Project Final Report: Sparse GeMM

Ryan Ahmed
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX
ryan.sadad02@utexas.edu

Tianfang Guo
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX
tg25697@utexas.edu

Minseo Park
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX
mp46753@utexas.edu

I. Objective

We aim to study sparse matrix multiplication (Sparse GeMM) algorithms and hardware accelerators optimized for this task. Sparse matrix multiplication is a fundamental operation in machine learning, scientific computing, and other computational fields. As neural network workloads increasingly adopt sparse representations, there is a growing need for efficient Sparse GeMM. Our objective is to evaluate different Sparse GeMM approaches, implement them on an FPGA, and analyze their performance compared to the standard GeMM inner product algorithm. We will investigate whether sparse computation provides significant speedup and energy efficiency benefits over traditional computations when mapped to hardware accelerators. We seek to understand the architectural trade-offs associated with different Sparse GeMM algorithms.

II. Background and Motivation

Recent research in neural networks has shown that pruning 80-90% of the parameters retains nearly the same accuracy while significantly reducing storage requirements. However, the computational benefits of exploiting sparse matrices for acceleration remain underexplored. Sparse GeMM algorithms can optimize computation in pruned networks by reducing redundant and unnecessary operations, leading to more efficient execution. This project will examine different Sparse GeMM algorithms, evaluate their feasibility for FPGA-based acceleration, and assess their performance benefits.

III. Research Plan

The research steps are as follows:

1) Study and analyze various Sparse GeMM algorithms from literature:
   a) Inner Product
   b) Outer Product
   c) Gustavson's Algorithm
2) Implement and optimize these Sparse GeMM algorithms on an FPGA using High-Level Synthesis (HLS) in Vitis.
3) Compare the performance of these algorithms.

Architecture and Evaluation Methodology:

1) Implementation Flow: C++ →Vitis HLS →FPGA Deployment
2) Performance Metrics: Profiling tools will be used to evaluate execution time, resource utilization, and energy efficiency.
3) Workloads: Testing will be conducted on matrices with varying sparsity levels, including highly sparse, moderately sparse, and dense matrices.

IV. Expected Outcome

1) A comparative analysis of Sparse GeMM algorithms implemented on FPGA.
2) Performance evaluation demonstrating speedup, resource efficiency, and feasibility between different algorithms.
3) Insights into the trade-offs between different Sparse GeMM techniques in an FPGA setting.

V. Significance/Impact

Sparse matrices appear in many applications such as machine learning and AI, scientific computing, graph analytics, and medicine. Thus, if sparse matrix multiplication algorithms can improve the efficiency of working with sparse data, then these algorithms may boost productivity in each of these fields. In addition, they inspire further research on how to improve their performance and efficiency to become more useful in these various domains.

By studying and implementing Sparse GeMM algorithms, we hope to gain first-hand experience on how these algorithms exploit the sparsity of the data. We also hope to learn about the bottlenecks and limitations of these algorithms so that we can have an idea of future work on improving these algorithms. [1] [2] [3] [4] [5]

VI. Dataflows for Matrix Multiplication

Matrix multiplication computes C = A x B, where A is an M x K matrix, B is an K x N matrix, and C is an M x N matrix [1]. This operation takes three nested loops, two of which traverse the M and N dimensions of matrix C and one of which traverses the shared dimension K by matrices A and B. The order in which these loops are scheduled is the dataflow for the computation.

We study three basic dataflows: inner-product (IP), outer-product (OP), and Gustavson's Algorithm. These three dataflows have different levels of reuse for the inputs and outputs, so they each handle sparsity operations differently. The intersection operation is generally created by the coiteration loop, and values produced by this loop must be reduced.

