ASA: Accelerating Sparse Accumulation in Column-wise SpGEMM

CHAO ZHANG, Lehigh University, USA
MAXIMILIAN BREMER, Lawrence Berkeley National Laboratory, USA
CY CHAN, Lawrence Berkeley National Laboratory, USA
JOHN SHALF, Lawrence Berkeley National Laboratory, USA
XIAOCHEN GUO, Lehigh University, USA

Sparse linear algebra is an important kernel in many different applications. Among various sparse general matrix-matrix multiplication (SpGEMM) algorithms, Gustavson's column-wise SpGEMM has good locality when reading input matrix and can be easily parallelized by distributing the computation of different columns of an output matrix to different processors. However, the sparse accumulation (SPA) step in column-wise SpGEMM, which merges partial sums from each of the multiplications by the row indices, is still a performance bottleneck. The state-of-the-art software implementation uses a hash table for partial sum search in the SPA, which makes SPA the largest contributor to the execution time of SpGEMM. There are three reasons that cause the SPA to become the bottleneck: 1) hash probing requires data-dependent branches that are difficult for a branch predictor to predict correctly; 2) the accumulation of partial sum is dependent on the results of the hash probing, which makes it difficult to hide the hash probing latency; and 3) hash collision requires time-consuming linear search and optimizations to reduce these collisions require an accurate estimation of the number of non-zeros in each column of the output matrix.

This work proposes ASA architecture to accelerate the SPA. ASA overcomes the challenges of SPA by 1) executing the partial sum search and accumulate with a single instruction through ISA extension to eliminate data-dependent branches in hash probing, 2) using a dedicated on-chip cache to perform the search and accumulation in a pipelined fashion, 3) relying on the parallel search capability of a set-associative cache to reduce search latency, and 4) delaying the merging of overflowed entries. As a result, ASA achieves an average of 2.25x and 5.05x speedup as compared to the state-of-the-art software implementation of a Markov clustering application and its SpGEMM kernel respectively. As compared to a state-of-the-art hashing accelerator design, ASA achieves an average of 1.95x speedup in the SpGEMM kernel.

CCS Concepts: • **Computer systems organization → Multicore architectures**.

Additional Key Words and Phrases: SpGEMM, Sparse accumulation, Sparse linear algebra, Markov clustering

ACM Reference Format:
Chao Zhang, Maximilian Bremer, Cy Chan, John Shalf, and Xiaochen Guo. 2021. ASA: Accelerating Sparse Accumulation in Column-wise SpGEMM. In . ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Graph analytics [8, 15, 17, 34, 49, 51] has emerged as one of the key computational methods to solve important problems with irregular structures that arise across a variety of scientific and engineering disciplines, including bioinformatics [6, 20, 33], social networks [5], and physical

ACM TACO, Vol. X, No. Y, 2021
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

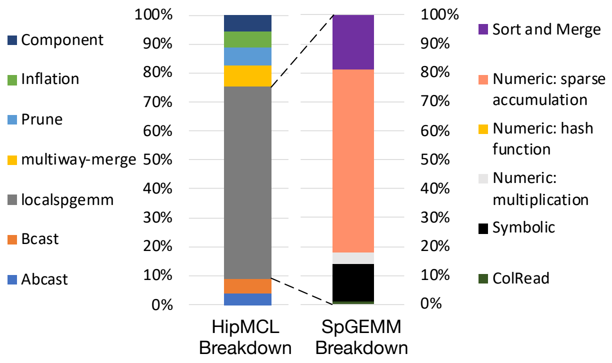


Fig. 1. Execution time breakdown of a High-performance Markov Clustering (HipMCL) [2] application with a protein bank [26] data input.

systems [3]. The graphs representing these problem spaces are typically large and sparse, which means that the connections among vertices are a small percentage (*i.e.,* typically less than 1% and hyper-sparse graphs have fewer connections than the number of vertices) of the total number of vertex pairs. For example, bioinformatics applications such as metagenome assembly [33] and protein clustering [28] work with sparse graphs of genetic and protein sequences that have 0.35% of non-zero connections [56]. General-purpose computer architectures that are optimized for dense computation and regular data access patterns struggle to attain high levels of computation throughput for graph analytic applications due to their innate data irregularity, which limits the capabilities to solve large and important problems in an affordable amount of time. As a result, there is a dire need to explore hardware acceleration for sparse graph analytic kernels.

To facilitate the optimization of these kernels in a way that can be applied across many domains, this work targets the GraphBLAS [12] specification, which recasts graph algorithms as sparse linear algebra operations. By developing optimized designs for these primitives, we can isolate changes to the GraphBLAS layer and use the accelerated functionality across multiple graph applications. The sparse general matrix-matrix multiplication (SpGEMM) is one of the most commonly used GraphBLAS kernels. This work focuses on accelerating SpGEMM and assesses the performance impact of the proposed design on HipMCL [2], which uses Markov clustering [53] to identify protein families. The HipMCL algorithm consists of an iterative loop, which updates cluster membership through an expansion, pruning, and inflation phase. As seen in Figure 1, expansion, which is represented as the local SpGEMM, is the most computationally expensive component.

The HipMCL library adopts Gustavson's column-wise SpGEMM [22] implementation (Algorithm 1), which multiplies non-zeros in columns of the second input matrix B with the columns of the first input matrix A and accumulates all of the partial sums through a sparse accumulation (SPA). The state-of-the-art software implementation of the column-wise SpGEMM (*e.g.,* GraphBLAS) uses a hash-based SPA with a symbolic-numeric method. The symbolic phase estimates the number of non-zeros in each output column and allocates a hash table for each column. In the subsequent numeric phase, partial sums are calculated by using the row index to look up the hash table to find the latest partial sums to add to. Before writing back the output column to the memory, all of the valid entries in the hash table are sorted by their row indices. As shown in Figure 1, the numeric phase takes the longest latency in the local SpGEMM computation, which is dominated by the hash-based SPAs. This is because linear probing is used when there are hash collisions that map indices to the same key. Processing hash lookups on general-purpose processors also suffer from hard-to-predict branches.

Prior work has proposed HTA [59] to accelerate common hash operations with ISA extensions. However, HTA is designed for general hash operations with a large memory footprint, whereas column-based SpGEMM can use matrix tiling to optimize for locality and to reduce hash table sizes. Using an HTA-like approach to accelerate the hashing operations in column-based SpGEMM would be overkill and cannot achieve the optimal efficiency. Moreover, accelerating hash operations alone cannot address SPA-specific computational challenges. Accelerators have been proposed for SpGEMM as well, in which hardware merger trees are used to sum up the multiplication results within a single pass [41, 58, 60]. In the merger design, the radix of the merger tree should be chosen carefully to balance between latency and area. A small-radix merger has to read the same input row multiple times when the merging factor exceed the radix, whereas a large-radix merger costs a large area overhead. For example, a merge tree in Sparch [60] costs more than 55% of the area and power.

This work proposes ASA, which is an in-core extension to a general-purpose processor for accelerating sparse accumulations in column-wise SpGEMM that maintains the generality of the multicore processors and adds minimum area overheads. The key contributions of the proposed ASA architecture are listed below:

- ASA extends the existing ISA to execute the partial sum search and accumulate with a single instruction, which improves the core utilization by eliminating hard-to-predict branches.
- ASA adds a small dedicated set-associative on-chip cache with an accumulator to hold partial sums and compute SPAs, which improves SPA throughput and reduces dynamic energy for cache lookups.
- ASA replaces hash linear probing with parallel search in the set-associative cache and delays merging of partial sum entries evicted from cache due to set conflicts.
- ASA provides a simple software interface to allow flexible use of the ASA hardware and easy integration with other software optimization of merging and sorting.

2 BACKGROUND

This section summarizes SpGEMM variants and existing works for sparse accumulation acceleration.

Sparse Matrix Matrix Multiplication (SpGEMM) is an important kernel in many applications, such as machine learning [21, 30, 54], numerical analysis [14], graph algorithms [45], *etc.* The broad use of the SpGEMM in data-intensive applications leads to many different parallel SpGEMM implementations.

An **inner product** implementation computes SpGEMM using a series of dot product operations between rows of the first matrix (A) and columns of the second matrix (B) for each element of the result matrix (C):

$$C_{[i,j]} = \sum_{k=0}^{N-1} A_{[i,k]} \times B_{[k,j]},$$

in which N is the matrix dimension, i and j are the row and column indices. Inner product SpGEMM has a good locality for matrix C and can be easily parallelized by sending different rows and columns to different cores without synchronization overhead. However, to select the non-zero elements from matrices A and B, it requires index matching before multiplication. The sparse storage format of A and B requires indirect memory accesses to load $B_{[k,j]}$ for each non-zero $A_{[i,k]}$. These dependent loads have poor spatial locality and are on the critical path of the computation, which can cause processor stalls and low core utilization even with an ideal tiling optimization [55].

