate* in the data structure, regardless of the underlying hardware representation (e.g., compressed or uncompressed). **SparseP** [23] is a state-of-the-art hardware accelerator that uses coordinate-based partitioning. It creates chunks that are contiguous in coordinate-space and are of roughly equal size. It achieves this by first dividing the matrix into $\sqrt{P}$ chunks of contiguous columns (where P is the total number of partitions) such that each chunk contains (approximately) the same number of nonzeros. Then, it further subdivides each column chunk into $\sqrt{P}$ chunks of contiguous rows such that each has the same number of nonzeros. This produces P final chunks, each with roughly the same number of nonzeros. Each partition contains rows and columns that are contiguous in coordinate-space, but each partition may contain a variable number of rows and columns. We select SparseP as a baseline to represent coordinate-based partitioning approaches.

Fig. 23 shows Azul's end-to-end throughput against all the baseline mappings on PCG. Azul widely outperforms baseline mappings on every matrix: it outperforms Round Robin by gmean 10.2×, Block by 13.5×, and SparseP by 25.2×.

These large speedups happen because prior methods are based on non-robust assumptions about matrix structure. Namely, they only effectively minimize inter-partition communication if a matrix is *spatially correlated*, i.e., adjacent rows contain similar nonzero column coordinates. In some cases, this assumption holds; SparseP and Block mappings are then somewhat suited to SpMV. However, this assumption does not hold universally. As a

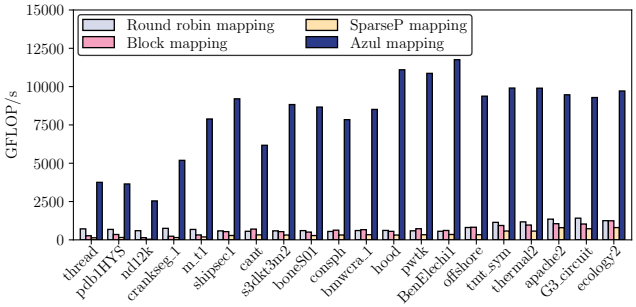

Fig. 23: End-to-end throughput comparison with prior mappings.

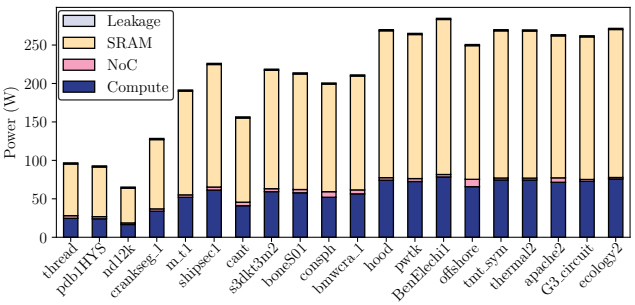

Fig. 24: Power breakdown by component.

result, Azul's mapping strategy achieves large reductions in NoC traffic: gmean 66× over Round Robin, 46× over Block, and 34× over SparseP. Furthermore, prior methods are not designed to efficiently handle triangular solves (or other computations with data dependences).

D. Data Mapping Algorithm Costs

Azul's data mapping approach *is* computationally expensive. Across the benchmark matrices, it takes an average of 6.16 minutes to map each of them to Azul's 4096 PEs. In contrast, it takes on average 0.25 minutes to map matrices with Block and 1.9 minutes to map them with Round Robin (Round Robin is much more expensive than Block because reduction trees become much more expensive to construct), and 0.6 minutes with SparseP. Despite Azul's mapping algorithm taking longer than baselines, this overhead is very well amortized: computationally expensive numeric algorithms perform linear solves at up to millions of timesteps, each using the same mappings. Furthermore, the preprocessing overhead can be amortized across *different* simulations using the same sparsity pattern. For example, if a user is solving for turbulent flow over an airplane wing in several situations, the connectivity of the meshed wing will be unchanged across simulations.

Prior work [62] has gone much further than Azul in terms of preprocessing costs: an FPGA-based accelerator, RSQP, goes as far as specializing an entire FPGA design to each input sparsity pattern, taking several hours per instance to compile.

Finally, Azul uses PaToH's `quality` preset. If mapping time is important, users could opt for a lower quality mapping by using the `default` or `speed` presets.

