2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00087

SpZip: Architectural Support for Effective Data Compression In Irregular Applications

Yifan Yang† Joel S. Emer†* Daniel Sanchez†
†Massachusetts Institute of Technology *NVIDIA
{yifany, emer, sanchez}@csail.mit.edu

***Abstract*—Irregular applications, such as graph analytics and sparse linear algebra, exhibit frequent *indirect, data-dependent accesses to single or short sequences of elements* that cause high main memory traffic and limit performance. Data compression is a promising way to accelerate irregular applications by reducing memory traffic. However, software compression adds substantial overheads, and prior hardware compression techniques work poorly on the complex access patterns of irregular applications.**

We present SpZip, an architectural approach that makes data compression practical for irregular algorithms. SpZip accelerates the traversal, decompression, and compression of the data structures used by irregular applications. In addition, these activities run in a *decoupled* fashion, hiding both memory access and decompression latencies. To support the wide range of access patterns in these applications, SpZip is *programmable*, and uses a novel *Dataflow Configuration Language* to specify programs that traverse and generate compressed data. Our SpZip implementation leverages dataflow execution and time-multiplexing to implement programmability cheaply. We evaluate SpZip on a simulated multicore system running a broad set of graph and linear algebra algorithms. SpZip outperforms prior state-of-the art software-only (hardware-accelerated) systems by gmean 3.0× (1.5×) and reduces memory traffic by 1.7× (1.4×). These benefits stem from both reducing data movement due to compression, and offloading expensive traversal and (de)compression operations.

I. Introduction

Irregular applications, like graph analytics and sparse linear algebra, are an increasingly important workload domain [66, 80]. These applications are often memory bandwidth-bound, as they suffer frequent memory accesses with poor locality. For example, graph algorithms process sparse graphs whose footprint far exceeds on-chip storage [26, 60] and take few instructions to process each vertex and edge, which requires reading or updating small elements scattered over gigabytes of memory.

Since many irregular algorithms are memory bandwidth-bound, *data compression* is an attractive way to accelerate them. Compressing data reduces main memory traffic by both reducing transfer sizes and increasing effective on-chip capacity.

However, data compression is hard to apply to irregular applications. Irregular applications use *sparse data structures* that store only meaningful relations, like nonzero elements in a matrix or neighbors in a graph. Sparse data structures induce frequent *indirect, data-dependent accesses to single or short sequences of elements*. These indirections arise either within a data structure (e.g., when traversing a sparse matrix) or across data structures (e.g., when visiting neighboring vertex data in a graph algorithm). These complex access patterns already limit performance, and software compression overheads would make data accesses even more expensive. Practical support for compression thus requires hardware acceleration.

In this work, we propose architectural support to make data compression practical for irregular algorithms. Our approach builds on two key insights. First, ideally, we should *optimize the representation and compression algorithm of each data structure for the data statistics and access patterns generated by the application*. For example, some graph algorithms, like BFS, only process the neighbors of a subset of vertices each iteration; in this case, we must support efficient access to the neighbors of a single vertex, but each neighbor set can itself be compressed. But other algorithms, like PageRank, traverse the neighbors of all vertices in sequence, so multiple neighbor sets can be compressed together, increasing efficiency. Second, specialized hardware should handle both the *traversal* and the *decompression* of these data structures, because these operations are naturally interleaved. For example, BFS as described above first accesses a frontier data structure to find which vertices to visit, then the graph data structure to fetch and decompress the neighbors of each vertex, and finally accesses each neighbor's data.

Following from these insights, we propose SpZip, specialized hardware support for traversing and generating compressed data structures. SpZip combines two desirable characteristics. First, it is *programmable* to handle a broad range of complex access patterns and data structures. Second, it exploits *decoupled execution* to hide the latencies of compression/decompression and memory accesses: cores offload data structure traversals to a separate unit, which runs them ahead of execution.

Prior work lacks some or all of these ingredients. Compressed main memory and cache architectures [7, 8, 9, 30, 55, 56, 57] compress individual cache lines or pages transparently to software. Though they support random accesses, they are unaware of application semantics, so they sacrifice significant opportunities to compress small, variable-sized chunks of data. Moreover, they do not exploit decoupled execution, slowing down critical-path accesses. As we will see in Sec. V-D, these systems achieve limited benefits on irregular applications. Some processors feature hardware compression engines, but they support only long data streams that are accessed sequentially [1, 4, 69]. Prior work has also proposed specialized fetchers and prefetchers [5, 6, 13, 44, 73] for specific access patterns (e.g., single indirections). Though these techniques reduce access latency, they are not programmable, so they are limited to specific access patterns and data formats, and do not support compression.

SpZip hardware consists of two main components:

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00087

1) The *SpZip fetcher* accelerates the data traversal and decompression tasks offloaded by the core. Each fetcher is decoupled from the core, and runs the traversal ahead of time to hide memory access and decompression latencies. Internally, the fetcher uses a novel design that exploits decoupling and time-multiplexing to implement programmability cheaply.
2) The *SpZip compressor* is the dual of the fetcher, compressing newly generated data before it is stored off-chip. This enables applications to compress read-write data, rather than being limited to read-only data structures.

To make SpZip programmable, we introduce the *Dataflow Configuration Language (DCL)*, which allows one to express programs that traverse or generate compressed data structures. DCL programs consist of an acyclic graph of simple, composable operators: memory-access operators that fetch/write data streams, and decompression/compression operators that transform these streams. SpZip adopts the DCL as its hardware-software interface, enabling it to support both a wide range of access patterns, and data representations tailored to these patterns.

To demonstrate SpZip's effectiveness, we apply it to several graph applications and sparse linear algebra kernels. We enhance a multicore CPU with SpZip, with per-core fetchers and compressors. We find that SpZip substantially accelerates irregular applications, due to both offloading costly data structure traversals and data compression. However, we find that compression has limited benefits on basic (Push or Pull) algorithms, because memory traffic is dominated by scattered accesses to single elements, where compression is ineffective. To boost the benefits of compression, we study three optimized execution strategies that were proposed to improve locality in irregular applications: Update Batching (a.k.a. propagation blocking) [15, 36], PHI [46], and preprocessing techniques [10, 21, 70]. These optimizations reduce or eliminate incompressible accesses, and in the case of preprocessing, improve value locality (clustering similar values), so compression often reduces data movement by over 2×. SpZip significantly benefits all execution strategies, including those that were the most effective, and compression sometimes changes which strategy is most effective.

We evaluate SpZip using detailed simulation on a wide range of irregular algorithms processing large inputs. Over a multicore without hardware support, SpZip improves performance by gmean 3.0× and up to 5.2×, and traffic by 1.7×. Over PHI [46], a state-of-the-art hardware technique to reduce data movement in irregular applications, SpZip improves performance by gmean 1.5× (1.8× with preprocessing) and up to 1.9×, and traffic by 1.4× (1.7× with preprocessing). Finally, SpZip is cheap, adding only 0.2% of area to each core. These results show that, with the right hardware support, compression is an effective approach to improve performance on irregular applications.

II. The SpZip DCL and its Applications

The *Dataflow Configuration Language* (DCL) is SpZip's hardware-software interface. In this section, we first give an overview of the DCL (Sec. II-A). Then, we present the DCL by example, showing how it can be used to traverse sparse data structures (which are common in irregular applications), optimize their representation (Sec. II-B), and to accelerate more complex traversals that span multiple data structures (Sec. II-C). Finally, we introduce prior optimizations for irregular applications and show that they are synergistic with SpZip, by making compression more effective (Sec. II-D). For concreteness, we focus on graph applications, though SpZip also supports other irregular applications (like sparse linear and tensor algebra).

This section serves a dual purpose: it presents all necessary background on irregular applications and motivates SpZip and the DCL by showing its generality and effectiveness. By introducing the DCL early, we can present this background under a common framework.

A. SpZip DCL overview

The SpZip DCL allows expressing algorithms that traverse or create data structures that may be compressed. The DCL encodes each algorithm as an acyclic graph of *memory-access and (de)compression operators*. Operators communicate through streams: each operator takes in a single input stream and streams outputs to one or more consumers.

Memory-access operators can be either *indirections* or *range fetches*. An *indirection* operator takes a sequence of indices i as input, and for each i, it fetches and outputs $A[i]$. A *range fetch* operator takes a sequence of index pairs $i : j$ as input, and for each pair, it fetches and outputs $A[i]$, $A[i+1]$, ..., $A[j-1]$. Each operator is configured with some static metadata, like the address of A and the size of each element.

Decompression/compression operators change the representation of their input stream, implementing decompression (for traversals) or compression. Each system may support multiple compression formats (e.g., delta encoding [57], run-length encoding [67], BPC [35], etc.), each resulting in a different compression and decompression operator.

Queues implement the input and output streams of operators. Queues connect operators and also serve as the inputs and outputs from/to cores. Queues enable *decoupled execution* [63], externally (the traversal runs ahead of the core) and *dataflow execution* [25] internally (each operators runs as soon as its inputs are available), hiding the latency of memory accesses and decompression and exposing pipeline parallelism. Because operators fetch and produce *variable-sized chunks* of data, queues include *markers* to denote the start and end of each chunk.

B. Applying the DCL to sparse data structures

Sparse data structures represent collections of mostly zero elements by representing only nonzero values and their coordinates. They are commonly used in irregular applications where the traversal of sparse data manifests as multiple levels of indirect, data-dependent accesses to single or short sequences of elements in the memory. Therefore, understanding sparse data structures helps to uncover the challenging access patterns and compression opportunities in irregular applications.

Fig. 1 illustrates the *Compressed Sparse Row (CSR)* format,[1] which encodes a sparse matrix row by row. CSR uses two arrays,

[1]The term *compressed* here means that zeros are not explicitly stored. To avoid confusion, in this paper we use the term compression only to refer to data (entropy) compression, and refer to CSR simply as a sparse data structure.

