Quartz: A Reconfigurable, Distributed-Memory Accelerator for Sparse Applications

Quartz: A Reconfigurable, Distributed-Memory Accelerator for Sparse Applications

Courtney Golden
MIT CSAIL
Cambridge, MA, USA
cgolden@csail.mit.edu

Axel Feldmann
MIT CSAIL
Cambridge, MA, USA
axelf@csail.mit.edu

Joel Emer
MIT CSAIL/NVIDIA
Cambridge, MA, USA
emer@csail.mit.edu

Daniel Sanchez
MIT CSAIL
Cambridge, MA, USA
sanchez@csail.mit.edu

Abstract

Iterative sparse matrix computations lie at the heart of many scientific computing and graph analytics algorithms. On conventional systems, their irregular memory accesses and low arithmetic intensity create challenging memory bandwidth bottlenecks. To overcome such bottlenecks, distributed-SRAM architectures are structured as an array of tiles, each with a processing element (PE) and a small local memory, to achieve very high aggregate memory bandwidth. However, current distributed-SRAM architectures suffer from either *poor programmability* due to over-specialized PEs or *poor compute performance* due to inefficient general-purpose PEs.

We propose Quartz, a new architecture that uses *short dataflow tasks* and *reconfigurable PEs* in a distributed-SRAM system to deliver both high performance and high programmability. Unlike traditional sparse CGRAs or on-die reconfigurable engines, Quartz allows reconfigurable compute to be highly utilized and scaled by (1) providing high memory bandwidth to each processing element and (2) introducing a task-level dataflow execution model that fits this new setting. Our execution model dynamically reconfigures each tile's PE in response to inter-tile messages to execute tasks on local data. This execution model enables fine-grained *data partitioning* across tiles. To make execution efficient, we explore novel data partitioning techniques that use graph and hypergraph partitioning to minimize network traffic and balance load in the face of both static-static and static-dynamic operand sparsity. To ensure programmability, we show how a wide range of Einsum-expressible computations and flexible data distributions can be systematically captured in small tasks for execution on Quartz.

Quartz's architecture, data partitioning techniques, and programming model together achieve gmean 21.4× speedup over a prior state-of-the-art system for six different iterative sparse applications from scientific computing and graph analytics.

CCS Concepts

• **Computer systems organization → Parallel architectures**.

Keywords

sparsity, distributed-memory architectures, reconfigurable computing, dataflow execution, scientific computing, graph analytics

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756035

ACM Reference Format:
Courtney Golden, Axel Feldmann, Joel Emer, and Daniel Sanchez. 2025. Quartz: A Reconfigurable, Distributed-Memory Accelerator for Sparse Applications. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756035

1 Introduction

Iterative sparse matrix computations lie at the heart of many modern applications across scientific computing and graph analytics, such as linear equation solvers, social network analysis, and bioinformatics [13, 41, 46, 79, 82]. Many such applications perform linear algebra operations on sparse matrices and sparse or dense vectors inside a loop that runs for many iterations. These workloads are characterized by (1) *high sparsity* (in the range of 99-99.9999% [39]), which introduces irregularity into memory access patterns, and (2) *low arithmetic intensity*, which increases the frequency of those accesses. These factors compound to place extreme pressure on memory bandwidth, which has led prior work to explore a plethora of accelerators that go beyond conventional memory hierarchies.

A natural fit for such memory-bound computations are architectures built around distributed, high-bandwidth local storage [2, 4, 20, 42, 59, 60, 71, 87, 91]. By placing a small memory at each of many parallel processing elements (PEs), such systems can achieve low-latency and low-energy memory accesses and very high aggregate bandwidth. Current technologies make SRAM best suited for this architectural style, with numerous commercial and academic works adopting tiled all-SRAM architectures for iterative workloads in scientific computing, graph analytics, and machine learning that store all data on-chip for reuse across iterations [2, 20, 30, 42, 59, 60].

However, existing distributed-SRAM architectures suffer from either **poor programmability** or **poor compute performance**. Some systems, like Azul [30], harden the control flow of a small number of kernels to achieve high performance in a narrow domain. However, the wide diversity of operations and control flow in iterative sparse computations demands flexible and programmable hardware. Unfortunately, existing *programmable* systems trade compute throughput for generality. Dalorex [59], for example, uses general-purpose cores as PEs and spends most instruction issue slots on bookkeeping and address calculation, forcing its compute rate below that which its memory bandwidth could sustain.

We propose a new architecture that uses short dataflow tasks and reconfigurable compute in a distributed-SRAM system to deliver both high performance and high programmability. The value of reconfigurability for sparse applications has been shown by prior work [24, 25, 45, 54], but existing systems are either bottlenecked on memory bandwidth, lack enough PEs to make use of their memory capacity, or feature execution models optimized for different design

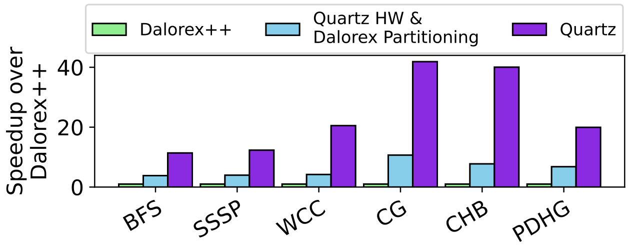

Figure 1: Contributions of compute hardware and partitioning techniques to Quartz's performance.

objectives that would hinder performance in a distributed-SRAM setting, such as by restricting data distribution choices across tiles. We enable reconfigurable compute to be highly utilized and scaled by (1) providing high memory bandwidth to each PE and (2) introducing a task-level dataflow execution model where PEs are dynamically reconfigured based on inter-tile messages to execute tasks on local partitions of data. This significantly raises the upper bound of performance over existing distributed-SRAM designs.

Our architecture, called Quartz, contains a tiled array of reconfigurable processing elements (PEs) that each access a private SRAM scratchpad and communicate through a scalable network. The iterative applications we target each yield a finite set of short local task types, and PEs are designed to efficiently execute instances of these tasks by configuring fabrics of functional units and operating on values from incoming network packets and the local scratchpad. Quartz decomposes the sparse matrix and vector computations found in its applications into a small set of sparse primitives and its PEs make their execution efficient. With these compute hardware design choices *alone*, Quartz outperforms a state-of-the-art distributed-SRAM architecture with general-purpose PEs by 5.7× and achieves 15.9× better performance per chip area (Fig. 1).

However, to fully harness the performance potential of reconfigurable distributed-SRAM systems, data must be carefully *partitioned* across tiles. Data partitioning is important for (1) *minimizing communication*, so that network traffic does not become a bottleneck, and (2) *balancing work* among tiles to avoid underutilization from waits for the "long pole" computation to finish. These objectives are well-studied and relatively straightforward to achieve in regular problems [85] but challenging and often directly at odds in sparse computations. Existing partitioning approaches either only address one objective [59, 60, 87] or address both only for cases in which all operands have *static* sparsity patterns (i.e., their distributions of nonzeros remain unchanged across iterations) [30]. *We are the first to both minimize communication and balance load for computations that also include one operand with dynamic sparsity.* We do so with an approach that includes graph and hypergraph partitioning and show that it improves performance by an additional 3.7× (Fig. 1).

To make such an architecture programmable, we show how to take computations expressed as cascades of Einsums [51, 57, 68] (using an extended notation to encompass graph algorithms and capture partitioning) and systematically decompose them into simple tasks that map well to reconfigurable logic. Our novel approach captures both computation and inter-tile communication in Einsums and their corresponding tasks. Operations are grouped into tasks differently than in existing graph analytics and scientific computing accelerators to enable Quartz's more flexible partitioning.

In summary, Quartz uses reconfigurable compute and task-level dataflow to capture the performance potential of high-bandwidth local storage for iterative sparse computations while maintaining flexibility across diverse workloads. Our novel data partitioning techniques make execution on this architecture significantly more efficient, and systematic task generation makes it programmable.

As a result, Quartz outperforms a state-of-the-art distributed-SRAM architecture, Dalorex++, by gmean 21.4× (Fig. 1) and an H100 GPU by gmean 93×. Although Quartz has a much larger area than an H100, it provides 21× higher performance per area.

We summarize our contributions as follows:

(1) A flexible programming model that translates sparse computations from Einsums (representing both computation and communication) into short, spatially partitioned tasks that can be run on reconfigurable compute units.
(2) A novel architecture that places task-driven reconfigurable compute engines in a tiled, distributed-SRAM array.
(3) The first exploration of partitioning methods that both balance load and minimize communication for computations with both static-static and static-dynamic operand sparsity.

2 Motivation and Background

In this section, we first describe the characteristics of iterative sparse matrix computations to motivate our architectural design choices and the need for programmability. We then show how Quartz fills a gap in a taxonomy of existing architectures in this space. Lastly, we provide background on the Einsum notation used in Sec. 3.

2.1 Characteristics of Iterative Sparse Matrix Computations

Iterative sparse matrix computations play a key role in scientific computing and graph analytics applications. Fig. 2 shows pseudocode for three example applications. Fig. 2(a) shows push-based Breadth-First Search (BFS). BFS finds the minimum path length from an origin node to all other nodes in a graph. Each iteration i takes the set of nodes that are i hops from the origin and produces the set at distance $i + 1$. Fig. 2(b) shows conjugate gradients (CG), an indirect iterative solver for linear systems of the form $Ax = b$. Each iteration refines a solution vector x via a sparse-matrix vector multiplication (SpMV) (line 5) followed by elementwise vector operations and dot products. SpMV is the computational bottleneck due to its irregular memory access patterns and lack of reuse on matrix elements. Fig. 2(c) shows Primal-Dual Hybrid Gradients (PDHG), which is the basis of many first-order methods for solving nonsmooth convex optimization problems [7]. It includes two SpMVs (lines 8 and 11) and vector operations of multiple lengths. All these applications are structured around loops with sequences of sparse matrix and vector operations (or operations of similar complexity). Due to their high sparsity and low reuse, such patterns do not map well to architectures with conventional memory hierarchies.

