SparseCore: Stream ISA and Processor Specialization for Sparse Computation

Gengyu Rao
gengyura@usc.edu
University of Southern California
Los Angeles, CA, USA

Jingji Chen
jingjich@usc.edu
University of Southern California
Los Angeles, CA, USA

Jason Yik
jyik@usc.edu
University of Southern California
Los Angeles, CA, USA

Xuehai Qian
xuehai.qian@usc.edu
University of Southern California
Los Angeles, CA, USA

ABSTRACT

Computation on sparse data is becoming increasingly important for many applications. Recent sparse computation accelerators are designed for specific algorithm/application, making them inflexible with software optimizations. This paper proposes SparseCore, the first general-purpose processor extension for sparse computation that can flexibly accelerate complex code patterns and fast-evolving algorithms. We extend the instruction set architecture (ISA) to make stream or sparse vector first-class citizens, and develop efficient architectural components to support the stream ISA. The novel ISA extension intrinsically operates on streams, realizing both efficient data movement and computation. The simulation results show that SparseCore achieves significant speedups for sparse tensor computation and graph pattern computation.

CCS CONCEPTS

• **Computer systems organization → Architectures**.

KEYWORDS

Stream ISA; Sparse computation acceleration; Graph analytics; Deep learning

ACM Reference Format:
Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2022. SparseCore: Stream ISA and Processor Specialization for Sparse Computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507705

1 INTRODUCTION

Sparse computation on data where a large fraction of values are zeros is important in scientific computing [21, 23], machine learning [31, 51], graph analytics [14], and simulation [10], etc. Sparse tensor algebra and graph pattern mining (GPM) are are two major applications that perform frequent sparse computation. The essential computation is the *set operations, i.e., intersection, subtraction, and merge (union)*, on two sparse vectors. Currently, accelerators have been developed for *specific application domains on fixed algorithm or dataflow*. For sparse tensors, these operations can be naturally extracted from algorithms and effectively accelerated. Sparse matrix-sparse matrix multiplication (SPMSPM) has rich algorithmic diversity [75]. Prior works have developed accelerators based on inner-product [24], outer-product [50], and Gustavson's algorithm [75]. The different choices of dataflow determine the computation schedules with different tradeoffs. Some dataflows have asymptotically worse performance on certain inputs. With dataflow embedded in accelerator architecture, adopting one accelerator prevents the usage of other dataflows. Graph Pattern Mining (GPM) [18, 58] only received research interests recently. GPM tasks a graph G and pattern specification p as inputs, and returns all subgraphs of G that are *isomorphic* to p. Two graphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$ are isomorphic if and only if there exists an one-to-one mapping $f : V_0 \rightarrow V_1$ such that $(u, v) \in E_0 \iff (f(u), f(v)) \in E_1$. A subgraph of G isomorphic to p is called an *embedding*. GPM enables various important applications such as functional modules discovery [53, 63], biochemical structures mining [12, 43, 47] and anomaly detection [6, 28, 29, 48, 54, 68] and many others [17, 30, 57, 64, 67, 71, 76]. The key challenge of GPM is the need to enumerate a large number of subgraphs, e.g., with WikiVote, a small graph with merely 7k vertices, the number of vertex-induced 5-chain embeddings can reach 71 billion. An efficient method for finding the embeddings of p from G is *pattern enumeration*, which constructs the embeddings that are isomorphic to p. Graphs are typically stored in sparse representation, e.g., compressed sparse row (CSR), which keeps the neighbor list of a vertex as a sparse array. The key operation of pattern enumeration is the intersection between neighbor lists—the essential primitive to construct embeddings based on pattern. For example, for two connected vertices v_1 and v_2, performing the intersection of their neighbor lists could identify triangles (v_1, v_2, v), where v is the neighbor of both v_1 and v_2. Pattern enumeration algorithms can be expressed as nested loops, in which each level extends the current subgraphs with a new vertex.

Different from sparse tensor and graph computation, for GPM, the *intersections on edge lists are deeply embedded in the nested loops*,

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507705

making it difficult to reuse existing accelerators for other domains to accelerate GPM algorithms. This difficulty gives rise to the recent interests of designing specific GPM accelerators. TrieJax [27] and GRAMER [73] are two early GPM accelerators. Their performance is limited since they did not accelerate the state-of-the-art algorithms. We provide detailed analysis in Section 2.3. FlexMiner [70] is the state-of-the-art pattern-aware software/hardware co-designed GPM accelerator. While achieving impressive speedups, the design is not flexible, i.e., the skeleton of its embedding exploration algorithm is implemented in hardware and fixed, making it hard to apply new algorithmic optimizations without hardware modification. SISA [7] is a set-centric Processing-In-Memory (PIM) based GPM accelerator. The set operations can be expressed in the programming model and off-loaded to PIM for execution.

In this paper, rather than designing a new accelerator for a particular domain based on specific algorithms/dataflows, we aim to enhance the *flexibility* of sparse computation acceleration architecture. We define a sparse vector as a *stream*, which can be a *key* or *(key,value)* stream. We propose *SparseCore*, which extends the instruction set architecture (ISA) to make stream first-class citizens, and develop efficient architectural components to support the stream ISA. The novel ISA extension *intrinsically operates on streams*, realizing both efficient data movement and computation. It can be considered as a natural extension to the traditional instructions for ordinary scalar values. Our approach accelerates all kinds of sparse computation with a *unified* architecture, which can adopt to different and evolving algorithms with software modifications, instead of developing specialized hardware every time.

Good flexibility can make the architecture easily adapt to complex and fast evolving algorithms and optimizations, which is particularly the case for GPM. The general-purpose processor can execute *any code patterns*, while the stream ISA extension can accelerate the critical intersection operations in a *seamless* manner. In this sense, our approach is similar to the successful SIMD ISA extension to multimedia applications decades ago. In comparison, FlexMiner [70] replaces some intersection with cmap, which is used to perform connectivity checking, but forces the implementation to use the less general notion that may not benefit from new optimizations. For example, FlexMiner is unable to support a new optimization based on Inclusion-Exclusion Principle that can accelerate pattern counting by up to 1110× in GraphPi [65], while SparseCore can easily benefit from it by implementing the optimization in software.

Moreover, extending ISA—the interface between software and hardware—allows compiler to analyze *existing* source codes and generate instructions in stream ISA to accelerate sparse computations. For GPM, the additional compiler pass can be smoothly integrated with compilation-based systems, such as AutoMine [46], GraphZero [45], and GraphPi [65], preserving the simple user interface. Specifically, users provide the input graph and pattern specifications, and the compiler synthesizes algorithm implementations automatically.

SparseCore architecture considers both computation and memory efficiency. We introduce Stream Mapping Table (SMT) that records the mapping between stream ID and stream register, and tracks the dependency between streams. For computation efficiency, we introduce stream unit (SU) to efficiently perform set operations including intersection, subtraction, and merge. The computations on sparse values determined by the indices in two streams are performed by Stream Value Processing Unit (SVPU). To accelerate a unique GPM algorithm pattern, we introduce nested instructions, which are implemented inside the processor by a sequence of micro-ops. To enable stream data reuse, a scratchpad is associated with SU. To ensure efficient data movements, we propose a Stream Cache (S-Cache) that can effectively prefetch streams based on the known sequential stream access pattern.

We develop a GPM compiler to generate codes with stream ISA. The new instructions are transparent to programmers and the compiler takes unmodified GPM codes. The main challenge for code generation is stream management (similar to register allocation in traditional compilers). For tensor computation, we modified TACO [32], the state-of-the-art tensor algebra compiler, to generate the baseline and optimized implementations with stream instructions.

We implement SparseCore ISA and its architectural components on zSim [61]. We evaluate the architecture with (1) GPM applications, including seven patterns (triangle/three-chain/tailed-traingle counting, 3-motif mining, 4/5-clique counting, and FSM) on ten real graphs, and (2) sparse tensor computation, including matrix multiplication with three algorithms, tensor times vector(TTV), and tensor times matrix(TTM) on 11 matrices and two tensors. For GPM, SparseCore significantly outperforms InHouseAutomine on CPU by on average 13.5× and up to 64.4×. SparseCore also outperforms FlexMiner and TrieJax by on average 2.7×and 3651.2×, up to 14.8×and 43912.3×respectively. For tensor computation, our experiments show that SparseCore archives 6.9×, 1.88×, 2.78× , 4.49×, and 2.44× speedup for inner-product, outer-product, Gustavson's algorithm, TTM, and TTV respectively.

2 BACKGROUND

2.1 Sparse Tensor Computation

While the dense tensor computation can be efficiently accelerated by GPU and SIMD instructions, the acceleration of sparse tensor computation is an open problem. Motivated by various applications, it has received intensive research interests [24, 25, 50, 56, 75, 77].

An important kernel is sparse matrix-sparse matrix multiplication (SPMSPM) $A_{mk} * B_{kn} = C_{mn}$, which can be commonly implemented via three algorithms: inner-product, outer-product, and Gustavson's algorithm. The difference among the three is the order of loops that iterate three indices: inner-product loops in the order of m, n, and k; outer-product loops in the order of k, m, and Gustavson's algorithm loops in the order of m, k, and n. The inner-product can be seen as multiplication of vectors, thus it can be implemented by intersection. For Outer-product and Gustavson's algorithm, the computation results are merged into the result matrix C, so they can be implemented by merge. Recent accelerators are developed based on different algorithms with different dataflows.

