InnerSP: A Memory Efficient Sparse Matrix Multiplication Accelerator with Locality-aware Inner Product Processing

Daehyeon Baek*, Soojin Hwang*, Taekyung Heo*, Daehoon Kim†, Jaehyuk Huh*
*School of Computing, KAIST
†Department of Information and Communication Engineering, DGIST
{dhbaek, sjhwang, tkheo}@casys.kaist.ac.kr, dkim@dgist.ac.kr, jhhuh@kaist.ac.kr

Abstract*—Sparse matrix multiplication is one of the key computational kernels in large-scale data analytics. However, a naive implementation suffers from the overheads of irregular memory accesses due to the representation of sparsity. To mitigate the memory access overheads, recent accelerator designs advocated the outer product processing which minimizes input accesses but generates intermediate products to be merged to the final output matrix. Using real-world sparse matrices, this study first identifies the memory bloating problem of the outer product designs due to the unpredictable intermediate products. Such an unpredictable increase in memory requirement during computation can limit the applicability of accelerators. To address the memory bloating problem, this study revisits an alternative inner product approach, and proposes a new accelerator design called InnerSP. This study shows that non-zero element distributions in real-world sparse matrices have a certain level of locality. Using a smart caching scheme designed for inner product, the locality is effectively exploited with a modest on-chip cache. However, the row-wise inner product relies on on-chip aggregation of intermediate products. Due to uneven sparsity per row, overflows or underflows of the on-chip storage for aggregation can occur. To maximize the parallelism while avoiding costly overflows, the proposed accelerator uses pre-scanning for row splitting and merging. The simulation results show that the performance of InnerSP can exceed or be similar to those of the prior outer product approaches without any memory bloating problem.

***Keywords*-sparse matrix multiplication, hardware accelerator, inner product**

I. Introduction

Sparse matrix multiplication has been one of the key computations used in a wide range of data analytics and high-performance computing. The applications of sparse matrices range from data analytics [3], recommendation systems [17], circuit simulation [6], machine learning [14], search engine [4], to computer graphics [25]. Although a sparse matrix can be represented in many different forms, the representation of sparsity itself incurs inherent indirect and thus irregular memory accesses for operations on sparse matrices. Due to the irregular patterns, memory access latency and bandwidth become a critical performance bottleneck.

To provide fast multiplications on large-scale sparse matrices, recent studies proposed accelerator designs combined with high bandwidth 3D-stacked DRAM [21], [27]. In the approaches, the key insight is that outer product computation can be more efficient with accelerators than conventional inner product approaches used in CPUs and GPUs. In the two prior accelerators, the outer product algorithm generates intermediate products which are not yet added to the final output, and writes them to the memory. The final accumulation of the intermediate outputs is done at the second stage. OuterSPACE [21] first proposed such outer product-based multiplication, and SpArch [27] improves the outer product approach with an on-chip partial merging mechanism to reduce the overheads of writing and reading a large amount of intermediate products.

The outer product approaches minimized input reads at the cost of increased memory traffic for partial products. However, this study identifies an important downside of the outer product approaches in its memory capacity usage. The generation of partial products requires a large amount of extra memory capacity, which can be multiples of the final output. In addition, the required memory capacity for a multiplication is unknown until the actual outer product operations are completed, since the partial product sizes depend on the distributions of non-zero elements. In sparse matrices for large-scale commercial applications, such a *memory bloating* problem can severely restrict the applicability of the accelerators.

This study first analyzes 755 sparse matrices to show the severity of the memory bloating problem. The analysis shows that the worst 20% of the real-world matrices require off-chip memory 16.9 times larger than the size of the final output matrix to store partial products if the outer product approach is used. In addition, the analysis also shows that many common sparse matrices have a certain level of access locality to be exploited during the inner product computation.

Based on the analysis, this study proposes a new accelerator design, called InnerSP, which revisits the conventional inner product approach. Unlike the outer product approach, InnerSP does not require any extra memory on top of two input and one output matrices. It employs a *row-wise inner product*, which multiplies one row of the first input matrix with the second matrix, producing a row of the output.

To improve the row-wise inner product, InnerSP enhances two performance-critical aspects, 1) exploiting locality in the second matrix, and 2) maximizing parallelism under a limited on-chip aggregation hash table. During a row computation step, part of the second matrix is fetched, and the intermediate results are merged in the on-chip hash table. With on-chip aggregation, the first input is read only once, and the final output is also written once. The second input can be fetched multiple times, potentially causing a large amount of memory traffic. However, there exists locality in the second matrix accesses because of the clustered column distribution of the first matrix. With the locality, a moderate on-chip cache can provide data reuse for the second matrix, reducing memory accesses significantly.

The second challenge in the acceleration with the row-wise inner product is the trade-off between the parallelism and potential overflows in the on-chip hash table for accumulating a row or rows of the output matrix. Due to high sparsity variance in the number of non-zero elements in each row, processing a single row or fixed number of rows at a time can suffer from the limited parallelism if not enough non-zero multiplications occur. On the other hand, it can suffer from the overflow of the on-chip accumulation hash table, if too many non-zero elements are generated. To address the problem, this paper proposes a pre-scanning technique. A quick first pass identifies the upper-bound number of non-zero elements for each output row. The result is used to determine the number of rows to be operated together. In addition, if a single output row cannot be stored in the on-chip accumulation hash table, the row is partitioned by column to avoid overflows in the on-chip accumulation hash table.

A recent study, MatRaptor [24], also investigated the potential of the conventional row-wise inner product approach with a new optimized sparse matrix format. It focuses on the area efficiency of the row-wise inner product while achieving better performance than OuterSPACE. Our study focuses on the memory bloating problem based on the extensive analysis of real square matrices from SuiteSparse Matrix Collection [5]. In addition, we investigate the locality of the second input matrix and the row merging and splitting technique to maximize the parallelism without hash table overflows. The simulation-based evaluation shows that INNERSP can exceed the performance of OuterSPACE by 4.57×, and MatRaptor by 2.45×. Compared to the best performing outer product approach (SpArch), INNERSP has 6.8% performance improvement without the memory bloating problem, not requiring any extra memory to store partial products during computation.

Compared to the prior approaches, the main contributions of this study are as follows:

- This paper analyzes a large number of real-world sparse matrices for identifying the partial product size and locality. Based on the analysis, it identifies the memory bloating problem of the outer product approaches.
- It proposes a new row-wise inner product accelerator, which can exploit the existing locality of the second matrix accesses.
- It proposes a pre-scanning technique to almost eliminate the overflow problem of the accumulation hash table in the inner product computation.

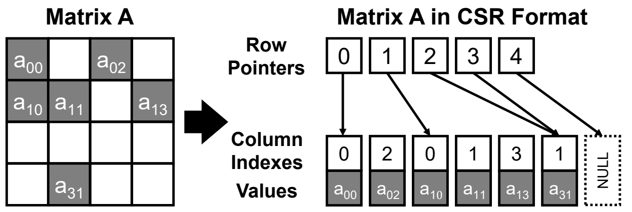


Figure 1: Compressed Sparse Row (CSR) format example

The rest of the paper is organized as follows. Section 2 describes the background in sparse matrix multiplication acceleration. Section 3 discusses the memory bloating problem of the outer product and potential locality in memory accesses of the inner product approach. Section 4 describes the proposed accelerator design, and Section 5 reports the experimental results. Section 6 presents the related work, and Section 7 concludes the paper.

II. BACKGROUNDS

A. Sparse Matrix and Multiplication

A sparse matrix is a matrix where most of its elements are zeroes. There have been several formats to represent sparse matrices in a memory-efficient manner by skipping zero elements. One of the most widely used formats is the Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) format. In this paper, we will use CSR as our primary format. Figure 1 illustrates an example of the CSR format. It consists of three arrays: *value*, *column-index*, and *row-pointer* arrays. The value array stores actual non-zero values in the sparse matrix in row major order. The column-index array (column array) has the column indices for corresponding elements in the value array. The row-pointer array (row array) points the starting index of each row in column-index and value array.