E. Area and Power Estimation

Methodology: We derive Azul's area and power at 7nm using the following methodology. We implement Azul's PE in RTL and synthesize it on ASAP7 (7.5-track [17]) using Synopsys Design Compiler with a 2 GHz target frequency.

For memory, we follow Dalorex's methodology and use figures from fabricated SRAM at 7nm [65], which achieves 29.2 Mb/mm^2 (3.75 MB/mm^2). To estimate SRAM energy, we use CACTI [6] to model each tile's memories. Since CACTI supports nodes down to 22nm, we scale energy down to 7nm. This yields 10.9 pJ per 96-bit read to a 36KB memory, which is similar to published results [33].

Component	Area
PEs	4096 × 0.0043 mm^2 = 17.8 mm^2
Routers	4096 × 0.0016 mm^2 = 6.6 mm^2
SRAMs	4096 × 0.0281 mm^2 = 115.2 mm^2
I/O	15 mm^2
Total	155 mm^2

TABLE V: Azul area estimates.

For the network, we estimate router and link area and energy using DSENT [56], which we scale from 22nm to 7nm, using ASAP7 technology. The 2D torus NoC links are short (two tile lengths, Fig. 19); and they use global wires that are routed above logic. ASAP7 global wires achieve 112ps/mm, so link traversal takes only 42ps, less than 10% of the cycle time.

Finally, for I/O, we conservatively estimate the area for a 512GB/s interface by using the area of an HBM2e PHY, 15 mm^2 [18, 48] (note that Azul has no off-chip memory).

Results: Table V shows the area breakdown of Azul. Overall, our Azul configuration has modest area at 7nm, about 155mm^2. As expected, SRAM takes most area, 74%.

Fig. 24 shows Azul's power consumption for each matrix, broken down by component (SRAM, compute, and network). We compute power by combining activity factors from simulation with energies from RTL synthesis and modeling tools, as described above. We include leakage and dynamic power. Azul consumes 210 W on average, and up to 288 W. SRAMs dominate energy due to the high rate of memory accesses.

Comparison with WSE-2: Estimating area and power in a modern technology node is necessarily approximate as we lack access to commercial PDKs and tools, like SRAM compilers. This said, we believe these estimates are reasonable given available details of fabricated designs. Specifically, the Cerebras WSE-2, built in TSMC N7, takes 38,000µm^2 per tile [39, slide 7]. Each tile has 48KB of SRAM, a core that's substantially more complex than Azul's PE, and a router. Each tile has a peak power of 30 mW at 1.1 GHz. Thus, a 4096-tile WSE-2 would take 155 mm^2 with a TDP of 123 W; Azul's peak power is higher due to its higher frequency and SRAM capacity.

Since Azul is SRAM-dominated, it's worth considering alternative design points. WSE-2's SRAM has somewhat lower density than our estimate above, 2.6 MB/mm^2 [39]. This is likely because the WSE-2 SRAM is built from smaller (6KB) banks to support two 128-bit reads and a write per cycle; even using this SRAM, Azul would take 195 mm^2, a reasonable area.

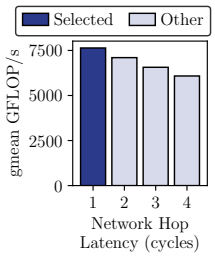

Fig. 25: Network latency sweep.

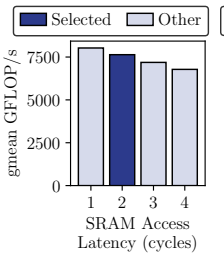

Fig. 26: SRAM access latency sweep.

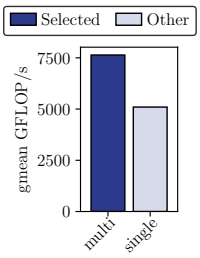

Fig. 27: Multithreading.

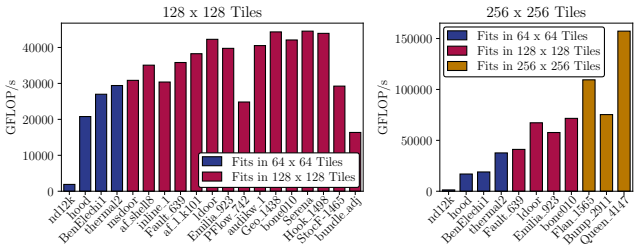

Fig. 28: PCG performance for scaled up Azul systems.