An **outer product** implementation [9] multiplies matrix A and B by decomposing the operation into outer product multiplications of pairs of columns of A and rows of B:

$$C = \sum_{i=0}^{N-1} C_i = \sum_{i=0}^{N-1} A_{[:,i]} \times B_{[i,:]},$$

where $A_{[:,i]}$ is the i-th column of A and $B_{[i,:]}$ is the i-th row of B. C_i is the partial matrix of the final result matrix C. The computation is divided into two phases: 1) $A_{[:,i]} \times B_{[i,:]}$ multiplication and 2) partial matrix merging. For the multiplication phase, every non-zero element in $A_{[:,i]}$ is multiplied with every non-zero element in $B_{[i,:]}$. Hence, the accesses to both matrix A and matrix B have good spatial locality and have short reuse distance. However, the partial matrix merging phase requires high synchronization overhead to merge the partial matrix products that are assigned to different cores. Other outer-product approaches, such as PB-SpGEMM [19], avoid the synchronization by streaming the intermediate partial matrices to memory for merging later (expand, sort, compress), which may generate substantial memory traffic.

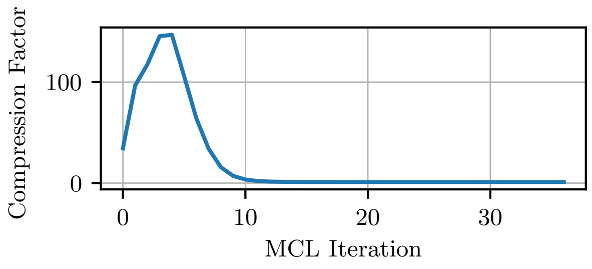


Fig. 2. Compression factor versus MCL iteration for eukarya network graph [2].

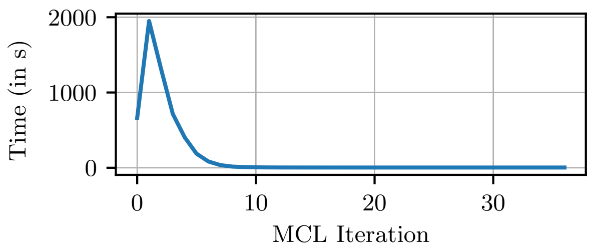


Fig. 3. Execution time versus MCL iteration for eukarya network graph [2]..

In **Gustavson's column-wise SpGEMM** [22] algorithm, columns of A are multiplied with the non-zeros of a column of B and the results are accumulated into a column of C using a sparse accumulator (SPA) [18].

$$C_{[:,j]} = \sum_{k=0}^{N-1} A_{[:,k]} \times B_{[k,j]}$$

$B_{[k,j]}$ is a non-zero element in a column of matrix B, $A_{[:,k]}$ is the corresponding columns in matrix A, $C_{[:,j]}$ is a output column of matrix C. In column-wise SpGEMM, different columns can be computed in parallel.

For SpGEMM, there is no single optimal formulation for all contexts, as the performance depends on the sparsity of the input matrices as well as the *compression factor*. Assuming the computation of $C = A \times B$ requires n_{flops} multiply-accumulate operations, and n_{nzc} equals the number of non-zeros in C, the compression factor is defined as n_{flops}/n_{nzc}, which corresponds to the average number of terms that must be summed up to produce a single non-zero entry of C. When the compression factor is low, the outer product formulation outperforms Gustavson's as the extra memory traffic incurred by splitting up the multiplication phase and the merging phase is relatively small [19]. But as the compression factor rises, the lower memory traffic of Gustavson's algorithm leads it to outperform the outer product based formulation.

Ultimately the SpGEMM implementation preference is application specific. In Figures 2 and 3, we show the average compression factor of the SpGEMM and wallclock time elapsed per MCL iteration for HipMCL on the Eukarya network with 32.4M vertices and 360M edges [2]. Comparing the two figures, 92% of the total execution time is spent in the first five MCL iterations, which consist of high compression factor SpGEMM multiplications, which favor the use of column-wise SpGEMM algorithms. Thus to maximize performance gains, this paper focuses on optimizing the performance of column-wise SpGEMM.

In SpGEMM, the pattern and number of non-zero elements of the output matrix is unknown before computation. But the memory allocation of the output matrix should be decided ahead of time. One way is to allocate large enough memory space, which might be inefficient. The other way is to use a symbolic-numeric method (Algorithm 1) [10, 39] to analyze output computation patterns, which is time-consuming. Alternatively, recently developed hash-based SpGEMM algorithm uses symbolic analysis for tiling and uses hash tables within each tile to record and lookup partial sums. The purpose of symbolic analysis is to precisely control the hash table size to reduce hash probing overhead. However, the hash operations are still the performance bottleneck due to high branch mis-prediction rate and poor spatial locality. As shown in the Figure 1, the numeric and symbolic phases dominate the execution time of the SpGEMM kernel. Therefore, in HipMCL [2] (High-performance Markov Clustering), more than 50% of the entire application runtime is devoted to computing SpGEMM hash operations.

Algorithm 1 shows the procedure of a column-wise SpGEMM. The bottleneck of this algorithm is the sparse accumulation at line 5-7 and 16-20 in Algorithm 1. Recent software implementations adopt many different data structures to do the accumulation, such as hash tables, vectorized hash tables, and heaps [2, 37, 44].

3 MOTIVATION AND KEY IDEAS

The proposed design is motivated by the computation challenges of the sparse accumulation in column-wise SpGEMM. The goal of this work is to overcome these challenges by designing a sparse accumulation accelerator that can be easily integrated into general-purpose multi-core architecture with minimum hardware overhead and a simple software interface. This section discusses the three key ideas of ASA to achieve this goal.

3.1 Extending ISA to Avoid Data-Dependent Branches

As discussed in Section 2, Hash probing is the bottleneck for both symbolic and numeric phases. One reason is that the core does not know whether the probing will hit, miss, or have a collision. When multiple keys are hashed to the same cell, this cell has collisions. A hash lookup typically compares keys that are mapped to the same cell one by one. The implementation of hash probing requires data-dependent branches, which are difficult to predict. Prior work [59] observed that mispredicted branches can be the performance bottleneck of many hash-intensive applications. To avoid these difficult-to-predict branches, this work proposes to extend an ISA with an hardware

Algorithm 1: Symbolic-Numeric SpGEMM.

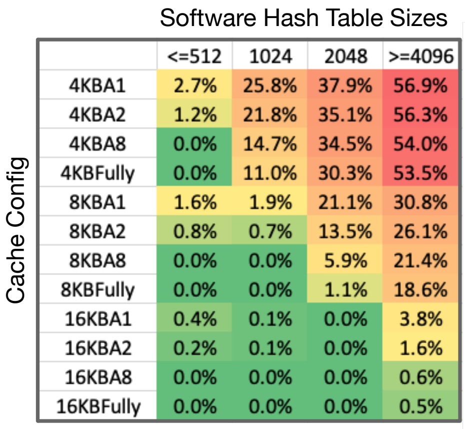
```
1  Procedure Symbolic(A,B):
2      for $B_{[k,j]}$ in $B_{[:,j]}$ do
3          for $A_{[i,k]}$ in $A_{[:,k]}$ do
4              $value = A_{[i,k]} \times B_{[k,j]}$
5              if $C_{[i,j]} \notin C_{[:,j]}$ then
6                  $nnzColC += 1$
7              end
8          end
9      end
10 return nnzColC
11
12 Procedure Numeric(A,B):
13     for $B_{[k,j]}$ in $B_{[:,j]}$ do
14         for $A_{[i,k]}$ in $A_{[:,k]}$ do
15             $value = A_{[i,k]} \times B_{[k,j]}$
16             if $C_{[i,j]} \in C_{[:,j]}$ then
17                 $C_{[i,j]} = C_{[i,j]} + value$
18             else
19                 insert $C_{[i,j]}$ into $C_{[:,j]}$
20             end
21         end
22     end
23 return tupleC
24
25 Procedure ColumnWiseSpGEMM(C,A,B):
26     for $B_{[:,j]}$ in matrix B do
27         $nnzColC = Symbolic(A, B)$
28         $AllocateHashTable(nnzColC)$
29         $Hashtable = Numeric(A, B)$
30         $tupleC = PairSort(Hashtable.begin(), Hashtable.end())$
31         $C += tupleC$
32     end
33 return C
```

probing and accumulation (HPA) instruction similar as other instruction extensions in [13, 43, 59]. As a result, lines 16-20 in Algorithm 1 can be consolidated into a single instruction, which helps to reduce the total instruction count, avoid branch misprediction penalty, and improve core utilization.