Generalized matrix multiplication (GEMM) produces the $M \times K$ size of output matrix (**C**) from the $M \times N$ size of first input matrix **A** and $N \times K$ size of second input matrix **B**. In this paper, **A**, **B**, and **C** will be used to denote the first and second input matrices and the output matrix. A typical inner product matrix multiplication computes a dot product of a row of **A** and a column of **B** to produce an element of **C**:

$$C[i,j] = \sum_{k=0}^{N-1} A[i,:] \cdot B[:,j]$$

SpGEMM accelerators: There have been several studies which proposed accelerators for sparse GEMM [21], [24], [27]. In the designs, an accelerator has a dedicated memory with high bandwidth, such as 3D-stacked High Bandwidth Memory (HBM). Input matrices in the CPU-side system

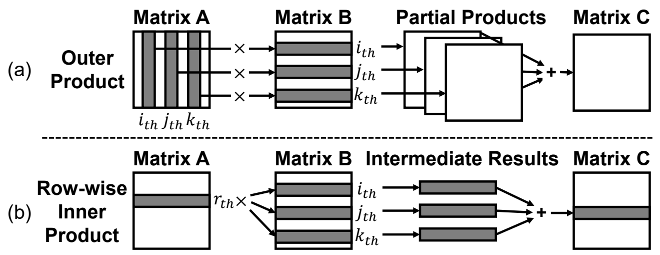


Figure 2: Comparison of outer product and row-wise inner product

memory are copied to the accelerator memory via DMA operations, and the output matrix is copied back to the system memory after computation. Since the accelerator memory capacity is fixed and the data transfer between the system DRAM and accelerator memory has much lower bandwidth than the accelerator memory, efficient utilization of the accelerator memory is important. Two common approaches for the accelerator algorithm are outer product and inner product techniques.

B. *Outer Product Accelerators*

Figure 2 (a) illustrates the outer product algorithm. In the outer product algorithm, the result matrix **C** is calculated by summing partial product matrices. Each partial product matrix is generated by multiplying a single column of **A** and a single row of **B**, when the column index of **A** and the row index of **B** match. Assuming the number of columns in **A** is equal to *N*, repeating the outer product calculation from the first to the last column of **A** and corresponding row of **B** produces *N* partial products. Final **C** is produced by adding all partial products.

$$C = \sum_{k=0}^{N-1} C_k, \text{ where } C_k = A[:,k] \times B[k,:]$$

A key advantage of the outer product algorithm is that it sequentially reads both **A** and **B** only once, eliminating repeated reads of **B**, which occur in the inner product algorithm. However, it needs to write all partial products to the memory and read them again to summarize them to the final output **C**.

OuterSPACE: OuterSPACE is the first outer-product-based sparse matrix multiplication accelerator [21]. It is based on the assumption that typical sparse matrices with a low density do not generate a large amount of partial products. If the density of input sparse matrices is low, then each non-zero element of the output **C** is accumulated from one or a small number of partial products. In such cases, the overall size of partial products may not be significantly larger than **C**. OuterSPACE proposed a two-phase algorithm with multiply and merge phases. The multiply phase produces all partial products by iterating *N* times. After all the partial products are written to the memory, they are read and accumulated to produce the final **C** matrix.

SpArch: SpArch improves the outer product approach by adding on-chip merging of partial products [27]. Unlike OuterSPACE, SpArch attempts to merge multiple partial products with on-chip parallel merge trees, and the partially merged intermediate products are written to the memory. Compared to OuterSPACE, it can reduce the memory write and read traffic for partial products with the on-the-fly merging mechanism. The effectiveness of the on-chip merge is dependent on the locality of row and column indices for partial products. It further improves the merging efficiency by condensing **A** to reduce the number of partial matrices while sacrificing the efficiency of reading **B**. To overcome the disadvantage, SpArch adds a prefetcher for **B** to hide read latency and reduce duplicated accesses of **B**.

C. *Row-wise Inner Product Based Solutions*

Another widely adopted matrix multiplication algorithm is a row-wise inner product. For its high parallelism and low memory footprint, the row-wise inner product is used in CPU-based solutions [10] and in GPU-based ones [2], [16], [19], [20]. Figure 2 (b) illustrates the row-wise inner product algorithm. A single row of the result matrix **C** is the sum of intermediate results, which are generated by multiplying each element of a row in **A** and row vectors of **B**. The intermediate row is the multiplication result of elements from the row of **A** whose column indices are matched with the row indices of corresponding rows in **B**. The amount of intermediate products which must be stored, is much smaller than the outer product algorithm, since they are merged for each **C** row and they are no longer needed after the completion of the output row.

$$C[i,:] = \sum(A[i,k] \times B[k,:])$$

MatRaptor: A recent study that uses the row-wise inner product approach is MatRaptor [24]. MatRaptor proposes C^2SR, a new hardware-friendly sparse matrix format to efficiently utilize the memory bandwidth. With this read optimization, MatRaptor uses three on-chip queues to merge and sort each intermediate row. However, this approach raises an exception to the CPU when the queues overflow, and the CPU handles the rest of the row that is not merged in the on-chip queues. Such overflows can lead to performance degradation for certain inputs. Although MatRaptor and InnerSP are both based on the row-wise inner products, this study investigates the memory bloating problem of outer product and the locality opportunities of **B**. In addition, this study addresses the overflow and underflow challenges of on-chip merging storage.

Table I shows the comparison of the prior three approaches and ours. OuterSPACE [21] and SpArch [27] use outer product approaches for calculation. Therefore, the two approaches have the memory bloating problem due to intermediate partial matrices. These partial matrices cannot be fit in on-chip memory, which causes additional

Methods	**Outer–SPACE** [21]	**SpArch** [27]	**MatRaptor** [24]	**INNERSP** (Ours)
Algorithm	Outer	Outer	Inner	Inner
# of B Reads	Once	Multiple	Multiple	Multiple
# of C Writes	Multiple	Multiple	Once	Once
Mem. Bloat Prevention	✗	✗	✓	✓
Caching B	✗	✓	✗	✓
Merge Ovf. Handling	✗	✗	✗	✓

Table I: Comparison to prior accelerators.

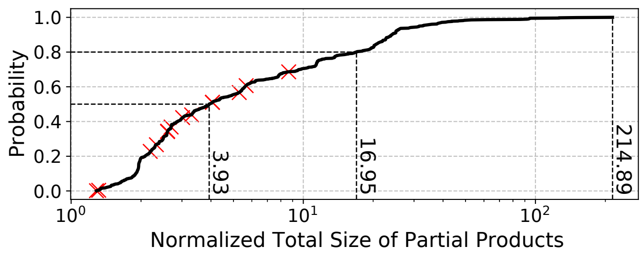


Figure 3: CDF of memory bloating factor. Red × marks present the matrices used in the evaluation of prior studies (comparison dataset) [21], [24], [27].

off-chip memory traffic. To reduce the off-chip memory traffic, SpArch also uses caching. MatRaptor [24] uses an inner product approach with on-chip row merging. It can prevent memory bloating, but it does not reduce repetitive **B** reads and does not address the performance degradation by overflowing output rows that cannot be fit in on-chip merging storage.

III. Motivation

A. Dataset

In this paper, we analyze 755 sparse matrices from SuiteSparse and SNAP [5]. From the suites, the 755 matrices are selected and used for evaluation to show the general performance of INNERSP. The selected matrices (*total dataset*) are square matrices, and the number of non-zero elements is within the minimum and maximum number of non-zero elements from the benchmark matrices used in the prior work [27]. Among them, 14 matrices (*comparison dataset*) are used to compare our performance result to the prior work, as they are the common matrices used by all three prior studies [21], [24], [27].