While the computations we target from scientific computing and graph analytics have unique operations and control flow that demand high programmability, they also share four key characteristics that inform our architectural design choices:

```
F = bitvec(NUM_NODES, 0) # frontier
F_next = bitvec(NUM_NODES, 0)
F[start_node] = 1
dist = vector(num_nodes, MAX_VALUE)
dist[start_node] = 0

i = 1
while F has at least one set bit:
  for src in range(0..NUM_NODES):
    if F[src]:
      for nbr of src:
        new_dist = i
        if new_dist < dist[nbr]:
          F_next[nbr] = 1
          dist[nbr] = new_dist
  F = F_next; reset F_next
  i++
```

Fig. 2(a): Pseudocode for push-based BFS.

```
matrices: A
vectors: x, r, p

for i in range(max_iter):
  Ap = matvec_mul(A, p)
  alpha = rs_prev / dot(Ap, p)
  x = x + alpha * p
  r = r - alpha * Ap
  rs = dot(r, r)

  if (rs < tol):
    return x

  beta = rs / rs_prev
  p = r + beta * p
  rs_prev = rs
```

Fig. 2(b): Pseudocode for CG.

```
matrices: K, KT
vectors: x, y, x_next, y_next, x_bar
scalars: p_resid, d_resid # residuals
constants: tau, sigma, c (vec), q (vec
      )
projection operators "clip" values

for i in range(max_iter):
  KTy = matvec_mul(KT, y)
  x_next = proj_x(x - tau * (c - KTy))
  x_bar = 2 * x_next - x
  Kx = matvec_mul(K, x_bar)
  y_next = proj_y(y + sigma * (q - Kx)
      )
  p_resid = norm(x_next - x)
  d_resid = norm(y_next - y)
  x = x_next; y = y_next
  if max(p_resid, d_resid) < tol:
    return x, y
```

Fig. 2(c): Pseudocode for PDHG.

(1) ***Iterative:*** The iterative nature of these computations is reflected in the matrix and vector operations inside loops that are executed hundreds or thousands of times. This allows us, like prior work [30, 59, 80], to amortize pre-processing overheads like data partitioning.
(2) ***Highly Sparse:*** Sparse matrices and vectors in scientific and graph workloads typically have 99-99.9999% sparsity [39].
(3) ***Static and Dynamic Sparsity:*** Operand sparsity patterns (i.e., the distribution of nonzeros) can either be *static*, remaining unchanged across iterations, or *dynamic*. In the algorithms we target, the largest (matrix) operands exhibit static sparsity, as they represent system state. In *all-active* algorithms, like CG and PDHG, all vectors contain nonzeros at every coordinate during every iteration. However, *non-all-active* algorithms, like BFS, contain dynamic vector sparsity. For example, the sparsity pattern of the frontier vector in BFS changes across iterations as nodes are added and removed. Dynamic vector sparsity creates *effectively dynamic sparsity* in the matrix operand, as the subset of the graph used during each iteration changes. Dynamic, time-varying behavior is challenging for data partitioning.
(4) ***Low Intra-Iteration Arithmetic Intensity:*** Within each iteration, the algorithms we target intrinsically have low arithmetic intensity (ratio of operations computed per byte of data loaded from memory) due to low intra-iteration reuse. For example, SpMV performs a single multiplication for each matrix element, suffering zero matrix reuse.

2.2 Taxonomy of Prior Architectures

Quartz fills a gap in existing literature by combining the throughput and flexibility of reconfigurable hardware with the per-PE bandwidth and capacity characteristics of distributed SRAM. Table 1 classifies existing accelerators applicable to iterative sparse matrix computations based on their compute and memory design choices. Columns indicate whether the key attribute that relieves memory bandwidth pressure is hierarchy (where most storage is in large, slow memories, with small caches or scratchpads near compute) or distribution (where scratchpads near compute are the primary storage element). We focus here on the rightmost column and bottom row to motivate Quartz; Sec. 7 covers the other categories.

Many commercial and academic systems adopt distributed-SRAM architectures for scientific computing, graph processing, and machine learning [2, 20, 30, 42, 59, 60]. However, prior work either uses specialized, fixed-function compute [30] that lacks programmability, or trades compute throughput for generality using general-purpose hardware. Such systems, like Dalorex [59], suffer from low memory access rates due to the large portion of instruction issue slots spent on bookkeeping and address calculation.

Table 1: Taxonomy of related work.

Compute Model	Shared Memory Hierarchy	Distributed High-Bandwidth Local Memory DRAM	 SRAM
Specialized Compute Units	GraphPulse [64] Graphicionado [37] ExTensor [39] Tensaurus [72] Sparsepipe [88] GUST [32]	GraphR [71] 3D-LiM [90]	Azul [30]
General-Purpose Cores	Prodigy [76] Pipette [53] DeSC [36] DSWP [66] TMU [70]	Tesseract [4] GraphP [87] GraphQ [91] NDC [63] TOP-PIM [86]	Dalorex [59]
Reconfigurable Compute Units	Alrescha [10] PolyGraph [24] Onyx [45] Fifer [54]	HRL [31] NDA [29]	Quartz (this work)

The potential for reconfigurable compute to balance performance with programmability for iterative sparse workloads has been demonstrated (last row of Table 1), but only in settings where *total system performance is bottlenecked by factors other than compute throughput* and, thus, different design tradeoffs make sense. For example, Alrescha [10] uses lightweight reconfigurable logic to accelerate the data-dependent portions of sparse computations, and PolyGraph [24] supports many different graph algorithm variants on CGRAs. However, sparse workloads remain memory-bound on such systems due to their conventional memory hierarchies, which caps performance and would underutilize highly scaled compute resources. In addition, the hardware and execution models of these systems are efficient when cache locality matters and load balance among data partitions (which are loaded from memory and processed over time) does not, but these goals are reversed in distributed-SRAM systems like Quartz.

Reconfigurable compute has also been explored in near-data processing (NDP) architectures (HRL [31], NDA [29]), which stack memory (DRAM in today's technology) atop compute to provide high per-PE bandwidth. However, the strict area and power budgets of stacked logic limits the number of PEs per memory stack, leaving too few PEs to make use of the large memory capacity. Distributing memory into smaller pieces (SRAM today) and scaling to many more PEs can increase parallelism and peak throughput, but necessitates a more flexible execution model. For example, vertex-based graph programming models co-locate all outgoing edges of a vertex and traverse them in a single task. We instead use fine-grained data distribution and optimize our hardware for short tasks.

2.3 Einsum Notation

To provide a formal mathematical definition of the sparse computations we target and our execution model, we use Einstein summation notation, or 'Einsums' [11, 68]. Einsums have been widely used to describe sparse tensor algebra [40, 44] and, recently, more general computations [51, 52, 57, 89].

Einsums describe operations on *tensors*, which are the multidimensional generalization of vectors and matrices. A tensor has one or more *ranks*, i.e., dimensions. For example, a vector is a rank-1 tensor, a matrix is rank-2. A *point* is a single data value in the tensor described by *coordinates* that index ranks. Ranks are represented with capital superscript and coordinates with lowercase subscript.

Nayak et al. [51] recently formalized the *operational definition of an Einsum*: "An Einsum specifies three things: (1) the input and output tensors involved and their ranks, (2) an iteration space containing a point for each computation to be performed, and (3) the specific computation to be performed at each point in the iteration space." For example, the Einsum for matrix-matrix multiplication is

$$Z_{m,n}^{M,N} = A_{m,k}^{M,K} \cdot B_{k,n}^{K,N}$$

There are two input tensors, A and B, on the right-hand side, and one output tensor, Z, on the left. Computing this Einsum requires first enumerating all valid coordinate tuples: here, $(M \times N \times K)$. At each point in this space, we multiply the specified points in the input operands, and populate the corresponding point in Z. If a value is already present, we *reduce* the two values (traditionally, using addition). Importantly, the Einsum specifies only *what* must be computed, with no requirements on the *order* in which the operations occur. Notably, for *sparse* tensors, some coordinates hold empty values; multiplications are only performed for coordinates that survive *intersecting* the two tensors' coordinate/value streams to find nonempty values.

In this work, we leverage two recent extensions to Einsum notation [57] that allow it to capture a wider range of applications: (1) replacing multiplication with any user-specified binary map operator and replacing addition with any reduce operator, and (2) using iterative ranks to represent changing tensor values over time.

3 Quartz Programming Model

Fig. 3 shows an overview of Quartz. Quartz is a tiled architecture with distributed SRAM and many parallel processing elements that communicate through a network. Computation is performed via task-level dataflow execution of short, per-tile tasks. In order for Quartz to be programmable for a wide range of applications, it is essential to derive a systematic method of converting applications to reconfigurable hardware-friendly tasks.

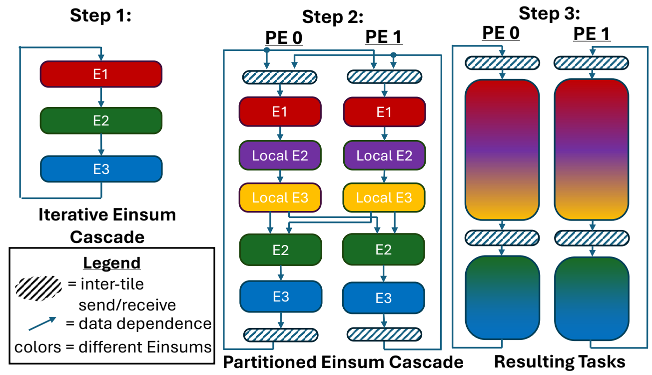

Figure 4: Example task construction process for Quartz's execution model. Box colors denote corresponding Einsums.

The programmer interface for Quartz is an Einsum representation of a problem specification. In this work, we define a systematic, highly automatable process for translating Einsums to task types (kernels of code that can then be translated to a CGRA-like bitstream for execution on a Quartz PE). Defining these steps represents the innovative part of Quartz's programming model; we leave it to future work to encode them as compiler passes. Partitioning (exact techniques described later in Sec. 5) and the instantiation of resulting tasks of each type are already automated.

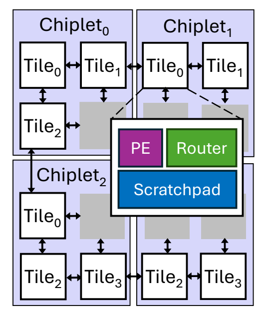

Figure 3: Overview of Quartz's architecture.

Fig. 4 illustrates Quartz's task generation process for an arbitrary iterative sparse matrix computation. In this section, we explain these steps and illustrate them for a simple example: a single sparse matrix-vector multiplication. At the end of the section, we show how this generalizes to more complex and meaningful applications like the ones evaluated in this work. Sec. 4 further details the hardware architecture used to execute these tasks.

3.1 Step 1: Einsum Cascades

We begin by converting a problem specification into a cascade of one or more Einsums, which abstract away implementation choices about execution order and data representation [51]. We represent iteration explicitly in the cascade using iterative ranks [57].

> **Example: SpMV** Sparse matrix-vector multiplication (Fig. 5) can be written as the following Einsum:
>
> $$Z_m^M = A_{m,n}^{M,N} \cdot B_n^N$$

3.2 Step 2: Partitioned Einsum Cascade

As described in Sec. 2, each Einsum prescribes multiple operations, consisting of map operations on the values at the projections of a point in the iteration space to a point in each of the operand tensors (e.g., multiplications in a traditional matrix-vector multiplication) and reduce operations (e.g., additions). To maximize performance on Quartz, not all operations associated with the same Einsum will take place on the same tile or at the same time. *Thus, operations must be grouped into tasks that execute across space and time. One Einsum may give rise to many tasks, and one task may contain operations from several Einsums.*

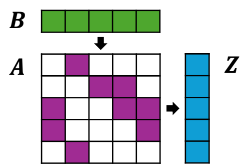

Figure 5: SpMV. Colored boxes represent nonzero values.