The collision resolution and overflow handling are performed by hardware and are hidden from the programmer. A programming interface is included in this design to provide key-value pairs to the sparse accumulator hardware, which will be discussed in Section 5.

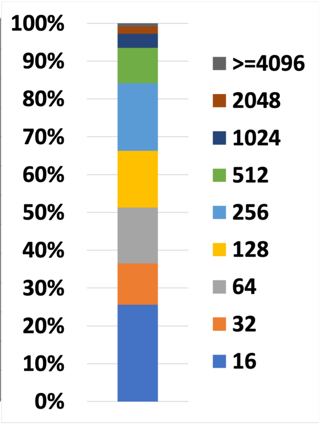
3.2 Dedicated Hardware for Probing and Accumulation

The ASA architecture adds a dedicated hardware cache to store partial sums and an accumulator per core to directly add the multiplication result to an intermediate partial sum with a matching key. The size of the cache should be small to allow fast lookup and minimize area overhead. In the symbolic-numeric method, the symbolic phase first identifies the total number of non-zeros


Software Hash Table Sizes
Cache Config

	<=512	1024	2048	>=4096
4KBA1	2.7%	25.8%	37.9%	56.9%
4KBA2	1.2%	21.8%	35.1%	56.3%
4KBA8	0.0%	14.7%	34.5%	54.0%
4KBFully	0.0%	11.0%	30.3%	53.5%
8KBA1	1.6%	1.9%	21.1%	30.8%
8KBA2	0.8%	0.7%	13.5%	26.1%
8KBA8	0.0%	0.0%	5.9%	21.4%
8KBFully	0.0%	0.0%	1.1%	18.6%
16KBA1	0.4%	0.1%	0.0%	3.8%
16KBA2	0.2%	0.1%	0.0%	1.6%
16KBA8	0.0%	0.0%	0.0%	0.6%
16KBFully	0.0%	0.0%	0.0%	0.5%

(a) Conflict rate with different hash table sizes and different cache configurations (4KBA2: 4KB, 2-way set-associative cache)



(b) Hash table sizes distribution

Fig. 4. (a) Cache conflict rate with different configurations and hash table sizes. ($ConflictRate = \frac{\#SetConflicts}{\#HWProbings}$) (b) Hash table sizes distribution. (HipMCL with a protein bank [26] data input)

in an output column of matrix C to allocate a hash table with an appropriate size. As a result, the software hash table size varies based on the number of non-zeros in the output column of matrix We adopted SUMMA [11], which is a distributed SpGEMM implementation, assigns each processor a submatrix of C and broadcasts input matrices A and B to different processors that can limit the size of the output matrix sizes. As a result, with a smaller output matrix to compute, the size for each hash table can be reduced. As shown in Figure 4 (b), more than 90% of the hash tables have fewer than 512 entries, which means they can fit into a small hardware cache. Instead of storing the entries into a private L1 cache, this work uses a dedicated on-chip cache with an accumulator for sparse accumulations (Partial Sum Cache). This design choice of using a cache smaller than the L1 cache reduces the energy of cache accesses. Having a dedicated cache and an accumulator also enables high-throughput hardware probing and accumulation via pipelining the cache lookup, addition, and write back.

Applying tiling algorithm [55] to the input and output matrices can help to fit the intermediate partial sums, which are stored in the hash table in the software implementation, into the partial sum cache. A set-associative cache is used to strike a balance between hardware complexity and set conflict rate. When set conflicts happen, the proposed design evicts a partial sum entry and handles these overflows later with a relatively small performance overhead. Figure 4 (a) shows the set conflict rates of different hash table sizes and under different cache configurations. An 8-way, 8KB cache with an 8B block size can accommodate more than 99% of the hash probing without set conflicts because most of the hash tables are smaller than 512.

3.3 Resolving Collisions in Hardware and Delaying Overflow Merging

A set-associative cache searches all of the tags (*i.e.,* hash keys) in parallel, which is an important reason to anticipate performance improvement when comparing ASA with the software hash probing that resolves collisions through linear search. The symbolic-numeric method in GraphBLAS [12] uses the symbolic phase to determine the hash table size to minimize collisions. It is helpful to allocate the hash table with an appropriate size for each output column, which saves space when the column is sparse and reduces collisions when the column is dense. The hardware cache has a fixed size and associativity. It is not necessary to use the symbolic phase to estimate the hash table size but it is helpful to apply a tiling algorithm. A fallback mechanism is essential to handle cache overflows due to set conflicts.

In the ASA architecture, a FIFO queue data structure is allocated through malloc function before the sparse accumulation. Evicted entries from the partial sum cache are inserted into the FIFO queue by using a hardware address generator to issue store requests. Using dedicated hardware to handle cache overflows avoids stalling the processor. After a partial sum entry is evicted from the cache, it will not be searched for the rest of the sparse accumulation. There could be multiple intermediate partial sums in the FIFO queue that have the key, which means they need to be added together to produce the final partial sums. An architecture register is added to keep track of the size of the FIFO queue. At the end of the sparse accumulation, the head and tail pointers of the FIFO queue are read by software and these overflowed entries are merged. By taking the merging of overflowed entries off the critical path of sparse accumulation, it is also to use the partial sum cache for one column while merging overflowed entries for another column. The detailed explanation of the overflow handling with FIFO queue is in Section 4.2.

3.4 Minimizing both software and hardware overhead

Multiple SpGEMM accelerators have been proposed recently [23, 38, 40, 58, 60], which can be used to execute limited applications. Serving multiple types of computational kernels in a single accelerator is challenging because different kernels prefer different system tradeoffs. An SpGEMM accelerators usually have a similar size of a CPU core. For example, SpArch [60] uses more than 55% of the area and power for building a merge tree to accumulate partial sums. To make the design cost-effective in terms of both performance and area, ASA can provide a competitive speedup with a lightweight software interface and negligible hardware area (less than 0.1% of the core area).

4 ASA ARCHITECTURE

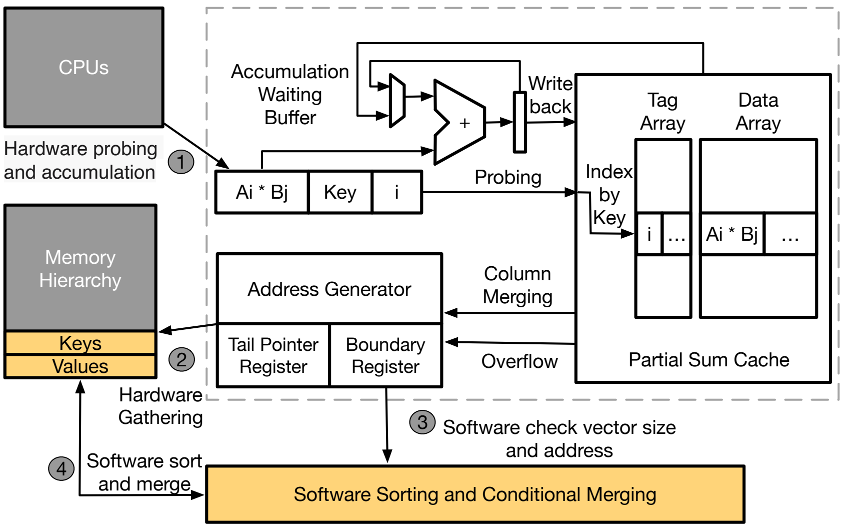


Fig. 5. An illustration of the ASA microarchitecture.

The proposed ASA architecture augments each core with an accumulation waiting buffer to store the multiplication results and its corresponding hash key, a floating-point adder, a small hardware cache to each core to store partial sums, and an address generator for both overflow handling and hardware gathering (Figure 5). The address generator has two architectural registers visible to the software: a tail pointer register recording the current tail position of the associated partial sum FIFO queue in memory, and a tail boundary register recording the boundary address of the allocated space.

4.1 Hardware Probing and Accumulation

A hardware probing and accumulation (HPA) instruction is similar to a store instruction, which has three source operands, which are the hash key for indexing cache sets, row index for tag comparison, and the multiplication result of a pair of non-zeros as the value. Similar to a store instruction an HPA instruction is issued from the load-store queue (LSQ) when the instruction is at the head of the reorder buffer and both operands are available. The hash key and the multiplication result of an issued HPA instruction will be stored in an accumulation waiting buffer to be added to a matching partial sum (❶ in Figure 5). The key is used to lookup the partial sum cache by first using the key to index to the corresponding cache set and then comparing the row index i with the stored tags. As shown in Figure 6, if the reference hits in the cache, the value of the matching entry is read and added with the multiplication result. If the reference misses in the cache, a new entry is allocated and the multiplication result is directly stored in the cache. If miss in the cache but the corresponding cache set is full, a cache overflow happens. An entry needs to be selected according to a replacement policy and evicted from the cache. A good choice of the replacement policy will help to prevent premature evictions. This work uses a least recently used (LRU) replacement policy.