B. Memory Traffic Analysis

In this subsection, we analyze the amount of required memory accesses for the base outer product and inner product approaches to show that the memory traffic of the base outer product commonly exceeds that of the inner product.

Base outer product: The outer product approach reads **A** and **B** only once, but it can generate read and write operations for partial products. In the base outer product, the entire partial products are written to the memory during multiplication phase, and fetched for accumulation phase. The amount of partial product writes and reads depends on the number and distribution of non-zero values. If the majority of elements of **C** are the accumulation of many partial product values, then the amplification of writes for partial products will be severe. Otherwise, the partial product traffic will be similar to the write traffic of **C**. We will use *size*() function to denote the memory size of a matrix including the row pointers, column indices, and values.

The total memory traffic required for partial matrices is the twice of the total sum of partial matrices (*size*(**P**)). One is for writing partial matrices to memory, and the other is for reading partial matrices to generate the result matrix **C**. The final output **C** will be written after that. Therefore, the total memory access of outer product is equal to *size*(**A**) + *size*(**B**) + 2 × *size*(**P**) + *size*(**C**). However, if intermediate products are partially merged inside the accelerator as used in SpArch, memory accesses will be reduced.

Row-wise inner product: With the row-wise inner product, if the on-chip accumulation storage is large enough to hold a row of **C**, the accumulation is completed without extra memory writes for partial products. Therefore, the final **C** needs to be written only once. The main source of performance degradation in the inner product approach is reading **B** multiple times. If **A** and **B** are dense matrices, **B** must be read N times, where N is the number of rows in **A**. However, due to the sparsity of **A**, only part of **B** is read from the memory. For a non-zero row of **A**, a row of **B** is read only if the row index of **B** matches the column index of a non-zero element in the row of **A**.

Therefore, the row-wise inner product requires one read for **A**, multiple reads for **B**, and one write for **C**. The amount of read accesses for nonzero elements of **B** is identical to *size*(**P**) of the base outer product. Therefore, the total memory accesses of row-wise inner product is equal to *size*(**A**) + *size*(**P**) + *size*(**C**) + $\alpha \times$*size*(**A**). The last term, $\alpha \times$*size*(**A**), is to read two adjacent row pointers of **B** for every non-zero element of **A**, which is used to retrieve the elements of a **B** row matching the column index of a non-zero **A** element. α is the ratio of the size of two row pointers of **B** and a column-value pair for a non-zero element of **A**. In our case, α is 2/3. The analysis shows that the base outer product commonly has more memory accesses than the inner product, and the difference is *size*(**B**) + *size*(**P**) - $\alpha \times$*size*(**A**).

C. Memory Bloating of Outer Product

A key problem of outer product is *memory bloating*. Memory bloating is defined as the extra off-chip memory usage in addition to store **A**, **B**, and **C**. This subsection quantifies the extra off-chip memory capacity required to

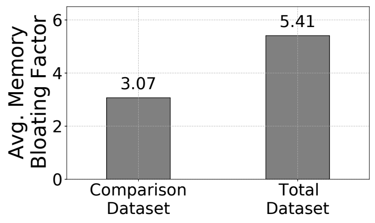


Figure 4: Average memory bloating factors of the comparison dataset and total dataset

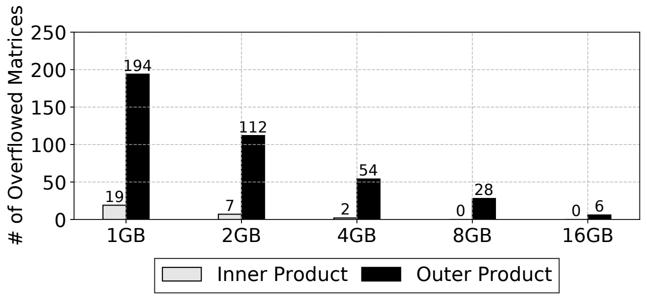


Figure 5: The numbers of sparse matrices that cannot be computed with given memory sizes and algorithms (lower is better)

hold the partial products for the *base outer product* approach. In this analysis, ***a memory bloating factor*** is the ratio of size of partial product and the size of the output.

Figure 3 presents the cumulative distribution function (CDF) of matrices with respect to their memory bloating factor. Even in the best case, the extra memory for partial products is similar to the size of matrix **C**. 50% of the matrices generate partial products that consume more than 3.93× memory capacity compared to final outputs. For the top 20% with the largest memory bloating, partial matrix sizes are 16.95 times larger than the sizes of **C**. Furthermore, this extra memory capacity required to store partial products is not known before multiplying sparse matrices, as its size is determined by the distribution of non-zero values in **A** and **B**.

In Figure 3, the fourteen matrices (comparison dataset) used in prior studies [21], [24], [27] are marked with red cross marks. The memory bloating factors of the comparison dataset are relatively low. Figure 4 compares the average memory bloating factors between the comparison set and total dataset. While the comparison set has a bloating factor of 3.07 on average, the total dataset has a bloating factor of 5.41.

This memory bloating problem may result in two critical problems. First, as the sizes of sparse matrices have been increasing for many real-world problems, the limited DRAM capacity of the accelerators cannot sustain the large extra memory required for the partial products. Second, the unpredictable extra memory size of the outer product approach requires dynamic memory allocation incurring extra overheads.

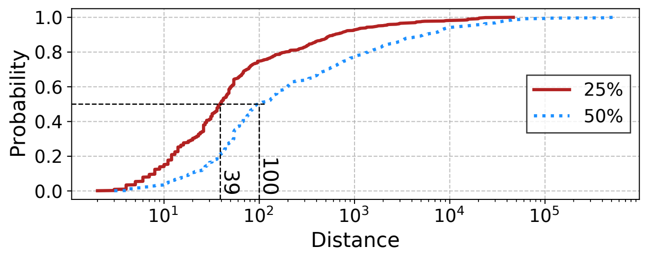


Figure 6: CDFs of distances presenting the locality of sparse matrix multiplications

Figure 5 shows the number of sparse matrices that cannot be computed because of memory overflows from the total dataset. We counted the number of overflowed matrices while varying the matrix multiplication algorithm and accelerator memory capacity. The figure shows that the numbers of non-computable matrices of the base outer product are significantly higher than those of the inner product approach. When the accelerator DRAM capacity is 4GB, the outer product approach cannot handle 54 matrices (out of 755 matrices), while the inner product approach cannot cover only two matrices.

SpArch proposed on-chip merging with a Huffman scheduler and pipelined merge tree to reduce the extra memory capacity that partial products consume. It mitigates the memory bloating problem of the base outer product approach. However, SpArch still needs to store some partial products in off-chip DRAM due to the limited merge tree and imperfect pipelining. Compared to outer product approaches, the row-wise inner product does not incur any extra memory for partial products, since all partial products are accumulated in the on-chip buffer and the final value is written to **C** only once.

D. *Locality in Inner Product*

One of the key assumptions of outer-product-based sparse matrix accelerators is the low locality of memory accesses. The locality of memory accesses in sparse matrix multiplications largely depends on the distribution of non-zero elements. Row-wise inner product involves multiplication between a row from **A** and rows from **B**, as described in Section II. As the non-zero element of i_{th} column of a row of **A** requires the i_{th} row of **B**, the frequent appearance of non-zero elements on the i_{th} column in **A** increases the locality of the i_{th} row of **B**. We define a metric, *distance*, which is the number of non-zero elements between two non-zero elements on the i_{th} column when presented in the CSR format. The shorter distance implies the higher locality of the i_{th} row of **B**.