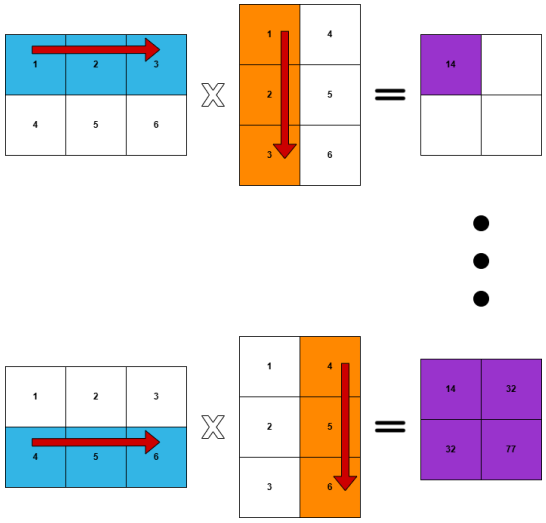

Fig. 1. IP Dataflow Example

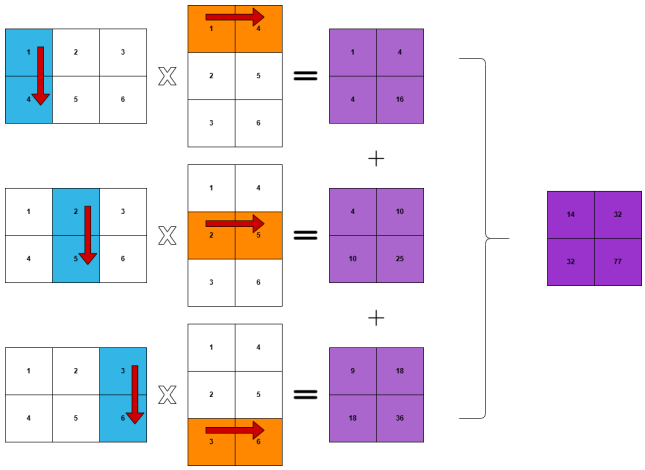

Fig. 2. OP Dataflow Example

A. Inner-Product

In the inner product dataflow, each element of the output matrix C is calculated as the dot product of a row of matrix A and a column from matrix B. For each position along the shared dimension (the number of elements in a row of the first matrix or a column of the second), multiply the corresponding element from the row of the first matrix by the element from the column of the second matrix. Sum each result from the multiplication step, and the final value is stored as the entry into the output Matrix. An example is shown in **Fig. 1**.

B. Outer-Product

In the outer product dataflow, the output matrix C is computed as a sum of rank-one matrices. In each iteration, take a column vector from the first matrix and a row vector from the second matrix. Multiply the column vector by the row vector to create a complete intermediate rank-one matrix. In this intermediate result, every entry is produced by multiplying an element from the column with an element from the row. Sum all intermediate rank-one matrices into the final product matrix. An example is shown in **Fig. 2**.

C. Gustavson's Algorithm

In Gustavson's algorithm, intermediate rows are calculated similar to outer product. Multiply each element in the current row of the first matrix by the corresponding row from the second matrix. Accumulate the products into corresponding columns of the resultant matrix. After all iterations, combine each complete row into the final output matrix. An example is shown in **Fig. 3**.

VII. Sparsity

Matrix multiplication in hardware accelerators must account for the nonuniform distribution of data in real-world applications. In this section, we examine two regimes of sparsity—mildly sparse and highly sparse matrices—each presenting unique challenges and opportunities for optimization.