Each HPA instruction takes three cycles to complete after being issued from the LSQ. One cycle for cache lookup, one cycle for accumulation, and another cycle for writing back to cache. In order to improve the throughput, a three-stage pipeline is implemented such that when one HPA instruction is computing accumulation, the following HPA issued back-to-back can lookup the cache. It is possible that the back-to-back HPAs have matching keys, hence the hash key also needs to compare with the keys of the previous outstanding HPAs. If the back-to-back HPAs have the same key, the previous accumulation result is forwarded to the input of the accumulator.

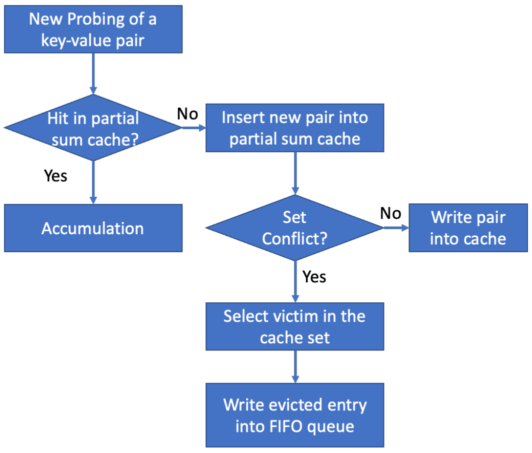

New Probing of a key-value pair
Hit in partial sum cache?
No
Insert new pair into partial sum cache
Yes
Accumulation
Set Conflict?
No
Write pair into cache
Yes
Select victim in the cache set
Write evicted entry into FIFO queue

Fig. 6. Hardware probing and accumulation with overflow handling.

4.2 Overflow Handling

A contiguous memory address space is pre-allocated to store both the overflowed key-value pairs of the partial sum entries evicted from the partial sum cache during the hardware probing and accumulation phase (❶ in Figure 5) as well as the writeback partial sum entries during the hardware gathering phase (❷ in Figure 5). The memory space pre-allocation is done in software in a FIFO queue data structure. During phase ❶ and ❷, the FIFO requires only insertion operations. An address generator is added to ASA to calculate the virtual addresses for these insertions to the FIFO queue, which is equipped with a tail pointer register storing the position of the next insertion and a tail boundary register storing the boundary address of the pre-allocated memory space. The virtual address range is sent to ASA when the memory space for the FIFO queue is pre-allocated. The tail pointer register is initialized to the start position (*i.e.,* head pointer) of the pre-allocated FIFO queue. Each eviction of the key-value pair will use the tail pointer value to calculate the addresses for the key and the value (Figure 6). ASA generates normal store instructions to write these entries into the FIFO queue through the memory hierarchy, which is the same as other store instructions from the load store queue. An evicted partial sum entry due to cache overflow can have a repeating key with another evicted entry or the same key as a writeback entry. In order to prevent stalling, the hardware overflow handling does not merge these entries with repeating keys at the eviction time. After each insertion to the FIFO queue, the tail pointer will increment and compare with the value of the tail boundary register. If the tail pointer equals to or exceeds the boundary address, the FIFO queue is full and an interrupt is triggered to allocate more space. After memory allocation, both the tail pointer and the tail boundary registers are reset accordingly for the newly allocated space.

4.3 Sorting and Merging

Elements in the output column need to be sorted. As shown in Figure 7, all valid entries in the partial sum cache are firstly added to the FIFO queue, which waits for a subsequent sorting operation. Implementing sorting logic in the hardware can be expensive, whereas the execution time of sorting non-zeros of the output column in software is relatively low for SpGEMM as compared to other operations [44]. Hence, this design keeps sorting in software. After the gathering phase ❷, all key-value pairs are written into FIFO queue in memory. A pair sort is used to sort all key-value pairs by its keys, which is the same as the original software implementation. The pair sort needs the start and end position of the unsorted FIFO queue. In phase ❸, the software reads the tail pointer register value before and after the hardware gathering. The sorted tupleC can be directly added to the output matrix C if there are no overflows during the column computation (phase ❹). If there are overflowed key-value pairs, additional merge operations can be performed to add the overflowed entries to the sorted array. In this case, entries with the same key would be merged first. After that, all key-value pairs would be sorted to the output tupleC, which takes additional O(N) of time complexity on top of the original software sorting with O(NlgN).

Overflow requires additional instructions, which can offset the benefits of using the proposed partial sum cache. The amount of the overflows can be well controlled if appropriate tiling algorithm is applied to the column-wise SpGEMM. This means breaking down the denser columns of the input matrix B into multiple smaller sub-columns.

4.4 Context Switch

Modern processors usually adopt Lazy FP State Save/Restore, which defers the save and restore of certain CPU context states on the task switch. Similarly, the content in the partial sum cache is part of the state that will be saved and restored lazily when the hardware resources are not required in a new context.

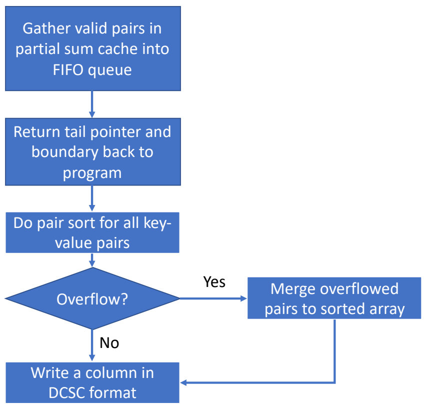

Gather valid pairs in partial sum cache into FIFO queue
Return tail pointer and boundary back to program
Do pair sort for all key-value pairs
Overflow?
Yes
Merge overflowed pairs to sorted array
No
Write a column in DCSC format

Fig. 7. Sort and merge flowchart.

5 PROGRAMMING INTERFACE

Table 1. ASA programming interfaces.

Names	Explanation
ASA.malloc(size)	Allocate space for the partial sum tuples and initialize ASA internal registers accordingly
ASA.insert(key,i,value)	Insert a pair of key and value into partial sum cache, accumulate if hit in the cache
ASA.gather()	Write all valid entries from partial sum cache into reserved address space
ASA.tail()	Return the tail pointer of the partial sum tuples
ASA.overflow()	Return whether there is any overflow
ASA.clear()	Invalidate all entries in the partial sum cache and clear the internal registers

The proposed design includes a simple programming interface to use ASA. Table 1 lists important procedure calls. Algorithm 2 shows an example implementation of SpGEMM using ASA. Compressed data format (e.g., DCSC [9]) is typically used for both the input and output matrices to reduce memory footprint by avoiding storing zeros. During the computation of each output column, the partial sums can be calculated in an order different from the index order. Hence, a series of key-value pairs are stored in a FIFO queue data structure (tupleC in Algorithm 2), where the key is the row index and the value is the partial sum. This FIFO queue is used when storing the intermediate partial sums as well as the final sorted non-zeros of an output column. The *size* of the memory allocation (line 2 in Algorithm 2) should be equal to or greater than the capacity of the partial sum cache. If more cache overflows are anticipated, a larger size can be pre-allocated. This parameter can be optimized for different matrices and partitions.

As discussed in Section 3, the symbolic phase can be removed when using ASA. In the numeric phase, the proposed design uses the FPU in the core for multiplication of $A_{[i,k]}$ and $B_{[k,j]}$. We discussed the design choice of not offloading the multiplication and hash key calculation in Section 7.2. Line 16-20 in Algorithm 1 now can be replaced with a simpler $ASA.insert(key, i, value)$ function at line 7 in Algorithm 2. The key is the hash value calculated by applying the hash function to the

Algorithm 2: An Example SpGEMM using ASA.

```
1  Procedure Numeric(A,B):
2      tupleC = ASA.malloc(size);
3      for $B_{[k,j]}$ in $B_{[:,j]}$ do
4          for $A_{[i,k]}$ in $A_{[:,k]}$ do
5              $value = A_{[i,k]} \times B_{[k,j]}$;
6              key = hash(i);
7              ASA.insert(key,i,value);
8          end
9      end
10 return tupleC
11
12 Procedure ColumnWiseSpGEMM(C,A,B):
13     for $B_{[:,j]}$ in matrix B do
14         $tupleC = Numeric(A, B)$;
15         sortStart = ASA.tail();
16         ASA.gather();
17         PairSort(sortStart, ASA.tail());
18         if ASA.overflow() then
19             AdditionalMerge(tupleC);
20         end
21         $C{+} = tupleC$;
22         tupleC.free();
23         ASA.clear();
24     end
25 return C
```

original row index i for $A_{[i,k]}$, which achieves better load balancing among cache sets than row indices does when used to index the hardware cache. $ASA.insert(key, i, value)$ will insert a pair of key and value to the partial sum cache (dedicated for sparse accumulation). If the key hits in the cache, it reads the current partial sum $C_{[i,j]}$, adds $value$ with $C_{[i,j]}$, and stores the new partial sum back to the cache. The cache lookup, addition, and write back are perceived as an atomic operation. If the key misses in the cache, it inserts a new entry into the cache. Cache overflow is handled by hardware, which will be discussed in detail in Section 4.2. The evicted entries will be stored in the pre-allocated tupleC. Regardless of partial sum cache hit, miss, or overflow, the ASA unit will handle it by hardware without data-dependent branches, which is one of the key advantages as compared to the original software implementation.