To determine how to best group operations into per-tile tasks, we must first account for the fact that data is *partitioned* among distributed memories. Representing partitioning explicitly in our cascade is important because (1) distributing computation gives rise to *new computations*, similar to how parallelizing a program across threads introduces new reductions and synchronization, and (2) partitioning implies *boundaries* between Einsums across which operations cannot be placed in the same task (as tasks are local to a tile).

We design Quartz's execution model to be flexible enough to represent all possible partitionings. This is done by Azul [30] for tensors in its specific domain but is unlike conventional graph programming models. For example, vertex-based models assume all outgoing edges of a vertex are co-located and thus group all operations from exploring a particular vertex into one task. Our model allows any nonzero to be assigned to any tile in our spatial architecture, and tensors can be independently partitioned or co-partitioned. Although individual task instances will process data in accordance with *specific* partitioning implementations, a flexible abstract model of partitioning is sufficient to define task *types*.

To represent partitioning a tensor in our Einsums, we add a rank that indicates the partition at which each nonzero element is stored. Since partitions represent different physical locations in hardware, some duplication and distribution of data elements becomes necessary. This gives rise to three types of partitioned tensors: (1) *strictly partitioned* tensors, where each nonzero element is stored in only one partition, (2) *duplicate partitioned* tensors, where each nonzero may be present (with the same value) in multiple partitions, and (3) *shard partitioned* tensors, where each nonzero may be present in many partitions and whose values will be reduced (i.e., partial sums in the case of an addition operator). We develop new notation, detailed below, that represents inter-partition data transfer using Einsums and we insert cascade steps to support local accumulations and transfers (additional boxes in Fig. 4 Step 2).

Example: SpMV As shown in the upper panel of Fig. 6, we *strictly partition* the input and output tensors to the computation, i.e., A, B, and Z. In order to make all operands for each map and reduce operation local, we must create a *duplicate partitioned* tensor LB to store local copies of elements of B needed for combination with elements of A. Since results are computed in different partitions, we must also add an intermediate *shard partitioned* tensor LZ to store local results that are later reduced across partitions to form Z. With the addition of a partitioning rank, the core operation of SpMV becomes:

$$LZ_{p,m}^{P,M} = A_{p,m,n}^{P,M,N} \cdot LB_{p,n}^{P,N}$$

However, to make this a complete computation, we need to compute LB from B (i.e., send each vector element B_n to all partitions whose owned elements of A include one or more with N-index n) and Z from LZ (i.e., send each vector element LZ_m to the partition storing Z_m and reduce it with other incoming values) (lower panel of Fig. 6). We develop the notion of a *distribution tensor* to represent these data transfers. $T1_{p,n} = 1$ denotes that partition p needs the value of element B_n to combine with locally stored elements of A, and $T2_{p,m} = 1$ if element Z_m is assigned to partition p. Note that these communication tensors can be represented efficiently in hardware: $T1$ is stored in a "coordinate/payload" representation that reduces to just a list of coordinates, and ownership information ($T2$) uses bitmaps. T-vectors and partitionings are computed offline. The resulting Einsum cascade is:

$$LB_{p,n}^{P,N} = T1_{p,n}^{P,N} \cdot B_{q,n}^{P,N} \quad (1)$$

$$LZ_{p,m}^{P,M} = A_{p,m,n}^{P,M,N} \cdot LB_{p,n}^{P,N} \quad (2)$$

$$Z_{p,m}^{P,M} = T2_{p,m}^{P,M} \cdot LZ_{q,m}^{P,M} \quad (3)$$

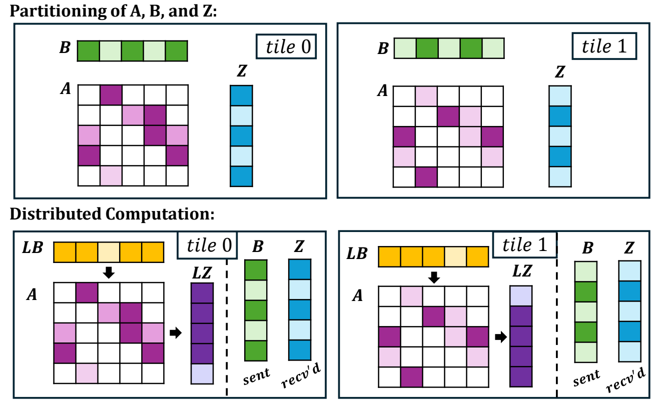

Figure 6: Partitioning of SpMV. Dark colors represent nonzero values stored locally, light colors represent nonzero values not present locally, and white indicates zeroes.

3.3 Step 3: Choice of Dataflow and Loop Fusion

After the full distributed Einsum cascade is constructed, we lower each Einsum that represents tile-local computation (i.e., that is not a send/receive Einsum) into a loopnest, as is done in TeAAL [51]. Each task is local to a tile and to an iteration, so to create a task from a loopnest, we must insert a fixed value for the partitioning rank ($P = p$) and the iteration rank ($I = i$). This eliminates loops involving those ranks. Since Quartz targets computations that operate on sparse vectors and matrices (nominally rank-2 tensors), at most two loops remain. Dataflow (loop order) and data representations for the remaining loops are chosen to maximize reuse (see example below). We design Quartz to have simple tasks that contain at most a single loop, because it simplifies our hardware design (see Sec. 4.2), so we split the execution of an outer loop across multiple tasks.

Example: SpMV Einsum 2 in the above cascade performs tile-local computation. For a particular partition p, we can write it as the following loopnest:

```
for n in N:
  for m in M:
    LZ_p[m] += A_p[m, n] * LB_p[n]
```

where `LZ_p`, `A_p`, and `LB_p` are local data structures with the subset of values at tile p (each tile will have different nonzeros from these structures, depending on how data is partitioned).

This computation can proceed in one of two dataflows: $m \rightarrow n$ or $n \rightarrow m$. Since column values (`LB_p[n]` values) arrive from the network at varied times and in a unknown order, we choose an $n \rightarrow m$ dataflow and split the iterations of the n loop into separate tasks. Thus, for each arriving `LB_p[n]`, we launch a task that scales all local values in that column of the matrix (loops over the m rank and reuses `LB_p[n]`) and updates their corresponding elements of the local output vector. A single task thus computes:

```
for m in M:
  LZ_p[m] += A_p[m, n] * LB_p[n]
```

Finally, as an optimization, once each non-transfer Einsum has been written as task loopnests, we fuse the loops of data-dependent tasks on the same tile to eliminate the need for additional intermediate storage. Tasks that exceed the size of the reconfigurable fabric are split into multiple smaller chunks. We are left with a set of *task types* (Fig. 4 Step 3), each triggered by the arrival of data or the completion of an earlier task at the same tile. Many instances of each task type are created on each tile, and additional optimizations can be applied in hardware to execute each one. The code described by each task is translated to a bitstream to configure Quartz's reconfigurable tiles, as is done in CGRAs.

Example: Task Types for SpMV

(1) **SendColumnValue** sends a column index n and corresponding B-value from the owning tile to all tiles containing elements of A in column n (Einsum 1).
(2) **ScaleColumn** scales all locally stored elements in a column of A by the vector element received, reducing each into their local row sum (Einsum 2).
(3) **SendPartialSum** sends an index m and its local partial sum to the tile containing the final element for that row (included in Einsum 3).
(4) **UpdateRowSum** reduces an incoming partial sum into a final row sum (included in Einsum 3).

3.4 Generalizing to More Complex Applications

We apply these steps to the more complex iterative sparse matrix applications we target. In practice, applications have between four and sixteen task types. A complete collection of the Einsum cascades and corresponding task types for all applications evaluated in this paper appears in [34]. Below we show the task types for BFS. Although it is more complex than SpMV, grouping operations from different Einsums allows us to still define just four types:

Example: Task Types for BFS

(1) **SendPriorFrontierElement** sends an index and distance value for a node from the prior frontier to all tiles containing its neighbors.
(2) **TraverseEdges** traverses the locally stored neighbors of a particular node k from the prior frontier, filters visited nodes based on local information, and updates a local copy of the surviving nodes' distances.
(3) **SendFrontierCandidate** sends the index of a local candidate for the next frontier and its pending distance to the tile owning the global copy of its distance.
(4) **UpdateFrontier** filters local node candidates for the next frontier based on (now locally available) global information, and updates their global distances.

Quartz's programming model offers more flexibility than that of Dalorex, which shards the original compressed sparse data structures of operands and splits code into tasks at the pointer indirections incurred during access. Such an approach is insufficient to support arbitrary data distributions and data representations.

4 Quartz Hardware Architecture

Quartz is a multi-chiplet spatial architecture with a large array of tiles connected by a 2D torus network. As shown in Fig. 7, each tile contains a processing element (PE), scratchpad SRAM, and router.

4.1 Task-Level Dataflow

Quartz tiles are designed to efficiently execute short dataflow tasks (tens of operations on average). Quartz adopts task-level dataflow execution primarily driven by message-triggered tasks using active messages [28] (some tasks are instead triggered by the completion of a local task, as described below). Message-driven execution is widely used in prior work [28, 56, 61]. Rather than executing a sequence of instructions stored in memory, as in Dalorex [59] and Azul [30], each task executes by configuring and using a fabric of functional units. Each task is triggered by a *single* input (a message or the output of a prior local task) and does not need to wait for other operands to arrive before executing. Any other needed operands can be retrived from the local scratchpad. As a result, Quartz avoids hardware overheads of conventional instruction-level dataflow, where computations wait for multiple inputs [9, 27].

All Quartz tasks have the following structure: (1) receive input data from the network and/or read local data from the scratchpad, (2) compute on that data, and (3) write data back to local storage and/or send data over the network to other tiles. Computing on data can also involve conditionally loading additional operands from

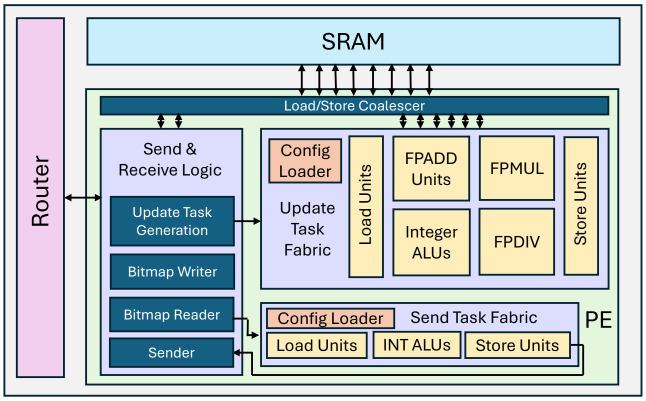

Figure 7: Architecture of one Quartz tile.

the scratchpad. For example, the **TraverseEdges** task from BFS (described in Sec. 3.4) is triggered by the arrival of a source node index from the network. It loads all locally stored outgoing edges of the source node to find candidate nodes for the next frontier, loads the local copy of the current distance to each of those candidate nodes, and uses the distance value of the source node (stored in the message argument) to update the distance to all candidate nodes for which this source node provides a shorter path from the origin.