To evaluate the locality of memory accesses of sparse matrix multiplications, we measure the distances of matrices in the total dataset. Figure 6 presents the CDFs of distances

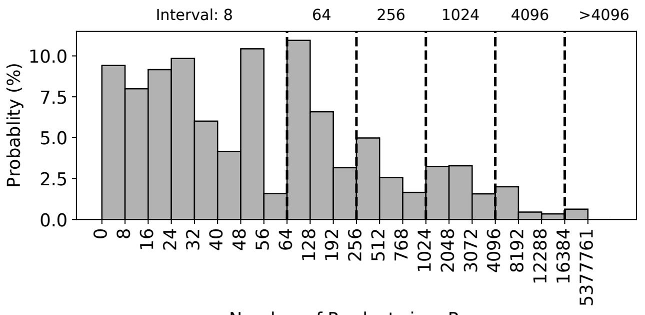


Figure 7: Distribution of the number of products generated in a row

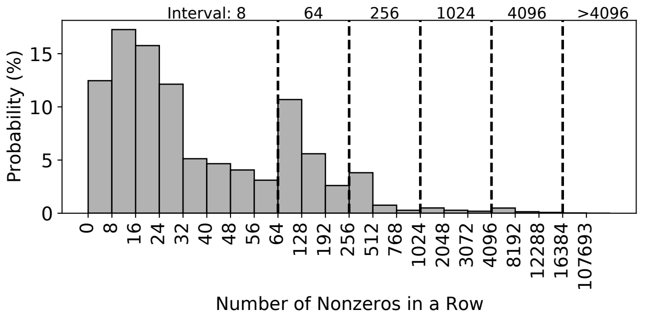


Figure 8: Distribution of non-zeros in a result row

on the 25th percentile and 50th percentile of each workload. The memory access localities in sparse matrix multiplication largely depend on the distribution of non-zero elements of matrices. The figure shows that real-world sparse matrices have locality that can be exploited by on-chip caching. For 50% of workloads, the distance on the 25th percentile is less than 39, and the distance on the 50th percentile is less than 100. Since **B** rows are reused within a relatively short distance, small on-chip caches can provide reused **B** data effectively.

E. Variance in Row-wise Sparsity

The row-wise inner product approach merges row-wise partial products with an on-chip storage, such as a hardware hash table. However, with the fixed size of the on-chip merging storage, dealing with the variance in sparsity per row is an important challenge. When the number of products is far less than the size of the merge storage, the storage is not only underutilized, but also the parallelism is not large enough to fully exploit the on-chip multiplication units. On the other hand, if the number of products is larger than the size of the merging storage, the off-chip memory must be used to write the overflowed elements that are not stored by the on-chip merging storage. Although the fallback mechanism allows the SpGEMM operation even under a high variance of sparsity, it can lead to a significant performance drop, due to the slow off-chip memory access latency.

Figure 7 shows a histogram for the distribution of the number of partial products per output row. Each region divided by dotted lines has different interval width for each bar as denoted in the graph, for efficient representation within a wide and sparse range. This result shows the multiplication parallelism for each row of output. The results are from the total dataset. As shown in the figure, there is a significant variance in the number of partial products per output row. The densities are high in the low range of the number of products, but there are still many output rows which can have more than 1000 partial products.

Figure 8 shows a histogram for the distribution of the number of elements per **C** row. As well as Figure 7, this histogram also has different bar ranges along regions. This analysis is directly related to the on-chip storage requirement for processing a single row of the output. Although the occurrence of overflow might not be frequent, the cost of handling an overflow is very expensive with off-chip memory accesses.

INNERSP addresses this variance of sparsity with new row merging and row splitting techniques. Using fast pre-scanning, INNERSP approximates the required merging storage. As long as on-chip storage is allowed, it merges rows to process them together. If an overflow is expected, it splits a row to avoid any overflow. With a 256KB on-chip merging storage, up-to 16,384 elements per row can be merged without any overflow. From the total dataset, we found 21,157 rows from 42 matrices that have rows that exceeds the 16,384 limit.

IV. ARCHITECTURE

A. Overview

INNERSP uses a row-wise inner product algorithm, and merges intermediate products on an on-chip hash table with accumulators as shown in Figure 9. The memory efficiency of the inner product algorithm eliminates the memory bloating problem of the outer product approach described in Section III-C. It allows much larger sparse matrices to be processed with a given fixed high bandwidth memory capacity attached to an accelerator.

In addition to 16 multiplication units in the multiplier array, two main components of INNERSP are the hardware hash table for accumulating intermediate results, and caching supports for the matrix **B**, exploiting the data locality in typical sparse matrices. The multiplication result is accumulated in the hash table, producing elements of **C**. Figure 10 describes the detailed architecture of the hash table. To support 16 outputs from the multiplication units in each cycle, the hash table consists of 16 banks. Each bank includes an adder for accumulation in addition to the hash storage.

A critical consideration for the hash table performance is a hash table overflow. If the number of non-zero elements of the current row-block exceeds the size of the hash table,

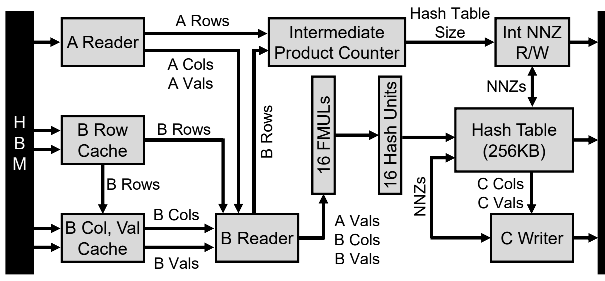


Figure 9: Overview of INNERSP architecture

some contents of the on-chip hash table should be evicted to the off-chip memory. With the backing memory, the computation still can be executed correctly, but the performance is significantly degraded due to off-chip memory access latency. To avoid such overflows for the hash table, INNERSP proposes a quick pre-scanning to identify the number of intermediate results for each row, and uses the intermediate results as an approximation of the hash table requirement.

Based on the pre-scanned analysis, a set of consecutive non-zero rows are processed together. We define a *row-block* as a batch of consecutive non-zero rows of **A** and corresponding rows of **C** processed together. The hash table accepts one row-block at a time to avoid overflows, and the size of each row-block is determined by the pre-scanning analysis. The hash table is indexed by the row and column indices of **C** to support the row-block processing.

INNERSP uses two caches for **B** to exploit the data locality of sparse matrices, *row pointer cache* and *column-value cache*. In INNERSP, the access pattern of the row pointer array is different from that of column and value arrays. For the **B** row pointer array, two adjacent row pointers are read for each row fetch, and the row pointer fetching has random patterns. However, **B** column-value arrays are read semi-sequentially for each row. Therefore, a small 32KB cache with 8B block size is used for the row pointer array, and the column and value cache capacity is 256KB or 512KB with a larger block size of 64B.

B. Major Components

The major components of INNERSP are the **A** reader, **B** reader, **B** caches, execution units, hash table, and **C** writer.

A reader: INNERSP reads the matrix **A** once in a sequential manner. It includes three simple FIFO queues which read row pointer, column index, and value arrays sequentially. Each FIFO queue can store 1024 entries to hide memory access latency.

B reader & B caches: To exploit the locality of accessing **B**, INNERSP uses two caches, *row cache*, and *column-value cache* for storing part of row pointer, column index, and value arrays of **B**. The row cache uses a 32KB 16-way associative cache to store the row array entries of **B**.

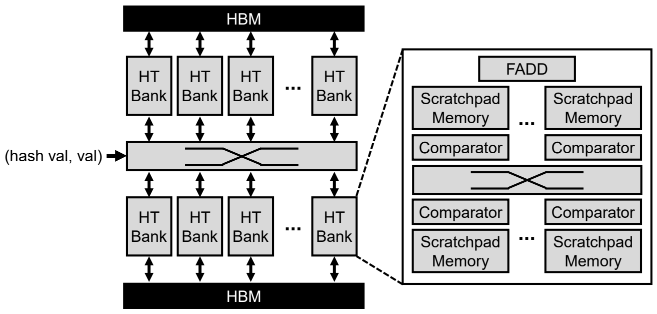


Figure 10: INNERSP hash table architecture. The hash table has 16 banks, and each bank has one adder and 16 pairs of 1KB scratchpad memory and comparators.