After the numeric phase, ASA.gather() (line 16) writes all of the valid entries into tupleC following evicted partial sums if there are cache overflows during the numeric phase. The tail pointer position is recorded before calling ASA.gather() to allow a pair sort function call to perform in-place sorting on non-repeating keys (line 17). If there are overflows for this column computation, additional software merges to tupleC will be used to merge overflowed key-value pairs to the sorted key-value pairs in tupleC (line 18-20) with O(N) time complexity, where N is the total number of the overflows. After this additional merging, the size of tupleC might be reduced and can be added to matrix C in the compressed storage format. Finally, the allocated space for tupleC is released and $ASA.clear()$ (line 23) is invoked to clear the partial sum cache and ASA internal registers.

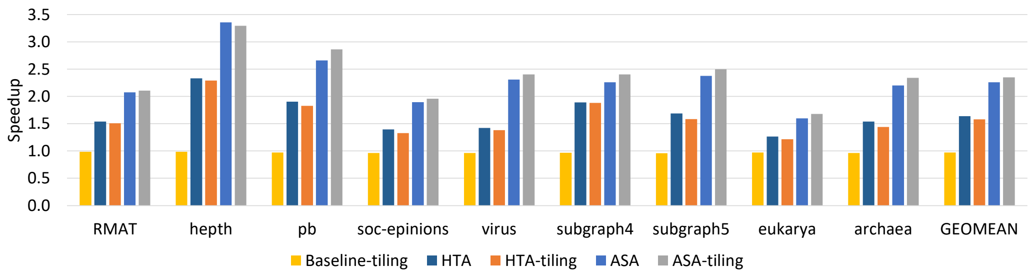


Fig. 8. HipMCL Performance comparisons for baseline software implementation, HTA, and ASA with and without tiling algorithm.

6 EXPERIMENTAL SETUP

To evaluate the proposed ASA architecture, this work modifies ZSim[42], which is a PIN-based [32], execution-driven, and cycle-accurate simulator, to model the proposed architecture and compare the performance with a baseline multi-core architecture and the HTA architecture. Cache sizes and latency of the baseline architecture are modeled based on Intel i7-6700 [24], which are listed in Table 2. The main memory timing parameters are based on the Micron MT40A2G4 [35] DDR4-2400-CL17 datasheet. This work uses CACTI 7.0 [4] to model the partial sum cache and estimate its area, latency, and power. Area overhead of other components is estimated based on logic synthesis using the FreePDK45nm [48] standard cell library, and is scaled to 14 nm. Accessing partial sum cache costs 0.004nJ per read and 0.012nJ per write of dynamic energy. The latency fits into one processor cycle at 2.6 GHz. McPAT [27] is used to estimate the energy consumption of other on-chip components. In the ASA architecture, each of the cores is augmented with an ASA unit for SPA acceleration, which includes an accumulation waiting buffer, a floating-point adder, a partial sum cache, and an address generator. To have a fair comparison with the baseline and the HTA, we increase the L1 cache size of the baseline and the HTA by the size of the partial sum cache. As a result, all three evaluated architectures have the same total on-chip cache capacity. For CPI breakdown analysis, ZSim does not execute wrong-path execution but recovers from mispredictions in a fixed 17 cycles and cancels in-flight data misses. [42]

Table 2. Baseline configuration for simulation.

Processors	8 cores, 2.6 GHz, 4-wide OoO, 256-entry ROB 64-entry LSQ
Partial Sum Cache	private, 8KB, 8-way, LRU, delay = 1 cycle
L1-Ds/Is	private, 32KB, 8-way, 8-entry MSHR delay = 4 cycles
L2s	private, 256KB, 8-way, 16-entry MSHR delay = 12 cycles
LLC	shared, 8 MB, 16-way, 128-entry MSHR delay = 38 cycles
Memory Controller	FCFS, Open page, read and write queue size = 64
Main Memory	4GB, 2400 Mhz, 1 channel, 1 rank, 16 banks tCL = tRCD = tRP = 17 cycles

Table 3. Datasets (compressed into CSC format).

Number	Names	#Vertices	#Edges	Sizes
1	RMAT [7]	6.72M	104M	840MB
2	pb [26]	36.4K	4.3M	55MB
3	soc-epinions [26]	75.8K	500K	11MB
4	hepth [26]	27.7K	350K	5.7MB
5	virus [2]	0.2M	4.5M	90MB
6	subgraph4 [2]	13.6M	41.3M	550MB
7	subgraph5 [2]	4.1M	10.3M	140MB
8	eukarya [2]	12.5M	56.2M	560MB
9	archaea [2]	0.7M	3.2M	47MB

This work evaluates a SpGEMM implementation from a highly optimized GraphBLAS [12] library, which powers HipMCL [2, 44]. The datasets of the SpGEMM are listed in Table 3, which covers representative inputs with different characteristics (*i.e.,* size, sparsity). As HipMCL runs the SpGEMM kernel iteratively, we mark the first five iterations as the ROI (region of interests) to save simulation time. Given the fact that the first few iterations take most of the MCL runtime (Figure 3), the sampled iterations are critical and representative of the entire application. The typical input graphs for HipMCL is too large (20 GB to 10 TB) to use on a simulator, and requires partitioning to run on real machines. This work focuses on accelerating the local SpGEMM, which makes it sufficient to use sub-matrices. Table 3 lists all of the graphs used in the evaluation. Subgraph4, subgraph5, eukarya, virus, and archaea datasets are subsampled by CombBLAS [10] matrix partition library from the original data set. The sampling algorithm randomizes the vertex labels before extraction of the sub-matrices. Besides sub-matrices, we also tested a synthetic graph (RMAT), a protein bank data (pb), a social network (soc-epinions), and a physical citation network graph (hepth). [1] For denser column computation, we break it down to multiple sub-columns as a simple tiling algorithm to reduce the number of partial sum cache overflows. HipMCL requires a larger memory footprint than the input size. For example, it uses more than 12GB of memory for a 500MB input (subgraph4 in the evaluation). When the ASA is used for larger inputs, tiling algorithms are expected to be applied to reduce memory usage, which will also reduce the overflows.

7 EVALUATION RESULTS

This section presents the evaluation results of the proposed ASA architecture on performance and energy. A roofline model analysis is performed to demonstrate the computation bottlenecks. Moreover, sensitivity studies are conducted on the partial sum cache configurations and alternative design choices on offloading computation to the hardware accelerator.

7.1 Performance

The performance benefit of the proposed ASA design comes from three aspects: 1) it avoids branch mis-prediction penalty in the baseline hash-based SpGEMM, 2) it reduces the total number of the instructions by consolidating hash probing, collision handling, and accumulation operations into a single instruction and removing symbolic phase, and 3) it provides a higher throughput for sparse accumulation by using a dedicated cache and accumulator.

Speedup. On average, ASA achieves a 2.25x speedup as compared to the baseline hash-based SpGEMM, which is 67% more than what HTA can achieve (Figure 8). As compared to HTA,

[1]Note that HipMCL computes matrix multiplication with input matrix $B = A^T$

which accelerates only hash operations, ASA uses a dedicated partial sum cache and a dedicated accumulator to provide higher throughput for sparse accumulation. HTA relies on a software rollback for collision and overflow handling. Overflows in the HTA table will trigger a software fallback path for an update, whereas ASA uses the address generator to write the overflowed entries to a pre-allocated memory space and merge overflowed partial sums in the end. HTA evicts a randomly chosen key-value pair to the next level to make space for a new one, which may cause premature eviction when hash probing has locality. ASA uses a LRU replacement policy to exploit locality in SpGEMM computation and minimize premature eviction of partial sums. HTA was designed for hash-intensive applications, especially those that have large hash tables, where poor locality causes cache thrashing and long memory stalls. In SpGEMM, the input matrices can be partitioned into tiles to allow non-zeros in a sub-column to fit into on-chip caches, the sparse accumulation throughput is a greater concern than cache thrashing. In fact, applying tiling does not help to improve performance for the baseline nor the HTA for the evaluated application and inputs. This is because the non-zeros in each output column can already fit in a L1 cache. Tiling does not provide more locality benefit for the baseline and HTA, but rather adds overheads due to increased number of branches and more irregular memory accesses in the tiled input matrix A.