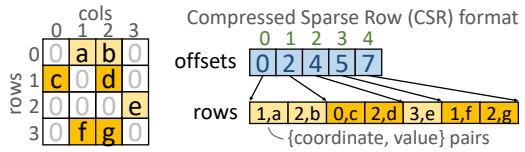


Fig. 1. Compressed Sparse Row (CSR) matrix format.

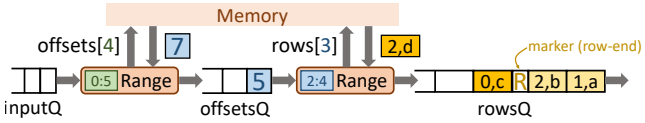


Fig. 2. DCL pipeline that traverses CSR matrix from Fig. 1.

`offsets` and `rows`. For each row index `i`, `offsets[i]` stores the starting location of the i^{th} row in the `rows` array. Each element of the `rows` array stores the column coordinate of a nonzero element and its value.

Fig. 2 shows how the DCL encodes the traversal of the sparse matrix in Fig. 1, using a simple pipeline with two **range fetch** operators. The first operator fetches the contents of the `offsets` array, and the second operator interprets these offsets as ranges to the `rows` array to fetch the contents of each row.

With this implementation, the core specifies *ranges of rows* to be fetched by enqueueing them to the input queue, and then consumes the rows by dequeueing from the output queue. In the example, the core has enqueued the range **0:5**, denoting a traversal of the whole matrix (traversing smaller ranges is also useful, e.g., with parallel processing). Note how queues decouple operation: the first range-fetch operator is currently fetching the last offset (**7**), the second operator is fetching the second element of the second row (**2,d**), and the core has not yet consumed anything. Because rows are variable-sized, a marker denotes the end of each row (**R** between **2,b** and **0,c**).

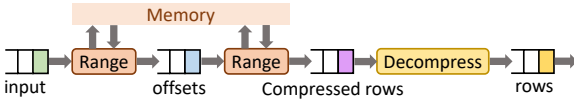


Fig. 3. DCL pipeline for matrix where each sparse row is entropy-compressed.

Data compression can further reduce the size of sparse data structures. For example, consider a variant of the CSR format in Fig. 1 where each row is individually compressed, e.g., with delta encoding, and the `offsets` array *points to the start of each compressed row*. Ligra+ [62] uses such a format to reduce bandwidth and graph size. But Ligra+ achieves small speedups because it decompresses rows in software. SpZip avoids these overheads by supporting decompression operators in the DCL.

Fig. 3 shows a DCL pipeline to traverse this CSR variant with data-compressed rows. Compared to Fig. 2, the key change is the addition of a **decompress** operator that takes compressed rows as input and produces uncompressed rows. As before, this decompress operator is decoupled from the core and other stages through queues, allowing SpZip to hide decompression latency.

DCL's generality: Though we have focused on a specific data format and access pattern, the DCL supports many other access patterns and data structures. For instance, compressing each row individually is sensible if we require accesses to random rows; for programs that access long chunks, we could compress several rows at once, and offsets could also be compressed. Fig. 1 shows coordinates and values stored contiguously, but applications often store them separately; the DCL can handle them with two parallel range fetch operators. The DCL can also handle many other sparse formats, which recent work has systematized as a composition of access primitives that the DCL supports [19, 37, 67], including matrices in DCSR, COO, DIA, or ELL; higher-dimensional tensors and tiled variants; and graphs in adjacency lists and their blocked variants, common in streaming graph analytics [27, 39]. Moreover, the DCL is not limited to a single data structure, as we will see next.

C. Applying the DCL to graph algorithms

Irregular applications are characterized by indirect, data-dependent accesses across or within data structures. So far we have seen how DCL expresses the traversal of a single sparse data structure. Since most irregular applications use *multiple* data structures, effective acceleration requires handling access patterns that span data structures, including read-only and read-write data. To illustrate this, we focus on graph algorithms.

Most graph algorithms proceed in iterations. On each iteration, the data of each vertex is updated based on the data of neighboring vertices. There are two basic execution styles: Push (source-stationary) and Pull (destination-stationary). In Push algorithms, source vertices (i.e., those whose values must be propagated) are processed one by one, and each vertex propagates (pushes) its update to its outgoing neighbors. In Pull algorithms, destination vertices are processed one by one, and each vertex reads (pulls) updates from its incoming neighbors.

Push and Pull algorithms use the same main data structures: the graph adjacency matrix, usually in CSR format as shown in Fig. 4, encodes the outgoing (Push) or incoming (Pull) edges of each vertex; and one or more arrays that hold algorithm-specific per-vertex or per-edge data.

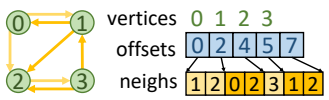


Fig. 4. Adjacency matrix in CSR.

```
1  def PageRankIter(Graph g, Array scores,
2                   Array contribs):
3    for src in range(g.numVertices):
4      for dst in g.neighs[g.offsets[src]:
5                          g.offsets[src+1]]:
6        scores[dst] += contribs[src]
```

Listing 1. Single iteration of Push PageRank.

Listing 1 shows pseudocode for one iteration of Push PageRank. In this code, vertices push updates (`contribs`) to their out-neighbors, adjusting their `scores` (in a second phase, not shown, the score and contrib of each vertex are updated for the next iteration). Accesses to destination `scores` require two indirections (`offsets`→`neighbors`→`scores`) that span two data structures (the adjacency matrix and the scores array).

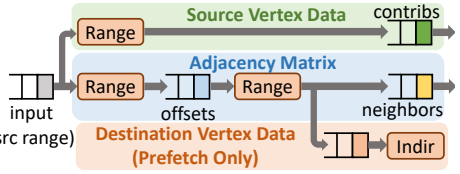


Fig. 5. DCL pipeline for Push PageRank.

```
1  def BFS(Graph g, VertexId root):
2    curDist = 0
3    frontier = [root], nextFrontier = []
4    Array dists(g.numVertices, INFINITY)
5    while !frontier.empty():
6      for src in frontier:
7        for dst in g.neighs[g.offsets[src]:
8                            g.offsets[src+1]]:
9          if dists[dst] == INFINITY:
10           dists[dst] = curDist
11           nextFrontier.append(dst)
12     curDist += 1
13     frontier = nextFrontier
14     nextFrontier = []
15   return dists
```

Listing 2. Push-based BFS.

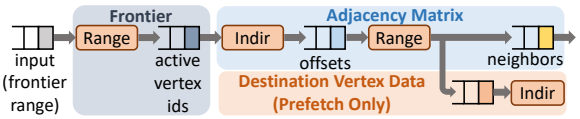


Fig. 6. DCL pipeline for non-all-active, Push BFS.

To accelerate PageRank with SpZip, the DCL encodes a program that traverses all data structures, and the core is dedicated to the compute operations (line 6 of Listing 1). Fig. 5 shows the DCL pipeline that achieves this, divided in three regions. The **blue** region traverses the adjacency matrix and fetches neighbor ids (`dst`); the **green** region fetches source vertex data (`contribs`); and the **orange** region *prefetches* destination vertex data (`scores`). In this case, destination vertex data are not passed to the core, because updates to scores must be done with atomics (as with parallel execution, multiple cores may update the same destination). This prefetching is accomplished with an indirection operator (which accesses `scores[dst]` for each `dst` at its input queue) that does not have an output queue. Because the fetcher is colocated with the core, this prefetch leaves data nearby (at the core's L2 cache in our implementation).

All-active and non-all-active algorithms: Some graph algorithms, like PageRank in Listing 1, are *all-active*: they process all graph vertices on each iteration. By contrast, *non-all-active* algorithms maintain a subset of active vertices, known as the frontier, and process only the active vertices on each iteration.

Listing 2 shows the code for a non-all-active algorithm, Push Breadth-First Search (BFS). BFS keeps the active source vertices in the `frontier` data structure. Fig. 6 shows the DCL program for BFS, which fetches neighbor (destination vertex) ids and prefetches their distances. Compared to PageRank, BFS has another level of indirection, shown in **grey**, to access the frontier and fetch active vertex ids. Due to this extra indirection, BFS has non-contiguous accesses to `offsets`.

Data compression can be added to DCL application pipelines just like we saw in Sec. II-B. In general, compression opportunities exist whenever the application accesses *contiguous sequences* of elements, i.e., after range fetch operators. For example, in PageRank, we could compress neighbors and offsets (or chunks of them), and in BFS, we could compress neighbors and the frontier (but not offsets, which are accessed non-contiguously). In this way, we can tailor the representation of each data structure to the application's specific access patterns, maximizing the benefits of compression.

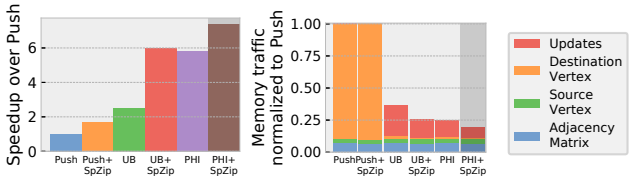


Fig. 7. Performance and memory traffic breakdown of BFS on `uk-2005`, normalized to Push.

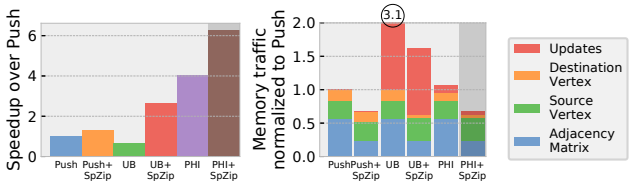


Fig. 8. Performance and traffic of BFS on `uk-2005` with preprocessing, normalized to Push with preprocessing.

SpZip benefits: Fig. 7 reports the performance (left) and memory traffic (right) for different variants of the non-all-active BFS, processing a large web graph on a 16-core system (see Sec. IV for methodology). The two leftmost bars compare a software-only Push implementation and the Push implementation enhanced with SpZip. SpZip is 1.7× faster than Push. However, this speedup stems almost exclusively from accelerating traversals, and compression barely helps: both variants incur about the same memory traffic.