In order to prevent application-level deadlock across tiles, we separate the execution of tasks triggered by arriving messages (**Update** tasks) from those that send data to other tiles (**Send** tasks) on separate reconfigurable fabrics. To bound the storage used by pending **Send** tasks, we use bitmaps (implemented as hierarchical bitmaps, so they can be scanned efficiently in hardware [43]).

4.2 Sparsity Support

Quartz uses three techniques to decompose the sparse matrix and vector computations found in its applications into a small set of sparse primitives and make their execution efficient.

(1) First, for computations with two ranks (e.g., matrix-vector multiplication), Quartz chooses a dataflow and then splits the execution of the outer loop across separate tasks. Each task executes one outer loop iteration, thus using only a single data value from one of the operands. Coordinate intersections become degenerate, and the task reduces to simply scaling a sparse vector. In Quartz's applications and execution model, only one of the two possible dataflows allows for reuse, so Quartz chooses that one. For example, in matrix-vector multiplication, each Quartz task uses a single data value from the vector and iterates over a column of values in the matrix. In **TraverseEdges**, a single source node is received and locally stored outgoing edges are traversed.
(2) Quartz leverages static sparsity to store a priori the tiles to which each vector coordinate/value pair must be sent to initiate such tasks. As described in Sec. 2.1, all matrix operands in our workloads have static sparsity. In BFS, for example, this allows each tile to store the indices of all other tiles containing outgoing edges of the nodes it owns.
(3) In other operations, Quartz leverages problem structure and partitioning to guarantee that the nonempty coordinates of one operand are a subset of the nonempty coordinates of the other. For example, **TraverseEdges** creates a sparse vector of candidate frontier nodes and must update the local copies of their distances. Step 2 of Quartz's task generation process created and partitioned a local distance vector that is guaranteed to contain all local candidate coordinates. Sometimes, as an optimization, we can guarantee that the nonempty coordinates of both operands are identical, yielding a trivial intersection. Quartz achieves this by identically partitioning operands of an elementwise operation if they are dense or have the same sparsity pattern, such as *x_next* and *x_bar* in PDHG (Fig. 2(c)).

Using these techniques, Quartz PEs efficiently support the sparse operations of a wide range of scientific and graph applications without the need for intersection units and mergers [39, 40, 62, 89].

4.3 Reconfigurable Fabrics

To execute tasks with the structure described above, Quartz PEs contain reconfigurable fabrics of "fabric units" connected by configurable routes. Each fabric unit is either a load/store unit or an arithmetic unit and contains functional circuitry and a small number of registers. To facilitate streaming data from the scratchpad, which provides a mechanism to execute loops, load/store fabric units are simple affine pattern generators, like those in Symphony [62] but with a single loop descriptor. Each can issue strided SRAM accesses using a decoupled address generator and response handler at a rate of one load/store per cycle. SRAMs are highly banked (as in the Cerebras WSE [20]). The SRAMs also use wide word widths (128 bits in our implementation) to enable Quartz to store additional data, like reduction destinations, alongside tensor values. and allow wide accesses. The scratchpad bus width is 128-bit to support this while compute unit widths are standard.

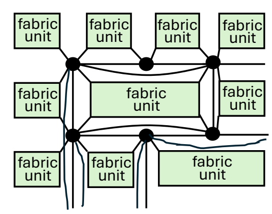

Figure 8: Example fabric units connectivity. Dots are connection nodes.

Fabric units are connected by simple *connection nodes*, which are configurable groups of muxes. We choose this because the widely differing physical areas of Quartz fabric units' functional units preclude placing them in a regular grid with a simple mesh network. Fig. 8 shows fabric units (green boxes) of varying physical areas connected by connection nodes (black dots). Each connection node connects to its adjacent fabric units and those two hops away to increase flexibility in single-hop inter-fabric unit routing. Connection nodes interface with fabric unit inputs and outputs, which are single-element FIFOs (registers plus valid/ready signals). Such ports allow for functional units of different latencies, backpressure from stalls due to memory port contention, and loop execution.

Only a few forms of control flow are necessary to support Quartz's simple tasks: (1) loops dictated by streaming arrays of data from memory and operating on them, and (2) conditional branches within or outside those loops. Quartz PEs support (1) via the pattern generation of load/store units and (2) via steering, like other dataflow architectures [17, 27, 74], where ALU comparison outputs toggle valid signals that route data.

Since Quartz PEs dynamically reconfigure as tasks arrive, reconfiguration must be fast. We perform reconfiguration as in Fifer [54], giving each fabric unit double-buffered configuration cells to preload the next task's configuration during execution of a prior task.

4.4 Inter-Tile Synchronization and Speculation

Quartz uses barrier synchronization between iterations for algorithms where data from one iteration must be written before it is read in later iterations, like CG and PDHG. Other algorithms, like BFS, can benefit from allowing later work to execute speculatively. Allowing this increases parallelism at the cost of work efficiency (from misspeculation). In such cases, Quartz allows interleaving tasks from two iterations at a time and prioritizes earlier work.

Regardless of interleaving, some global synchronization is necessary in non-all-active algorithms to indicate iteration completion, since work per iteration is data-dependent. For this, a hierarchically min-reduced signal is sent from each PE to a designated "control tile" to track the minimum present iteration with negligible communication cost, similar to barrier networks in supercomputers [3].

4.5 Scalability

Quartz scales to many tiles (16,384 in our configuration) across multiple chiplets, enabling significant parallelism. Tiles are connected by a 2D torus network made up of inter- and intra-chiplet links. To reduce network congestion, we build *multicast trees* and *reduction trees*, similar to prior work [14, 21, 35, 60]. At compilation/partitioning time, we define hierarchical tree-like paths for distributing data (multicast trees) and sending partial output values (reduction trees). Execution of tree-based communication is implemented via extra task types to avoid additional hardware costs. Intermediate nodes in reduction trees filter or reduce values, thus reducing network traffic. Local bubble routing along each torus ring prevents network-level deadlock [18].

5 Quartz Partitioning Techniques

5.1 Objectives

In a distributed-memory architecture like Quartz, the method of partitioning data across memories significantly impacts performance because it directly determines network traffic and load balance. We partition at a *single-element* granularity, allowing any nonzero of each operand to be stored at any tile. To achieve high performance, our data placement strategy must meet two objectives: (1) **load balance work** among tiles, and (2) **minimize inter-tile communication**. Prior work either addresses only one objective or addresses both only for all-active workloads with entirely static sparsity. Quartz's heuristics optimize for *both* objectives simultaneously, even for workloads with *dynamic, time-varying* behavior.

Minimizing inter-tile communication means trying to place data elements that are involved in the same operation on the same partition and, as a result, *increasing reuse within a PE* for values involved in multiple operations. Consider executing partitioned BFS: Fig. 9 shows its data structures (analogous to those for SpMV in Sec. 3) and the two types of communication that arise. Matrix A represents the adjacency matrix (rows are source nodes and

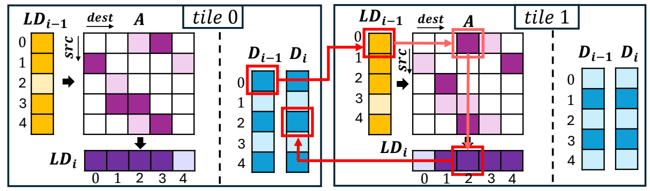

Figure 9: Data dependences for BFS.

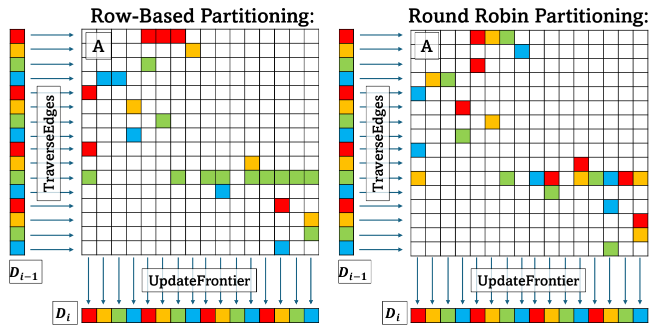

Figure 10: Visualization of single-objective prior partitioning strategies for an example graph's adjacency matrix and distance vector. Colors denote tile assignment. The vector is shown twice, along side and bottom (like Fig. 9), so locality along both rows and columns can be visualized.

columns are destination nodes), vector D holds distances, and LD is a local copy of the distances that arises from partitioning. As in SpMV, minimizing inter-tile communication means co-locating elements from the same *row* m of matrix A with the corresponding element m of vector D, while also co-locating elements in each *column* n with element n of D. In our example, tile 0 owns the distance value for node 0 and tile 1 stores an outgoing edge of node 0, so tile 0 needs to send the current distance to node 0 to tile 1 at the beginning of each iteration. To prevent this communication from going over the network, we ideally would place all elements in row 0 of A on the same tile as element D_0. Once tile 1 traverses the edge from node 0 to node 2, it needs to update node 2's distance by sending a message to tile 0. Ideally, we would co-locate all elements in column 2 of A with D_2.

5.2 Prior Approaches

5.2.1 Simple Communication Minimization. Some prior systems, especially those accelerating graph algorithms, focus only on minimizing communication by applying coordinate-space [75] *tiling* techniques [39, 55] to the matrix A. One common technique is to assign all nonzeros in each *row* of A to the same partition as the corresponding element of D (shown in Fig. 10 (left)). Unfortunately, the number of nonzeros per row can be highly uneven (e.g., in power-law graphs), causing significant load imbalance. and significant communication can still arise between elements of a *column*.

5.2.2 Simple Load Balancing. Alternatively, other work [59] focuses only on load balance by distributing nonzero data values from A and D equally among tiles in a round-robin fashion (shown in Fig. 10 (right)). However, with this approach, elements in the same

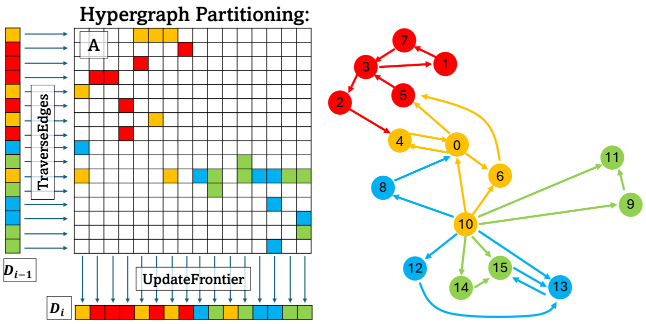

Figure 11: Visualization of hypergraph partitioning for the same example as Fig. 10. Colors denote tile assignment.