F. Sensitivity to Hardware Parameters

To demonstrate Azul's robustness to variations in hardware implementation choices, we sweep the NoC hop latency in Fig. 25 and SRAM access latency in Fig. 26. Increasing NoC hop latency decreases in gmean throughput by 4% per extra cycle, when simulating 1–4 cycles/hop. Increasing SRAM latency decreases gmean throughput by 3% per extra cycle. These show that Azul is barely sensitive to higher latencies.

In addition, we evaluate how much benefit our fine-grained multithreading provides. Fig. 27 shows that multithreading provides a 1.5× speedup over single-threaded PEs due to avoiding stalls on data dependences, as described in Sec. V-A.

G. Scaling Up

One of Azul's key limitations is that it can only accelerate linear solves for matrices that fit in its distributed SRAM. To solve larger problems, we must scale Azul up, moving to multi-die or wafer-scale technologies. In Fig. 28 we show the performance of running PCG on Azul configurations that are 4× and 16× larger than the default. We include matrices that do not fit into the 64×64 tile version of Azul, as well as some of the smaller system, to see how size affects scalability. The 16× larger system can fit all SuiteSparse matrices (Table IV).

In the 128×128-tile system, all but one matrix (`nd12k`) that fit into the 64×64 tile system have a $> 2\times$ speedup. `nd12k` is parallelism limited even on 4096 PEs, so it is not surprising that its performance does not improve on larger systems.

Similarly, many matrices become *parallelism limited* when moving from 128×128 to 256×256. Large matrices that only fit into 256×256 tiles, however, achieve very high throughput (up to 157 TFLOPs, 60% of peak), showing that Azul's techniques scale gracefully to large problem sizes.

VII. Conclusion

Iterative solvers for sparse linear systems of equations are an important and performance-critical class of algorithms. However, they are inefficient on existing architectures due to a lack of intra-iteration data reuse and the need for low-latency fine-grained synchronization. We have presented Azul, a hardware accelerator with distributed on-chip memory that exploits reuse *across* solver iterations and therefore overcomes the algorithms' memory bottlenecks. To effectively utilize Azul's hardware, we also present novel parallelism- and communication-aware data mapping algorithms. Through this hardware-software codesign, Azul achieves 217× gmean speedup over a GPU baseline and 159× gmean speedup over a prior accelerator architecture.

Acknowledgments

We thank Hyun Ryong Lee, Nikola Samardzic, Shabnam Sheikha, Fares Elsabbagh, Maggie Du, Alex Krastev, Taewoo Han, and our anonymous reviewers for their feedback on the paper. We also thank Andrew Ilyas and Alex Cohen for their valuable input on partitioning algorithms and Mark Hamilton for his help with benchmarking. This work was funded in part by the National Science Foundation under grant CCF-2217099, and by a Wistron research grant.

References

[1] D. Abts, J. Kim, G. Kimmell, M. Boyd, K. Kang, S. Parmar, A. Ling, A. Bitar, I. Ahmed, and J. Ross, "The Groq software-defined scale-out tensor streaming multiprocessor: From chips-to-systems architectural overview," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022.

[2] L. M. Adams and H. F. Jordan, "Is SOR color-blind?" *SIAM Journal on Scientific and Statistical Computing*, 1986.

[3] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A modern linear operator algebra framework for high performance computing," *ACM Transactions on Mathematical Software (TOMS)*, 2022.

[4] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, "ALRESCHA: A lightweight reconfigurable sparse-computation accelerator," in *Proc. HPCA-26*, 2020.

[5] O. Axelsson, "A generalized SSOR method," *BIT Numerical Mathematics*, 1972.

[6] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2017.

[7] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park, "An energy-efficient processor architecture for embedded systems," *IEEE Computer Architecture Letters*, 2008.

[8] J. Barbic, "Course notes FEM simulation of 3D deformable solids: A practitioner's guide to theory, discretization and model reduction. part 2: Model reduction (version: August 4, 2012)," in *SIGGRAPH*, 2012.