2.2 GPM Methods and Optimizations

Figure 1 shows the GPM problem which finds the pattern (triangle) (a) in the input graph (b). Figure 1 (c) shows the memory

access and computation of pattern enumeration. From two connected vertices, their edge lists are accessed, followed by the *intersection* between them. In this example, each common neighbor forms a triangle embedding matching the pattern. During execution, edge list accesses are followed by the computation (intersection) that is much more complex than graph computation and cannot be efficiently performed in current processors. Specifically, starting from the first vertex ID of two edge lists, if they mismatch, the processor advances the pointer of one of the lists, checks the boundary, fetches the next vertex ID, and compares again. This code pattern contains branches and data dependencies in a tight loop, making it difficult to predict the branches and exploit instruction level parallelism.

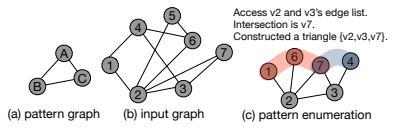

Figure 1: Pattern Enumeration

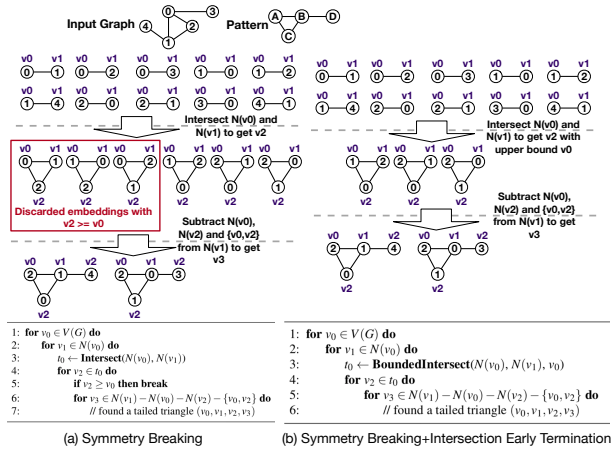

Figure 2: Tailed-triangle mining

Symmetry breaking in pattern enumeration avoids counting the same embedding for multiple times due to symmetry by enforcing a set of restrictions among vertices during embedding construction. A tailed-triangle mining example is shown in Figure 2. We denote the first/second/third/forth matched vertex of an embedding as v_0-v_3. Symmetry breaking requires $v_2 < v_0$ so that the a unique embedding is enumerated only once, i.e., $(v_0, v_1, v_2, v_3) = (2, 1, 0, 4)$ is the same as $(0, 1, 2, 4)$. As shown in (a), symmetry breaking first obtains all v_2 that is a common neighbor of v_0 and v_1 by intersecting $N(v_0)$ and $N(v_1)$, where $N(v)$ is the neighbor vertex set of v stored in edge list. Then, it discards all v_2 that are no less than v_0 to satisfy the restriction (line 5-6 of the algorithm (a)). This can be further improved by *early termination* of intersections since only the elements smaller than v_0 in $N(v_0) \cap N(v_1)$ are needed, indicated as `BoundedIntersect()` in (b). This optimization not only reduces computation and accessed data, but also eliminate branches in the next loop level. Due to the richness of the algorithm, the optimizations are fast evolving.

2.3 Existing Architectures on GPM

Pattern enumeration relies on various optimizations to achieve good performance, which make the codes very complex and cannot be efficiently executed on accelerators designed for sparse tensor [24, 25, 50, 56, 75, 77] and machine learning [16, 22, 52]. On the other side, it is also difficult to map the complex GPM algorithms on the more "general" specialized architectures such as Sparse Processing Unit (SPU) [13, 62, 69] which offers specialized supports for stream-join (similar to intersection) based on a systolic decomposable granularity reconfigurable architecture (DGRA).

SPU requires manually rewriting C codes and describing data flow graph (DFG) with the language extensions for DGRA. The computations are mapped to the systolic DGRA by analyzing the DFGs. Due to complex data and control flow dependencies, porting the same state-of-the-art GPM algorithms that execute on SparseCore to SPU leads to large DFGs. For example, four-motif needs up to 112 nodes in the DFG (48 computation nodes and 64 memory nodes), however, each SPU core can only support 20 computation nodes. Thus, it is infeasible to directly execute such large DFGs on an SPU core. To overcome the challenge, one solution is to partition the DFG into smaller partitions that an SPU core can accommodate, and distribute them to multiple cores for collaborative execution. Unfortunately, SPU has very limited mechanisms for cross-core collaboration, especially, poor synchronization support: the only way to synchronize SPU cores is via the "host" RISC-V core. Another possible solution is to execute GPM applications with multiple phases, each of which executes a DFG partition. Between different phases, the SPUs have to be reconfigured. To achieve good performance, the costly reconfiguration should not be performed frequently. Unfortunately, GPM applications switch between different parts of the DFG frequently, leading to extremely high overhead.

In summary, SPU's inefficiency for GPM applications is essentially due to its inflexibility to handle the complicated GPM codes with large DFGs. In comparison, SparseCore, as *a general-purpose processor extension*, is not restricted by the DFG-based programming model. It can execute the GPM codes in the similar way as the commodity processors do, except that the set operations are executed with ISA extension and special functional unit, i.e., Stream Unit (SU). In some sense, this approach resembles how SIMD instructions accelerate data-parallel computations.

Pipette [49] is a recent architecture that supports pipeline parallelism with ISA extension. Merge-intersect, as a stage in the pipeline, performs intersection between two streams. It is possible to port GPM applications on Pipetee, which requires less manual effort than SPU. However, programmers still need to express GPM algorithms in pipeline parallelism, leading to more significant code modifications than SparseCore. In comparison, SparseCore provides higher efficiency on unmodified codes with nested intersection and seamless integration of specialized acceleration components with CPUs.

TrieJax [27] is based on a variant of the Worst Case Optimal Join algorithm. Its performance is limited due to the lack of symmetry breaking support and blindly processing graph data as a database table. GRAMER [73] is based on a much slower pattern-oblivious algorithm with expensive isomorphic check. Its execution time after

speedup is even longer than directly executing pattern enumeration on commodity machines.

Based on more advanced algorithms, the software/hardware co-designed FlexMiner [70] achieves much better performance. Its software component analyzes the user-specified patterns, generates intermediate representations (IR) that contain the necessary information for embedding exploration (e.g., symmetry breaking restrictions), and passes them to a hardware embedding exploration engine. The downside is that, the implementation based on FlexMiner has to use the less general notion that may not benefit from new optimizations. SISA [7] is the first set-centric Processing-In-Memory (PIM) based GPM accelerator. It takes a similar approach by extending ISA but the execution of the set operations is performed on PIM. While SISA can effectively boost the performance of GPM and reduce data movements, it does not support sparse tensor computations, which not only require set operations on sparse vectors, but also perform floating point computations on values based on the outcome indices of set operations.

3 STREAM INSTRUCTION SET EXTENSION

3.1 Stream Definition

In general, we define a sparse vector as a *stream*, which can be: (1) a *key stream*—a list of *keys*, such as the edge list in graph representation; or (2) a *(key,value) stream*—a list of *(key,value)*, such as the pair of indices of non-zero elements and their values in a sparse tensor representation. We propose a novel instruction set extension that intrinsically operates on streams, supporting both data movement and computation.

3.2 Register Extension

The stream ISA extension represents stream as the first-class data type. The processor uses N *stream registers* to maintain stream information, where N is the maximum number of active streams supported. A stream is *active* between its initialization and free—each can be performed by an instruction. A stream register stores the stream ID, the stream length, the start key address, the start value address, stream priority, and a valid bit. The stream registers cannot be accessed by any instruction and are setup up when the corresponding stream is initialized. The program can reference a stream by the stream ID, the mapping between a stream ID and its stream register is managed internally in the processor with the *Stream Mapping Table (SMT)*. The key and value address of a stream register are only used by the processor to refer to the keys and values when the corresponding stream ID is referenced.

SparseCore also includes three auxiliary graph format registers (GFR_0, GFR_1, GFR_2) to support various graph representations. The content of these registers are interpreted based on the specific format. For simplicity, this paper considers compressed sparse row (CSR) [9], which has two arrays: (1) vertex array stores vertices sequentially with each entry pointing to the start of the vertex's edge list; and (2) edge array stores the edges of each vertex sequentially. In this case, GFR_0, GFR_1 and GFR_2 hold the CSR index, CSR edge list, and CSR offset, which can be loaded to GFRs by an instruction. CSR index and CSR edge list store the address of vertex array and edge array, respectively. The CSR offset stores the offset of the the smallest element larger than the vertex itself in the neighbor list. It is used to support the nested intersection, and the symmetric breaking optimization.

3.3 Instruction Set Specification

Table 1 lists the instruction set extension for streams. The instructions can be classified into three categories: (1) stream initialization and free; (2) stream computation; and (3) stream element access. The input operands for all instructions are general purpose registers containing stream ID.

`S_READ` and `S_VREAD` are the instructions to initialize a key stream and (key,value) stream, respectively. The operands are general purpose registers containing start key address (also start value address for `S_VREAD`), stream length, stream priority, and stream ID. After they are executed, if the stream ID is not active, an unused stream register (valid bit is 0) will be allocated to the stream and the new mapping entry is created and inserted into SMT. If the stream ID is already active, the previous mapping is overwritten with the current stream information. After creating the mapping to a stream register, both instruction will also trigger the fetching of key stream to the stream cache (see details in Section 4.3). Thus, if the current stream overwrites the previous one, the content in the stream cache will also be updated.