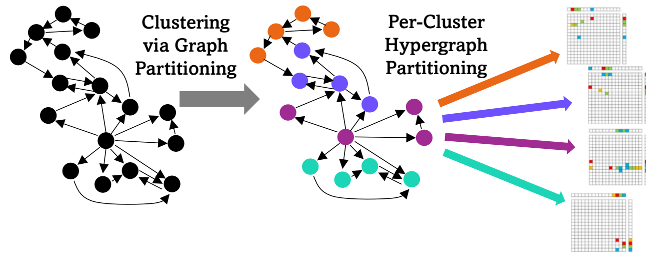

Figure 12: Schematic of Quartz's two-phase partitioning.

row or column are almost always scattered across tiles, causing high network traffic. Other systems use *position-based* tiling: tiles of nonuniform size (in coordinate space) that group consecutive nonzeros and equalize the number of nonzeros per tile. For example, prior systems often flatten a tensor and then assign blocks of contiguous nonzeros, even if they span multiple rows [15, 50], or construct 2D tiles of nonuniform coordinate dimensions (SparseP [33]). Unfortunately, as noted by Azul [30], position-based tiling does not minimize communication unless matrices are spatially correlated, which is not always the case.

5.2.3 Addressing Both for All-Active Algorithms. Prior work has addressed both objectives simultaneously only for *all-active* algorithms with entirely static sparsity [30]. For such computations, partitioning strategies that minimize communication and balance load for just *one* iteration will work well for the entire algorithm.

We use the all-active, static-only sparsity partitioning method employed in Azul [30] as one step in our partitioning process, so we describe it in some detail here using our BFS example. This method involves representing the data-dependences between operands as a *hypergraph*. A hypergraph is a generalization of a graph where each edge (*hyperedge*) can connect arbitrarily many nodes instead of just two [16]. We construct a hypergraph H (separate from the original input graph to BFS) that contains a node for each nonzero of A and element of D and add hyperedges connecting elements involved in the same computations. Under this formulation, if connected nodes were placed in separate partitions, inducing a "cut" in the hyperedge, a message would need to be sent over the network between tiles. To minimize communication, we can partition the nodes of this hypergraph into groups of roughly equal size while minimizing the total number of "cuts" in hyperedges (i.e., minimizing the number of messages that must be sent over the network). Mature hypergraph partitioning algorithms exist to solve this problem [19]. Fig. 11 illustrates hypergraph partitioning by showing the effect on both the adjacency matrix and the original input graph (H is not shown). Data elements that share a row or column are more often assigned to the same tile than with round-robin partitioning, and data elements are more evenly balanced across tiles compared to row-partitioning.

5.3 Quartz's Approach

Quartz's strategy involves (1) extending the all-active approach above to account for physical locality in placement that improves performance on large systems and (2) introducing a two-phase approach to capture dynamic behavior in non-all-active algorithms.

5.3.1 Physical Locality in Placement. While the hypergraph partitioning approach above minimizes the number of inter-tile messages sent for all-active computations (which have only static sparsity), it does not ensure that inter-tile messages do not need to travel long physical distances. To do so, we want dependent data elements that are assigned to different partitions to be placed in *nearby* tiles. We achieve this by applying hypergraph partitioning *recursively*—first across chiplets, then across tiles in each chiplet.

5.3.2 Data-Dependent Behavior. Recursive hypergraph partitioning alone is not sufficient to load balance *non-all-active* computations because they contain dynamically sparse vector operands and introduce time-variation in the *effective* sparsity pattern of even statically sparse operands. In BFS, the sparsity pattern of the frontier vector changes across iterations, so only a subset of the elements of A are used during each iteration. If we execute BFS using the tile assignments shown in Fig. 11, we will be activating only one or two PEs during many of the iterations. Even if the execution of iterations is overlapped, data should be assigned to tiles such that all tiles are equally busy *at any point in time.*

As a larger example, consider running BFS on a road graph. Hypergraph partitioning will map local chunks of the graph to partitions (e.g., mapping the roads of each city to a single tile). This hurts load balance, as the BFS traversal will only operate on a few partitions at once (i.e., a few tiles). To achieve a good partitioning, we must balance locality and load balance. In the road graph, this can be achieved by creating clusters that are larger than a single partition but still capture locality (e.g., the roads of an entire state rather than a city) and distributing each of those clusters more widely (e.g., across all tiles). Our partitioning algorithm builds on this insight to address the locality-balance tradeoff.

Our key contribution is that although it is not known a priori which elements will be used at a given point in time, there are useful heuristics leveraging the insight above that can be applied to partition at *compile-time* to load balance even in the face of dynamic *run-time* behavior. Quartz uses the following two-phase approach:

(1) Identify data elements that are likely to be used in the same iteration and place them into clusters.
(2) Independently partition each cluster across all of Quartz's tiles, minimizing intra-cluster communication.

For example, in BFS, element usage timing depends on the starting vertex. To make Quartz useful for dynamically serving graph-search queries, the partitioning must be efficient across *all* possible starting vertices. To solve this, we formulate the clustering problem as a *graph* partitioning problem using the original input graph *as-is* (without constructing any new data structures). Graph partitioning separates nodes into equally sized groups optimizing for a minimal edge cut. We use standard methods [19] to partition the input graph. Each resulting cluster then approximates the behavior of an all-active algorithm, so we apply our recursive hypergraph partitioning technique (from Sec. 5.3.1) to distribute each cluster among all tiles. Fig. 12 illustrates our approach. Note that this technique *increases* NoC traffic over hypergraph alone, but allows for much better load balance, achieving higher performance (Sec. 6).

6 Evaluation

6.1 Methodology

We evaluate the multi-chiplet Quartz configuration in Table 2. We provision 2,048 tiles per chiplet and scale to 16,384 tiles total across 8 chiplets. Each tile has 452 KB of SRAM (a 448 KB data SRAM and a 4 KB task queue/configuration SRAM), providing aggregate capacity of 7.1 GB. Data SRAMs are highly banked (as in Cerebras WSE [48]), with 8 banks each that support 128-bit (or narrower) accesses.

We clock Quartz at 1 GHz, and model 2-cycle SRAM accesses, consistent with published 7nm SRAM data [84]. On-chiplet links are short and ASAP7 global wires support 112ps/mm, so link traversal and links between fabric units each fit comfortably in one cycle. Modern silicon interposer technology can allow for 0.1ns inter-chiplet hops [22], which also fits within one cycle.

6.1.1 Simulator. We use a cycle-accurate simulator to understand performance and resource utilization, where each system component is represented as an object that ticks according to a global clock signal. Our simulator accounts for contention in the network by explicitly modeling routers, messages, and backpressure, and we validate functional correctness against CPU implementations.

6.1.2 Applications. We evaluate Quartz on six applications from graph analytics and scientific computing. We implement push-based breadth-first search (BFS), Moore's algorithm [69] for single-source shortest path (SSSP), and label-propagation [38] for weakly-connected components (WCC). We run two types of linear solvers, conjugate gradients (CG) and Chebyshev iteration (CHB), both with the identity preconditioner. We also run Primal-Dual Hybrid Gradient (PDHG), a popular first-order method for non-smooth convex optimization problems [7]. For graph applications, we follow standard practice in evaluating traversed edges per second (or giga-TEP/s) [1]. We compute *effectual* work (*unique* edges traversed).

6.1.3 Datasets. For our graph algorithms, we evaluate on both real-world and synthetic graphs. Real-world graphs are chosen from SuiteSparse [26] to include a range of sizes and application domains (social networks, roads, webpages, citation patterns, DNA analysis, and computational geometry). For BFS, we use unweighted directed graphs; for SSSP, weighted directed graphs; and for WCC, undirected graphs. We also add a synthetic graph, Kron-21 [47]. For CG and CHB, we select symmetric, positive-definite matrices from SuiteSparse [26] that represent physics and engineering simulations.

Table 2: Evaluated Quartz hardware configuration.

Chiplet Configuration	8 chiplets
Tile Configuration	2048 tiles per chiplet (16384 total), Reconfiguration: 147 B and 33 B for update & send fabrics Update Fabric Units: 8 load units, 8 store units, 1 FPMUL, 6 FPADD, 1 FPDIV, 16 INTALU Send Fabric Units: 3 load units, 3 store units, 2 INTALU
SRAM Scratchpads	(448+4) KB/tile (904 MB/chiplet; 7.1GB total), 2 cycles per access, pipelined, 8 ports per data SRAM, 128b wide
Clock Frequency	1 GHz
Network	2D bidirectional torus, 128-bit links, 1 cycle/hop on both inter-chiplet and intra-chiplet links, 8 TB/s bisection bandwidth

Table 3: Matrices used to evaluate Quartz. Rows/cols specified separately for PDHG because all others are square.

BFS Matrix	nnz	n	CG Matrix	nnz		n
amazon0302	1.23e6	2.62e5	G3_circuit	7.66e6		1.59e6
delaunay_n23	5.03e7	8.39e6	bundle_adj	2.02e7		5.13e5
GAP_road	5.77e7	2.39e7	StocF_1465	2.10e7		1.47e6
Kron21	6.35e7	1.24e6	bone010	4.79e7		9.87e5
soc_LiveJournal1	6.90e7	4.85e6	Hook_1498	5.94e7		1.50e6
cage15	9.92e7	5.15e6	Serena	6.41e7		1.39e6
wikipedia	1.01e8	4.21e6	Flan_1565	1.14e8		1.56e6
SSSP Matrix	**nnz**	**n**	**CHB Matrix**	**nnz**		**n**
appu	1.85e6	1.40e4	G3_circuit	7.66e6		1.59e6
web_Google	8.64e6	9.16e5	bundle_adj	2.02e7		5.13e5
italy_osm	1.40e7	6.69e6	StocF_1465	2.10e7		1.47e6
sx_stackoverflow	3.62e7	2.60e6	bone010	4.79e7		9.87e5
Kron21	6.35e7	1.24e6	Hook_1498	5.94e7		1.50e6
cage15	9.92e7	5.15e6	Serena	6.41e7		1.39e6
wikipedia	1.01e8	4.21e6	Flan_1565	1.14e8		1.56e6
WCC Matrix	**nnz**	**n**	**PDHG Matrix**	**nnz**	**rows**	**cols**
auto	6.63e6	4.49e5	bab3	3.30e6	2.31e4	3.94e5
web_Google	8.64e6	9.16e5	ds_big	4.62e6	1.04e3	1.75e5
hugetrace_00000	1.38e7	4.59e6	rmine25	7.18e6	2.95e6	3.27e5
coPapersDBLP	3.05e7	5.40e5	square41	1.36e7	4.02e4	6.22e4
Kron21_undir	6.35e7	1.24e6	ns1663818	2.04e7	1.72e5	1.25e5
com_LiveJournal	6.94e7	4.00e6	mspp16	2.77e7	5.62e5	2.93e4
mycielskian17	1.00e8	9.83e4	L2CTA3D	3.00e7	2.10e5	1.00e7

For PDHG, we take linear programs from the Mittelman Suite (included in SuiteSparse). Table 3 lists these datasets.