To see why, Fig. 7 breaks down traffic by data type.[2] First, *scatter updates* to **destination vertex data** dominate, consuming over 80% of traffic. Because these are accesses to single elements, SpZip does not compress them. Second, most of the remaining traffic is to the **adjacency matrix**, for which SpZip compresses neighbor sets. However, this graph has neighbor ids that are highly scattered, so compression barely helps.

Overall, SpZip outperforms Push because it relieves cores from costly traversals and misses, effectively saturating memory bandwidth. To improve performance further, we must reduce memory traffic. To this end, we now discuss optimized execution strategies that improve locality and make compression more effective.

D. Graph processing optimizations

Update batching (UB) [15, 36] also known as propagation blocking [15] or DRAM-conscious clustering [36], improves the spatial locality of Push algorithms by transforming the cache misses from scatter updates, which often dominate traffic (Fig. 7), into batches of sequential memory accesses.

Listing 3 shows an example UB application, serial PageRank. UB splits execution into two phases, *binning* and *accumulation*. In the binning phase, each source vertex generates updates (`contribs`) for its neighbors from its vertex data. Instead of directly applying these updates to destination vertices as Push does, updates are buffered in *bins* (as {`dst, contribs[src]`} tuples). Each bin collects updates destined to a *cache-fitting range of destination vertices*. In the accumulation phase, the algorithm applies updates bin by bin to update the vertex `scores`.

[2]This BFS is a different variant from the one in Listing 2: it builds the BFS tree (Sec. IV), so it has **source vertex data**, as Fig. 7 shows.

```
1  def PageRankIter(Graph g, Array scores,
2                   Array contribs):
3    # Binning phase
4    for src in range(g.numVertices):
5      for dst in g.neighs[g.offsets[src]:
6                          g.offsets[src+1]]:
7        bId = dst / verticesPerBin
8        bins[bId].append({dst,contribs[src]})
9
10   # Accumulation phase
11   for bin in bins:
12     for {dst, contrib} in bin:
13       scores[dst] += contrib
```

Listing 3. Serial PageRank using Update Batching (UB).

UB reduces traffic thanks to spatial locality: though binned updates are large and are spilled to main memory, each bin is written sequentially, resulting in streaming writes that use full cache lines, unlike scatter updates. In the accumulation phase, these scatter updates still happen, but they mostly result in *hits* because each bin is restricted to a cache-fitting slice of vertices.

The DCL pipeline for the binning phase of UB-based PageRank (Listing 3) is the same as Fig. 5 except it does not prefetch destination vertex data. The DCL pipeline for the accumulation phase consists of one range fetch operator for the bins (and one decompression operator after it if the bins are compressed).

Fig. 7 shows that UB reduces memory traffic significantly, by 2.7×. Traffic is now dominated by sequential accesses to **updates**, and **destination vertex data** traffic is small since it enjoys perfect reuse. Overall, UB is 2.5× faster than Push.

More importantly, UB *enables effective compression*: each bin is accessed sequentially, so bins can be compressed well, reducing the dominant contribution to memory traffic. (We will see how SpZip compresses bins in detail in Sec. III-C.) Fig. 7 shows that UB+SpZip reduces update traffic by 1.6× and overall traffic by 1.4× over UB. Since SpZip also handles UB's traversals, it is 2.4× faster than UB and 6.0× faster than Push.

PHI [46] builds on UB and further reduces traffic by adding hardware support, and provides state-of-the-art data movement reductions. PHI builds on the observation that the updates in many algorithms are *commutative* and *associative*. PHI leverages this to opportunistically coalesce updates to the same destination vertex in the cache hierarchy before binning and spilling them off-chip. Cores push updates to caches, which buffer and coalesce them. Updates are binned lazily: when a cache line with updates is evicted from the LLC, its updates are written into bins. Bins are then spilled to main memory.

PHI and SpZip offer complementary ways to reduce data movement. Fig. 7 shows that, on BFS, PHI and UB+SpZip achieve similar traffic and performance. But PHI and SpZip can be combined to yield even larger benefits: PHI+SpZip reduces **update** traffic by a further 1.3× and improves performance by 26%. Overall, PHI+SpZip is 7.4× faster than Push.

Preprocessing techniques reorder sparse data structures to improve locality [31, 70, 74, 78, 79, 82]. In graph algorithms, preprocessing reorders vertex ids in the adjacency matrix.

Preprocessing is beneficial only when the graph is reused enough times to amortize preprocessing cost, which is often much higher than the algorithm itself. Thus, prior work has proposed lightweight and heavyweight preprocessing techniques. Several lightweight techniques use *degree sorting* [10, 28, 79]. They group high-degree vertices, which are accessed more frequently, improving spatial locality. Other lightweight techniques use *topological sorting*, reordering vertices according to their topology, e.g., through a BFS [21] or DFS [12, 74] traversal order. These algorithms group vertices with many connections, improving temporal and spatial locality. Heavyweight techniques like GOrder [70] improve locality further, but use complex algorithms that are orders-of-magnitude slower [10, 45].

Preprocessing improves locality and makes compression more effective. Fig. 8 shows results for the same experiments as in Fig. 7, but when the graph is preprocessed with DFS. Preprocessing dramatically reduces Push's destination vertex traffic, as most updates happen to nearby data. By contrast, because UB does not exploit temporal locality, preprocessing does not reduce memory traffic, making it far worse than Push. PHI achieves similar memory traffic as Push but higher performance thanks to hardware support.

The SpZip variants of each algorithm significantly improve performance and reduce traffic by about 1.5× thanks to compression. In both Push and PHI, the **adjacency matrix** now dominates traffic. But while SpZip compression was ineffective in the original graph, the preprocessed graph has high value locality: since related vertices are placed nearby, each neighbor set often has similar ids. Thus, SpZip reduces its size by 2.3×. Overall, PHI+SpZip provides the highest speedups: 6.3× over Push with preprocessing, and 7.4× without preprocessing.

These results show that the best algorithm variant to use depends on the graph and available optimizations (e.g., whether preprocessing is practical). We will also later see (Sec. V) that compression favors topology-based preprocessing algorithms because they improve value locality further than degree-sorting preprocessing, which is more widely used without compression.

III. SpZip Design

A. SpZip overview

SpZip consists of two key components: *fetchers* that traverse and decompress data structures and feed them to cores, and *compressors* that compress new data before they are stored off-chip. We implement SpZip in a multicore system. Fig. 9 highlights SpZip's additions and shows how they fit into the system: each core has a fetcher, which issues accesses to the core's private L2 cache, and a compressor, which issues accesses to the LLC.

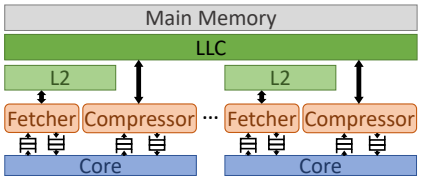


Fig. 9. SpZip architecture overview.

Each core communicates with its fetcher and compressor through queues: the input and output queues of these units are exposed to the core. The core uses `enqueue` and `dequeue` instructions to move data between queues and registers.

We first present SpZip's fetcher (Sec. III-B) and compressor (Sec. III-C), then discuss system-level issues (Sec. III-D), and conclude by analyzing SpZip's small area costs (Sec. III-E).

B. SpZip fetcher

The SpZip fetcher runs DCL programs to traverse and decompress data structures, as we saw in Sec. II. Each core's fetcher issues accesses to its L2 cache. This keeps data in compressed form in the L2 and LLC, increasing their effective capacity.

To begin execution, the core loads a DCL program in the fetcher, and enqueues some initial inputs to it. The fetcher then runs *decoupled* from the core: it issues accesses autonomously and runs the traversal and data decompression ahead of the core, filling its output queues with data that the core can then dequeue. Importantly, unlike prior indirect prefetchers [5, 6, 13, 73], SpZip does *not* monitor the core's accesses to infer which accesses to perform. This is simpler, more efficient, and supports decompressing data in ways that prefetchers cannot. However, it does require changes to programs.

We first discuss the microarchitecture of the fetcher, then explain how to modify programs to use it. (Though we show application code using SpZip directly for illustration purposes, our implementation uses SpZip through a Ligra-style runtime, and leaves application code unchanged—see Sec. III-D.)

Fetcher microarchitecture: The key challenge in implementing the SpZip fetcher is *how to implement programmability efficiently*. Prior fetch units [38, 40, 44] implemented either a fixed-function pipeline, or provided some configurability by disabling or configuring components within a fixed pipeline. But SpZip requires a more flexible approach.

A possible fetcher implementation would be to adopt a reconfigurable design, with multiple fetch and decompression operators and queues that can be wired through a configurable interconnect to implement DCL programs. However, this is costlier than needed. Instead, we observe that, when each operator is implemented as a specialized unit with reasonable throughput (e.g., a range fetcher or decompressor that can produce up to 32 bytes/cycle), operators have relatively low activity factors: to consistently run ahead of cores, it suffices for one operator to fire 33% of the cycles in our applications.

This observation leads us to a *time-multiplexed* fetcher design, shown in Fig. 10, where operators and queues share the same physical units. The fetcher has three types of components:

- The *scratchpad* stores the queues of the DCL program. Each queue uses a region of the scratchpad, and is managed as a circular buffer, with head and tail pointers as shown in Fig. 10.
- Two functional units (FUs), the *access unit* and the *decompression unit*, implement the functionality needed by the memory-access and decompression DCL operators.
- The *scheduler* chooses which operator to execute each cycle. It keeps multiple *operator contexts*. Each context holds the configuration and input/output queue ids of a single operator in the DCL program. The scheduler tracks which operators are ready to execute, and picks one each cycle.