`S_VREAD` does not load the values, which will be triggered when the computation instruction for (key,value) stream (`V_VINTER`) is executed. The values are accessed and fetched through the ordinary memory hierarchy rather than the stream cache. `S_FREE` is used to free a stream. When it is executed, the processor finds the SMT entry for the stream ID indicated in the operand and set the valid bit to 0. If such entry is not found, an exception is raised.

The stream ISA contains nine instructions for *stream computation*. `S_INTER`, `S_INTER.C`, `S_SUB`, `S_SUB.C`, `S_MERGE`, `S_MERGE.C` perform simple computation on key stream—intersection and subtraction. The suffix "`.C`" indicates the variants of the corresponding instructions that do not output the result stream but just the *count* of non-zeros in the result stream. If the output is a stream, the stream ID of an initialized stream should be given in one of the input registers. The stream ID is then added into SMT. These instructions take a upper-bound operand `R3` to support early intersection/subtraction termination. Once all output stream elements smaller than `R3` have been produced, the instruction terminates the computation early. For unbounded operations, `R3` is set to -1.

`S_VINTER` performs user-defined intersected value computations. These instructions compute the intersection of the keys of the two input (key,value) streams, and then performs the *computation on the values* corresponding keys. For example, the key intersection of two (key,value) streams $[(1, 45), (3, 21), (7, 13)]$ and $[(2, 14), (5, 36), (7, 2)]$ is 7. The instruction performs the computation on the corresponding values: assuming the computation is multiply-accumulation (`MAC`) specified in `IMM`, the result is $13 \times 2 = 26$ in `R2`. The other types of computation can be specified in `IMM`, such as `MAX` (choose the maximum and accumulate), `MIN` (choose the minimum and accumulate), or any reduction operation.

`S_VMERGE` performs merged value computations and outputs a key value stream. It computes the merged keys, multiplies the corresponding value with a given scale, and adds multiplication results of the same key. For example, given two inputs streams $[(1, 4), (3, 21)]$

Table 1: Stream ISA Extension. R0-R4 are general-purpose registers, F0,F1 are FP registers, IMM is an immediate value.

Instruction	Description	Operands
`S_READ R0, R1, R2, R3`	Initialize a key stream	R0:start key address, R1:stream length, R2:stream ID, R3: priority
`S_VREAD R0, R1, R2, R3, R4`	Initialize a (key,value) stream	R0:start key address, R1:stream length, R2:stream ID, R3:start value address, R4: priority
`S_FREE R0`	De-allocate a stream	R0:stream ID
`S_FETCH R0, R1, R2`	Return one element of a key stream	R0:stream ID, R1:element offset, R2: returned element
`S_SUB R0, R1, R2, R3`	Subtraction of two streams, use stream of id R0 to subtract stream of id R1	R0,R1: input stream IDs, R2:output stream ID, R3: upper-bound of the subtracted result
`S_SUB.C R0, R1, R2, R3`	Return # of elements in subtraction of two streams, use stream of id R0 to subtract stream of id R1	R0,R1: input stream IDs, R2:returned result, R3: upper-bound of the subtracted result
`S_INTER R0, R1, R2, R3`	Intersection of two streams	R0,R1: input stream IDs, R2:output stream ID, R3: upper-bound of the intersected result
`S_INTER.C R0, R1, R2, R3`	Return # of elements in intersection of two streams	R0,R1: input stream IDs, R2: returned result, R3: upper-bound of the intersected result
`S_VINTER R0, R1, R2, IMM`	Sparse computation using the values of two (key,value) streams	R0,R1: input stream IDs, R2:returned result, IMM: specify user-defined op
`S_MERGE R0, R1, R2`	Merge of two streams	R0,R1: input stream IDs, R2:output stream ID
`S_MERGE.C R0, R1, R2`	Return # of elements in merge of two streams	R0,R1: input stream IDs, R2:output stream ID
`S_VMERGE F0, F1,R0, R1, R2`	Sparse computation with two (key,value) streams	F0,F1: multiplication scale, R0,R1: input stream IDs, R2:output stream ID
`S_LD_GFR R0, R1, R2`	Initialize GFRs based on graph representation	R0, R1, R2: content to be loaded into GFR_0, GFR_1, and GFR_2
`S_NESTINTER R0, R1`	Nested intersection	R0: stream ID, R1: returned result

and $[(1,1),(5,36)]$, the result keys would be $[1,3,5]$. Assume scales are 2 and 3, each value would be $[(4*2+1*3),(21*2),(36*3)]$. Thus the result stream would be$[(1,11),(3,42),(5,108)]$.

In SparseCore, computation on values is performed by a dedicated functional unit, which can be easily extended to perform new operations. The two instructions are useful in sparse tensor computation, where the keys indicate the positions of the non-zeros and the actual computations are performed on these values. If any input stream ID is not a (key,value) stream, an exception is raised.

`S_NESTINTER` performs the *nested intersection*. It is an instruction specialized for GPM. Let the input stream (an edge list) be $S = [s_0, s_1, ..., s_k]$, where each s_i corresponds to a vertex. Let us denote the edge list of each s_i as $S(s_i)$, and the result of the instruction as C. This instruction performs the following computation: $C = \sum_{i=0}^{i=k} count((S \cap S(s_i)))$, where $\cap$ is the intersection between two key streams, and *count* returns the length of a stream. The intersections are bounded by the value of s_i. Thus, this instruction implements *dependent stream intersection*. Given a stream S, the other streams to be intersected with it are determined by the keys (vertices) of S. The generation of the dependent streams corresponding to each s_i is performed by the processor using the information in the three GFR registers, which are loaded once using `S_LD_GFR` before processing a graph.

The `S_FETCH` instruction performs the stream element access—returning the element with a specific offset in a stream, which can be either the output stream of an intersection operation or an initialized stream loaded from memory. Typically, the offset is incremented to traverse all elements in a stream. When it reaches the end of the stream, `S_FETCH` will return a special "End Of Stream (EOS)" value.

In Table 1, we included all parameters needed by an instruction as operands for clarity. For some instructions, the total number of operands might be too large to be encoded in the instruction format of a given processor. It does not pose fundamental issue. In an implementation, we can include shared registers to hold the priority (`R3` of `S_READ` and `S_VREAD`) and scales (`F0` and `F0` of `S_VMERGE`), and remove them from the operands. We can introduce simple instructions to set these registers, which can be used before these instructions. This solution is feasible and correct since such information is obtained when the instructions are decoded, which happens in order. We do not include such details in the instruction specification to avoid diluting the essential ideas.

```
for (Vertex v0: graph) {
  Set n0 = v0.neighbours();
  // equivalent to
  // cnt += NestedIntersect(n0);
  for (Vertex v1: n0) {
    Set n1 = v1.neighbours();
    cnt += Intersect(n0, n1).size();
  }
}
```

```
… // for (v0: graph)
// R1-R4: start_addr, len, id, priority of n0
S_READ R1, R2, R3, R4
S_NESTINTER R3, R5
S_FREE R3
ADD R6, R5, R6 // cnt += NestedIntersect(n0);
```

(a) Nested Intersection (triangle)

```
for (Vertex v0: graph) {
  Set n0 = v0.neighbours();
  for (Vertex v1: n0) {
    Set n1 = v1.neighbours();
    Set t0 = BoundedIntersect(n0, n1, v0);
    for (Vertex v2: t0)
      {… // calculate N(v1)-N(v0)-N(v2)-{v0, v2} }
  }
}
```

```
… // for (v0: graph) for (v1: n0)
// R1-R4: start_addr, len, id, priority of n0
// R5-R8: start_addr, len, id, priority of n1
// R9: id of t0, R10: v0
S_READ R1, R2, R3, R4 // create the input streams
S_READ R5, R6, R7, R8
S_INTER R3, R7, R9, R10 // R10=v0 is the upper bound
S_FREE R3; S_FREE R7 // free the input streams
… // for (v2: t0) …
```

(b) Bounded Intersection (tailed-triangle)

Figure 3: Pattern enumeration with Stream ISA

```
while (true) {
  cmp = stream1[i1] - stream2[i2];
  if (cmp == 0) {
    i1++;i2++;
    output += value1[i1]*value2[i2];
  } else if (cmp < 0) {
    i1++;
  } else {
    i2++;
  }
  if (i1>boundary1 || i2>boundary2)
    break;
}
```

(a) Inner Product in C

```
…
//mov index addr, length, id, value addr, priority of stream 1 to R8-R12
S_VREAD R8, R9, R10, R11, R12
//mov index addr, length, id, value addr, priority of stream 2 to R8-R12
S_VREAD R8, R9, R10, R11, R12
//mov id of stream 1, stream 2 to R8-R9
S_VINTER R8, R9, R10, MAC
//mov id of stream 1 to R8
S_FREE R8
//mov id of stream 2 to R8
S_FREE R8
…
```

(b) Inner Product in Our ISA

```
while (true) {
  cmp = stream1[i1] - stream2[i2];
  if (cmp == 0) {
    out_v[idx] = v1[i1]*V1+v2[i2]*V2;
    out[idx] = stream1[i1];i1++;i2++;
  } else if (cmp < 0) {
    out[idx] = stream1[i1];
    out_v[idx] = v1[i1]*V1; i1++;
  } else {
    out[idx] = stream1[i2];
    out_v[idx] = v2[i2]*V2; i2++;
  }
  idx++;
  if (i1>bound1 || i2>bound2) {//tail
      stream = (i1>bound1)?stream2:stream1;
      v = (i1>bound1)?v2:v1;
      i = (i1>bound1)?i2:i1;
      copy(&out[idx],&stream[i],left);
      copy(&out_v[idx],&v[i],left);
    break;
  }
}
```