The cache consists of 16 banks with 8B block size, since row pointer accesses require only two adjacent row pointers (4B+4B) with low spatial locality. From the cache with 16 banks, the **B** row fetcher can fetch 16 row pointer pairs simultaneously in each cycle.

The column-value cache is used to store column and value entries of **B**. The column-value cache can store both of the column and value arrays. For each row fetch, the **B** column-value reader sequentially reads the columns and values of an entire row of **B**. To maximize the sequential read bandwidth, the column-value cache uses 64B block size.

Execution units: The multiplier array is composed of four SIMD units. Each SIMD unit consists of four double precision floating point multipliers, and the total number of multipliers is 16. In addition, there are 16 hash units to compute hash keys for subsequent hash table accesses. While multipliers generate double-precision floating point values, the hash units compute hash keys from row and column indices. With 16 multipliers and adders in 1GHz frequency, the ideal performance of INNERSP is 32 GFLOPS. shows the maximum performance of 32 GFLOPS.

Hash table: The hash table merges multiple intermediate results in parallel for different elements of **C**. Therefore, we designed the hash table as a group of 16 hash table modules with 16KB scratchpad memory in each module. Every module has a double-precision floating point adder that accumulates intermediate results, and 16 hash comparators to compare indices of non-zero elements in parallel. The hash table module also has a memory port to access the off-chip hash table buffers in the external memory when overflows occur.

Each hash entry contains 8B for the row (4B) and column (4B) indices, and 8B for value. The size of the hash table is 256KB or 512KB. The 256KB hash table contains 16,384 entries. For every request, a hash comparator lookups the corresponding bucket for the request, with the following three cases:

- **Hash Hit**: The comparator finds the hash entry that has the same row and column index in the scratchpad

Algorithm 1 Counting the number of intermediate products per output row

```
1: $input$: $A.row$, $A.col$, $B.row$
2: $output$: $intermediate\ product\ count\ (icnt)$
3: for $i = 0 \to A.nrow$ do
4:     read $A.row[i]$, $A.row[i+1]$
5:     $icnt[i] = 0$
6:     for $j = A.row[i] \to A.row[i+1]$ do
7:         read $colA = A.col[j]$
8:         read $B.row[colA]$, $B.row[colA+1]$
9:         $icnt[i] = icnt[i] + B.row[colA+1] - B.row[colA]$
10:     end for
11:     if $icnt[i] > B.ncol$ then
12:         $icnt[i] = B.ncol$
13:     end if
14: end for
```

memory. In this case, the comparator sends the value from the multiplier and the value from the existing hash entry to the floating-point adder. The added value is written back to the scratchpad memory.

- **Hash Insertion**: The comparator finds an empty entry from scratchpad memory for insertion. The empty entry is updated to the newly added row index, column index, and value.
- **Overflow**: If the hash table is full without the matching indices, then the comparator sends the new value to the DRAM hash table buffer.

All hash table requests with the same hash key go into the same hash comparator. The comparator serializes all requests to ensure atomicity for every hash insertion request to maintain the correctness of parallel hash insertion. For each request, a hash comparator reads its scratchpad memory in a linear-probing manner until a hit occurs or it inserts new entry. If a comparator reads all of 1KB scratchpad memory, and there is no room for the request, then the comparator fallbacks on the DRAM-resident hash table buffer.

With the 16 comparators for each 16 hash modules, the hash table can handle 256 hash entry accesses simultaneously, which is sufficient to receive 16 new entries per cycle with a small chance of pipeline stalls.

C writer: The **C** writer is composed of a row writer, a column writer, and a value writer. When writing **C**, the **C** row writer first receives the number of non-zero elements from the hash table, and accumulates them to generate row index pointers before writing. After that, the row writer writes a row pointer of **C**, and the column and value writers write the corresponding column indices and values to the memory. To hide the write latency, the column and value writers have a queue of 1024 entries for each writer.

C. Pre-scanning for Output Size Approximation

To avoid hash table overflows while maximizing the parallelism by increasing the row-block size, it is necessary to identify non-zero output elements for each row of **C** before computing **C**. To find the accurate number of non-zero elements of **C**, the algorithm must traverse the entire row and column arrays of **A** and **B**, consuming a significant amount of memory bandwidth. The time complexity of the algorithm is *O((number of nonzero elements in* ***A****)×(number of nonzero elements in* ***B****))*, which is equal to the complexity of index matching in real SpGEMM operation. To reduce the time complexity, InnerSP uses an approximate pre-scanning method which reads only the **A** row and column arrays, and the **B** row array. The approximate algorithm only finds the upper-bound of the number of non-zero elements for **C** by calculating the number of intermediate products. The number of non-zero elements is always less than or equal to the number of intermediate products for each **C** row.

Algorithm 1 shows how the numbers of intermediate products are counted from the row pointers and column indices of **A**, and the row pointers of **B**. For each row of **A**, column indices of non-zero elements of the row are read. For each retrieved column index, the number of non-zero elements of the **B** row corresponding to the column index is accumulated to the intermediate product counters (*icnt[]*). The number of intermediate products of each row is used for determining the amount of row splitting and merging.

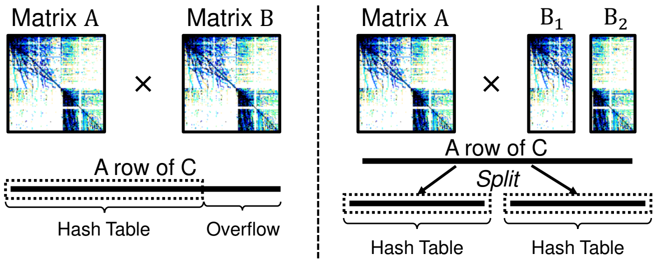


Figure 11: *Row Splitting* avoids hash table overflows by dividing the matrix **B** into smaller ones.

D. Row Splitting and Merging

Splitting a row to prevent overflows: If the number of non-zero elements in a **C** exceeds the hash table capacity, the overflowed **C** elements must be stored in the off-chip memory hash table, as shown in the left-side of Figure 11. However, such overflows can incur a significant performance degradation.

To prevent hash table overflows, InnerSP finds an upper-bound of non-zero elements in each **C** row with the pre-scanning step as described in Algorithm 1. If the upper-bound number of non-zero elements of a **C** row exceeds the hash table limit, InnerSP divides the rows of **C**, and processes split rows separately. For example, as shown in the right side of Figure 11, if there is a chance of overflow by the upper-bound analysis from the pre-scanning, **C** is divided into two split rows. To compute a split row, only the corresponding columns of **B** are multiplied with a row

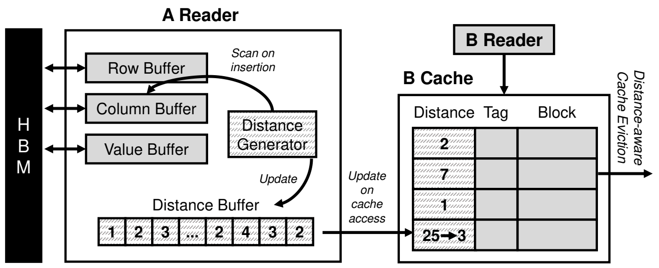


Figure 12: B cache structure with P-OPT. Distance generator utilizes the **A** column buffer data to generate next access times. This data is used in B cache for optimal eviction.

of **A**, producing a split **C** row. Until the prior split row computation is completed, the computation of next split row is postponed. However, to prefetch inputs, next input values can still be fetched and stored in on-chip buffers until the buffers are full.