This design achieves high throughput by exploiting *dataflow execution* and by *decoupling operators* through queues: each operator fires only when its input queue has an element to process and its output queues have sufficient space, and operators can run ahead and buffer results in queues, hiding memory and

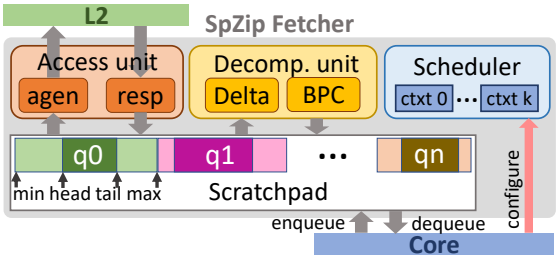


Fig. 10. SpZip programmable fetcher microarchitecture.

decompression latencies. This design supports any DCL program with as many operators as contexts and as many queues as the scratchpad can track (16 each in our implementation).

Access Unit (AU): The AU implements both memory-access operators: *indirections* and *range fetches*. The AU is internally decoupled to support many outstanding memory requests, as shown in Fig. 10: the *address generator* (agen) issues a memory request for each indirection or range fetch, and the *response handler* (resp) tracks of outstanding requests and writes data to output queues as responses are received. Memory requests are sent to the L2 cache, but they often miss, so the AU supports multiple outstanding requests (up to 8 cache lines in our implementation) to hide memory latency.

Decompression Unit (DU): The DU implements decompression operators. Its implementation includes decompression logic for all supported compression algorithms.

We use two algorithms in our design, delta encoding [57] and Bit-Plane Compression (BPC) [35], because we observe that no algorithm dominates. Our delta-encoding implementation simply subtracts the previous and current inputs, and emits an N-byte output if their delta (plus a small length prefix) fits within N bytes (some prior work calls this a byte code [62]). BPC uses a more sophisticated encoding but needs longer chunks (e.g., 32 elements) to compress effectively. Thus, delta encoding is preferable on short streams, like individual neighbor sets, and BPC on longer chunks, like bins in Update Batching.

Scheduler: The scheduler decides which operator context to execute and arbitrates reads and writes to the queues. It uses queue occupancy to determine which operators are ready. An operator is ready if its input queue has an element, its output queues have sufficient free space, and its functional unit (FU) is available (e.g., the AU may be filled with inflight requests). The scheduler follows a round-robin policy among ready operators. To issue an operation, the scheduler feeds the head of the input queue to the FU, along with the context id. (Though contexts are shown separately in Fig. 10, each context is specialized to an FU, and tracks FU-specific information, e.g., the current index for each range fetch.) The scheduler also arbitrates writes to queues performed by the access and decompression units.

Queues and markers: Queues are held in the scratchpad. Each queue is managed as a circular buffer, and tracked with the usual pointers: min/max and head/tail. Queues take and serve elements of various widths, as specified by their operators. For example, a range fetch can consume 8, 16, 32, or 64-bit indexes.

Queues implement *markers* to support variable-length chunks: each 32-bit word is tagged with a *marker* bit; if set, the word is

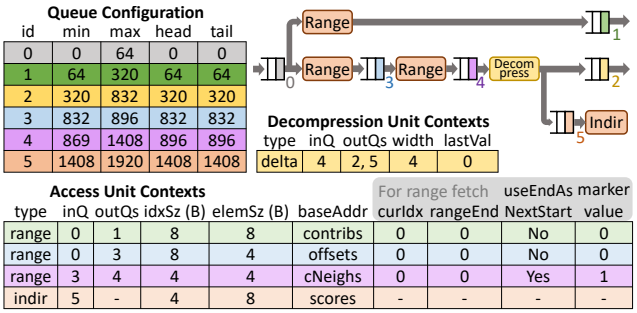


Fig. 11. DCL pipeline (top right) and fetcher configuration (queue configuration and FU contexts) for Push PageRank with compressed neighbors.

```
1  def PageRankIter(Graph g, Array scores,
2                   Array contribs):
3    spzip_fetcher_cfg(...) # load Fig.11 pipe
4    inQ = 0, contribsQ = 1, neighQ = 2
5    enqueue(inQ, {0, g.numVertices})
6    while contrib = dequeue(contribsQ)
7                    is not marker:
8      while dst = dequeue(neighQ)
9                      is not marker:
10       scores[dst] += contrib
```

Listing 4. Serial Push PageRank using SpZip fetcher.

interpreted as a marker instead of data. The range fetch operator emits a marker after finishing each range, and the decompression operator interprets each range as the end of a compressed chunk. Each range fetch operator can specify a value associated with each marker, and all operators have *pass-through* semantics for markers: each marker at the input is copied to the output queues. This lets the core infer the semantics of dequeued elements. For example, in the BFS pipeline from Fig. 6, the core can identify the end of each neighbor set thanks to markers produced by the neighbor range fetch, and the end of each frontier range thanks to markers produced by the earlier frontier range fetch.

Fetcher usage and API: To load a DCL program in the fetcher, the core writes the configuration (sizes and locations) of all queues, as well as the configuration of each operator context. Fig. 11 shows the configuration for a variant of Fig. 5's PageRank pipeline where, in addition, neighbor sets are compressed. This configuration is done rarely, through memory-mapped I/O.

Listing 4 shows pseudocode for an iteration of serial Push PageRank (similar to Listing 1) using the DCL program from Fig. 11. The `spzip_fetcher_cfg(...)` runtime function configures the fetcher as described above, and `enqueue()` and `dequeue()` translate to specialized instructions. The initial enqueue starts a traversal of the full compressed graph (by enqueueing the range `{0, g.numVertices}` to queue 0 in Fig. 11). The code then dequeues contribs and neighbor ids, and applies each source's contrib to its neighbors' scores. The dequeue instruction reads the value and whether it is a marker, and the code exits each loop upon finding a marker. (Our implementation uses x86, so the marker bit is stored in the overflow flag; ISAs without flags can use a second output register instead.)

C. SpZip compressor

The SpZip compressor is the dual of the fetcher: it compresses newly generated data before it is written back to main memory. The compressor shares many features with the fetcher: it is a decoupled engine; the core communicates with it using queues, and configures it using memory-mapped I/O; it is programmable; and it uses a similar time-multiplexed implementation. However, the compressor has two key differences. First, it issues LLC accesses rather than L2 accesses. This avoids polluting private caches with data that is unlikely to be reused before being evicted, and lets the engine use the larger LLC to buffer yet-to-be-compressed data. Second, compressors have a different mix of operators, to compress and write data streams.

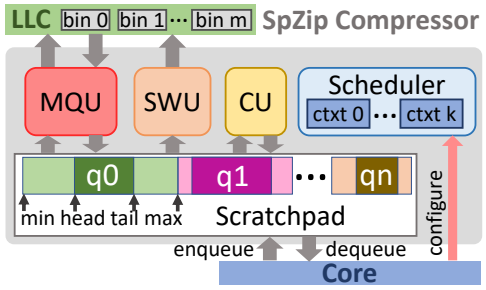


Fig. 12. SpZip compressor microarchitecture.

We focus our compressor design on handling either a single stream (e.g., the frontier in BFS) or many streams (e.g., the update bins in Update Batching). With simple extra functional units, the compressor could produce all elements of compressed sparse data structures (like the CSR with compressed rows from Sec. II-B; the compressor can currently produce the rows, but not the offsets, which a core would need to produce). But we do not need this in the applications we study.

Compressor microarchitecture: Fig. 12 shows the design of the compressor, which includes a scratchpad for local queues, several functional units, and a scheduler, just like the fetcher.

The functional units include a *compression unit* (CU), a *stream writer unit* (SWU), and a *memory-backed queue unit* (MQU). The first two are used to compress a single stream, and the last one uses (cached) memory to implement a large number of queues. This is needed when compressing many streams so that we can leverage the LLC to buffer them. We explain these units through DCL pipelines for one and many streams.

Compressing a single stream: Fig. 13 shows the DCL pipeline for compressing a single stream, like the frontier or a sequence of rows. The *compression* operator processes the uncompressed input stream, and the *stream writer* writes it sequentially to (cached) memory, starting at a configured address, and tracks the length of the compressed stream. The input stream uses markers to delimit the chunks to compress (e.g., to separate rows or denote the end of the frontier).

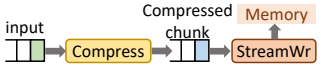


Fig. 13. DCL pipeline to compress a single stream.

Compressing many parallel streams: Fig. 14 shows the DCL pipeline for compressing the bins in Update Batching (UB), which is the most impactful use of on-the-fly compression in our applications. Conceptually, each bin is a separate stream. But UB needs many parallel streams, because each bin is limited to a *cache-fitting* slice. For example, a graph with 32 GB of vertex data and a 32 MB LLC needs 1024 bins. This would take 2048

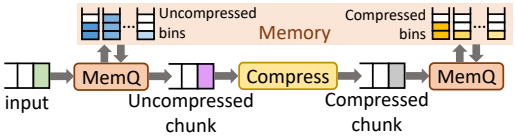


Fig. 14. DCL pipeline to compress UB bins (multiple streams).

queues in the fetcher if we followed the single-stream approach in Fig. 13 (which uses two queues per stream). This would demand a too-large queue scratchpad.

Instead, the *memory-backed queue unit* (MQU) implements queues in conventional memory (without changing the internals of the cache): *both the queue storage and its pointers are kept in memory*. The MQU interacts with memory through conventional loads and stores (issued to the LLC in our implementation). The MQU context is simply the number of queues, and the starting address of an array that holds min, tail, and max pointers of each queue. The MQU interprets its input queue as `(queueId, value)` tuples, and enqueues `value` to the queue with `queueId`. This requires reading and writing the tail pointer, and writing the value to memory. When an in-memory queue fills up or the input is a marker, the MQU streams the queue through its output (as the `queueId` followed by the contents of the in-memory queue), and marks the `queueId` queue empty; or, if MQU has no output queue, it quiesces and interrupts a core (e.g., to allocate more space for the queue).