(c) Gustavson in C

```
…
//mov index addr, length, id, value addr, priority of stream 1 to R8-R12
S_VREAD R8, R9, R10, R11, R12
//mov index addr, length, id, value addr, priority of stream 2 to R8-R12
S_VREAD R8, R9, R10, R11, R12
//mov index addr, length, id, value addr, priority of stream 2 to R8-R12
S_VREAD R8, R9, R10, R11, R12
//mov id of stream 1, stream 2, result stream to R8-R10
//mov scale factor to F1 and F2
S_VMERGE F1, F2, R8, R9, R10
//mov id of stream 1 to R8
S_FREE R8
//mov id of stream 2 to R8
S_FREE R8
…
```

(d) Gustavson in Our ISA

Figure 4: Different SpMSpM Dataflows with Stream ISA

3.4 Code Examples

Figure 3 (a) shows triangle counting implementation using stream ISA. The v1 for-loop is essentially a nested intersection operation on `n0`, which can be implemented by `S_NESTINTER`. Figure 3 (b) shows

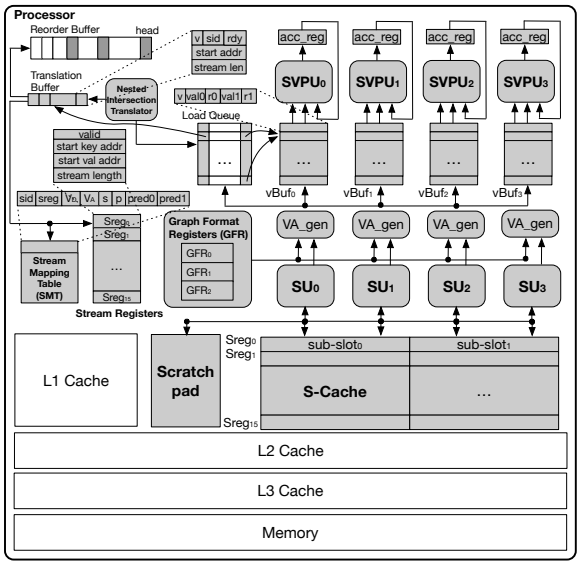

Figure 5: SparseCore Architecture

the implementation of tailed-triangle mining (shown in Figure 2 (b)) with intersection early termination. We use `S_INTER` to intersect the two edge lists with an upper-bound `v0` (stored in `R10`) so that the intersection can terminate early. Figure 4 (a)(b) show the inner product implementation with our stream ISA extension. Line 7 performs the multiply-accumulation on the values of the intersected keys. Figure 4 (c)(d) show the Gustavson implementation. Line 10 performs the merge computation. With SparseCore, the same architecture and ISA can flexibly implement different algorithms.

The ISA allows different loop iterations to use the same stream IDs, similar to the same variable names. The processor keeps track of the active streams in both front-end (after instruction decoding) and back-end (at instruction commit time), and will recognize the same stream IDs in different iterations as different streams.

4 SPARSECORE ARCHITECTURE

The SparseCore architecture is composed of specialized structures built on conventional processor architecture and memory hierarchy that implement the stream ISA extensions. Figure 5 shows a detailed overview with stream related components highlighted in gray color. All instructions in Table 1 except `S_NESTINTER` occupy one entry in the Reorder Buffer (ROB).

4.1 Stream ID Mapping

In SparseCore, each stream ID (*Sid*) specified in an instruction is mapped to an internal stream register (*Sreg*). This mapping is performed at the front-end after instruction decoding and the mapping relation is kept in SMT. Besides the stream ID and its mapped stream register, each SMT entry contains: (1) two valid bits: V_D, indicating the *define* point of the stream, and V_A, indicating whether the stream is *active*; (2) the *start (s)* and *produced (p)* bit, which indicate whether S-Cache contains the keys from the start of the stream and whether the data for the whole stream is produced (so that it can be used by the dependent streams); and (3) the *pred0* and *pred1*: the IDs of the streams that the current stream depends on.

Initially, both V_D and V_A are 0 and SMT is empty. Both V_D and V_A are set after decoding a `S_READ` or a `S_VREAD` instruction and the SMT entry indicates that the `Sid_i` in the last operand of the instruction is mapped to `Sreg_j`. Both V_D and V_A are set to one, they indicate that the instruction defines Sid_i and it is active. Later, when `S_FREE Sid_i` is decoded, the SMT is examined and an entry for `Sid_i` should be found (otherwise an exception is raised), and its V_D is reset, while V_A is unchanged. This means that `Sid_i` is no longer defined—the instructions after `S_FREE Sid_i` should not be able to reference `Sid_i`—but the stream is still active since `S_FREE Sid_i` has not been retired. When `S_FREE Sid_i` is retired, V_A is reset and the entry becomes free. When a new stream is mapped, the processor checks SMT and finds an entry with $V_A = 0$, which implies $V_D = 0$. Note that is not true vise versa—$V_D = 0$ does not imply $V_A = 0$.

Our design expects the codes to call `S_FREE` after a stream is no longer used, so that its SMT entry can be released. It can be easily ensured by the compiler. When all stream registers are occupied ($V_A = 1$), the instruction that initializes a new stream will be stalled. The current design with 16 stream registers is enough for all our applications. The larger (or even unlimited) number of stream IDs can be supported by virtualization: When a thread attempts to allocate too many streams, newer entries will be saved to a special memory region to release SMT space. The deadlock may happen when an intersection instruction is blocking the ROB and the related streams are swapped out. To avoid this scenario, we prioritize the streams used by the first intersection instruction in ROB and swap in its operand streams.

The design can naturally support the stream operations in loop iterations of GPM. Typically, inside an iteration, some streams are initialized and computations on them are performed before `S_FREE`s at the end of the iteration (refer to Figure 3 (c) for an example). Different iterations can use the same stream IDs, which are mapped to different SMT entries.

SMT does not increase the latency of CPU pipeline, and can be implemented in a pipelined manner similar to the register rename stage in CPU. The mapping from architecture registers to physical registers is similar to the mapping from *Sids* to *Sregs*, with the "readiness" of stream IDs.

4.2 Stream Unit and Stream Reuse

Stream Unit (SU) performs stream operations, we applied extensive optimizations to achieve high performance. Figure 6 illustrates the parallel comparison inside SU. For simplification, we only show 4 elements, which have all been loaded into the internal buffer of SU. We set the buffer size as 16 and use double buffer to avoid stall when one buffer is occupied by moving data into SU. For intersection, at Cycle 1, the first element in each stream will be loaded and compared with all the elements in the other stream. For stream A, its 3rd element is found to be equal to the first element of B, thus at the next cycle, the 3rd element

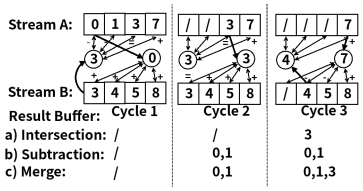

Figure 6: Parallel Comparison

of A will be loaded for parallel comparison. For stream B, the first element of A is less than any of its elements, so no action is needed. At Cycle 2, the 3rd element of A and the first element of B are found equal, so the element 3 will be put into the result buffer. Finally, at Cycle 3, each stream will use their next element for parallel comparison. The same optimization also applies to subtraction and merge. For subtraction and merge, the parallel comparison may generate multiple elements at one cycle, while only zero or one element can be produced per cycle for intersection. For example, at Cycle 1, the first two elements in stream A are less than the first element of B, thus they will be put into result buffer for subtraction and merge. We use a buffer to keep output stream, and write back when a full S-cache line of elements are generated.

In both GPM and tensor computation, a stream can be reused many times or immediately used by the following computation. It is either based on the algorithm or because such stream keeps the intermediate results. To avoid unnecessary data movements between S-Cache, a scratchpad shared among all SUs is used to store the stream with high stream priority. The stream priority (R3 of S_READ and S_VREAD) can be assigned by compiler after program analysis.

4.3 Stream Cache

In SparseCore, the keys for each active stream are loaded into a special *stream cache (S-Cache)*, which lies on top of L2 cache together with L1. The values in (key,value) stream are fetched through the normal memory hierarchy. When the stream keys are accessed using the stream instructions, the data will not pollute L1. Since the keys of a stream are accessed sequentially, the data can be effectively prefetched to S-Cache without a complex prefetcher, thanks to the *known* access pattern. Each stream register has a slot that holds a fixed number keys of the stream. We use the 64-key slot which leads to 256 byte slot size. With 16 stream registers, the total size of S-Cache is 4KB.

When an S_READ is executed, the first 64 keys are fetched to the S-Cache, and the *start* bit in SMT for the stream is set. Unless the length of the stream is no more than 64, at this point the S-Cache only contains the first portion of the stream. The *start* bit indicates that the instructions that depend on the stream can use the data in the S-Cache slot. Referring to Table 1, our ISA does not contain any instruction that explicitly stores to a stream: only S_INTER and S_SUB produce the results in the output stream. When these instructions are executed, the result keys are written to the S-Cache slot in group of 64. If the result stream contains more than 64 keys, the slot will contain *the most recently produced 64 keys* while the previous slot is written back to L2 and the *start* bit is cleared. When the whole result stream is generated by the computation instruction, the *produced* bit is set, which is used to trigger the dependent instructions.