Merging multiple rows to improve parallelism: To maximize parallelism when the hash table has enough space, INNERSP merges the computation of multiple rows of **C** into a single row-block. A row-block can be one or more rows of **C**.

Based on the upper-bound numbers of non-zero elements for rows of **C**, row-block sizes are adjusted for different rows of **C**. Similar to split rows, until a row-block computation is completed, the computation of the next row-block is blocked. After the completion of a row-block computation, the values in the hash table are written to the memory by the **C** writer. Note that even if the multiplications and hash table accesses are blocked for next row-blocks until the prior one is completed, the A and B readers can continue to fill the queues for the next row-blocks until the queues are full. Such decoupling can support pipelined data fetching and execution to hide long memory latencies.

E. Improving B Cache Replacement Policy

To further improve the utilization of the **B** cache, INNERSP exploits the computation pattern of sparse matrix multiplication, and enhances its replacement policy. This scheme is based on P-OPT [1], which improves the cache replacement policy for a large CPU last-level cache, when the CPU is running graph computation. Based on the observation from P-OPT, we modify and apply the technique to the B cache in our sparse matrix multiplication accelerator. In the inner product computation, for a column index of **A**, the corresponding row of **B** is needed for multiplication. If the distance between the same column indices across different rows of **A** is known, the reuse distance of the corresponding row of **B** can be inferred.

INNERSP identifies the column distance of **A** from the on-chip buffer of **A** column-value reader. The buffer contains up-to 4096 entries of column-value pairs. As shown in Figure 12, the distance is computed from the buffer, if the same column index appears in the buffer. The distance is stored in a separate distance buffer. With 4096 entries in the buffer, 12 bits are needed to represent each distance. For every insertion of column of **A** in the buffer, the distance generator updates distances of the nearest entry with the same column index.

Whenever the **B** row and column-value caches are accessed, the distance information is updated to the replacement policy bits in the caches. The cache has two bytes of age-bits to track eviction order in each block. The age-bits are divided into upper 12 bits for distance values and lower 4 bits for LRU position in a 16-associativity set. With every read access to a cache block, the upper 12 bits are updated to the new distance value. In a set, the block with the largest age value is evicted. When a block is evicted, the lower 4-bit LRU position is updated. In this scheme, the caches can evict a block near-optimally to minimize misses, since the replacement policy is based on near future access pattern.

Component	Description
A Row Fetcher	Queue with 4B, 1024 entries
A Column Value Fetcher	Queue with (4B, 8B) pair, 4096 entries
B Row Fetcher	Queue with (4B, 4B) pair, 1024 entries
B Column Value Fetcher	64 queues with (4B, 8B) pair, 128 entries each queue.
B Row Cache	32KB 16-way cache, 8B block size
B Column Value Cache	256 or 512KB 16-way cache, 64B block size 16 ports to main memory.
Multiplier	4 $\times$ 4 double-precision floating point multipliers with 1GHz frequency. Total 16 GFLOPS.
Interm. NNZ R/W	2 queues with 4B x 1024 entries.
Hash Table	16 $\times$ 16KB hash table modules Each module is composed of 16 $\times$ 1KB scratchpad memory and 16 hash processors, & a double-precision floating point adder. 16 GFLOPS total for addition.
C Row Writer	Queue with 4B, 128 entries.
C Column Value Writer	Queue for 4B column indices (1024 entries) Queue for 8B values (1024 entries).
Main Memory	HBM with 16 64-bit pseudo-channels each at 8GB/s. 80ns minimum latency, 100ns average latency.

Table II: Architectural simulation parameters

V. EVALUATION

A. Methodology

We implemented an in-house cycle-accurate simulator combined with DRAMsim3 [13] to evaluate the performance of INNERSP. Table II describes the simulation parameters used for the simulator. For the performance comparison, we normalize the results to that of the common baseline, Intel Math Kernel Library (MKL) [10]. We run experiments with MKL on a real system with Intel Core-i7 5930k processor, which is the same processor used in SpArch [27]. The main dataset used in the evaluation section is the comparison dataset described in Section III-C. Additionally, we show the speedups over MKL for the total dataset. We evaluate two configurations of INNERSP. `InnerSP-256` has a 256KB

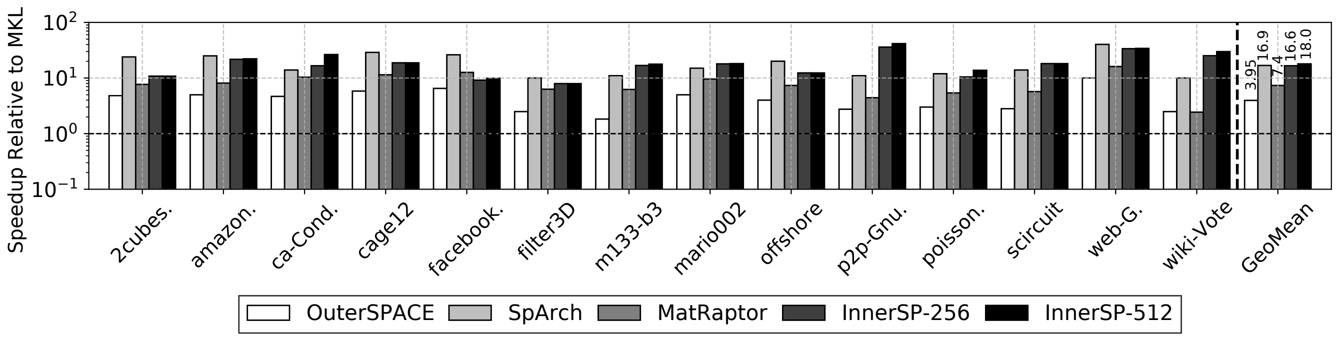


Figure 13: Speedup of InnerSP compared with previous works normalized to MKL (comparison dataset)

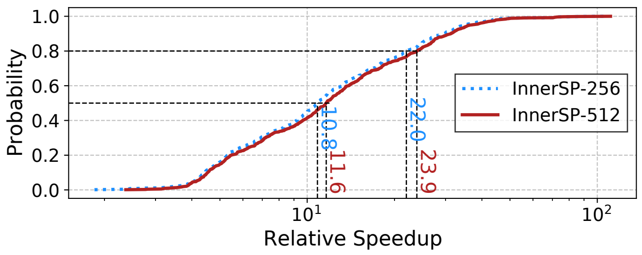


Figure 14: CDF of speedup with `InnerSP-256` and `InnerSP-512` for the total dataset, normalized to MKL

B column-value cache, and `InnerSP-512` has a 512KB cache, while the other configurations are equal.

B. Speedup

Figure 13 presents the performance of OuterSPACE, SpArch, MatRaptor, and InnerSP on the comparison dataset. The y-axis shows the relative speedup with respect to MKL. `InnerSP-512` shows 11.7 GFLOPS and 18.0× better performance than MKL on average. It outperforms OuterSpace by 4.57×. SpArch is the best-performing prior technique, and `InnerSP-512` outperforms SpArch by 6.8% without the memory bloating problem. Compared to the prior inner product scheme, MatRaptor, InnerSP shows 2.45× better performance. Therefore, InnerSP can achieve both performance and memory capacity efficiency goals, compared to the prior inner and outer product approaches.

Figure 14 shows the performance of `InnerSP-256` and `InnerSP-512` on the total dataset with a CDF graph, normalized to Intel MKL. With the total dataset, InnerSP reaches 14.0 GFLOPS in the geometric mean and 15.5 GFLOPS in the arithmetic mean with standard error ±6.00 GFLOPS. Compared with Intel MKL, `InnerSP-256` is 10.8× faster in the median, and shows more than 22.0× performance boost on the top 20% of matrices. `InnerSP-512` shows higher performance improvement than `InnerSP-256`, as it is 11.6× faster in the median.