[9] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*. SIAM, 1994.

[10] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, "Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems," *Scientific Programming*, 2012.

[11] J. Brown, "Efficient nonlinear solvers for nodal high-order finite elements in 3D," *Journal of Scientific Computing*, 2010.

[12] Ü. V. Çatalyürek and C. Aykanat, "Patoh (partitioning tool for hypergraphs)." 2011.

[13] Cerebras Systems Inc., "The second generation wafer scale engine," https://cerebras.net/wp-content/uploads/2021/04/Cerebras-CS-2-Whitepaper.pdf, 2021.

[14] E. Chow, "Massive asynchronous parallelization of sparse matrix factorizations," U.S. Dept. of Energy, Tech. Rep., 2018.

[15] E. Chow and Y. Saad, "Experimental study of ILU preconditioners for indefinite matrices," *J. Comput. Appl. Math.*, 1997.

[16] L. T. Clark, V. Vashishtha, D. M. Harris, S. Dietrich, and Z. Wang, "Design flows and collateral for the ASAP7 7nm FinFET predictive process design kit," in *Proc. of the 2017 IEEE International Conference on Microelectronic Systems Education (MSE)*, 2017.

[17] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm FinFET predictive process design kit," *Microelectronics Journal*, 2016.

[18] S. Dasgupta, T. Singh, A. Jain, S. Naffziger, D. John, C. Bisht, and P. Jayaraman, "Radeon RX 5700 series: The AMD 7nm energy-efficient high-performance GPUs," in *Proc. of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2020.

[19] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, 2011.

[20] B. D. de Dinechin, "Kalray MPPA®: Massively parallel processor array: Revisiting DSP acceleration with the kalray MPPA manycore processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015.

[21] J. W. Demmel, *Applied numerical linear algebra*. SIAM, 1997.

[22] A. Feldmann and D. Sanchez, "Spatula: A hardware accelerator for sparse matrix factorization," in *Proc. MICRO-56*, 2023.

[23] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, "SparseP: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures," *Proc. of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*, 2022.

[24] G. H. Golub and H. A. van der Vorst, "Closer to the solutions: Iterative linear solvers," *The state of the art in numerical analysis*, 1997.

[25] G. H. Golub and C. F. Van Loan, *Matrix computations*, 4th ed. Johns Hopkins University Press, 2013.

[26] Y. Gui and G. Zhang, "An improved implementation of preconditioned conjugate gradient method on GPU," *Journal of Software*, 2012.

[27] A. Gupta, "WSMP: Watson sparse matrix package part iii: Iterative solution of sparse systems version 7.11," IBM, Tech. Rep., 1997.

[28] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *Proc. ICCAD*, 2016.

[29] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos National Lab, Tech. Rep., 2008.

[30] M. T. Heath, *Scientific computing: an introductory survey*, 2nd ed. McGraw-Hill, 2018.

[31] M. A. Heroux, J. Dongarra, and P. Luszczek, "HPCG benchmark technical specification," Sandia National Lab, Tech. Rep., 2013.

[32] M. Joshi, A. Gupta, G. Karypis, and V. Kumar, "A high performance two dimensional scalable parallel algorithm for solving sparse triangular systems," in *Proc. of the Fourth International Conference on High-Performance Computing*, 1997.

[33] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped Google's TPUv4i: Industrial product," in *Proc. ISCA-48*, 2021.

[34] G. Karypis, "hMETIS 1.5: A hypergraph partitioning package," http://www.cs.umn.edu/~metis, 1998.

[35] C. T. Kelley, *Iterative methods for linear and nonlinear equations*. SIAM, 1995.

[36] D. S. Kershaw, "The incomplete Cholesky–conjugate gradient method for the iterative solution of systems of linear equations," *Journal of computational physics*, 1978.

[37] F. Kjolstad, S. Kamil, J. Ragan-Kelley, D. I. W. Levin, S. Sueda, D. Chen, E. Vouga, D. M. Kaufman, G. Kanwar, W. Matusik, and S. Amarasinghe, "SimIt: A language for physical simulation," *ACM Transactions on Graphics*, 2016.

[38] J. Li, Y. Zhang, H. Zheng, and K. Wang, "FDMAX: an elastic accelerator architecture for solving partial differential equations," in *Proc. ISCA-50*, 2023.