Tiling helps to improve performance for ASA by reducing cache overflows in the small partial sum cache. In general, input graphs that observe a large reduction on cache overflows (*e.g.*, pb, subgraph4, subgraph5, eukarya, and archaea according to Figure 11) have performance benefit from tiling.

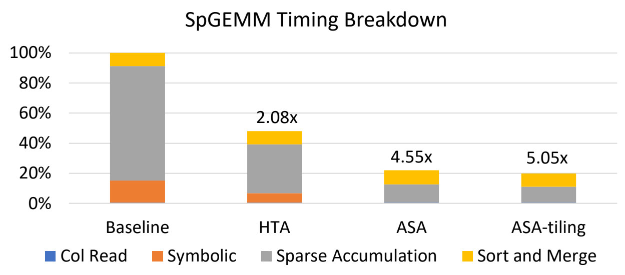


Fig. 9. Average execution time break down for SpGEMM kernel across all inputs.

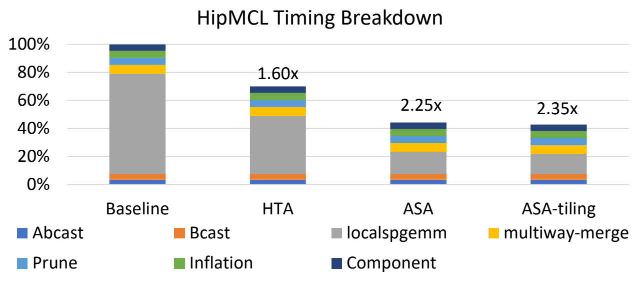


Fig. 10. Average execution time break down for HipMCL application across all inputs.

Speedup breakdown. To understand how well the proposed ASA helps with improving hash probing throughput and how much overhead comes from the cache overflows, we break down the SpGEMM kernel timing in Figure 9. On average, the proposed ASA can achieve a 4.55x of performance speedup for the SpGEMM kernel. The symbolic phase takes 14.5% of the execution time for the baseline, which can be eliminated from ASA-enabled SpGEMM. The sparse accumulation (hash-based numeric phase) takes 76% of the baseline runtime, which can be reduced by 6.33x. The performance overhead of the ASA is when there are overflows and costs conditional merging for all of the overflowed entries. In the baseline SpGEMM, sorting and merging takes 8.7% of the total execution time, whereas in ASA, sorting and merging now takes 9.5% relative to the baseline execution time. As a result, the conditional merging only costs 0.8% of the total performance overhead because the selected partial sum cache allows most of the hash probing to be overflow-free. Applying tiling can further reduce the number of overflows and hence reduce the sorting and merging latency to 8.45% of total execution time. This is because sorting multiple small chunks takes less time than sorting all chunks together.

We further break down HipMCL workload, the localspgemm in Figure 10 refers to the total execution time of the SpGEMM kernel. The overall performance of the MCL algorithm can be improved by 2.25x because of the speedup from the SpGEMM kernel.

7.2 Offload Hash and Multiplication

The proposed ASA architecture does not offload hash key calculation and multiplication to hardware for a cost-effective design. Adding more hardware resources for hash key calculation and multiplication (line 5-6 in Algorithm 2) can further achieve an average of 15.8% and a 5.3% of additional speedup for the SpGEMM kernel and the HipMCL application respectively.

There are three reasons to keep hash key computation and multiplication in software: 1) the programmer can have the flexibility to explore different hash functions, which may result in different optimal choices for different problem domains. The choice of the hash function will influence the load balancing among different cache sets, which can result in different number of cache overflows due to set conflicts. The evaluated design uses a prime number modulo hash function. 2) Multiplications of the non-zero elements can be vectorized using an existing vector engine inside the core to achieve higher throughput such as the Intel AVX-512. The evaluated design of ASA uses the existing floating-point unit (FPU) to reduce area overhead. And 3) offloading hash function and multiplication to dedicated hardware logic only achieves an incremental improvement according the simulation results of the selected inputs.

7.3 Overflow Rate

Instead of precisely split tiles based on the number of nonzeros, we use a simple tiling algorithm that breaks dense output columns into multiple sub-columns. During the actual computation, if a column of C will cause overflow, the column is broken up into several chunks. The chunks span uniform parts of A, e.g. if A has 2 million rows and we'd like to break the column into 2 chunks, the first chunk will contain entries [0,1e6) and the second chunk [1e6,2e6). We assume that the distribution of the non-zeros is not particularly skewed towards either chunk. The SpGEMM then proceeds to fully compute one chunk at a time. As shown in Figure 11, the overflow rate can be significantly reduced by applying a simple tiling algorithm.

7.4 Partial Sum Cache Configurations

The size and associativity of the partial sum cache can influence the set conflict rate as shown in Figure 4. We found that the performance is more sensitive to the cache capacity than associativity (Figure 12). In our design, the partial sum cache is implemented as a fine-grained cache with a block size equal to the word size of a partial sum (*i.e.,* 8B). The ASA can achieve a good speedup with a 4KB cache, which can save up to 512 key-value pairs. The tiling algorithm selects the size of

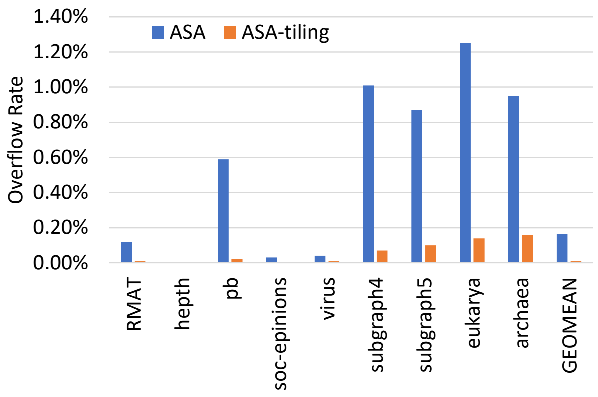


Fig. 11. Overflow rate for ASA with and without tiling.

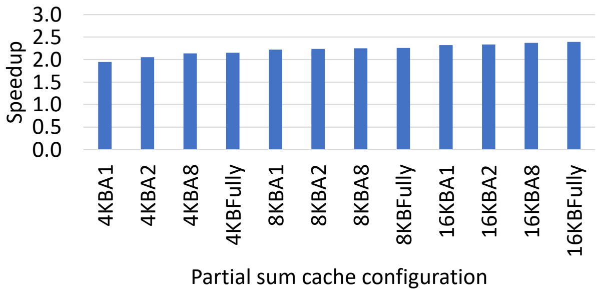


Fig. 12. Performance with different partial sum cache configurations. (The L1 cache size is kept as 32KB)

the sub-matrices to fit the number of non-zeros into the cache. The smaller the cache, the faster the cache lookup and the lower the lookup energy. However, if the cache is too small, the input sub-matrices of A would have fewer rows and hence increase the amount of irregular accesses due to the DCSC storage format used in HipMCL. Ideally, the partial sum cache should be large enough to allow the input data broadcasting to saturate the system memory bandwidth, yet small enough to allow fast cache lookup to match with the demand sparse accumulation throughput. This optimal design point of cache size depends on the sparsity and merging factor of the matrices. Luckily, for the evaluated graphs, the selected 8KB cache is within this optimal range.

7.5 Roofline Modeling

Sparse accumulation is the bottleneck of the baseline system that prevents it from achieving a higher throughput. A roofline model for HipMCL application with different inputs is shown in Figure 13, which includes bandwidth ceilings of different levels of the memory hierarchy and a computation ceiling. The original HipMCL implementation does not fully utilize the memory bandwidth nor the processing throughput. This is because the sparse accumulation is bounded by the data-dependent branches. The proposed ASA eliminates those hard-to-predict branches and improves sparse accumulate throughput using dedicated partial sum caches. The performance is improved by more than 2x. As a result, for all of the inputs, their positions on the roofline graph

[2]The FLOP calculation does not include the accumulations done by ASA.

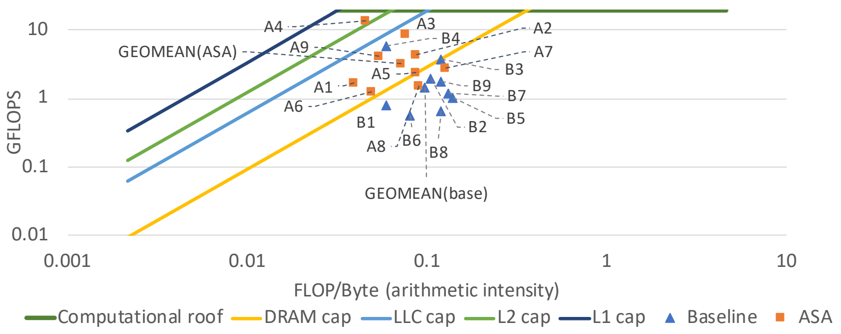


Fig. 13. Roofline modeling (orange colored dots represented ASA kernels, and blue colored dots represented baseline kernels) (A1 represent ASA with input RAMT-input number 1 in Table 3).[2]

are shifted toward the upper left. After using ASA, all of the inputs are closer to the rooflines. Most inputs are bounded by the memory and last level cache throughput.