C. Row-Block Adjustment Technique

InnerSP employs row splitting and row-merging techniques to avoid hash table overflows and to improve the parallelism. This subsection discusses the effectiveness of those techniques with the comparison dataset and five additional sparse matrices.

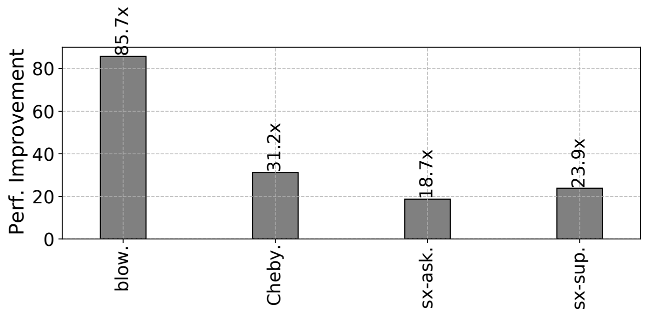


Figure 15: Row splitting: Performance improvements with `InnerSP-256`

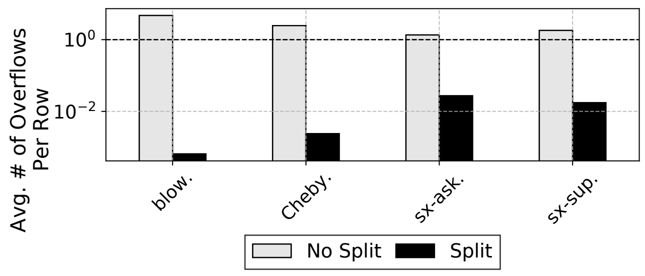


Figure 16: Row splitting: Comparison of average number of external memory accesses per row in each matrix in `InnerSP-256`

Row Splitting: Although the hash table overflow may not occur frequently, its performance impact is very high if it happens. The following four matrices are used to show the cases that suffer from hash table overflows: `bloweya` [8], `Chebyshev4` [18], `sx-askubuntu`, and `sx-superuser` [12]. The result matrices of the four matrices contain many rows that are larger than the hash table size. Figure 15 shows the performance improvements for the four cases by row splitting. Figure 16 shows the average number of overflows per row without and with row splitting. As shown in the figure, the reduction of memory traffic from hash table overflows from more than 1.33 memory accesses

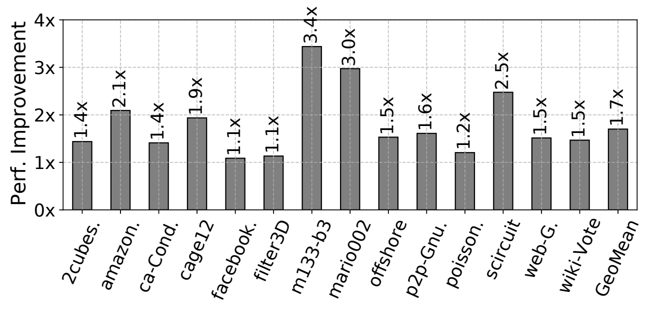


Figure 17: Row merging: Performance improvements with `InnerSP-256`

per row to less than 0.027 accesses per row, making the performance of InnerSP jumps by from 18.7× to 85.7×. The results validate that InnerSP can effectively handle large matrices with many concentrated non-zero elements in a row of the output matrix.

Row Merging: Row merging improves the performance by enhancing available parallelism, by processing as many rows as possible without causing hash table overflows. Figure 17 shows the performance improvement from row merging compared to the baseline without the row merging technique. On average, it can improve the performance by 1.7×. For example, in `wiki-Vote`, 37.3% of rows from the result matrix do not have any non-zero elements. For such workloads, row-block sizes can be increased significantly to improve the utilization of the execution units and hash table.

D. Performance Sensitivity

B column-value cache latency: In our configuration, the latency of the **B** column-value cache is set to 4ns. To evaluate the performance sensitivity to the latency, we increase the latency from 4ns to 8ns. Figure 18 shows the average performance using the comparison dataset with 4ns, 6ns, and 8ns latencies for the **B** column-value cache. The performance is normalized to that with 4ns. As shown in the figure, the latency change does not significantly affect the overall performance. 8ns reduces the performance only by 0.66%. As data fetching, execution, merging, and C writing are pipelined, the performance is highly dependent on the memory bandwidth than the latency. Therefore, the primary benefit of caching is the reduction of memory bandwidth consumption.

Non-square matrices: The comparison and total datasets only use square matrices. To evaluate the effect of non-square matrices, we randomly selected 13 non-square matrices from SuiteSparse Matrix Collection [5]. To evaluate non-square matrices, we evaluate the multiplication of a matrix and the transpose of the same matrix, since the row and column dimension must match for multiplication. Figure 19 shows the speedup normalized to MKL. The figure shows two results. The first bar is the performance of the multiplication of a matrix and the transpose of the same matrix, and the second bar is the performance of the multiplication of the transpose of a matrix and the original matrix. The average speedup is 20.6 and 8.9 for the two scenarios. The reason the second one has a much lower speedup is due to the dimension of the matrices. In most of the non-square matrices, the number of columns is much bigger than the number of rows. Therefore, the second one produces much bigger output matrices after multiplication.

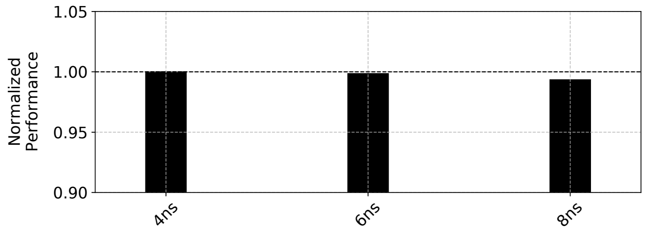


Figure 18: Average performance with various latency settings on **B** column-value cache in `InnerSP-256`, normalized to the 4ns (comparison dataset)

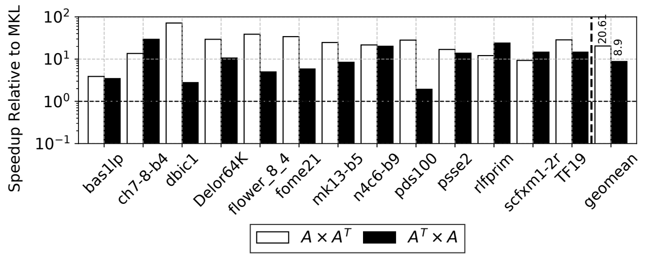


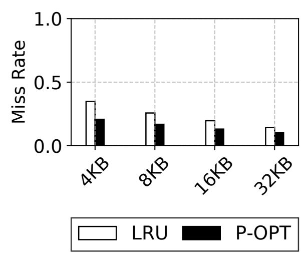
Figure 19: Speedup of `InnerSP-256` with non-square matrices normalized to MKL

E. Exploiting Data Locality

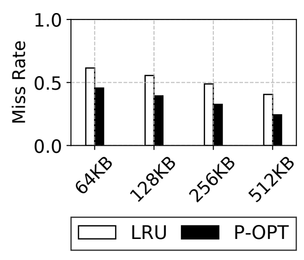
InnerSP exploits the data locality of sparse matrices with the **B** caches. To evaluate the effectiveness of caches, we conducted experiments with various sizes of **B** cache with the comparison dataset. Figure 20 presents average miss rates with various sizes of row and column-value caches. The figures also compare LRU and P-OPT policies. For Figure 20 (a), the size of **B** column-value cache is fixed to 256KB. For Figure 20 (b), the size of **B** row cache is fixed to 32KB.