The typical code pattern is that two streams are initialized by S_READ before the intersection operation is performed. In this case, data fetching from L2 to S-Cache and transfer to SUs for computation can be pipelined. To support that, we use the idea of double buffer and divide each slot into two sub-slots. When a sub-slot is fetched from L2, the keys in the other sub-slot can be prefetched to SU simultaneously and the intersection computation can be overlapped. Stream cache can send two cache line of data to two SUs at each cycle.

With multiple SUs, the parallel execution time of multiple intersections can be better overlapped with the data fetching time of these streams. When multiple SUs (4 in our design) need data to perform computations, S-Cache has to schedule the data transfer to different SUs. We use a simple round-robin policy: at each cycle, S-Cache schedules the transfer to a different SU that is waiting for the data. Each SU is able to perform the intersection on the partial key streams received.

4.4 Stream Data Dependency

Two streams may have dependency due to: (1) stream ID, where an instruction uses the output stream of a previous computation (S_INTER or S_SUB) as an input stream; or (2) the overlapped memory regions of two streams. It is easy to handle the first scenario: after the stream IDs are available after decoding, the dependency can be handled in the similar manner to the data dependency on general registers. When a dependency is identified, the consumer instruction can only execute after the producer instruction. It is enforced by filling the *pred0* and *pred1* in SMT of the consumer instruction. When the producer instruction finishes, its SMT entry's *produced* bit is set. Each cycle the processor checks the status of the producer instruction(s) and triggers the consumer instruction when all operands' *produced* bit are set. If the key stream produced is less than 64 keys, the whole stream is in S-Cache with the *start* bit set, the consumer instruction reads directly from S-Cache; otherwise, the slot will be refilled from L2.

For the second scenario, we can check the potential dependency conservatively by leveraging the fact that *the length of the output stream is less than the sum of the length of two input streams*. Thus, we can conservatively deduct the maximum length of the output stream. The possibly overlapped stream memory regions can be detected using the start key address and stream length of different streams. The dependent stream instructions need to be executed sequentially, which is enforced using the same mechanism as the first scenario.

4.5 Sparse Computation on Values

The sparse computation on values is suppored by the coordination between SU, value buffer (vBuf), load queue, and Stream Value Processing Unit (SVPU). When S_VINTER is executed, an SU starts with key intersection calculation and the output keys are given to the *Value Address Generator (VA_gen)* associated with the SU (refer to Figure 5). VA_gen generates the value addresses for each key in the intersection. These addresses are sent to load queue to request the values through the normal memory hierarchy, rather than S-Cache. Each value request is also allocated with an entry in the vBuf, which will collect the two values returned from the load queue (val0 and val1). Each entry has a ready bit (r) for each value, which is set when the load queue receives the value.

For S_VINTER, we assume that the operation is commutative (e.g., multiply-accumulate) thus the computation using val0 and val1 can be performed by SVPU as soon as both ready bits are set. We do not need to enforce any order on the accumulation.

The *acc_reg* is used to keep the accumulated partial results. While performing substantial amount of computations, this instruction only takes one entry in ROB. After the final result is produced in the acc_reg of the corresponding SU, it will be copied to the destination register, and then the instruction will retire from the processor when it reaches the head of ROB. The execution flow for S_VMERGE is similar, VA_gen and vBuf are also used for value requests. However, SU will perform merge rather than intersection. Also, instead of accumulate results, S_VMERGE will output each value.

4.6 Nested Intersection

For S_NESTINTER, we use the *Nested Instruction Translator* to generate the instruction sequence of other instructions of stream ISA to implement it. Based on the input key stream, the translator first generates the stream information based on each key element. The memory addresses of the streams information are calculated based on the GFR registers, then the memory requests are sent through load queue. For each stream, an entry is allocated in the *translation buffer*, its ready bit (rdy) is set when the stream information is returned at load queue. Similar to the pointer to vBuf entry, each load queue entry also keeps a pointer to the translation buffer entry. For each nested stream, three instructions are generated: S_READ, S_INTER.C, and S_FREE. An addition instruction is generated to accumulate the counts. Each instruction takes an entry in the translation buffer. The start address and stream length fields are only used in S_READ. When the stream information is ready, the three instructions are inserted into ROB. The translation is stalled when the translation buffer is full, which can be due to either ROB full or waiting for the stream information. In either case, the space will be released because eventually the instructions in ROB will retire and the requested data will be refilled. These events do not wait for the translation procedure and cause no deadlock.

5 IMPLEMENTATION AND SOFTWARE

5.1 Implementation Considerations

S_NESTINTER is translated into a variable length instruction sequence by the *Nested Instruction Translator* and takes multiple ROB entries. To ensure precise exception, the processor takes a checkpoint of registers before the instruction. If an exception is raised during the execution, processor rolls back to the checkpoint and raises the exception handler. It is similar to the mechanisms for atomic block execution [11, 55]. Besides information such as general registers, the checkpoint includes the content of SMT, stream registers and GFR registers.

In SparseCore architecture, stream cache does not participate in the coherence protocol. For the applications that SparseCore targets, the data (such as graph or sparse tensors) are read-only, thus there is no correctness issue. S-Cache itself is not read-only, many of our applications indeed write intermediate data to stream cache. For data synchronization, normal CPU instructions should not access stream data. An implementation raises an exception if this is violated. It ensures that S-Cache is only be accessed via S_FETCH.

5.2 Hardware Cost

We implemented the key components of SparseCore, including S-Cache, SU, SMT, and stream registers, using Chisel[2]. We used Synopsys Design Compiler with the Open-Cell 15nm design library[44] to synthesize the Verilog code generated by Chisel. These component can achieve a frequency of 4.35Ghz, indicating that our architectural extensions will not affect the latency of the baseline processor. We estimate area numbers of the SRAMs in our Scratchpad using CACTI[5]. We use the 22nm technology, which is the closest from 15nm available in CACTI. The total area of our S-Cache with 12 slots, 4 SUs, SMT, Scratchpad and Sregs takes $0.73mm^2$, while the area pf an Intel SkyLake server core (14nm) is close to $15mm^2$ [27].

5.3 Compiler

For GPM, we developed a compiler to generate GPM implementation with stream ISA. The compiler takes the user-specified patterns as input, synthesizes the corresponding intersection based GPM algorithms (e.g., those in Figure 2), and translates them to C++ implementations embedded with stream ISA assembly instructions. For tensor computation, we modified tensor algebra compilers TACO [32], which takes user-specified math expression as input and generates C++ implementations embedded with stream ISA instructions.

One major challenge is stream management during code generation (similar to register allocation in traditional compilers). To implement an intersection, the compiler may generate instructions that introduce up to three active streams—two input streams loaded by S_READ and one output stream produced by S_INTER. We release these created streams eagerly, since resources used to maintain actives streams (e.g. s-cache and stream registers) are limited. The streams created by S_READ are released by S_FREE after the intersection operation, and the compiler will insert S_FREE instructions to free the stream produced by S_INTER once it is no longer needed. If the number of actives streams reaches its limit (i.e., the number of stream registers), the compiler simply falls back to generate scalar ISA based intersection code, and print outs a warning message. In practice, we notice that such a "fall-back" scenario is rare (did not happen for all applications evaluated) thanks to our aggressive stream freeing strategy.

6 EVALUATION

6.1 Simulator and Configuration

We simulate SparseCore on zSim [61]. We implement all SparseCore architectural components into the simulator. Our configuration is listed in Table 2. For GPM, we compare SparseCore with the recent accelerators. For TrieJax [27], we implemented the partial join results (PJR) cache and simulated their cache hierarchy. The access patterns are analyzed and simulated according to the operational flow description. For Flexminer [70],

Table 2: Architecture Conf.

Number of cores	6
ROB size	128
loadQueue size	32
cache line size	64B
l1d cache size	32KB,8-way
L2	256KB,8-way
L3	12MB,16-way
S-Cache slot size	256B
scratchpad size	16KB

Table 4: Graph Datasets

name	#V	#E	avg D	max D
citeseer (C) [4, 20, 60]	3.3K	4.5K	1.39	99
email-eu-core (E)[40, 74]	1.0K	16.1K	25.4	345
soc-sign-bitcoinalpha (B) [1, 34, 35]	3.8K	24K	6.4	511
p2p-Gnutella08 (G) [41, 59]	6k	21k	3.3	97
socfb-Haverford76 (F) [60]	1.4K	60K	41.3	375
wiki-vote (W) [37, 38]	7k	104k	14.6	1065
mico (M) [19]	96.6K	1.1M	11.2	1359
com-youtube (Y) [72]	1.1M	3.0M	2.6	28754
patent (P) [39]	3.8M	16.5M	8.8	793
livejournal (L) [3, 42]	4.8M	42.9M	17.7	20333

we implemented the cmap and simulated their access patterns. For GRAMER [73], we implemented its specialized memory hierarchy and simulated the access patterns. For TrieJax, Flexminer, and GRAMER, we all assume full overlapping of any non-dependent data access.