6.1.4 Baselines. We compare quantitatively to the state-of-the-art distributed-SRAM architecture that is programmable enough to support general applications, Dalorex [59], and to an NVIDIA H100 GPU. We model Dalorex by replacing the Quartz specialized PEs with a model of an in-order RISC core and extracting operation latencies from Dalorex's open-source simulator [58]. We generously give the baseline multicast and reduction trees, calling it "Dalorex++", and provision it with the same memory capacity and PE count as Quartz. For comparisons to an H100 GPU, we use the best state-of-the-art libraries openly available for each application: Gunrock [81] for BFS, cuGraph [5] for SSSP and WCC, Ginkgo [6] for CG and CHB, and cuPDLP [49] for PDHG. We also compare qualitatively to other architectures with shared memory hierarchies and near-data processing (DRAM-based) systems.

6.1.5 Partitioning. We use the PaToH v3.3 [19] library to perform graph and hypergraph partitioning.

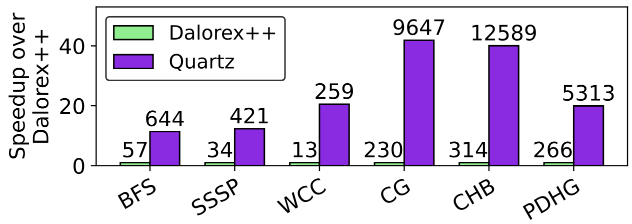

Figure 13: Overall comparison of Quartz to baseline. Bar heights show relative speedup and values on bars show absolute performance. Performance is in GTEP/s for BFS, SSSP, WCC and GFLOP/s for CG, CHB, PDHG. All are gmeans.

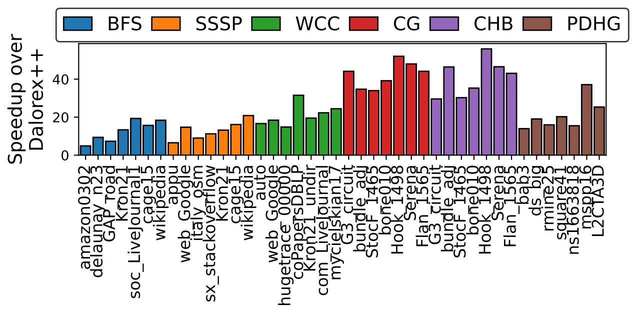

Figure 14: Quartz performance results by dataset.

6.2 Comparison to Baselines

6.2.1 Dalorex++. Fig. 13 shows the performance of Quartz relative to Dalorex++ for each application. Each bar reports gmean performance across inputs. Quartz substantially outperforms Dalorex++ on all applications, achieving a gmean 21.4× speedup. On the best-performing application, Chebyshev iteration, Quartz achieves 38% utilization of peak throughput (16 TMUL/s).

Fig. 14 reports speedups on individual inputs. Matrices are ordered in terms of ascending number of nonzeros. The performance of each input largely depends on how well Quartz's partitioning can exploit the sparsity structure of the matrix. For example, Quartz achieves good performance on `web_Google` for SSSP and WCC because its low diameter and clustered structure allow graph and hypergraph partitioning to extract significant locality.

6.2.2 Other Architectures. Architectures that optimize shared memory hierarchies, while attaining reuse of values in local storage, are ultimately bottlenecked on memory bandwidth. PolyGraph [24] and Alrescha [10] which use reconfigurable logic with a shared memory hierarchy, have 512 GB/s and 288 GB/s of memory bandwidth, respectively. In contrast, Quartz utilizes on average 280 TB/s of memory bandwidth. Even conservatively accounting for reuse in these workloads (e.g., through tiling and preprocessing) and optimistically assuming saturated memory bandwidth, Quartz would be about two orders of magnitude faster. Similarly, HRL [31], a state-of-the-art NDP system that uses reconfigurable logic within stacked DRAM, contains 8 stacks with an aggregate bandwidth of 1 TB/s. Even if this bandwidth is saturated, Quartz significantly outperforms HRL's peak throughput because it distributes memory capacity among more PEs and effectively utilizes its higher aggregate bandwidth. Additionally, Quartz achieves gmean 93× speedup over an H100 GPU. Table 4 shows a comparison of Quartz and GPU performance for each application.

Table 4: Comparison of Quartz and H100 GPU.

	Matrix	GPU Execution Time (ms)	Quartz Speedup	Energy Efficiency Comparison (Quartz Perf/J vs. GPU Perf/J)	Area Efficiency Comparison (Quartz Perf/mm^2 vs. GPU Perf/mm^2)
BFS	wikipedia	6.72	117.97	1.95e3	26.67
	soc_LiveJournal1	3.91	108.99	2.77e3	24.64
	amazon0302	3.48	368.32	2.97e4	83.28
	GAP_road	325.39	624.79	8.90e4	141.27
	delaunay_n23	61.18	384.98	3.42e4	87.05
	cage15	5.24	95.31	2.77e3	21.55
	Kron21	17.51	478.74	3.88e4	108.25
SSSP	italy_osm	2.78e3	5.91e3	8.81e6	1.34e3
	cage15	40.59	507.89	7.60e4	114.84
	web_Google	12.44	1.77e3	5.56e5	399.30
	sx_stackoverflow	13.58	524.64	5.72e4	118.63
	Kron21	10.00	271.06	1.41e4	61.29
	appu	13.27	575.23	5.68e4	130.07
	wikipedia	72.53	206.25	8.08e3	46.63
WCC	auto	2.82	40.36	235.49	9.13
	Kron21_undir	5.03	31.05	194.95	7.02
	mycielskian17	3.86	39.39	578.32	8.91
	web_Google	5.32	240.92	8.20e3	54.48
	coPapersDBLP	3.36	59.04	557.45	13.35
	hugetrace_00000	6.05	11.31	23.51	2.56
	com_LiveJournal	4.95	26.93	154.60	6.09
CG	Flan_1565	1.18	57.79	1.21e3	13.07
	Serena	0.80	67.69	1.67e3	15.31
	Hook_1498	0.77	73.39	1.91e3	16.60
	bone010	0.77	54.83	1.13e3	12.40
	StocF_1465	0.44	53.70	1.23e3	12.14
	bundle_adj	0.33	60.01	1.44e3	13.57
	G3_circuit	0.27	65.34	848.78	14.77
CHB	Flan_1565	1.03	55.79	1.08e3	12.61
	Serena	0.71	70.83	1.70e3	16.01
	Hook_1498	0.69	82.58	2.18e3	18.67
	bone010	0.68	50.97	954.72	11.53
	StocF_1465	0.40	63.75	1.35e3	14.41
	bundle_adj	0.31	91.61	2.62e3	20.71
	G3_circuit	0.25	118.47	3.39e3	26.79
PDHG	L2CTA3D	0.71	13.67	128.90	3.09
	mspp16	0.46	27.32	411.44	6.18
	ns1663818	0.23	13.64	115.47	3.08
	rmine25	0.20	22.66	255.87	5.12
	ds_big	0.09	26.22	228.46	5.93
	square41	0.16	21.75	203.89	4.92
	bab3	0.10	19.19	132.93	4.34
	gmean	2.36	93.02	2.40e3	21.03

6.3 Performance Breakdown

6.3.1 Ablation. Fig. 1 (Sec. 1) shows the contributions to Quartz's performance that result from its reconfigurable PEs and partitioning strategies, respectively. Distributed-SRAM with many thousands of PEs offers high aggregate bandwidth and hardware parallelism that can translate to high performance. Quartz achieves an average of 4.4 TMUL/s on floating-point applications (28% of peak) and 6.6 TOP/s on integer applications.

Relative to Dalorex's in-order cores, Quartz PEs eliminate many of the cycles spent on instruction fetch and address calculation. In addition to those control differences, however, Quartz PEs also

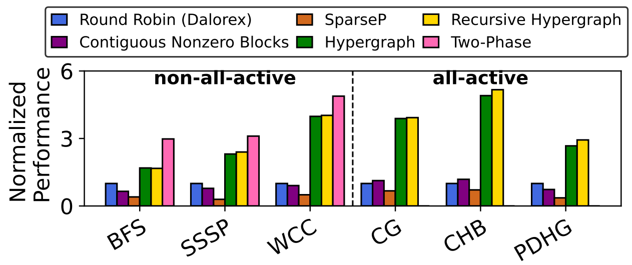

Figure 15: Comparison of partitioning strategies using Quartz hardware. The two-phase approach only makes sense for non-all-active algorithms.

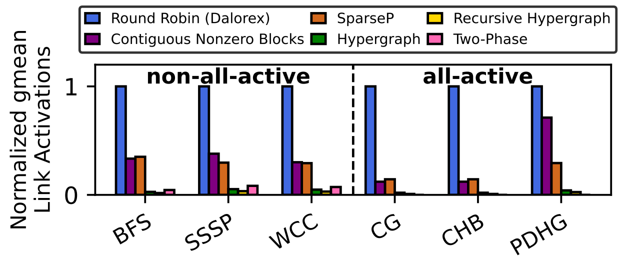

Figure 16: Effect of partitioning strategies on NoC traffic, using Quartz hardware. The two-phase approach only makes sense for non-all-active algorithms.

provide more functional units that can be configured as a custom pipeline. This can spatially execute an entire loop iteration for most Quartz tasks, allowing a throughput of 1 element/cycle. In contrast, producing each output element in Dalorex requires sending multiple data-dependent instructions down a generic pipeline. This hardware change provides 5.7× speedup. Quartz PEs add minimal reconfiguration time overhead, and their performance comes at a modest area cost since SRAM dominates chip area (see Sec. 6.5).

Once this compute bottleneck is cleared, network traffic limits the performance of "Quartz HW & Dalorex Partitioning". Quartz partitioning techniques for both all-active and non-all-active algorithms boost performance by an additional 3.7× over Dalorex++.

6.3.2 Partitioning Strategies. Fig. 15 shows the effect of different partitioning strategies on performance, using Quartz PE hardware. Each bar shows gmean performance across inputs. Fig. 16 reports the reduction in NoC traffic across techniques, shown in number of link traversals relative to the baseline. We run round-robin (RR), position-based tilings (Contiguous Nonzero Blocks (CNB) and 2D variable-sized SparseP [33]), plain hypergraph, and Quartz's recursive hypergraph partitioning on all benchmarks, plus Quartz's two-step partitioning on non-all-active algorithms. Plain hypergraph was used in prior specialized systems [30]; for some domains, including graphs, we are the first to apply it to an accelerator.

CNB partitioning reduces link activations over RR by grouping nonzeros along rows, but sees worse load balance due to its distribution of vector elements. Although each tile receives an equal number of vector elements, elements confer different numbers of reductions/multicasts (depending on matrix sparsity pattern). Clumping or spatial nonuniformities in the matrix make distributing contiguous blocks of vector elements (as in CNB) induce load imbalance. SparseP underperforms CNB. In some cases this is due to increased communication because neither row nor column communication is fully minimized; other times is due to load imbalance from vector distribution, as elements are distributed to the first tile containing a dependent matrix nonzero. Since it makes fewer incorrect assumptions about sparsity patterns, hypergraph partitioning outperforms all three techniques.