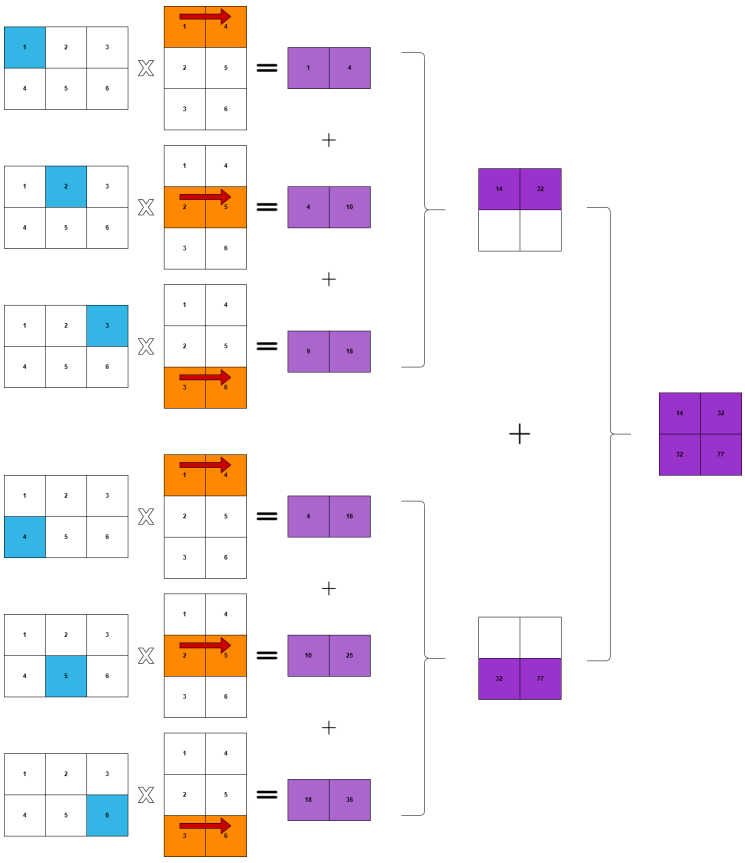

Fig. 3. Gustavson Dataflow Example

A. Mildly Sparse Matrices

Mildly sparse matrices contain about 1-90% nonzeros [1]. They are common in neural networks because of weight pruning and activation functions like ReLU, so accelerating matrix multiplication with mildly sparse matrices has merit in accelerating neural networks.

Using an IP-based dataflow, the matrix multiplication achieves element-level intersections and reductions. The matching nonzero elements in a row of A and column of B are paired together for multiplication, and all results are reduced with an accumulator to produce one output. This makes reductions simple, but there are many intersections to cover (M x N of them), and each of these intersections become more inefficient as sparsity grows as matching pairs become rare.

By contrast, an OP-based dataflow achieves matrix-level intersections and reductions. This reduces the number of intersections to cover (K of them), and each intersection is useful if there's at least one nonzero pair between a column of A and row of B. However, the reduction complexity is increased because more data movement is needed for the M x N partial results.

The Gustavson dataflow achieves a balance between the IP-based and OP-based dataflows, obtaining row-level intersections and reductions. Each nonzero element in a row of A is intersected with a row of B, and each intersection is useful if B's row contains at least one nonzero element. Like the OP-based dataflow, there are fewer intersections to cover (M of them). And although the reductions aren't as simple as in IP-based dataflow, they are simpler than the OP-based dataflow because accumulators are needed for just a row rather than an entire matrix. Thus, this dataflow strikes a nice balance for intersections and reductions among the three dataflows.

For mildly-sparse inputs, the IP dataflow can still be an efficient choice depending on the representation of the data. For example, bitvectors can be used to represent nonzero coordinates in a matrix, so performing an intersection can be done with a cheap AND operation. However, as sparsity increases, the IP dataflow becomes inefficient because matches become rare, and a different dataflow would be more efficient.

B. Highly Sparse Matrices

Highly sparse matrices contain less than 1% nonzeros [1]. These matrices are common in graph analytics and scientific computing, so accelerating these types of matrices have merit for these fields. However, accelerating highly sparse matrix multiplication is fundamentally different from dense or mildly sparse multiplication because there is low arithmetic intensity and data reuse. Thus, data movement is the key concern.

IP dataflow is inefficient for highly sparse matrix multiplication because of the low number of elements covered by each intersection. So the OP-based dataflow and Gustavson-based dataflow handle this type of matrix multiplication better.