6.2 Graph Mining Algorithms and Data Sets

We execute our InHouseAutomine to mine different patterns. It is because the codes of AutoMine [46] is not publicly available. For GPM, we choose several popular applications listed in Table 3 to evaluate SparseCore. They can be divided into four categories. (1) *Pattern counting* applications, which include triangle (T), three-chain (TC), and tailed-triangle counting (TT). We use T, 4C, and 5C to denote the implementations with nested intersection, while TS, 4CS, and 5CS refer to the corresponding implementations without this optimization. (2) ***k**-Motif mining*, which counts the embeddings of all connected patterns with a given size k. (3) ***k**-Clique mining*, which discovers all size-k complete subgraphs of the input graph. (4) *Frequent subgraph mining (FSM)*, which aims to discover all vertex-labeled frequent patterns. A pattern is considered as *frequent* if and only if its *support* is no less than a user-specified threshold.

Table 3: GPM Apps

Triangle counting (T)
Three chain counting (TC)
Tailed triangle counting (TT)
3-Motif (TM)
4-Clique (4C)
5-Clique (5C)
Frequent subgraph mining (FSM)

Similar to Peregrine [26], we choose the minimum image-based (MINI) support metric [8] and only discover frequent patterns with no more than three edges. It is important to note InHouseAutomine and our compiler for SparseCore implement the same algorithm. The only pass of SparseCore compiler in addition to InHouseAutomine compiler is to emit code using stream ISA extension.

Table 4 lists the real-world graphs we used from various domains, ranging from social network analysis to bioinformatics.

For sparse tensor computation, we implemented the three algorithms for SPMSPM, tensor times vector (TTV, $Z_{ij} = \sum_k A_{ijk}B_k$) and tensor times matrix (TTM, $Z_{ijk} = \sum_l A_{ijl}B_{kl}$). These applications make use of SparseCore's sparse value computation ability. We use state-of-the-art tensor algebra compiler TACO[32] to generate tensor kernel. We use the matrices and tensors listed in Table 5. We conduct more comprehensive evaluations of GPM applications than tensor computation since the general-purpose processor based design is motivated by the complex GPM code patterns.

Table 5: Matrix and tensor Datasets

Name	Dimensions	Nonzeros	Density
Circuit204 (C)[15]	1020×1020	5883	0.57%
Email-Eu-core(E)[40, 74]	1005×1005	25571	2.5%
Fpga_dcop_26 (F)[15]	1220×1220	5892	0.40%
Piston (P) [15]	2025×2025	100015	2.4%
Laser (L)[15]	3002×3002	5000	0.055%
Grid2 (G)[15]	3296×3296	6432	0.059%
Hydr1c (H) [15]	5308×5308	23752	0.084%
California (CA) [33, 36]	9664×9664	16150	0.017%
ex19 (EX) [15]	12005×12005	259577	0.18%
gridgena (GR) [15]	48962×48962	512084	0.021%
TSOPF (T) [15]	18696×18696	4396289	1.26%
Chicago Crime (Ch)[66]	$6.2k \times 24 \times 2.4k$	5.3M	1.46%
Uber Pickups (U) [66]	$4.3k \times 1.1k \times 1.7k$	3.3M	0.0385%

6.3 Overall Performance of GPM

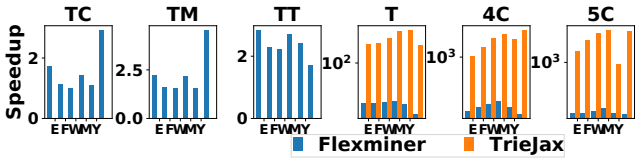

Figure 7: Speedup of SparseCore over Flexminer and TireJax

6.3.1 Comparison with Flexminer, TireJax, and GRAMER. To make the fair comparison, we only enable one computation unit in each accelerator and one SU in SparseCore. For Flexminer, the area of a PE is $0.18mm^2$ without the shared cache of 4MB. In comparison, the average area for each of SU in SparseCore, including scratchpad, S-Cache and all other added components, is $0.183mm^2$. For TrieJax, the total area of the architecture is $5.31mm^2$ for 32 internal threads. On average, each thread is $0.166mm^2$—very similar to area per SU for SparseCore. It is important to note Flexminer and SparseCore implement the same algorithm. We compare the performance of SparseCore, Flexminer, and TrieJax in Figure 7. Results related to TrieJax are shown in log scale. TrieJax can not support the Three chain, 3-Motif, and Tailed-triangle patterns, which are vertex-induced [70]. Intuitively, an edge-induced subgraph of G is formed by taking a subset of G's edges; while a vertex-induced subgraph of G is formed by taking a subset of G's vertices and *all* edges among them. The vertex/edge-induced pattern requires that the embeddings from the input graph be vertex/edge-induced subgraphs. Since TrieJax only supports join primitive, it can only support edge-induced patterns. We only evaluate clique counting, of which edge-induced and vertex-induced clique patterns happen to be the same.

On average, SparseCore outperforms TrieJax by 3651.2×. The performance gap is due to TrieJax's inferior algorithm design and the lack of graph structure support. TrieJax processes the graph data as a database table, which leads to unnecessary binary search and significant redundancy in GPM. For example, TrieJax lacks support for symmetry breaking, which means for Triangle and 4/5-Clique, they will count the same embedding for 6, 24, 120 times. Moreover, let us consider triangle counting, when extending from v_1 to v_2, TrieJax will try to find the edgelist of v_2 via its LUB unit, which would perform binary search on the table based on Worst Case Optimal Join. This operation may take up to $O(logN)$ time. However, for the CSR graph based approach, the same operation only takes $O(1)$ time.

Even though TrieJax has a partial join results (PJR) cache, we believe this design is inefficient and failed to exploit the access pattern of graph mining. PJR cache will deallocate large entries that exceed 1KB, which corresponds to only 256 vertices. However, for GPM applications, vertices with high degrees are more likely to be accessed[73], which usually cannot be placed in PJR. Hence, the PJR cache fail to cache the most frequently accessed data. For

example, the largest degree of email-eu-core, a small graph with merely 1K vertices, could be 345, which exceeds the capability of an PJR entry.

On average, SparseCore outperforms Flexminer by 2.7×. This speedup comes from the parallel comparison design inside SU, which provides the advantage in the basic stream computation. For GRAMER, since it is based on a pattern-oblivious algorithm with much more redundant computation. It is even slower than our baseline CPU benchmark. Based on our results, SparseCore outperforms GRAMER on average 40.1×and up to 181.8×.

6.3.2 Comparison with CPU. Further performance comparison among SparseCore (with/without nested intersection) and the CPU baseline are shown in Figure 8. TS, 4CS, and 5CS refer to the triangle counting, 4-Clique, and 5-Clique implementations without nested intersection. On average, enabling nested intersection speeds up these applications by 1.65×. It is because with nested intersection instructions, the normal instructions used to explicitly manage the corresponding loops, graph structure accesses, and embedding counting are eliminated. Nested intersection instructions allow more intersections to be executed on-the-fly simultaneously, thanks to the reduction of normal instructions that would have occupied more ROB entries. Besides, note that SparseCore achieves less speedup for FSM. It is because the support calculation in FSM is costly, and thus the intersection/subtraction operations that our architecture accelerates only take a smaller portion of execution time.

Comparing across different datasets, SparseCore achieves higher speedups on graphs with higher average degree. This could be explained by Amdahl's law. On graphs with higher degrees, the operand lengths of intersection/subtraction operations are generally longer. As a result, these operations are more computation-intensive and take up a larger portion of execution time. Recall that SparseCore only speedups intersection/subtraction operations, and thus achieves higher performance improvement on denser graphs. Also, a higher degree means the stream can be reused more often, leading to better utilization of the scratchpad and a significant reduction of access to the normal cache hierarchy.

6.4 Cycle Breakdown Analysis

We analyze the source of SparseCore's performance gain by analyzing the breakdown of the execution cycle for CPU and SparseCore as shown in Figure 9 and Figure 10. We can see from Figure 9 that branch misprediction cost takes a significant portion of the total cycles for CPU. This is due to the code pattern of intersections, which contains branches in a tight loop, making it difficult to predict the branches. As shown in Figure 10, the branch misprediction cost is significantly reduced for SparseCore, due to our specialized instructions. The *computation* categories are counted as the summation of cycles when any functional unit of the CPU is busy. This category can be further divided into two parts, *Other computation*, and *Intersection*. The *Intersection* represents the cycles when the CPU or Stream Unit is executing an intersection or subtraction operation. The *Other computation* category includes cycles for all other computations. It is worth noting that SparseCore can overlap *Other computation* with *Intersection*, since SparseCore is based on out-of-order CPU core. For SparseCore, the *Other computation* category takes a higher proportion of the reduced total cycles.

6.5 Comparing to GPU

We also compare SparseCore with GPU (Nvidia Tesla K40m). We assume the clock frequency of SparseCore to be 1Ghz. We compare the performance of SparseCore (with symmetry breaking) with two GPU implementations with or without symmetry breaking optimizations. The optimization in general adds more branches, and we want to study, with massive parallelism, whether the redundant enumeration with less branch divergence can overshadow less computation with more branches. Figure 11 shows the results. We can see that: 1) SparseCore outperforms the GPU implementations significantly, thus, even with a more powerful GPU, the results should stay the same; and 2) symmetry breaking is also effective in GPU, and the massive parallelism on more computation cannot overweight less computation with more branches. Using Nvidia profiling tools, we find that the reason for low performance of pattern enumeration on GPU is two-fold: 1) low warp utilization (about 4.4%) due to the branches and the different loop sizes (edge list length) for different threads; and 2) low global memory bandwidth utilization (about 13%) since threads access edge lists at different memory locations. Based on our results, it is no surprise that all existing pattern enumeration based graph mining system are based on CPU.