For all-active algorithms, recursively applying hypergraph partitioning reduces average hops per message by 21% over regular hypergraph, boosting overall gmean throughput by 5%.

For non-all-active algorithms, recursively applying hypergraph partitioning also reduces network traffic, leading to performance benefits on many inputs. However, recursive partitioning can sometimes introduce greater load imbalance, since each partitioning call allows some amount of load imbalance and recursive calls compound these tolerances. These two factors compete to determine performance.

Nonetheless, for non-all-active algorithms, Quartz's two-phase approach outperforms all other techniques, by 3.6× over the baseline round-robin strategy, and by 1.4× over recursive hypergraph, due to its ability to still exploit locality while load balancing better. It *increases* NoC traffic, by about 2.5× over recursive hypergraph, but achieves better load balance and thus improves performance.

6.4 Partitioning Costs

Quartz's partitioning techniques take gmean 18.5 minutes across all applications and datasets. Although this cost is high, partitioning time is amortized over *both* the many iterations of each algorithm *and* many runs of the algorithm with the same input sparsity pattern. For example, navigation and route planning applications operate on road network graphs that change rarely but serve many user queries [78] (i.e., millions of runs with a single partitioning, which could each take thousands of iterations). Similarly, physics simulations solve a linear system at each timestep, reusing the same matrix (which describes the physical system) across iterations, across timesteps, and across runs with different initial conditions. Simulating a system with 1us-timesteps for 10 simulated seconds with 1000 iterations per solve already requires 10 billion iterations [30].

Prior Graph500 submissions and linear solver accelerators also pay extremely high preprocessing costs. The top Graph500 submission from November 2024, using the Fugaku supercomputer, spends 1000× longer in preprocessing than the runtime of a single BFS (678.9 seconds preprocessing for a scale-43 graph, while BFS for it takes 0.69 seconds with the reported throughput) [8]. Similarly, prior work on optimization problem accelerators has gone as far as synthesizing entire FPGA designs based on a single input sparsity pattern, which takes far longer than Quartz's partitioning [80]. The hypergraph partitioning used in Azul [30] is used for similarly sized problems as Quartz. But by decomposing the problem into smaller pieces, Quartz's recurisve and two-step partitionings actually take *less* time than plain hypergraph (which takes gmean 20.9 minutes).

Furthermore, prior work has shown that it is often possible to develop cheaper heuristics that achieve a large fraction of the

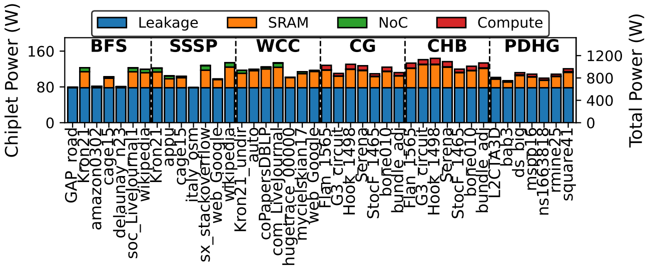

Figure 17: Power breakdown for Quartz.

performance gains of partitioning [12]. We believe it is valuable to identify the best high-quality partitioning methods available and leave it to future work to develop faster heuristics for our methods.

6.5 Area, Power, and Energy

6.5.1 Methodology. We estimate Quartz's area and power in a 7nm process node [23]. SRAM area is calculated using FN-CACTI [67] in ASAP7 [23]. Following prior work [30, 59], we estimate SRAM per-access energy and leakage power by scaling published data on fabricated 7nm SRAM [84]. This gives 11.6 pJ per 128-bit read and 18.2 pJ per 128-bit write to a 56KB SRAM bank. PE areas and energies are estimated from synthesizing functional units and individual components in ASAP7 [23] using a combination of Yosys [83] and Synopsys Design Compiler. Network area and energy are estimated using DSENT [73] at a 22nm process node that we scale down to 7nm. For I/O, we conservatively add a single 512 GB/s HBM2e PHY sized at $15mm^2$ [65]. Importantly, note that this is purely used for initial configuration and loading data, as Quartz *does not* fetch data from off-chip memory during program execution.

6.5.2 Results. Table 5 shows the area breakdown of a Quartz chiplet. Each chiplet is 68% SRAM by area and 32% compute. Quartz uses this significant on-chip memory to support meaningful problem sizes, and can be scaled to larger problem sizes with more chiplets. Quartz achieves gmean 21× higher area efficiency (FLOPS/mm^2) than an H100 GPU. Table 4 shows the improvement for each application.

Table 5: Area estimates for one Quartz chiplet.

Unit	Area
PEs	$2048 \times 0.0668mm^2 = 136.88mm^2$
Routers	$2048 \times 0.0016mm^2 = 3.28mm^2$
SRAMs	$2048 \times 0.3721mm^2 = 294.13mm^2$
I/O	$15\ mm^2$
Total	$449mm^2$

Fig. 17 shows Quartz's power consumption breakdown. Because Quartz partitions data effectively, NoC energy is a much smaller contributor to power consumption than in Dalorex [59]. Rather, SRAM heavily dominates power consumption, as in prior all-SRAM designs with specialized partitioning methods [30]. Aside from SRAM, note that different algorithms display different power-usage characteristics. For floating point computations such as CG, compute consumes a significant fraction of the remaining power. For integer computations like BFS, no complex arithmetic is executed and network traffic instead dominates. Quartz achieves gmean 2223× higher energy efficiency (FLOPS/J) than an H100 GPU. Table 4 shows the improvement for each application.

7 Additional Related Work

All the accelerators with shared memory hierarchies from Table 1 implement optimizations to increase reuse on sparse computations. Systems like Prodigy [76] and DeSC [36] use decoupling or prefetching. Others [37, 64] hide latency with custom pipelines. ExTensor [39] and Tensaurus [72] target large sparse kernels with high internal reuse (e.g., in matrix multiplication). However, all are memory-bottlenecked for workloads with no/limited reuse.

Quartz faces different design tradeoffs than reconfigurable systems with shared memory hierarchies. Alrescha [10] uses simple coordinate-space tiling; partitions are loaded from memory and processed over time, so cache locality is important and load balance among blocks is not. In contrast, Quartz's high hardware parallelism makes partition load balance important, and its fine-grained partitioning makes original matrix cache line locality irrelevant.

ARENA [77] explores task-based execution of HPC applications on reconfigurable nodes, but is specialized for much smaller systems (e.g., 16 nodes, not 16K). Quartz allows for greater scalability via point-to-point messages, and having many short tasks eliminates the need for ARENA's complex multi-threading techniques.

While HRL [31], NDA [29], and Quartz all use PEs based on CGRAs, they contain substantial architectural differences arising from different memory and task models. HRL and NDA are designed around sending DRAM requests (sometimes remote to non-local stacks) and thus contain extra buffering to hide long latencies. In contrast, Quartz guarantees all accesses are local so PE design can be simplified. Also, HRL and NDA use MapReduce-based compute models with a few long-running phases that may contain complex control flow. In contrast, Quartz PEs execute short, simple dataflow tasks, and thus contain hardware support for fast reconfiguration and omit unnecessary structures for more complex control flow.

In addition to the techniques described in Sec. 5, temporal and spatial graph slicing does not suffice to meet our objectives. For example, PolyGraph [24] slices graphs using bounded depth-first search followed by round-robin partitioning of each slice. But slices are processed over time so load balance is unimportant. GraphR [71] traverses small sections of input graphs in row- or column-major order, which fails to fully exploit locality among data elements.

8 Conclusion

Iterative sparse matrix computations are pervasive across graph analytics and scientific computing. Although prior distributed-memory architectures overcome memory bottlenecks, they suffer from either poor programmability or poor compute performance. Quartz uses short dataflow tasks and reconfigurable compute engines, combined with novel data partitioning techniques, to deliver both high programmability and high performance. Our Quartz implementation achieves gmean 21.4× speedup over a prior state-of-the-art system across six sparse workloads.

Acknowledgments

We thank Hyun Ryong Lee, Fares Elsabbagh, Xingran Du, Aleksandar Krastev, Viansa Schmulbach, Alicia Golden, and our anonymous reviewers for their feedback on this paper. This work was funded in part by a National Science Foundation (NSF) Graduate Research Fellowship Award #2142064, and by NSF grant CCF-2217099.

References

[1] 2024. The Graph 500 List 2024. https://graph500.org/?page_id=1285

[2] Dennis Abts, John Kim, Garrin Kimmell, Matthew Boyd, Kris Kang, Sahil Parmar, Andrew Ling, Andrew Bitar, Ibrahim Ahmed, and Jonathan Ross. 2022. The Groq software-defined scale-out tensor streaming multiprocessor: From chips-to-systems architectural overview. In *2022 IEEE Hot Chips 34 Symposium (HCS)*.

[3] N.R. Adiga, G. Almasi, G.S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A.A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T.M. Cipolla, P. Crumley, K.M. Desai, A. Deutsch, T. Domany, M.B. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M.E. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R.A. Haring, D. Heidel, P. Heidelberger, L.M. Herger, D. Hoenicke, R.D. Jackson, T. Jamal-Eddine, G.V. Kopcsay, E. Krevat, M.P. Kurhekar, A.P. Lanzetta, D. Lieber, L.K. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J.E. Moreira, B.J. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R.B. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R.K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B.D. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R.B. Tremaine, M. Tsao, A.R. Umamaheshwaran, P. Verma, P. Vranas, T.J.C. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C.T. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M.K. Seager, J.S. Vetter, and K. Yates. 2002. An Overview of the BlueGene/L Supercomputer. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*.

[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proc. ISCA-42*.

[5] RAPIDS AI. 2025. cuGraph - RAPIDS Graph Analytics Library. https://github.com/rapidsai/cugraph.

[6] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S Quintana-Ortí. 2022. Ginkgo: A modern linear operator algebra framework for high performance computing. *ACM Transactions on Mathematical Software (TOMS)* (2022).

[7] David Applegate, Mateo Diaz, Oliver Hinder, Haihao Lu, Miles Lubin, Brendan O' Donoghue, and Warren Schudy. 2021. Practical Large-Scale Linear Programming using Primal-Dual Hybrid Gradient. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. 20243–20257.

[8] Junya Arai, Masahiro Nakao, Yuto Inoue, Kanto Teranishi, Koji Ueno, Keiichiro Yamamura, Mitsuhisa Sato, and Katsuki Fujisawa. 2024. Doubling Graph Traversal Efficiency to 198 TeraTEPS on the Supercomputer Fugaku. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41406.2024.00107

[9] Arvind and R.S. Nikhil. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.* 39, 3 (1990), 300–318. https://doi.org/10.1109/12.48862

[10] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator. In *Proc. HPCA-26*.