Early accelerators tended to use OP-based dataflow. However, these accelerators didn't scale very well due to the large overhead of moving the partial results. Instead, modern accelerators like MatRaptor, Gamma, Flexagon, and Spada, used a Gustavson-based dataflow for handling highly sparse matrix multiplication. With this dataflow, these accelerators had simpler reductions and leveraged the structure in the matrices better. For example, nearby rows in the matrix A tended to have nonzeros in nearby indices, so these accelerators would exploit this property and obtain greater data reuse out of matrix B.

In general, highly sparse matrix multiplication lacks arithmetic intensity and thus compute throughput. So modern accelerators instead focus on reducing the amount of data transfer to accelerate this application. To this end, they focus on storage, such as caches and buffers. They also feature flexible interconnects, such as crossbars, so that the few matches are paired efficiently. However, these structures consume a significant amount of area, leaving less area for the PEs. Thus, these accelerators would struggle with denser matrices because of their lack of compute hardware. More recent accelerators try to support multiple dataflows and balance their hardware structures so that they can handle highly sparse matrices efficiently without trading off too much performance with the denser matrices.

VIII. Preliminary Evaluation

A. Methodology

A Python script was written to perform preliminary behavioral simulations of the dataflows. This allowed us to gain a better understanding of the data flows that we plan to implement. We initially implemented the three dataflows naively, operating on standard 2D matrices. Next we converted the 2D matrices into CSR format, and fed those into Gustavson's algorithm.

TABLE I
Preliminary simulation results

Dataflow Type	*totalOPs*
Inner Product	1000
Outer Product	1000
Gustavson's	52
Inner Product with CSR x CSC formatted inputs	192
Outer Product with CSC x CSR formatted inputs	63

B. Results

To gather performance characteristics, a counter variable *totalOPs* was created for each dataflow function. This counter increments every time an operation is performed on the input matrices; line 15 in **Fig. 4** demonstrates this process in the Inner Product function. **Table 1** shows the results of our preliminary simulations. Gustavson's algorithm performed 19.2x better than Inner and Outer Product on 2D matrices. Notably, Inner and Outer Product also gain a large performance increase when using sparsity aware inputs; 5.2x and 15.9x respectively. **Fig. 5** demonstrates Inner Product using CSR x CSC formatted inputs.

C. Analysis

To be clear, these numbers are somewhat deceptive. Naively, Inner and Outer Product iterate over both indices in their entirety, while Gustavson's algorithm iterates over nonezero entries only. The extra overhead used for checking nonzero entries isn't accurately represented in the script. Additionally, Gustavson's is very suitable for CSR format inputs as it is a row-based accumulation approach, while Inner and Outer Product both operate on standard 2D matrices with no further optimizations. However, we believe the script still demonstrates the fact that Gustavson's is a good algorithm for SpMSpM calculations. We plan to find an accurate overhead to fill in to the calculations, as then we would be able to demonstrate the level of sparsity needed for Gustavson's to become cost effective.

```
def inner_product(A, B):
m = len(A)          # number of rows in A
p = len(A[0])       # common dimension
n = len(B[0])       # number of columns in
    B
totalOPs = 0;

# Initialize result matrix with zeros.
C = [[0 for _ in range(n)] for _ in range(
    m)]

for i in range(m):
    for j in range(n):
        total = 0
        for k in range(p):
            total += A[i][k] * B[k][j]
            totalOPs += 1;
        C[i][j] = total

print("inner product number of operations:
    ", totalOPs)
return C
```

Fig. 4. Python code for Inner Product.

IX. C++ Simulator and HLS Accelerator Implementation

After completing our initial Python simulations, we implemented a full-system C++ simulator to better model accelerator behavior and simulate cycle-level performance before translation to hardware. The simulator includes:

- A **Simulator** class that coordinates data transfer and computation.
- A **CPU** module for matrix preprocessing and communication.
- Accelerators for **Inner Product**, **Outer Product**, and **TRGT** algorithms.
- A CSRMatrix struct for sparse data representation.