7.6 Instruction Reduction

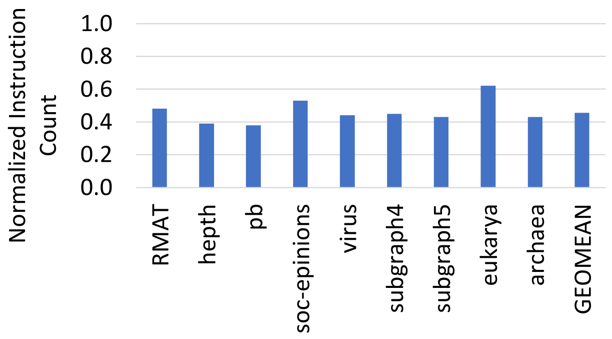


Fig. 14. Normalized instruction count after adopting ASA.

The proposed ASA reduces the total number of instructions by 1) packing complicated hash probing and collision handling into a single instruction, and 2) removing the symbolic execution as the implementation does not require allocating the hash table from software anymore. On average, the HipMCL algorithm running on ASA architecture observes a 54.4% dynamic instruction reduction as compared to the baseline (Figure 14). Although additional instructions are expected when there are cache overflows, the frequency of overflows remains low for all evaluated inputs. As a result, it does not contribute a large portion to the total instruction count.

7.7 On-chip Energy

HTA reduces the energy consumption and achieves a better performance as compared to the baseline. The proposed ASA architecture reduces more energy as compared to HTA, shown in Figure 15. There are three reasons for this further energy reduction. (1) The reduced instruction counts contribute to a reduction in energy associated with instruction fetching and decoding. (2) hardware hash probings in ASA use a smaller partial sum cache, which has a lower access energy than the access energy of a L1 cache. And (3) the reduced execution time in ASA reduces energy

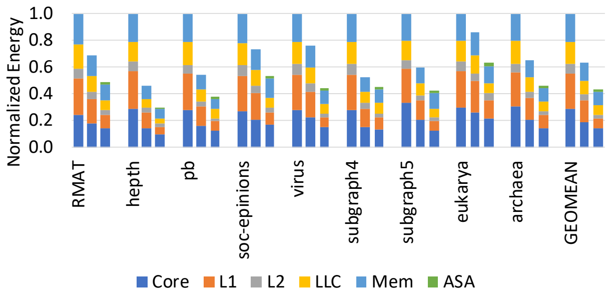


Fig. 15. Energy consumption normalized to the baseline. (Left bar: Baseline, mid bar: HTA, right bar: ASA)

associated with leakage power. As a result, the ASA reduces the total on-chip energy by 57.1% as compared to the baseline, which is a nearly 20% more reduction than HTA does.

7.8 Execution time breakdown

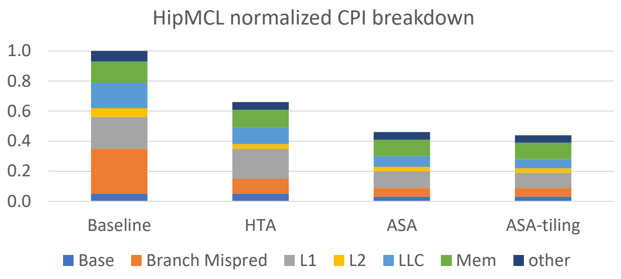


Fig. 16. Average normalized execution time breakdown.

To quantify the performance bottleneck, a execution time breakdown is shown in Figure 16, which followed the CPI stack methodology [16]. Each bar represent the average stalling time across cores and memory hierarchy.

ASA can reduce the stalling by branch mispredictions more than HTA does. This is because ASA can handle collision and data-dependent accumulation automatically by hardware. Moreover, ASA offloads the sparse accumulation to the partial sum cache. Execution time on L1 and LLC cache is also significantly reduced as compared to baseline.

7.9 Area overhead

The area overhead of the ASA architecture consists of four major components: 1) the partial sum caches, 2) the additional FP adders, 3) the accumulation waiting buffers, and 4) the address generators. The total area overhead is $0.014mm^2$ at 14nm, which occupies 0.013% of an 8-core processor die (100.708 mm^2).

8 RELATED WORK

To the best of our knowledge, ASA is the first tailored accelerator design for sparse accumulation in column-wise SpGEMM. In this section, we discuss the pros and cons of related works that target SpGEMM.

Hardware-accelerated index matching. SMASH [25] proposed a novel software encoding based on a hierarchical bitmap and used a bitmap management unit per core for highly efficient indexing of sparse matrices for the inner-product method. It does not fundamentally reduce the dependent loads as each multiplication is always followed by an index matching of two elements. Moreover, the hierarchical bitmaps occupy a larger memory footprint as compared to the CSR [31] format. The method requires converting the data format from CSR to the hierarchy of bitmaps as a pre-processing step, which adds a 4-30% of the end-to-end execution with different inputs.

Hardware-accelerated scatter updates. PHI [36] added compute logic at each cache level to buffer and coalesce these commutative updates throughout the hierarchy. As a result, it exploits temporal locality with low cost of synchronizations. In column-based SpGEMM, especially SUMMA [11] , the input matrices are partitioned and broadcast to different cores, so that most of the scatter updates will hit in the cache with minimal synchronizations with other cores. In our works, ASA has the following differences that makes it a better fit for column-based SpGEMM. 1) ASA extends the ISA for reducing branch-misprediction cost while PHI did not improves the branch predictions. 2) ASA supports hardware hash-based SpGEMM, which can effectively reduce the footprint of the scatter updates, while PHI focused on scatter updates to larger memory space. And 3) ASA leverages the partial sum cache with a higher throughput as compared to L1 cache to improve the speed of the sparse accumulations.

Hardware-accelerated hash lookup. HTA [59] proposed to extend the ISA for hash-specific operations, including hash lookup, insertion, and deletion, which helps to reduce the hard-to-predict branches and improves the core utilization. As compared to HTA, the proposed ASA can accelerate sparse accumulation in addition to hash operation. ASA handles hash collisions and cache overflows by hardware. Instead of leveraging existing caches to store hash table entries as HTA does, ASA adds dedicated on-chip storage as partial sum cache to provide higher throughput. With the same total on-chip cache capacity (data cache + partial sum cache), ASA outperforms HTA by more than 62.2% for a Markov clustering application with a SpGEMM kernel. SPX64 [46] proposed to add an on-chip scratchpad for accelerating hash lookups in redo logging for the transactional memory. However, it does not resolve data-dependent branches.

Processing-in-memory. PIM [29, 47, 61] exploits high internal bandwidth in/near memory. In 3D-stacked memory technology, cores on the logic layer can access the memory layer much faster than cores on the processors die do, which is helpful for memory-intensive applications. LiM [61] added CAM-based SpGEMM cores at the logic layer for fast lookup in sparse accumulation. As compared to the ASA design, LiM requires more hardware modifications and advanced technology.

SpGEMM accelerators. Software/hardware co-designs [23, 38, 40, 58, 60] of the outer-product based SpGEMM have shown significant improvements over the implementations on the traditional architectures. The OuterSPACE [38] architecture uses reconfigurable coalescing caches to separately optimize the multiplication and merging phases [38]. This is improved upon by SpArch [60], which proposed a dedicated merger and uses a Huffman tree scheduler to merge together partial sums in a way that reduces memory traffic. Gamma [58] leverages Gustavson's algorithm to accelerate SpGEMM, which manages data through fiber cache explicitly and improves the row traversal schedule to improve the data reuse. The proposed ASA architecture is an in-core extension to accelerate SpGEMM with good performance (5× speedup), which maintains the generality of multicore processors and requires a small area overhead (0.1%). Dedicated SpGEMM accelerators can achieve more than 20× speedup at a cost of more than 200× area overhead as compared to ASA.

Heap-based sparse accumulation [1, 10]. Instead of a hash table, some SpGEMM implementations use priority queues (heap) that are indexed by row indices for sparse accumulation. The advantage of using a heap-based accumulator is it does not require a sort and merge operation because the entries in the heap are sorted already. However, heap-based SpGEMM can be expensive because it requires logarithmic time to extract elements from the heap. As a result, heap-based

SpGEMM is more suitable for SpGEMMs with a low compression factor. In CombBLAS [10], a hybrid method is used to dynamically select whether to use a heap- or hash-based approach based on the symbolic phase analysis result. However, hash probing is still the bottleneck in CombBLAS when the applications have a high compression ratio.