6.6 The Distribution of Stream Lengths

We further analyze the length distribution of involved streams in different GPM algorithms. Figure 14 left shows the cumulative distribution function (CDF) of stream lengths in different graph mining algorithms on the email-eu-core graph. Even on the same graph dataset, different applications could lead to different stream length distributions. We notice that clique applications (i.e., 4-Clique/5-Clique counting) in general introduce shorter stream lengths. The reason is that in clique applications, the input operands of intersection operations are usually the intersection results of other streams. And these operands tend to have shorter stream lengths.

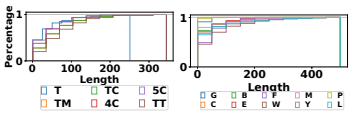

Figure 14: Length Distribution

We also fix the graph mining application to triangle counting and analyze the stream length distribution on various datasets. The results are reported in Figure 14 right. For this figure, we cut off the counting for stream larger than 500. The observation is intuitive– the longest stream length on datasets with larger maximal degrees (e.g., LiveJournal, Youtube) are longer. Besides, there are more long streams on denser datasets like E (email-eu-core) and F (socfb-Haverford76).

6.7 Varying the Number of Stream Unit

We characterize the performance of SparseCore by varying the number of SUs. Figure 12 shows the results with 1 to 16 stream units. When the number of SUs is no more than 4, increasing it will generally improve SparseCore's performance. However, with

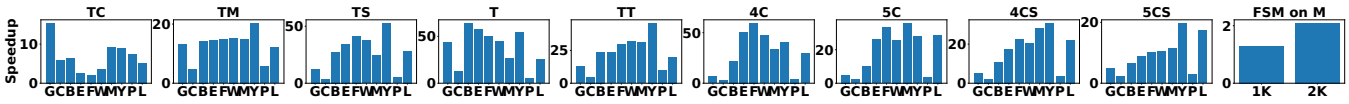

Figure 8: Speedups over CPU

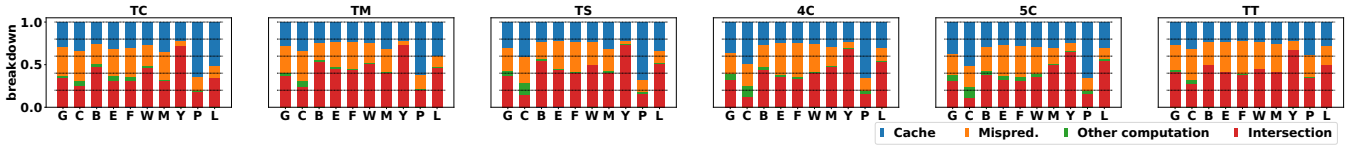

Figure 9: CPU execution breakdown

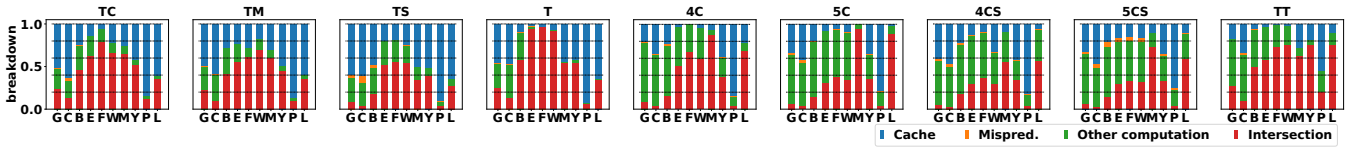

Figure 10: SparseCore execution breakdown

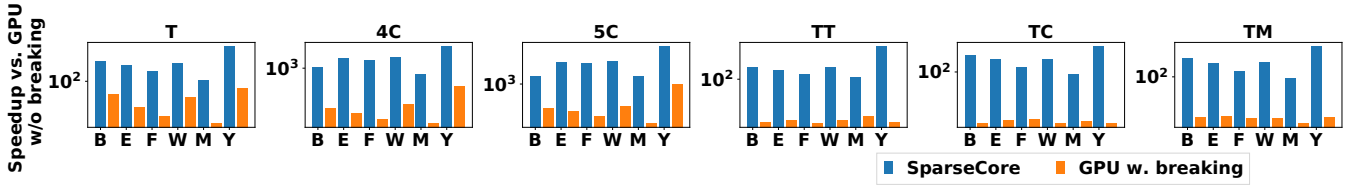

Figure 11: SparseCore compared to GPU implementations (log scale)

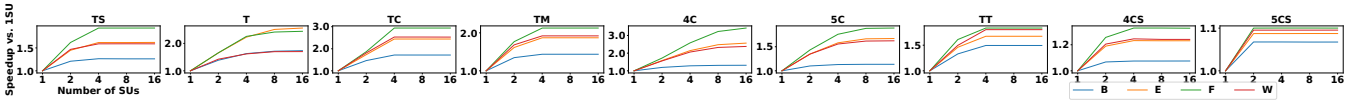

Figure 12: Varying the Number of SUs

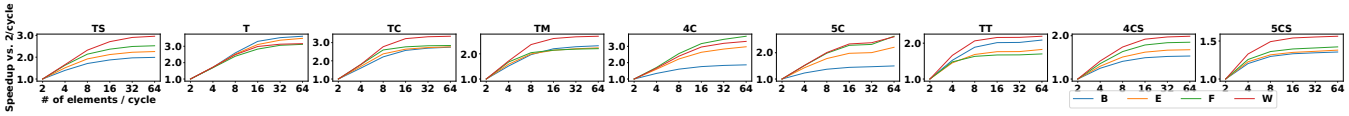

Figure 13: Varying S-Cache Bandwidth

more than 4 stream units, adding SUs introduces significantly less benefit.

6.8 Analysis on Bandwidth

We characterize SparseCore's performance with different bandwidth. Figure 13 shows the performance of SparseCore with aggregated S-Cache and Scratchpad bandwidth varying from 2 elements per cycle to 64 elements per cycle. Increasing aggregated bandwidth can improve SparseCore's performance. However, there is a point of diminishing return. For example, for the TC (three-chain counting) application, increasing the bandwidth from 32 to 64 elements/cycle introduces almost no benefit. It is because there are not enough concurrent active stream intersection/subtraction operations to saturate the bandwidth. The number of concurrent active stream operations is determined by the application and implementation. Triangle counting (T) and 4/5-Clique (4/5C) counting use the nested intersection instruction to trigger intersection operations in a bursty manner, leading to more simultaneously on-the-fly intersections. Hence, T/4C/5C benefits more from bandwidth increase than algorithm/implementation without the nested instruction (e.g., 4CS, 5CS). Moreover, each algorithm has a unique stream operation pattern, which leads to different number of simultaneously on-the-fly stream operations. Therefore, each algorithm would benefit from the bandwidth increase differently.

6.9 Tensor Computation Performance

6.9.1 Comparison with CPU. SparseCore's speedups against the CPU baseline are shown in Figure 15. For sparse matrix multiplication, it achieves on average 6.9×, 1.88×, and 2.78× speedup for inner-product, outer-product, and Gustavson's algorithm. Comparing across algorithms, Gustavson executes faster than the other two algorithm on CPU, e.g., 93.0× and 2.13× faster

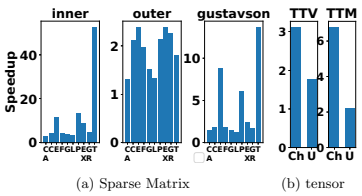

Figure 15: Tensor Computation Speedup

than inner- and outer-product on ex19. However, SparseCore archives the highest speedups for inner-product. It is because inner-product's data access pattern can be better accelerated by our data reuse. SparseCore achieves higher speedups for Gustavson's algorithm than outer-product for similar reason. After SparseCore's acceleration, Gustavson's algorithm still has the highest performance but the gap between inner-product becomes smaller, e.g., 24.9× and 2.13× faster than inner- and outer-product on ex19.

Comparing across different datasets, the speedup of TSOPF with inner-product and Gustavson's algorithm is much higher than the other matrices. This is because TSOPF has more non-zero elements per column, which leads to longer streams and more efficient data reuse. This is similar to our observations in Section 6.3.2, where SparseCore achieves higher speedup on graphs with a higher average degree. For tensor computation, SparseCore achieves on average 4.49× and 2.44× speedup for TTM and TTV respectively. Similar to matrices, for tensor with higher density, SparseCore can achieve higher speedup.

6.9.2 Comparison with OuterSPACE, ExTensor, and Gamma. Similar to the comparison with GPM accelerators, we only enable one computation unit in each accelerator and one SU in SparseCore. For OuterSPACE, as stated in their paper, the latency of allocation is typically hidden and they use scratchpad to hide the latency of grabbing new elements from the main memory. Thus, we mainly model their cache/scratchpad, their PE, and HMC transfer. For a fair comparison, we configured the latency of their cache/scratchpad to be the same as the latency of SparseCore's L1d cache. For ExTensor, we model their PE and the transfer cost of DRAM to LLB and Partial Output Buffer to DRAM with the same configuration in their paper. We model their PE with the same number of parallel comparators as SparseCore for a fair comparison. For Gamma, as stated in the paper, the FiberCache uses *fetch* to hide the memory access latency of a cache miss. For simplification, we model the FiberCache as "always hit". For the PE, we modeled it with one-element-per-cycle throughput. We compare the performance of SparseCore, OuterSPACE, ExTensor, and Gamma in Figure 16. We observe that that: 1) SparseCore with a better algorithm is faster than accelerators with worse algorithms, i.e., SparseCore with Gustavson's algorithm is faster than accelerators with outer-product, and inner-product; and 2) For each algorithm, the specialized accelerators are faster than SparseCore with 5.2× for inner-product, 3.1× for outer-product, and 2.4× for Gustavson's algorithm. This demonstrates the key trade-off between flexibility and performance: SparseCore can easily adapt to various algorithms with reasonable speedups—not significantly slower than the full specialized accelerators with the fixed dataflow. In particular, even if SparseCore's performance of Gustavson's algorithm is slower than Gamma, it is faster than OuterSPACE and ExTensor, thanks to the algorithmic advantages.