Fig. 14 begins with an MQU that manages one queue per bin. These queues store uncompressed bin data, and their purpose is to build large enough chunks (e.g., 32 updates per bin) that can be compressed efficiently. Once a queue fills up, the chunk is passed through the compressor from the first MQU. The compressed chunk is then handled the second MQU, which manages the compressed bins. Software allocates an initial amount of space per bin (e.g., 2 MB), and allocates more space on demand, when the bin fills up, using the interrupt mechanism. With this approach, the first MQU manages a limited amount of space that the LLC holds easily, whereas the second MQU appends to the much larger compressed bins. Compressed bins typically exceed the LLC's capacity, so conventional evictions displace them to main memory.

Compression optimizations for order-insensitive data: As described in Sec. III-B, we implement delta-encoding and BPC [35] compression. In addition, we find that in many cases compressed data is often *order-insensitive*: the order in which it is encoded is irrelevant. For instance, bins store sets of updates, and the frontier stores the set of active vertices, so reordering the elements in these streams does not affect semantics.

We leverage this by optionally sorting the elements of a chunk (32 elements in our implementation) before compressing it. This places similar values nearby, which improves compression ratios in both delta encoding and BPC.

Compressor usage and API: Like the fetcher, the compressor is configured by writing its queue and operator contexts through memory-mapped I/O. Listing 5 shows pseudocode for serial UB PageRank. In the binning phase, the fetcher fetches and decompresses contribs and neighbor ids (like in Listing 4), and the compressor compresses each bin. In the accumulation phase, the fetcher fetches each compressed bin, and the code applies the updates in each bin to neighbor scores.

```
1  def PageRankIter(Graph g, Array scores,
2                   Array contribs):
3    spzip_fetcher_cfg(...) # Fig.11 w/o indir
4    spzip_comp_cfg(...) # Fig.14 pipe
5    inQ=0, contribsQ=1, neighQ=2, binsQ=8
6    enqueue(inQ, {0, g.numVertices})
7    # Binning phase
8    while contrib = dequeue(contribsQ)
9                    is not marker:
10     while dst = dequeue(neighQ)
11                   is not marker:
12       binId = dst / verticesPerBin
13       enqueue(binQ, {binId, {dst,contrib}})
14   # close bins & wait till they're produced
15   for binId in range(numBins):
16     enqueue(binQ, {binId, endMarker})
17   spzip_comp_drain() # wait
18
19   # Accumulation phase
20   for cBin in compressedBins:
21     spzip_fetcher_cfg(...)# decompress cBin
22     spzip_comp_cfg(...) # Fig.13 pipe
23     enqueue(inQ, {0, cBin.size})
24     while {dst, contrib} = dequeue(binQ)
25                          is not marker:
26       scores[dst] += contrib
```

Listing 5. Serial UB PageRank using SpZip compressor and fetcher.

D. System-level integration

Using SpZip: While we have so far shown code that uses SpZip explicitly, we do not expect application programmers to write DCL programs directly. Instead, application code can use specialized runtimes or compilers. In fact, we use a Ligra-style graph processing framework (Sec. IV), and change only framework code to use SpZip, not application code. The DCL program is manually extracted via analyzing the data structures and their traversal pattern in the graph processing framework. Moreover, domain-specific languages like TACO [37] and GraphIt [18, 80] encode all the needed information to automatically generate DCL programs from high-level code.

Parallelism and load balancing: Although prior examples show serial execution, we use SpZip in a parallel fashion. Our runtime divides either the vertices (in all-active) or frontier (in non-all-active algorithms) into chunks, and divides them among threads. Threads then enqueue traversals to fetchers chunk by chunk, and perform work-stealing of chunks to avoid load imbalance. In Update Batching, in the binning phase, threads (and their compressors) produce bins in parallel; in the accumulation phase, each bin is also chunked and consumed in parallel by multiple threads (and their fetchers).

Virtual memory: SpZip operates on virtual addresses. Like prior indirect prefetchers and fetchers, each SpZip fetcher and compressor use their core's address translation hardware [5, 44, 73]. Specifically, fetcher and compressor use the core's L2 TLB. If a unit causes a page fault, it interrupts the core, so the OS can handle the page fault. The unit stops issuing accesses after a fault, and the OS reactivates it after the fault is handled.

Coherence and consistency: SpZip issues coherent memory accesses, and does not affect coherence or consistency. But SpZip engines operate autonomously, like separate threads,

without any synchronization other than through queues. This suffices for SpZip to fully handle both read-only data (e.g., the adjacency matrix) and read-write data that is produced and consumed in different phases (e.g., bins in UB and the frontier in non-all-active algorithms). Some shared read-write data, like destination vertex data in Push, incurs parallel read-modify-writes that must happen atomically. In this case, the fetcher does not serve this shared read-write data directly, though it still prefetches it (as we saw in Sec. II-C), and leaves the atomics to the core. These prefetches hide most of the cost of these accesses, which are frequently served by main memory.

Context switches: Similar to prior fetchers and systems with explicit queues [42, 44, 49], fetcher and compressor have architecturally visible state that must be saved and restored when a thread is context-switched. If the OS deschedules a thread, it needs to quiesce its fetcher and compressor, save their contexts, and restore them when the thread is rescheduled. This needs to be done only when switching to a different process, not on exceptions or syscalls (just like these do not save FPU state).

E. Area analysis

We implement the fetcher and compressor in RTL and synthesize them using yosys [71] and the 45nm FreePDK45 library [48]. We use CACTI [11] to estimate SRAM area. Each engine uses a 2 KB scratchpad for queues. Our BPC [35] implementation supports 32- or 64-bit elements, and uses a simple byte-level symbol encoding for each bitplane.

TABLE I
SPZIP AREA BREAKDOWN.

Fetcher	Area(μm^2)	Compressor	Area (μm^2)
AccU	10.1k	MQU & SWU	5.8k
DecompU	22.5k	CompU	25.0k
Scratchpad	6.8k	Scratchpad	6.8k
Scheduler	7.9k	Scheduler	7.9k
Total	47.3k	Total	45.5k

Table I shows the area cost of each engine. Fetcher and decompressor add minimal overheads, requiring 0.2% of the area of a general-purpose core (the Intel Haswell core used in the evaluated system, scaled to the same technology node).

IV. Experimental Methodology

Simulation infrastructure: We perform microarchitectural, execution-driven simulation using zsim [59]. We simulate a 16-core system with parameters given in Table II. The system uses out-of-order cores modeled after and validated against Intel Haswell cores. Each core has private L1 and L2 caches, and all cores share a banked 32 MB last-level cache. The system has four memory controllers, like Haswell-EP systems [32].

Applications We use seven benchmarks. Three are all-active algorithms: PageRank (`PR`) ranks vertices in a graph [53]; *Degree Counting* (`DC`) computes the incoming degree of each vertex and is often used in graph construction [14]; and *Sparse Matrix-Vector Multiplication* (`SP`) is a key sparse linear algebra kernel. Four are non-all-active: PageRank Delta (`PRD`), is an optimized `PR` variant that only processes vertices with enough change in their PageRank score each iteration; *Breadth-First Search* (`BFS`) produces the breadth-first tree from a root vertex [14]; *Connected Components* (`CC`) partitions vertices of a graph into disjoint subsets (or components) so that no edge crosses subsets

TABLE II
CONFIGURATION OF THE SIMULATED SYSTEM.

Cores	16 cores, x86-64 ISA, 3.5 GHz, Haswell-like OOO [59]
L1 caches	32 KB per core, 8-way set-associative, split D/I, 3-cycle latency
L2 cache	256 KB, core-private, 8-way set-associative, 6-cycle latency
L3 cache	32 MB, shared, 16 banks, 16-way hashed set-associative, inclusive, 24-cycle bank latency, DRRIP replacement
Global NoC	4×4 mesh, 128-bit flits and links, X-Y routing, 1-cycle pipelined routers, 1-cycle links
Coherence	MESI, 64 B lines, in-cache directory, no silent drops
Memory	4 controllers, FR-FCFS, DDR3 1600 (12.8 GB/s per controller)

[20]; and *Radii Estimation* (`RE`) performs parallel BFS's from a few vertices to estimate the radius of each vertex [43].

Datasets: We evaluate graph algorithms on five large web and social graphs shown in Table III. For SpMV, we use a matrix representative of structured optimization problems.

TABLE III
INPUT DATASETS.

Graph	Vertices(M)	Edges(M)	Source
arb	22	640	arabic-2005 [24]
ukl	39	936	uk-2005 [24]
twi	41	1468	Twitter followers [41]
it	41	1150	it-2004 [24]
web	118	1020	webbase-2001 [24]
nlp	27	760	nlpkkt240 [24]

We evaluate preprocessed and non-preprocessed variants of each input. Since several input graphs are already preprocessed, for the non-preprocessed variants we randomize the vertex ids of the input graph. Preprocessed variants use DFS by default [12], and we also study the sensitivity to preprocessing techniques.

Graph algorithms run for several iterations. Since these graph are large, to avoid long simulation times, we use *iteration sampling*, simulating every 5th iteration and fast-forwarding others. This is accurate since the characteristics of graph algorithms change slowly over iterations. Even with iteration sampling, we simulate over 100 billion instructions for the largest graph.

Schemes: We use three baselines: Push, Update Batching (UB), and PHI. We use optimized implementations from the authors of Propagation Blocking [15] for UB, and from PHI [46].