Branch predication. Predicated execution [52] removes conditional branches from the instruction stream by conditionally execute instructions based on the results of the boolean conditions, which can effectively reduce the branch misprediction penalty. However, predication increases the number of executed instructions and energy consumption. The ASA design replaces the software hash lookups by the hardware cache lookups. As a result, there is no branch in sparse accumulations. Moreover, ASA reduces the total number of instructions by packing complicated hash probing and collision handling into a single instruction, which can further improve performance and reduce energy.

Indirect memory prefetching. Indirect memory prefetching [50, 57] can hide front-end stalling time by prefetching dependent data ahead of the time. In this work, the columns of the output matrix is tilled to fit into the partial sum cache. Hence, prefetching does not help in the sparse accumulation phase. Input columns of matrix A that are indexed by column values in matrix B can potentially benefit from indirect memory prefetching, which is complementary to ASA.

9 CONCLUSION

This work proposes ASA, an in-core extension for accelerating sparse accumulations in column-base SpGEMM. By using a single instruction to compute sparse accumulation for each multiplication result of a pair of non-zeros, ASA can reduce the total number of dynamic instructions, execution time, and energy. ASA adds a small dedicated on-chip set-associative cache with an accumulator to compute hash probing and accumulate to achieve a high throughput. Linear hash probing is replaced by parallel tag search in the set-associative cache. And cache overflows are stored to a pre-allocated data structure in memory to avoid stalling by allowing delayed merging of partial sums. The proposed ASA architecture has a simple programming interface that allows further software optimizations. As compared to the baseline as well as a state-of-the-art hashing accelerator design, ASA achieves better performance and energy efficiency with the same total on-chip cache capacity.

REFERENCES

[1] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2016. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 38, 6 (2016), C624–C651.

[2] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic acids research* 46, 6 (2018), e33–e33.

[3] Radhakisan Baheti and Helen Gill. 2011. Cyber-physical systems. *The impact of control technology* 12, 1 (2011), 161–166.

[4] 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), 14.

[5] David L Banks and Nicholas Hengartner. 2008. Social networks. *Encyclopedia of Quantitative Risk Analysis and Assessment* 4 (2008).

[6] Andreas D Baxevanis, Gary D Bader, and David S Wishart. 2020. *Bioinformatics.* John Wiley & Sons.

[7] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[8] Urban Borštnik, Joost VandeVondele, Valéry Weber, and Jürg Hutter. 2014. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Comput.* 40, 5 (2014), 47–58.

[9] Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.

[10] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509.

[11] Aydin Buluç and John R Gilbert. 2012. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191.

[12] Aydin Buluc, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017. The graphblas c api specification. *GraphBLAS. org, Tech. Rep* (2017).

[13] Nguyen Dao, Andrew Attwood, Bea Healy, and Dirk Koch. 2020. FlexBex: A RISC-V with a Reconfigurable Instruction Extension. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 190–195. https://doi.org/10.1109/ICFPT51103.2020.00034

[14] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 3–10.

[15] J. Austin Ellis and Sivasankaran Rajamanickam. 2019. Scalable Inference for Sparse Deep Neural Networks using Kokkos Kernels. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[16] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. 2006. A performance counter architecture for computing accurate CPI components. *ACM SIGPLAN Notices* 41, 11 (2006), 175–184.

[17] Md Abdul Motaleb Faysal, Shaikh M. Arifuzzaman, Maximilian Bremer, Doru Popovici, and John Shalf. in press 2021. HyPC-Map: A Hybrid Parallel Community Detection Algorithm Using Information-Theoretic Approach. In *Proceedings of the 2021 IEEE High Performance Extreme Computing Virtual Conference*. IEEE.

[18] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.

[19] Zhixiang Gu, Jose Moreira, David Edelsohn, and Ariful Azad. 2020. Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. 293–303.

[20] Giulia Guidi, Marquita Ellis, Daniel Rokhsar, Katherine Yelick, and Aydın Buluç. 2021. BELLA: Berkeley Efficient Long-Read to Long-Read Aligner and Overlapper. In *Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 123–134.

[21] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of facebook's DNN-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–501.

[22] Fred G Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269.

[23] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs.. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE.

[24] Intel. [n. d.]. Intel i7-6700 (Skylake), 4.0 GHz (Turbo Boost), 14 nm. https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html.

[25] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-Designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 600–614. https://doi.org/10.1145/3352460.3358286

[26] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[27] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. , 469–480 pages.

[28] Weizhong Li and Adam Godzik. 2006. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics* 22, 13 (2006), 1658–1659.

[29] Yan Liao, Huaqiang Wu, Weier Wan, Wenqiang Zhang, Bin Gao, H-S Philip Wong, and He Qian. 2018. Novel in-memory matrix-matrix multiplication with resistive cross-point arrays. In *2018 IEEE Symposium on VLSI Technology*. IEEE, 31–32.

[30] Greg Linden, Brent Smith, and Jeremy York. 2003. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing* 7, 1 (2003), 76–80.

[31] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[33] Aravind Madhavan, Raveendran Sindhu, Binod Parameswaran, Rajeev K Sukumaran, and Ashok Pandey. 2017. Metagenome analysis: a powerful tool for enzyme bioprospecting. *Applied biochemistry and biotechnology* 183, 2 (2017), 636–651.

[34] Michael McCourt, Barry Smith, and Hong Zhang. 2015. Sparse Matrix-Matrix Products Executed Through Coloring. *SIAM J. Matrix Anal. Appl.* 36, 1 (2015), 90–109.

[35] Micron. [n. d.]. 8Gb: x4, x8, x16 DDR4 SDRAM Features. Micron Technology, Inc. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.

[36] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1009–1022.

[37] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on intel knl and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. 1–10.

[38] 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.

[39] Sergio Pissanetzky. 1984. *Sparse Matrix Technology-electronic edition*. Academic Press.

[40] 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.

[41] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 347–358.

[42] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. , 475–486 pages.

[43] Naser Sedaghati, Renji Thomas, Louis-Noël Pouchet, Radu Teodorescu, and P. Sadayappan. 2011. StVEC: A Vector Instruction Extension for High Performance Stencil Computation. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 276–287. https://doi.org/10.1109/PACT.2011.59

[44] Oguz Selvitopi, Md Taufique Hussain, Ariful Azad, and Aydın Buluç. 2020. Optimizing high performance markov clustering for pre-exascale architectures. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 116–126.

[45] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[46] Abhishek Singh, Shail Dave, Pantea Zardoshti, Robert Brotzman, Chao Zhang, Xiaochen Guo, Aviral Shrivastava, Gang Tan, and Michael Spear. 2021. SPX64: A Scratchpad Memory for General-Purpose Microprocessors. *ACM Trans. Archit. Code Optim.* 18, 1, Article 14 year=2020, (Dec. 2021), 26 pages. https://doi.org/10.1145/3436730

[47] Sriseshan Srikanth, Anirudh Jain, Joseph M. Lennon, Thomas M. Conte, Erik Debenedictis, and Jeanine Cook. 2019. MetaStrider: Architectures for Scalable Memory-Centric Reduction of Sparse Data Streams. *ACM Trans. Archit. Code Optim.* 16, 4, Article 35 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355396

[48] James E Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W Rhett Davis, Paul D Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, et al. 2007. FreePDK: An open-source variation-aware design kit. , 173–174 pages.

[49] G. Szarnyas, D. A. Bader, T. A. Davis, J. Kitchen, T. G. Mattson, S. McMillan, and E. Welch. 2021. LAGraph: Linear Algebra, Network Analysis Libraries, and the Study of Graph Algorithms. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. SIAM, 243–252.

[50] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. 2021. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 654–667.

[51] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing Communication in Graph Neural Network Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, Article 70, 17 pages.

[52] Gary Scott Tyson. 1994. The effects of predicated execution on branch prediction. In *Proceedings of the 27th annual international symposium on Microarchitecture*. 196–206.

[53] Stijn Marinus van Dongen. 2000. *Graph clustering by flow simulation.* Ph. D. Dissertation. University of Utrecht.
[54] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S Schreiber. 2013. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 197–210.
[55] Richard W Vuduc and Hyun-Jin Moon. 2005. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *International Conference on High Performance Computing and Communications*. Springer, 807–816.
[56] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 1–12.
[57] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. 178–190.
[58] 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* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702
[59] Guowei Zhang and Daniel Sanchez. 2019. Leveraging caches to accelerate hash tables and memoization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 440–452.
[60] 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.
[61] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti. 2013. Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2013.6670336