Figure 20 (a) shows the **B** row cache has relatively low miss rates due to the small footprint of the row pointer array. With 32KB capacity and LRU policy, the average miss rate of the **B** row cache is only 14.2%. As shown in Figure 20 (b), the **B** column-value cache has higher miss rates than the row cache. However, with 512KB capacity and LRU policy, the miss rate is reduced to 40.5%, showing the significant locality of **B** accesses captured by the modest cache capacity.

In the figure, we compare the base LRU replacement policy with the P-OPT policy. As shown in the figure, P-OPT can reduce miss rates for both **B** row and column-value caches. The column-value cache benefits more from



(a) B row cache



(b) B column-value cache

Figure 20: Average miss rates with various B cache sizes (comparison dataset)

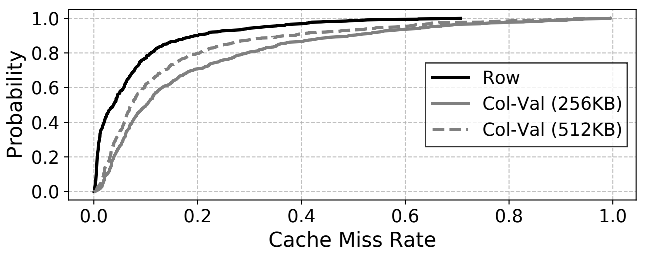


Figure 21: CDF of B cache miss rates with `InnerSP-256` and `InnerSP-512` for the total dataset

the P-OPT optimization than the row cache. The result shows the efficacy of the P-OPT policy for reducing the outgoing memory bandwidth consumption. For the 512KB **B** column-value cache, P-OPT reduces the miss rate to 24.3%.

Figure 21 presents the cumulative distributions of miss rates of **B** row and column-value caches, collected for the total dataset. With the total dataset, the **B** row cache and **B** 512KB column-value cache show the miss rates of 3.1% and 7.7% in geomean, respectively.

F. Area & Power Analysis

Table III compares the area and power overhead of INNERSP with prior studies. We estimate the area overhead and power consumption using CACTI for SRAM components and the synthesis results from the prior double-precision floating point multipliers and adders [7]. We estimate the area and power overhead of FIFO queues, caches, hash tables as an SRAM structure for CACTI and estimate the DRAM power consumption using the reported power consumption from the HBM specification [11]. INNERSP requires only 33% (`InnerSP-256`), and 66% (`InnerSP-512`) of the area of SpArch.

VI. Related Work

Sparse matrix accelerators: In addition to OuterSpace, SpArch, and MatRaptor, there have been other accelerator studies for sparse matrix-matrix multiplication and sparse matrix-vector multiplication. 3D-LiM was designed for the logic layer of 3D-stacked memory to accelerate sparse matrix multiplications [28]. It uses Doubly Compressed Column Format (DCSR) for input matrix **A**, and Coordinate List Format (COO) for input **B**. Sadi et al. proposed a sparse matrix-vector (SpMV) multiplication accelerator for highly sparse matrices [23]. SpaceA investigated SpMV multiplication accelerators optimized for processing-in-memory [26].

	OuterSPACE [21]	**SpArch** [27]	**MatRaptor** [24]	**INNERSP-256/512** (Ours)
Technology	32nm	40nm	28nm	40nm
Area	87 mm^2	28.49 mm^2	2.257 mm^2	9.41/18.7 mm^2
Power	12.39W	9.26W	1.34W	9.80/13.12 W
DRAM	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s

Table III: Area and power comparison with prior studies

Sparse neural network accelerators: SIGMA accelerates sparse matrix-matrix multiplication for deep neural network applications [22]. It maps its workloads into their flexible dot product engine dynamically via rich interconnect fabrics. UCNN accelerates sparse convolutional neural network layers by maximizing reuse of matrices in convolution operations [9]. Both approaches target deep neural network applications with 10-90% density, which is much higher than the traditional SpGEMM applications our study is targeting.

GPU software techniques: For GPU computing, inner product has been used as a common algorithm. BHSPARSE is a software framework for GPUs which applies different algorithms based on the number of intermediate products for each row to improve load balancing [15]. Nsparse uses a hash table with an inner-product approach, utilizing GPU shared memory for merging intermediate products with a two-pass algorithm to save memory usages [19].

VII. Conclusion

This study identified the memory bloating challenge of outer product approaches and proposed an accelerator design based on the row-wise inner product algorithm. To mitigate the weakness of inner product approaches, it proposed to utilize locality in sparse matrix multiplication and to avoid overflows in the on-chip merging table while maximizing execution unit utilization. Based on the optimizations, this study showed that the inner product acceleration can be a viable alternative to the outer production approach with efficient memory usage.

VIII. Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF-2019R1A2B5B01069816) and the Institute for Information & communications Technology Promotion (IITP2017-0-00466). Both grants are funded by the Ministry of Science and ICT, Korea. This work was also partly supported by Samsung Electronics Co., Ltd.

References

[1] V. Balaji, N. Crago, A.Jaleel, and B. Lucia, "P-OPT: Practical Optimal Cache Replacement for Graph Analytics," in *Proceedings of the 27th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[2] N. Bell and M. Garland, "CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations," https://cusplibrary.github.io/, [Online; accessed 25-November-2020].

[3] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics," in *Proceedings of the 30th ACM International Conference on Supercomputing (ICS)*, 2016.

[4] A. Coady, "Process and System for Sparse Vector and Matrix Representation of Document Indexing and Retrieval," 2004, US Patent 6,751,628.

[5] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[6] T. A. Davis and E. P. Natarajan, "Sparse Matrix Methods for Circuit Simulation Problems," in *Proceedings of the 8th Scientific Computing in Electrical Engineering (SCEE)*, 2010.

[7] S. Galal and M. Horowitz, "Energy-Efficient Floating-Point Unit Design," in *IEEE Transactions on Computers*, vol. 60, no. 7, 2010, pp. 913–922.

[8] N. Gould, Y. Hu, and J. Scott, "GHS indef collection," ftp://ftp.numerical.rl.ac.uk/pub/matrices/symmetric/, [Online; accessed 20-August-2021].

[9] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition," in *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018.

[10] Intel, "Intel Math Kernel Library," https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[11] JEDEC, "High Bandwidth Memory (HBM) DRAM JESD235D," 2021.

[12] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, jun 2014.

[13] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 110–113, 2020.

[14] T.-K. Lin and S.-Y. Chien, "Support Vector Machines on GPU with Sparse Matrix Format," in *Proceedings of the 9th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2010.

[15] W. Liu and B. Vinter, "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[16] W. Liu and B. Vinter, "A Framework for General Sparse Matrix–Matrix Multiplication on GPUs and Heterogeneous Processors," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 47–61, 2015.

[17] X. Luo, M. Zhou, S. Li, Z. You, Y. Xia, and Q. Zhu, "A Nonnegative Latent Factor Model for Large-Scale Sparse Matrices in Recommender Systems via Alternating Direction Method," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 3, pp. 579–592, 2015.

[18] B. Muite, "Ill-conditioned Chebyshev integration matrices," [Online; accessed 20-August-2021].

[19] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU," in *Proceedings of the 46th International Conference on Parallel Processing (ICPP)*, 2017.

[20] M. Naumov, L. S. Chien, P. Vandermersch, and U. Kapasi, "CUSPARSE Library," https://developer.nvidia.com/cusparse.

[21] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator," in *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[22] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training," in *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[23] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization," in *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[24] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product," in *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[25] J. Theiler, G. Cao, L. R. Bachega, and C. A. Bouman, "Sparse Matrix Transform for Hyperspectral Image Processing," *IEEE Journal of Selected Topics in Signal Processing*, vol. 5, no. 3, pp. 424–437, 2011.

[26] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator," in *Proceedings of the 27th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021.

[27] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient Architecture for Sparse Matrix Multiplication," in *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[28] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.