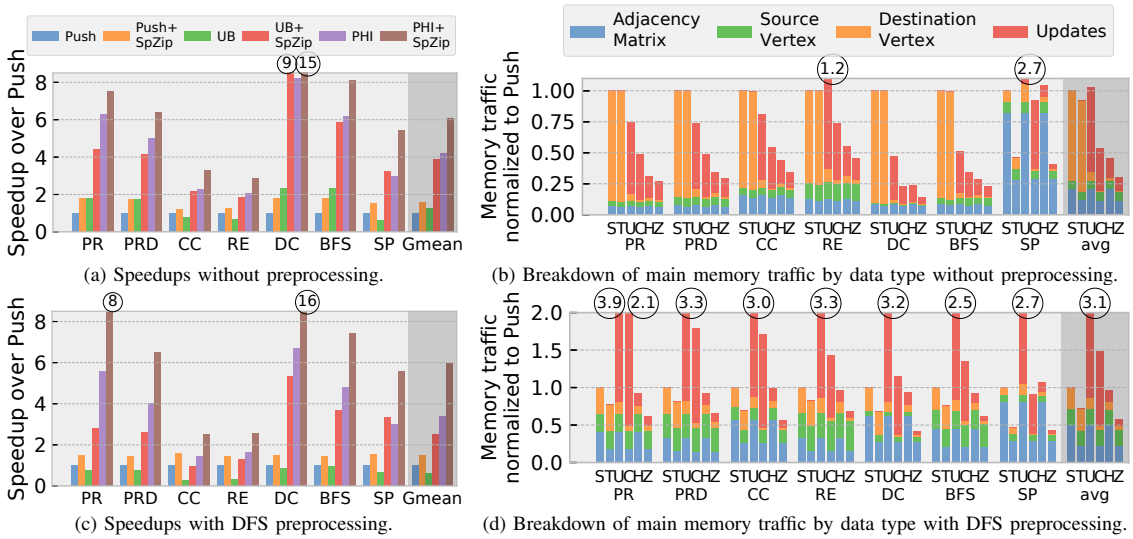
We enhance these strategies with SpZip. For Push, we compress the adjacency matrix, but not vertex data; for UB and PHI, we compress all structures (adjacency matrix, update bins, and vertex data); destination vertex data is compressed after applying each bin in the accumulation phase. We compress the adjacency matrix using delta encoding, and each application uses the best of BPC [35] and delta encoding for the other structures. To integrate PHI and SpZip, we change PHI so that, when a line with updates is evicted from an LLC bank, we send the line's updates to the compressor in the same chip tile.

Framework: We implement all algorithms on a carefully optimized framework based on Ligra [61]. We modify the framework's code to implement all of the above schemes; application code remains unchanged.

V. Evaluation

A. SpZip improves performance and reduces memory traffic

Fig. 15 reports the performance (higher is better) and off-chip memory traffic (lower is better) for all schemes without and with DFS preprocessing. All results are normalized to Push. Each bar groups shows results for all schemes on one application, averaged across all inputs. The last group of each plot



(a) Speedups without preprocessing.

(b) Breakdown of main memory traffic by data type without preprocessing.

(c) Speedups with DFS preprocessing.

(d) Breakdown of main memory traffic by data type with DFS preprocessing.

Fig. 15. Per-application speedups and memory traffic breakdowns for all schemes. S:Push, T:Push+SpZip, U:UB, C:UB+SpZip, H:PHI, Z:PHI+SpZip. Normalized to Push, averaged across all inputs.

(shaded darker) shows results across applications. Averages are geometric means for speedups and arithmetic means for traffic.

Results without preprocessing: Fig. 15a shows that SpZip improves performance for all schemes: SpZip accelerates Push by gmean 1.6×, UB by 3.0×, and PHI by 1.5×. PHI+SpZip is consistently the fastest technique: it is gmean 6.1× faster than Push (up to 15.4× on DC), and 4.8× faster than UB, the best-performing software-only scheme. If one does not wish to modify the cache hierarchy, which PHI does, UB+SpZip offers similar speedups (PHI is gmean 8% faster) without PHI's changes. These results show SpZip is highly versatile and effective.

Fig. 15b shows that SpZip's impact on off-chip traffic tracks what we saw in Sec. II: Push+SpZip barely reduces traffic over Push because compression is ineffective (except in SP, where the input is more structured). However, SpZip reduces traffic substantially over UB and PHI because their access patterns are more compressible. Across benchmarks, Push and UB achieve nearly the same traffic (UB is worse on RE and SP). UB+SpZip reduces traffic by 1.9× on average over Push, PHI by 2.2×, and PHI+SpZip by 3.3× (and up to 7.2× on DC). Fig. 15b shows that SpZip's data movement reductions stem primarily from compressing updates, and secondarily from compressing vertex data. Compression benefits all applications. Its benefits are more muted on PR and PRD because they have floating-point values with little value locality, making them harder to compress.

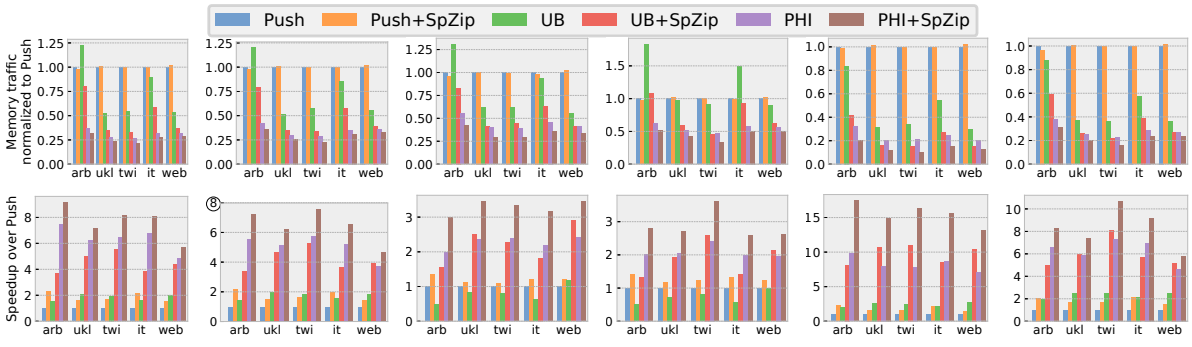
Overall, memory traffic reductions track speedups for Push+SpZip, UB+SpZip, and PHI+SpZip, as well as for PHI. This is because these schemes saturate memory bandwidth, showing that SpZip effectively avoids core bottlenecks by offloading traversals. By contrast, Push and UB often do not saturate memory bandwidth, as traversals bottleneck cores.

Results with DFS preprocessing: Preprocessing changes the tradeoffs among techniques like we saw in Sec. II: Fig. 15c and Fig. 15d shows that Push now outperforms UB, which is 41% slower and incurs 3.1× more traffic. This is because preprocessing improves locality for destination vertex data in Push, but does not help UB, which streams all updates to memory even when they have high locality. Despite its traffic gains, Push is only 41% faster than UB due to the overheads of atomic updates to shared vertex data. PHI's traffic is similar to Push's because coalescing updates exploits temporal locality, and is 3.4× faster than Push because it avoids synchronization overheads.

Regardless of these changes, SpZip substantially improves performance across all techniques: SpZip accelerates Push by gmean 1.5×, UB by 4.2×, and PHI by 1.8×. PHI+SpZip is consistently the fastest technique: it is gmean 5.9× faster than Push (up to 16.9× on DC). Despite UB being worse than Push, UB+SpZip outperforms Push+SpZip because it avoids synchronization overheads, as compression reduces the overhead of streaming updates to memory. UB+SpZip is also close to PHI (gmean 36% slower), making it a reasonable approach if implementing PHI is not desirable.

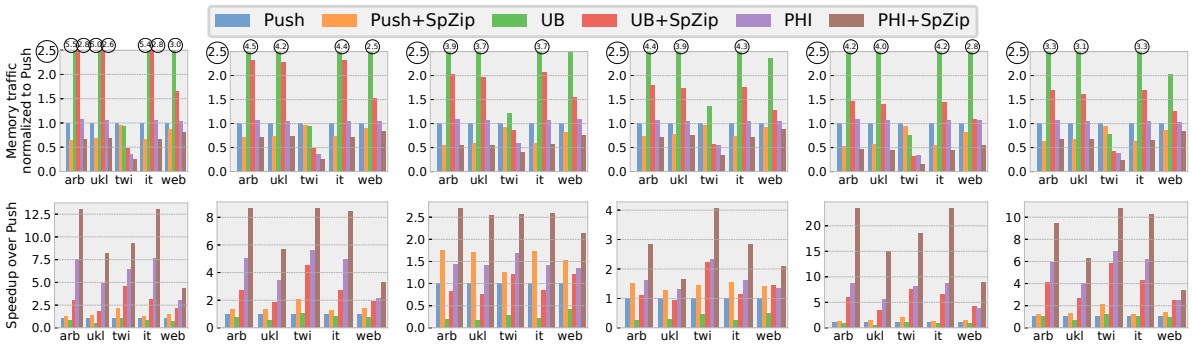
Fig. 15d shows that SpZip's compression now benefits all techniques. In Push, compression significantly reduces adjacency matrix size and traffic, by 2.3× on average, because neighbor sets now have similar ids. Thus, Push+SpZip reduces total memory traffic by 1.4× over Push. UB and PHI also enjoy the reductions in adjacency traffic, and in addition, see lower update and vertex data traffic. Overall, UB+SpZip reduces traffic by 2.1× over UB, and PHI+SpZip by 1.7× over PHI.

Results on individual inputs: Fig. 16 and Fig. 17 show results across all graph applications and inputs without and with DFS preprocessing. While speedups change per input, the above trends remain. First, PHI+SpZip is the fastest on all applications and inputs. Without/with preprocessing, PHI+SpZip achieves up to 17×/24× speedup (DC on arb) and up to 9.5×/6.2× traffic reduction (DC on twi) over Push. Second, UB+SpZip and PHI+SpZip yield consistent speedups and bandwidth savings over Push, UB, and PHI. Third, UB+SpZip is nearly as competitive as (and sometimes better than) PHI most of the time.



(a) PR (b) PRD (c) CC (d) RE (e) DC (f) BFS

Fig. 16. Per-input memory traffic (top) and performance (bottom) without preprocessing, normalized to Push.