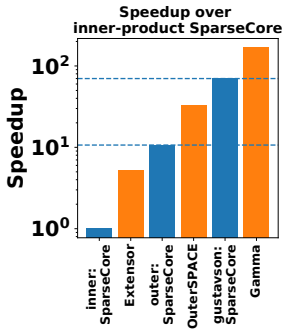

Figure 16: Gmean speedup of OuterSPACE, ExTensor, Gamma, and SparseCore with outer-product, and Gustavson's algorithm over SparseCore with inner-product

7 CONCLUSION

This paper proposes SparseCore, which extends the instruction set architecture (ISA) to make stream first-class citizens and accelerates sparse computation with a unified architecture. We develop the SparseCore architecture composed of specialized mechanisms that efficiently implement the stream ISA extensions. We implement the ISA extension and architecture on zSim [61]. The results show that SparseCore outperforms the recent more specialized GPM accelerators and achieves decent speedups on tensor computation.

ACKNOWLEDGEMENTS

We thank our shepherd Prof. Daniel Sanchez and the anonymous reviewers for their insightful comments and suggestions. This work is supported by National Science Foundation (Grant No. CCF-2127543, CCF-1750656, and CCF-1717754).

REFERENCES

[1] 2018. Bitcoin Alpha network dataset – KONECT. http://konect.cc/networks/soc-sign-bitcoinalpha
[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, 1212–1221.
[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 44–54.
[4] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2013. *Graph partitioning and graph clustering*. Vol. 588. American Mathematical Society Providence, RI.
[5] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.
[6] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 16–24.
[7] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, et al. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. *arXiv preprint arXiv:2104.07582* (2021).
[8] Björn Bringmann and Siegfried Nijssen. 2008. What is frequent in a single graph?. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 858–863.
[9] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.
[10] Andrew Canning, Giulia Galli, Francesco Mauri, Alessandro De Vita, and Roberto Car. 1996. O (n) tight-binding molecular dynamics on massively parallel computers: an orbital decomposition approach. *Computer Physics Communications* 94, 2-3 (1996), 89–102.
[11] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk disambiguation of speculative threads in multiprocessors. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 227–238.
[12] Young-Rae Cho and Aidong Zhang. 2009. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *IEEE Transactions on information technology in biomedicine* 14, 1 (2009), 30–36.
[13] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

924–939.
[14] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.
[15] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.
[16] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. GoSPA: An Energy-efficient High-performance Globally Optimized SParse Convolutional Neural Network Accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1110–1123.
[17] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. 2005. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Transactions on Knowledge and Data Engineering* 17, 8 (2005), 1036–1050.
[18] Alexandra Duma and Alexandru Topirceanu. 2014. A network motif based approach for classifying online social networks. In *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, 311–315.
[19] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment* 7, 7 (2014), 517–528.
[20] Robert Geisberger, Peter Sanders, and Dominik Schultes. 2008. Better approximation of betweenness centrality. In *2008 Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 90–100.
[21] John R Gilbert, Steve Reinhardt, and Viral B Shah. 2006. High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*. Springer, 260–269.
[22] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–165.
[23] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780.
[24] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.
[25] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher Fletcher. 2018. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 674–687.
[26] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[27] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. 2020. The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1217–1231.
[28] U Kang, Leman Akoglu, and Duen Horng Chau. 2014. Big graph mining for the web and social media: algorithms, anomaly detection, and applications. In *Proceedings of the 7th ACM international conference on Web search and data mining*. 677–678.
[29] U Kang, Leman Akoglu, and Duen Horng Polo Chau. 2013. Big graph mining: Algorithms, anomaly detection, and applications. *Proceedings of the ACM ASONAM* 13 (2013), 25–28.
[30] Hisashi Kashima, Hiroto Saigo, Masahiro Hattori, and Koji Tsuda. 2011. Graph kernels for chemoinformatics. In *Chemoinformatics and advanced machine learning perspectives: complex computational methods and collaborative techniques*. IGI Global, 1–15.
[31] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. 2019. Sparse deep neural network graph challenge. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[32] 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)*. 943–948. https://doi.org/10.1109/ASE.2017.8115709
[33] Jon M Kleinberg. 1998. Authoritative sources in a hyperlinked environment. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*. 668–677.
[34] Srijan Kumar, Francesca Spezzano, V. S. Subrahmanian, and Christos Faloutsos. 2016. Edge Weight Prediction in Weighted Signed Networks. In *Proc. Int. Conf. Data Min.* 221–230.
[35] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173
[36] Amy N Langville and Carl D Meyer. 2006. A reordering for the PageRank problem. *SIAM Journal on Scientific Computing* 27, 6 (2006), 2112–2120.
[37] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*. 641–650.
[38] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1361–1370.
[39] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 177–187.
[40] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 2–es.
[41] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 2–es.
[42] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.
[43] Avi Ma'ayan. 2009. Insights into the organization of biochemical regulatory networks using graph theory analyses. *Journal of Biological Chemistry* 284, 9 (2009), 5451–5455.
[44] Mayler Martins, Jody Maick Matos, Renato P Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. Open cell library in 15nm FreePDK technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design*. 171–178.
[45] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 21–37.
[46] Daniel Mawhirter and Bo Wu. 2019. AutoMine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.
[47] Tijana Milenković and Nataša Pržulj. 2008. Uncovering biological network function via graphlet degree signatures. *Cancer informatics* 6 (2008), CIN–S680.
[48] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 215–224.
[49] Quan M Nguyen and Daniel Sanchez. 2020. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 596–608.
[50] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.
[51] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, A. Puglielli, Rangharajan Venkatesan, B. Khailany, J. Emer, S. Keckler, and W. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), 27–40.
[52] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 27–40.
[53] Srinivasan Parthasarathy, Shirish Tatikonda, and Duygu Ucar. 2010. A survey of graph mining techniques for biological datasets. In *Managing and mining graph data*. Springer, 547–580.
[54] Pallabi Parveen, Jonathan Evans, Bhavani Thuraisingham, Kevin W Hamlen, and Latifur Khan. 2011. Insider threat detection using stream mining and graph mining. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*. IEEE, 1102–1110.
[55] Xuehai Qian, Benjamin Sahelices, and Josep Torrellas. 2014. OmniOrder: Directory-based conflict serialization of transactions. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 421–432.
[56] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.
[57] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. 2005. Graph kernels for chemical informatics. *Neural networks* 18, 8 (2005), 1093–1110.

[58] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. 2019. A Survey on Subgraph Counting: Concepts, Algorithms and Applications to Network Motifs and Graphlets. *arXiv preprint arXiv:1910.13011* (2019).

[59] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. 2002. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *arXiv preprint cs/0209028* (2002).

[60] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. http://networkrepository.com

[61] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[62] Karthikeyan Sankaralingam, Anthony Nowatzki, Vinay Gangadhar, Preyas Shah, and Newsha Ardalani. 2019. Systems and methods for stream-dataflow acceleration. US Patent App. 16/384,819.

[63] Matthew C Schmidt, Andrea M Rocha, Kanchana Padmanabhan, Zhengzhang Chen, Kathleen Scott, James R Mihelcic, and Nagiza F Samatova. 2011. Efficient α, β-motif finder for identification of phenotype-related functional modules. *BMC bioinformatics* 12, 1 (2011), 1–15.

[64] Daron R Shaw. 1999. The methods behind the madness: Presidential electoral college strategies, 1988-1996. *The Journal of Politics* 61, 4 (1999), 893–913.

[65] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 1418–1431.

[66] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. http://frostt.io/

[67] Shahadat Uddin, Liaquat Hossain, et al. 2013. Dyad and triad census analysis of crisis communication network. *Social Networking* 2, 01 (2013), 32.

[68] Ya-Nan Wang, Jian Wang, Xiaoshi Fan, and Yafei Song. 2020. Network Traffic Anomaly Detection Algorithm Based on Intuitionistic Fuzzy Time Series Graph Mining. *IEEE Access* 8 (2020), 63381–63389.

[69] Zhengrong Wang and Tony Nowatzki. 2019. Stream-based memory access specialization for general purpose processors. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 736–749.

[70] Chen Xuhao, Huang Tianhao, Xu Shuotao, Bourgeat Thomas, Chung Chanwoo, and Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[71] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: A frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 335–346.

[72] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *arXiv preprint arXiv:1205.6233* (2012).

[73] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. 2020. A Locality-Aware Energy-Efficient Accelerator for Graph Mining Applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 895–907.

[74] Hao Yin, Austin R Benson, Jure Leskovec, and David F Gleich. 2017. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 555–564.

[75] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

[76] Luming Zhang, Yahong Han, Yi Yang, Mingli Song, Shuicheng Yan, and Qi Tian. 2013. Discovering discriminative graphlets for aerial image categories recognition. *IEEE Transactions on Image Processing* 22, 12 (2013), 5071–5084.

[77] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.