[39] S. Lie, "Cerebras architecture deep dive: First look inside the hw/sw co-design for deep learning : Cerebras systems," in *IEEE Hot Chips 34 Symposium (HCS)*, 2022.

[40] Y. Liu, M. Jacquelin, P. Ghysels, and X. S. Li, "Highly scalable distributed-memory sparse triangular solution algorithms," in *Proc. of the SIAM Workshop on Combinatorial Scientific Computing (CSC)*, 2018.

[41] R. T. Mills, M. F. Adams, S. Balay, J. Brown, and A. Dener, "Toward performance-portable PETSc for GPU-based exascale systems," *Parallel Computing*, 2021.

[42] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-Machine multicomputer: An architectural evaluation," in *Proc. ISCA-20*, 1993.

[43] D. P. O'Leary, "Ordering schemes for parallel processing of certain mesh problems," *SIAM Journal on Scientific and Statistical Computing*, 1984.

[44] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, "Dalorex: A data-local program execution and architecture for memory-bound applications," in *Proc. HPCA-29*, 2023.

[45] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, "Tascade: Hardware support for atomic-free, asynchronous and efficient reduction trees," *arXiv preprint arXiv:2311.15810*, 2023.

[46] B. A. Page and P. M. Kogge, "Scalability of hybrid sparse matrix dense vector (SpMV) multiplication," in *Proc. of the 2018 intl. conf. on High Performance Computing & Simulation (HPCS)*, 2018.

[47] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Proc. ISCA-40*, 2013.

[48] Rambus Inc., "White paper: HBM2E and GDDR6: Memory solutions for AI," 2020.

[49] J. W. Ruge and K. Stüben, "Algebraic multigrid," in *Multigrid methods*, 1987.

[50] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, 1986.

[51] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[52] S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz, and P. Sanders, "High-quality hypergraph partitioning," *ACM Journal of Experimental Algorithmics*, 2023.

[53] G. L. Sleijpen, H. A. Van der Vorst, and D. R. Fokkema, "BiCGstab(l) and other hybrid Bi-CG methods," *Numerical Algorithms*, 1994.

[54] J. Solomon, *Numerical algorithms: methods for computer vision, machine learning, and graphics*. CRC Press, 2015.

[55] E. M. Stewart and L. Anand, "Magneto-viscoelasticity of hard-magnetic soft-elastomers: Application to modeling the dynamic snap-through behavior of a bistable arch," *Journal of the Mechanics and Physics of Solids*, 2023.

[56] C. Sun, C.-H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "DSENT: A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Proc. of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip (NOCS)*, 2012.

[57] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient processing of deep neural networks*. Morgan & Claypool, 2020.

[58] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS," in *Proc. ISSCC*, 2007.

[59] J. Verley, E. R. Keiter, and H. K. Thornquist, "Xyce: Open source simulation for large-scale circuits." Sandia National Lab, Tech. Rep., 2018.

[60] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proc. ISCA-19*, 1992.

[61] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *Computer*, 1997.

[62] M. Wang, I. McInerney, B. Stellato, S. Boyd, and H. K.-H. So, "RSQP: Problem-specific architectural customization for accelerated convex quadratic optimization," in *Proc. ISCA-50*, 2023.

[63] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, Sep 2007.

[64] M. Woo, T. Jordan, R. Schreiber, I. Sharapov, S. Muhammad, A. Koneru, M. James, and D. Van Essendelft, "Disruptive changes in field equation modeling: A simple interface for wafer scale engines," *arXiv preprint arXiv:2209.13768*, 2022.

[65] Y. Yokoyama, M. Tanaka, K. Tanaka, M. Morimoto, M. Yabuuchi, Y. Ishii, and S. Tanaka, "A 29.2 mb/mm2 ultra high density SRAM macro using 7nm FinFET technology with dual-edge driven wordline/bitline and write/read-assist circuit," in *Proc. of the 2020 IEEE Symposium on VLSI Circuits (VLSI)*, 2020.

[66] A. Yoo, A. H. Baker, and R. Pearce, "A scalable eigensolver for large scale-free graphs using 2d graph partitioning," in *Proc. of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.