(a) PR (b) PRD (c) CC (d) RE (e) DC (f) BFS

Fig. 17. Per-input memory traffic (top) and performance (bottom) with DFS preprocessing, normalized to Push.

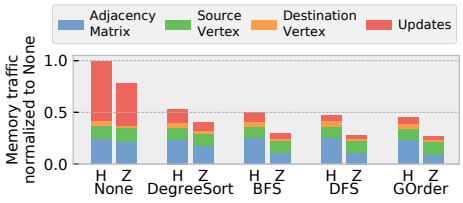
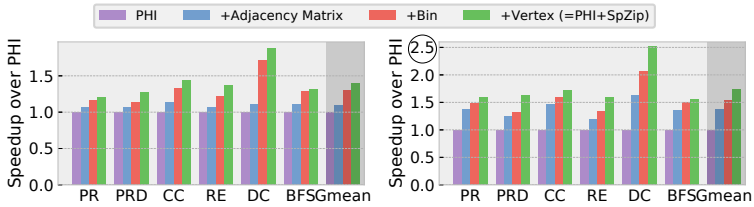


Fig. 18. Memory traffic breakdown of `uk-2005` with different preprocessing algorithms. H:PHI, Z:PHI+SpZip. Normalized to PHI without preprocessing (None/H), averaged across all graph applications.



(a) Speedup factor analysis w/o preprocessing (b) Speedup factor analysis w/ preprocessing

Fig. 19. SpZip compression factor analysis over PHI. +Adjacency Matrix: enables adjacency matrix compression, +Bin: adds update compression, +Vertex: adds vertex compression. Averaged across all inputs.

Without preprocessing, trends are similar across inputs. Preprocessing benefits inputs differently: in some, preprocessing uncovers plentiful structure, and Push enjoys good locality. But other graphs, especially `twi`, have little community structure, and preprocessing is less effective: Push suffers significant destination vertex data misses, the adjacency matrix compresses less well, and UB and PHI are preferable because batching updates is a better tradeoff, especially with SpZip's compression.

B. *Impact of preprocessing technique*

We have so far used DFS preprocessing; we now study the effect of compression on preprocessing techniques. Fig. 18 shows traffic breakdowns (lower is better) of graph `uk-2005` for PHI and PHI+SpZip without preprocessing and with four preprocessing algorithms: degree sorting, BFS, DFS, and GOrder. The results are averaged across all six graph applications.

Degree sorting is widely used [10, 28, 29, 79], and without compression (PHI alone), it achieves similar memory traffic to other preprocessing techniques. But with compression, degree sorting has a 1.3× gain, whereas the benefits of BFS, DFS, and GOrder increase to about 1.7×. This is because these algorithms improve the compression ratio of the adjacency matrix, by 2.3× for BFS, and 2.4× for DFS and GOrder, vs. only 1.4× for DegreeSort. This happens because topological sorting and GOrder place highly connected vertices nearby, improving value locality in neighbor sets: many neighbor ids have similar values, easing compression. Since DFS and DegreeSort are similarly lightweight [70, Table 9], and DFS nearly matches heavyweight GOrder, we use DFS in other experiments.

C. *Sensitivity studies*

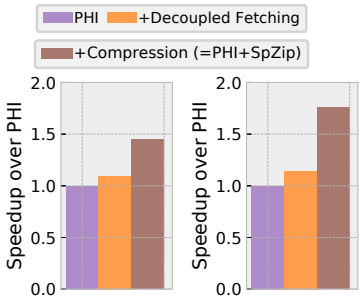
Impact of compression: Fig. 19 shows the impact of compressing different data structures. Each bar group reports performance for a single application, averaged across inputs like in Fig. 15. Within each group, the leftmost bar is PHI, and the other bars show how performance grows as more data structures are compressed (and handled by SpZip): first the adjacency ma-

trix, then the update bins, and finally vertex data. The rightmost bar, where all data structures are compressed, is PHI+SpZip.

As shown in Fig. 19, compressing each structure helps performance. Without preprocessing, compressing the bins helps the most overall, as they are the dominant contribution to traffic. With preprocessing, compressing the adjacency matrix helps the most overall, as it is the dominant contribution to traffic and preprocessing makes it compressible. Nonetheless, compressing vertex data also helps performance by reducing data movement, especially in all-active applications where both source and destination data are compressed, or in DC, where degree counts are small and highly compressible integers.

We also evaluate the impact of reordering order-insensitive data by running CC on UB. Across all inputs, sorting binned updates in UB improves their compression ratio from 1.26× to 1.55×. We see similar trends on other applications.

Decoupled fetching vs. compression: Fig. 20 compares the effect of SpZip's decoupled fetching and compression, showing that *compression is responsible for most of SpZip's benefits*. Each bar reports performance averaged across all applications and inputs. The middle bar (**+Decoupled Fetching**) uses SpZip without any compression. Decoupling improves PHI's performance by a modest 9% and 14% without and with preprocessing, respectively. By contrast, PHI+SpZip achieves speedups of 1.5× and 1.8×. Since the system is already memory bandwidth-bound, decoupling yields minor gains, and compression is needed to improve performance further.



(a) w/o preprocessing. (b) w/ preprocessing.
Fig. 20. Effect of decoupled fetching vs. compression.

Sensitivity to queue sizes: Fig. 21 shows the performance of PHI+SpZip on CC processing the uk-2005 graph with different fetcher scratchpad sizes: 1 KB, the default 2 KB, and 4 KB. Queues use the whole scratchpad in all cases. When going from 1 KB to 2 KB, performance improves by 2.6% (no preprocessing) and 10% (preprocessing), as deeper queues enable more decoupling. But there is negligible benefit from further decoupling: a 4 KB scratchpad has nearly the same performance. This shows that our default scratchpad size offers sufficient decoupling to hide latency, but does not incur needless overheads.

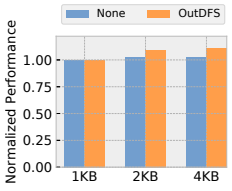
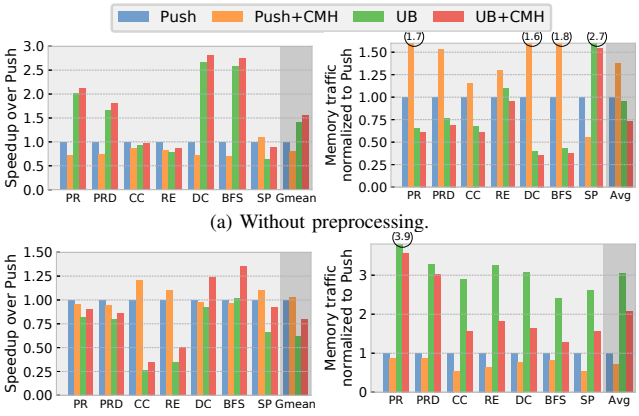


Fig. 21. Sensitivity to fetcher scratchpad size.

D. Benefits of compressed memory hierarchies

Fig. 22 shows results for Push and UB on the same 16-core system with a compressed memory hierarchy (CMH), consisting of a compressed last-level cache and main memory. This system uses a VSC [7] LLC with 2× the tags and the Base-Delta-Immediate (BDI) [57] compression algorithm, and



(a) Without preprocessing.
(b) With DFS preprocessing.
Fig. 22. Speedup and memory traffic of a compressed memory hierarchy (CMH) over Push on a normal system, averaged across all inputs.

main memory is compressed using Linearly Compressed Pages (LCP) [56]. LCP can reduce bandwidth by fetching multiple compressed cache lines per DRAM access [56, Sec. 5.1]). Each bar in Fig. 22 compares the performance and memory traffic of an application over the baseline system without a compressed hierarchy using Push. Results are averaged across all inputs.

Without preprocessing, CMH yields no speedups on Push and accelerates UB by only 11%. Because Push is dominated by compression-unfriendly scatter updates (Fig. 15b), CMH does not reduce memory traffic and compression hurts access latency; by contrast, SpZip's prefetching boosts Push's performance. With preprocessing (which is more friendly to compression), CMH accelerates Push and UB by only 3% and 28% (vs. 1.5× and 4.2× in SpZip). Memory traffic reductions show that CMH achieves much smaller compression ratios than SpZip, since it does not exploit application semantics. For example, it compresses fixed-size lines, which causes larger deltas across neighbor sets. More importantly, these systems *are not decoupled*, so they are forced to use simple compression algorithms and layouts to reduce decompression overhead. For example, to support fast addressing, LCP compresses all lines in a page to the same size. Thus, a few incompressible lines in the page make LCP ineffective on the whole page.

By contrast, SpZip does not have these limitations, and improves both performance and memory traffic by tailoring compression to the access pattern and exploiting decoupling to hide decompression latency.

VI. Additional Related Work

We now discuss additional related work not covered so far.

Accelerators for irregular applications: Prior work proposes specialized accelerators for graph processing and sparse linear algebra using ASICs [3, 31, 33, 47, 52, 54, 58, 64, 65, 76, 77, 81] or FPGAs [22, 23, 34, 50, 51, 72]. Though we have prototyped SpZip on a multicore, it would also benefit these accelerators, in two ways. First, these accelerators do not compress data, so SpZip would reduce memory traffic. Second, these accelerators use fixed-function pipelines to fetch and write data, which limits the data structures and algorithms they support. SpZip's

programmability would enable them to support more formats and workloads.

Memory hierarchy optimizations for irregular applications: Prior work has proposed indirect prefetchers to handle indirections in irregular workloads [5, 6, 13, 73]. These prefetchers are limited in the access patterns they handle. Concurrently with SpZip, recent work has proposed programmable prefetchers [68, 75] that overcome this problem, but they do not reduce data movement, and only hide memory latency. As we have seen, irregular applications often saturate memory bandwidth, so improving performance requires reducing data movement.