[11] Brett W. Bader and Tamara G. Kolda. 2007. Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM J. Sci. Comput.*

[12] Vignesh Balaji and Brandon Lucia. 2018. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 203–214. https://doi.org/10.1109/IISWC.2018.8573478

[13] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*.

[14] Amanda Bienz, Luke Olson, and William Gropp. 2019. Node-Aware Improvements to Allreduce. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*.

[15] Guy E. Blelloch. 1990. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA.

[16] Alain Bretto. 2013. Hypergraph theory. *An introduction. Mathematical Engineering. Cham: Springer* (2013).

[17] M. Budiu, P. V. Artigas, and S. C. Goldstein. 2005. Dataflow: A Complement to Superscalar. In *Proc. ISPASS*.

[18] C. Carrion, R. Beivide, J.A. Gregorio, and F. Vallejo. 1997. A flow control mechanism to avoid message deadlock in k-ary n-cube networks. In *Proceedings Fourth International Conference on High-Performance Computing*. 322–329. https://doi.org/10.1109/HIPC.1997.634510

[19] Ümit V Çatalyürek and Cevdet Aykanat. 2011. PaToH (Partitioning Tool for Hypergraphs).

[20] Cerebras Systems Inc. 2021. The second generation wafer scale engine. https://8968533.fs1.hubspotusercontent-na1.net/hubfs/8968533/Whitepapers/Cerebras-CS-2-Whitepaper.pdf.

[21] R.C. Chalmers and K.C. Almeroth. 2003. On the topology of multicast trees. *IEEE/ACM Transactions on Networking* (2003).

[22] Shuangliang Chen, Saptadeep Pal, and Rakesh Kumar. 2024. Waferscale Network Switches. In *Proc. ISCA-51*.

[23] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm FinFET predictive process design kit. *Microelectronics Journal* (2016).

[24] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *Proc. ISCA-48*.

[25] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proc. MICRO-52*.

[26] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* (2011).

[27] Jack B. Dennis and David P. Misunas. 1974. A preliminary architecture for a basic data-flow processor. In *Proc. ISCA-2*.

[28] T.v. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. 1992. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. ISCA-19*.

[29] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *Proc. HPCA-21*.

[30] Axel Feldmann, Courtney Golden, Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2024. Azul: An Accelerator for Sparse Iterative Solvers Leveraging Distributed On-Chip Memory. In *Proc. MICRO-57*.

[31] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proc. HPCA-22*.

[32] Armin Gerami and Bahar Asgari. 2025. GUST: Graph Edge-Coloring Utilization for Accelerating Sparse Matrix Vector Multiplication. In *Proc. ASPLOS-XXX*.

[33] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. 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).

[34] Courtney Golden. 2025. *A Reconfigurable, Distributed-Memory Accelerator for Sparse Applications.* Master's thesis. Massachusetts Institute of Technology, Cambridge, MA.

[35] Richard L. Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, Ami Marelli, Valentin Petrov, Evyatar Romlet, Yong Qin, and Ido Zemah. 2020. Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)TM Streaming-Aggregation Hardware Design and Evaluation. In *High Performance Computing*.

[36] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proc. MICRO-48*.

[37] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. MICRO-49*.

[38] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. 2017. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition* 70 (2017), 25–43. https://doi.org/10.1016/j.patcog.2017.04.018

[39] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proc. MICRO-52*.

[40] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*.

[41] Trey Ideker, Owen Ozier, Benno Schwikowski, and Andrew F. Siegel. 2002. Discovering regulatory and signalling circuits in molecular interaction networks. *Bioinformatics* (2002).

[42] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking. https://arxiv.org/abs/1912.03413

[43] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *Proc. MICRO-52*.

[44] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[45] Kalhan Koul, Maxwell Strange, Jackson Melchert, Alex Carsello, Yuchen Mei, Olivia Hsu, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Zhouhua Xie, Christopher Torng, Joel Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. 2024. Onyx: A 12nm 756 GOPS/W Coarse-Grained Reconfigurable Array for Accelerating Dense and Sparse Applications. In *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 1–2.

[46] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web*.
[47] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. [n. d.]. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.* ([n. d.]).
[48] Sean Lie. 2021. Multi-Million Core, Multi-Wafer AI Cluster. In *2021 IEEE Hot Chips 34 Symposium (HCS)*.
[49] Haihao Lu and Jinwen Yang. 2024. cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia. arXiv:2311.12180 [math.OC] https://arxiv.org/abs/2311.12180
[50] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/2145816.2145832
[51] Nandeeka Nayak, Toluwanimi O. Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. 2023. TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators. In *Proc. MICRO-56*.
[52] Nandeeka Nayak, Xinrui Wu, Toluwanimi O. Odemuyiwa, Michael Pellauer, Joel S. Emer, and Christopher W. Fletcher. 2024. FuseMax: Leveraging Extended Einsums to Optimize Attention Accelerator Design. https://arxiv.org/abs/2406.10491
[53] Quan M. Nguyen and Daniel Sanchez. 2020. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *Proc. MICRO-53*.
[54] Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *Proc. MICRO-54*.
[55] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
[56] Michael D Noakes, Deborah A Wallach, and William J Dally. 1993. The J-Machine multicomputer: An architectural evaluation. *ACM SIGARCH Computer Architecture News* (1993).
[57] Toluwanimi O. Odemuyiwa, Joel S. Emer, and John D. Owens. 2024. The EDGE Language: Extended General Einsums for Graph Algorithms. https://arxiv.org/abs/2404.11591
[58] Marcelo Orenes-Vera, Esin Tureci, Margaret Martonosi, and David Wentzlaff. 2024. Muchisim: A Simulation Framework for Design Exploration of Multi-Chip Manycore Systems. In *Proceedings of 2024 International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://doi.org/10.48550/arXiv.2312.10244
[59] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. 2023. Dalorex: A data-local program execution and architecture for memory-bound applications. In *Proc. HPCA-29*.
[60] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. 2023. Tascade: Hardware support for atomic-free, asynchronous and efficient reduction trees. *arXiv preprint arXiv:2311.15810* (2023).
[61] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered instructions: a control paradigm for spatially-programmed architectures. In *Proc. ISCA-40*.
[62] Michael Pellauer, Jason Clemons, Vignesh Balaji, Neal Crago, Aamer Jaleel, Donghyuk Lee, Mike O'Connor, Angshuman Parashar, Sean Treichler, Po-An Tsai, Stephen W. Keckler, and Joel S. Emer. 2023. Symphony: Orchestrating Sparse and Dense Tensors with Hierarchical Heterogeneous Processing. *ACM Trans. Comput. Syst.* 41, 1–4, Article 4 (Dec. 2023), 30 pages.
[63] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Proc. ISPASS*.
[64] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *Proc. MICRO-53*.
[65] Rambus Inc. 2020. White paper: HBM2E and GDDR6: Memory Solutions for AI.
[66] R. Rangan, N. Vachharajani, M. Vachharajani, and D.I. August. 2004. Decoupled software pipelining with the synchronization array. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* 177–188.
[67] Divya Praneetha Ravipati, Rajesh Kedia, Victor M. Van Santen, Jörg Henkel, Preeti Ranjan Panda, and Hussam Amrouch. 2022. FN-CACTI: Advanced CACTI for FinFET and NC-FinFET Technologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2022).
[68] M.M.G. Ricci and Levi-Civita T. 1901. Méthodes de calcul différentiel absolu et leurs applications. *Math. Ann.*
[69] Alexander Schrijver. 2012. On the history of the shortest path problem. *Documenta Mathematica* 17, 1 (2012), 155–167.
[70] Marco Siracusa, Víctor Soria-Pardos, Francesco Sgherzi, Joshua Randall, Douglas J. Joseph, Miquel Moretó Planas, and Adrià Armejach. 2023. A Tensor Marshaling Unit for Sparse Tensor Algebra on General-Purpose Processors. In *Proc. MICRO-56*.
[71] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *Proc. HPCA-24*.
[72] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[73] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. 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)*.
[74] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. 2003. WaveScalar. In *Proc. MICRO-36*.
[75] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. *Efficient Processing of Deep Neural Networks*. Springer Charm.
[76] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. 2021. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *Proc. HPCA-27*.
[77] Cheng Tan, Chenhao Xie, Tong Geng, Andres Marquez, Antonino Tumeo, Kevin Barker, and Ang Li. 2021. ARENA: Asynchronous Reconfigurable Accelerator Ring to Enable Data-Centric Parallel Computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 12 (2021), 2880–2892. https://doi.org/10.1109/TPDS.2021.3081074
[78] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The more the merrier: efficient multi-source graph traversal. *Proc. VLDB Endow.* (2014).
[79] Ali Tizghadam and A. Leon-Garcia. 2009. A graph theoretical approach to traffic engineering and network control problem. *Teletraffic Congress*.
[80] Maolin Wang, Ian McInerney, Bartolomeo Stellato, Stephen Boyd, and Hayden Kwok-Hay So. 2023. RSQP: Problem-specific Architectural Customization for Accelerated Convex Quadratic Optimization. In *Proc. ISCA-50*.
[81] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (Aug. 2017), 3:1–3:49.
[82] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. 2009. User interactions in social networks and their implications. In *Proceedings of the 4th ACM European Conference on Computer Systems*.
[83] Claire Wolf. [n. d.]. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.
[84] Yoshisato Yokoyama, Miki Tanaka, Koji Tanaka, Masao Morimoto, Makoto Yabuuchi, Yuichiro Ishii, and Shinji Tanaka. 2020. 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 *2020 IEEE Symposium on VLSI Circuits*.
[85] Daochen Zha, Louis Feng, Liang Luo, Bhargav Bhushanam, Zirui Liu, Yusuo Hu, Jade Nie, Yuzhen Huang, Yuandong Tian, Arun Kejariwal, and Xia Hu. 2023. Pre-train and Search: Efficient Embedding Table Sharding with Pre-trained Neural Cost Models. In *Proceedings of Machine Learning and Systems*.
[86] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: throughput-oriented programmable processing in memory. *International Symposium on High-Performance Parallel and Distributed Computing*.
[87] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *Proc. HPCA-24*.
[88] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. 2024. Sparsepipe: Sparse Inter-operator Dataflow Architecture with Cross-Iteration Reuse. In *Proc. MICRO-57*.
[89] Kai Zhong, Zhenhua Zhu, Guohao Dai, Hongyi Wang, Xinhao Yang, Haoyu Zhang, Jin Si, Qiuli Mao, Shulin Zeng, Ke Hong, Genghan Zhang, Huazhong Yang, and Yu Wang. 2024. FEASTA: A Flexible and Efficient Accelerator for Sparse Tensor Algebra in Machine Learning. In *Proc. ASPLOS-XXIX*.
[90] Qiuling Zhu, Berkin Akin, H. Ekin Sumbul, Fazle Sadi, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2013. A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing. In *2013 IEEE International 3D Systems Integration Conference (3DIC)*.
[91] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proc. MICRO-52*.