HATS [44] is a specialized fetcher that performs locality-aware graph traversals to reduce data movement. HATS achieves some of the benefits of preprocessing by running bounded-depth DFS traversals. HATS and SpZip are complementary: SpZip's fetcher could be enhanced to perform locality-aware traversals, and HATS does not perform compression.

GRASP [29] reduces data movement with a combination of preprocessing and hardware support: it applies a variant of degree-sorting (DBG) to segregate high-degree vertices, and changes the replacement policy to prioritize them. OMEGA [2] applies the same approach to a hybrid memory hierarchy, pinning high-degree vertices to a scratchpad instead.

Graph compression: Prior work introduces compression algorithms tailored to the adjacency matrix. These software techniques mainly seek to reduce memory *capacity*, not bandwidth: since the adjacency matrix is the largest structure, compression enables in-memory processing of very large graphs. Ligra+ [62] compresses neighbor sets with simple techniques. But software decompression overheads limit its speedup to 14%. WebGraph [16, 17] achieves order-of-magnitude capacity savings, but makes graph operations many times slower. SpZip could adopt complex compression formats like WebGraph, though maximizing performance requires handling data beyond the adjacency matrix.

VII. Conclusion

We have presented SpZip, an architectural technique that makes data compression practical for irregular algorithms. SpZip targets the wide gap that exists between two extremes: conventional compression algorithms that work only on long sequential streams, and compressed memory hierarchies that support random accesses but are access-pattern-unaware. In the middle, there are many applications that access short but compressible data chunks, of which irregular applications are a key example. Since data accesses and (de)compression operations are naturally interleaved, SpZip accelerates the traversal, decompression, and compression of data structures. To achieve high performance, these activities run in a *decoupled* fashion, hiding both memory access and decompression latencies. To support a wide range of access patterns, SpZip is *programmable*, and uses a novel Dataflow Configuration Language to specify programs that traverse and generate compressed data. As a result, SpZip achieves large performance gains and data movement reductions on a wide set of irregular applications and optimized execution strategies.

Acknowledgments

We sincerely thank Maleen Abeydeera, Nithya Attaluri, Axel Feldmann, Anurag Mukkara, Quan Nguyen, Nikola Samardzic, Vivienne Sze, Po-An Tsai, Yannan Nellie Wu, Victor Ying, Guowei Zhang, and the anonymous reviewers for their helpful feedback. This work was supported in part by DARPA SDH under contract HR0011-18-3-0007 and by Semiconductor Research Corporation under contract 2020-AH-2985. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola *et al.*, "Data compression accelerator on IBM POWER9 and z15 processors: Industrial product," in *Proc. ISCA-47*, 2020.

[2] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco, "Heterogeneous memory subsystem for natural graph analytics," in *Proc. IISWC*, 2018.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. ISCA-42*, 2015.

[4] K. Aingaran, S. Jairath, and D. Lutz, "Software in silicon in the Oracle SPARC M7 processor," in *IEEE Hot Chips Symposium*, 2016.

[5] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proc. ICS'16*, 2016.

[6] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proc. ASPLOS-XXIII*, 2018.

[7] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proc. ISCA-31*, 2004.

[8] A. Arelakis, F. Dahlgren, and P. Stenstrom, "HyComp: A hybrid cache compression method for selection of data-type-specific compression methods," in *Proc. MICRO-48*, 2015.

[9] A. Arelakis and P. Stenstrom, "SC2: A statistical compression cache scheme," in *Proc. ISCA-41*, 2014.

[10] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *Proc. IISWC*, 2018.

[11] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM TACO*, 2017.

[12] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza, "Clustering a DAG for CAD databases," *IEEE TSE*, vol. 14, no. 11, 1988.

[13] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao *et al.*, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *Proc. HPCA-25*, 2019.

[14] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv:1508.03619 [cs.DC]*, 2015.

[15] S. Beamer, K. Asanovic, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *Proc. IPDPS*, 2017.

[16] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. WWW-13*, 2004.

[17] P. Boldi and S. Vigna, "The webgraph framework II: Codes for the world-wide web," in *Proc. DCC*, 2004.

[18] A. Brahmakshatriya, E. Furst, V. Ying, C. Hsu, C. Hong, M. Ruttenberg *et al.*, "Taming the Zoo: The unified GraphIt compiler framework for novel architectures," in *Proc. ISCA-48*, 2021.

[19] S. Chou, F. Kjolstad, and S. Amarasinghe, "Automatic generation of efficient sparse tensor format conversion routines," in *Proc. PLDI*, 2020.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. MIT press, 2009.

[21] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proc. ACM*, 1969.

[22] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph processing framework on FPGA—A case study of breadth-first search," in *Proc. FPGA-24*, 2016.

[23] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proc. FPGA-25*, 2017.

[24] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM TOMS*, vol. 38, no. 1, 2011.

[25] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proc. ISCA-2*, 1974.

[26] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proc. SPAA*, 2017.

[27] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *Proc. HPEC*, 2012.

[28] P. Faldu, J. Diamond, and B. Grot, "A closer look at lightweight graph reordering," in *Proc. IISWC*, 2019.

[29] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *Proc. HPCA-26*, 2020.

[30] E. G. Hallnor and S. K. Reinhardt, "A unified compressed memory hierarchy," in *Proc. HPCA-11*, 2005.

[31] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. MICRO-49*, 2016.

[32] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, 2014.

[33] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik *et al.*, "ExTensor: An accelerator for sparse tensor algebra," in *Proc. MICRO-52*, 2019.

[34] S. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "GraFBoost: Using accelerated flash storage for external graph analytics," in *Proc. ISCA-45*, 2018.

[35] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *Proc. ISCA-43*, 2016.

[36] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proc. PACT-25*, 2016.

[37] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "The Tensor Algebra Compiler," in *Proc. OOPSLA*, 2017.

[38] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proc. MICRO-46*, 2013.

[39] P. Kumar and H. H. Huang, "GraphOne: A data store for real-time analytics on evolving graphs," in *Proc. FAST*, 2019.

[40] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips, "SQRL: Hardware accelerator for collecting software data structures," in *Proc. PACT-23*, 2014.

[41] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. WWW-19*, 2010.

[42] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M. Kaashoek, "Exploiting two-case delivery for fast protected messaging," in *Proc. HPCA-4*, 1998.

[43] C. Magnien, M. Latapy, and M. Habib, "Fast computation of empirically tight bounds for the diameter of massive graphs," *JEA*, vol. 13, 2009.

[44] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proc. MICRO-51*, 2018.

[45] A. Mukkara, N. Beckmann, and D. Sanchez, "Cache-Guided Scheduling: Exploiting caches to maximize locality in graph processing," in *AGP'17*, 2017.

[46] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates," in *Proc. MICRO-52*, 2019.

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

[48] Nangate Inc., "The NanGate 45nm Open Cell Library," http://www.nangate.com/?page_id=2325, 2008.

[49] Q. M. Nguyen and D. Sanchez, "Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism," in *Proc. MICRO-53*, 2020.

[50] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe *et al.*, "GraphGen: An FPGA framework for vertex-centric graph computation," in *Proc. FCCM-22*, 2014.

[51] T. Oguntebi and K. Olukotun, "GraphOps: A dataflow library for graph analytics acceleration," in *Proc. FPGA-24*, 2016.

[52] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proc. ISCA-43*, 2016.

[53] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[54] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti *et al.*, "OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator," in *Proc. HPCA-24*, 2018.

[55] B. Panda and A. Seznec, "Dictionary sharing: An efficient cache compression scheme for compressed caches," in *Proc. MICRO-49*, 2016.

[56] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons *et al.*, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proc. MICRO-46*, 2013.

[57] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. PACT-21*, 2012.

[58] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das *et al.*, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for dnn training," in *Proc. HPCA-26*, 2020.

[59] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. ISCA-40*, 2013.

[60] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan *et al.*, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. SIGMOD*, 2014.

[61] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. PPoPP*, 2013.

[62] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with Ligra+," in *Proc. DCC*, 2015.

[63] J. E. Smith, "Decoupled access/execute computer architectures," in *Proc. ISCA-9*, 1982.

[64] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. HPCA-24*, 2018.

[65] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. MICRO-53*, 2020.

[66] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi *et al.*, "GraphMat: High performance graph analytics made productive," in *Proc. VLDB*, 2015.

[67] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks," *Synthesis Lectures on Comp. Arch.*, 2020.

[68] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis *et al.*, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *Proc. HPCA-27*, 2021.

[69] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM memory expansion technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, 2001.

[70] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proc. SIGMOD*, 2016.

[71] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proc. Austrochip-21*, 2013.

[72] Y. Yang, Z. Li, Y. Deng, Z. Liu, S. Yin, S. Wei, and L. Liu, "GraphABCD: Scaling out graph analytics with asynchronous block coordinate descent," in *Proc. ISCA-47*, 2020.

[73] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proc. MICRO-48*, 2015.

[74] P. Yuan, C. Xie, L. Liu, and H. Jin, "PathGraph: A path centric graph processing system," *IEEE TPDS*, 2016.

[75] Y. Yuan, Y. Wang, R. Wang, R. B. R. Chowhury, C. Tai, and N. S. Kim, "QEI: Query acceleration can be generic and efficient in the cloud," in *Proc. HPCA-27*, 2021.

[76] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication," in *Proc. ASPLOS-XXVI*, 2021.

[77] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen *et al.*, "GraphP: Reducing communication for PIM-based graph processing with efficient data partition," in *Proc. HPCA-24*, 2018.

[78] Y. Zhang, X. Liao, H. Jin, L. Gu, and B. B. Zhou, "FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping," *IEEE TKDE*, 2018.

[79] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. Amarasinghe, "Making caches work for graph analytics," *IEEE BigData*, 2017.

[80] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A high-performance graph DSL," in *Proc. OOPSLA*, 2018.

[81] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *Proc. HPCA-26*, 2020.

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