Once validated, each accelerator was translated to C++ HLS code using Vitis HLS. We constrained all designs to 8 multipliers and compared them with a dense baseline to

```
def inner_product_csr_csc(csrA, cscB):
dataA, indicesA, indptrA, shapeA,
    input_conversion_overheadA = csrA
dataB, indicesB, indptrB, shapeB,
    input_conversion_overheadB = cscB
conversion_overhead =
    input_conversion_overheadA +
    input_conversion_overheadB
totalOPs = 0
m, kA = shapeA
kB, n = shapeB

if kA != kB:
    raise ValueError("Inner dimensions do
        not match for multiplication.")

# Initialize the result matrix as a dense
    array
C = [[0 for _ in range(n)] for _ in range(
    m)]

# For each row of A (in CSR format)
for i in range(m):
    row_start = indptrA[i]
    row_end = indptrA[i + 1]
    row_indices = indicesA[row_start:
        row_end]
    row_values = dataA[row_start:row_end]

    # For each column of B (in CSC format)
    for j in range(n):
        col_start = indptrB[j]
        col_end = indptrB[j + 1]
        col_indices = indicesB[col_start:
            col_end]
        col_values = dataB[col_start:
            col_end]

        # Compute the dot product of A[i
            ,:] and B[:,j]
        dot = 0
        a_ptr, b_ptr = 0, 0
        while a_ptr < len(row_indices) and
             b_ptr < len(col_indices):
            a_index = row_indices[a_ptr]
            b_index = col_indices[b_ptr]
            totalOPs += 1;
            if a_index == b_index:
                dot += row_values[a_ptr] *
                     col_values[b_ptr]
                a_ptr += 1
                b_ptr += 1
                totalOPs +=1;
            elif a_index < b_index:
                a_ptr += 1
            else:
                b_ptr += 1

        C[i][j] = dot

print("inner product with CSRxCSC inputs
    number of operations:", totalOPs+
    conversion_overhead, "(", totalOPs, "
    +", conversion_overhead, ")")
return C
```

Fig. 5. Python code for Inner Product.

analyze how each dataflow handled varying matrix sparsity levels.

A. Vitis HLS: Baseline

In this section, we describe the translation of these Python algorithms to corresponding accelerators written in C++ and compiled with Vitis HLS. In particular, we design one accelerator for each of the dataflows and analyze their performance and resource utilization.

To establish a baseline for our sparse GeMM accelerators, we implemented a dense GeMM accelerator. This accelerator used standard dense representations of the matrices during data transfer and used an outer-product-based flow to perform the computation. We limit this design to eight multiplier units, which we enforce across our sparse GeMM implementations to get a fair analysis of the sparsity-aware optimizations, which would no longer be influenced by the computation resources available. Moreover, we limit the matrices to be fixed at 32x32 elements of unsigned integers to limit the scope of this project to computation-based optimizations involving sparsity.

Table II shows the performance and resource utilization of this Dense GeMM baseline. The design takes about 1000 cycles to load data, 4098 cycles to perform the main computation, and 1000 cycles to send the data back to memory. This computation time is optimal for dense matrices in our test case. Assuming that it takes 1 cycle to perform a multiply-and-accumulate operation, it would take 32,768 cycles to perform the matrix multiply computation with 32x32 matrices and just one MAC unit because matrix multiplication is an $O(N^3)$ operation. So with 8 MAC units, we bring the computation down to just 32,768 / 8 = 4,096 cycles, which is virtually our computation cycle count.

In terms of resource utilization, the design consumes 24 DSPs because the tool uses 3 DSPs to construct a multiplier for 32-bit numbers. Moreover, the design uses 32 BRAMs, 2490 FFs, and 5940 LUTs to store the data and facilitate the computation. These resource and performance metrics will serve as the reference point to evaluate the sparse accelerators.

TABLE II
DENSE GEMM (BASELINE) METRICS

Total Cycles	6317
Compute Cycles	4098
BRAM	32
DSP	24
FF	2490
LUT	5940

B. Inner Product Accelerator Design

In this section, we discuss the design of the inner-product accelerator as implemented in Vitis HLS. Recall that in an inner-product flow, the dot product of a row of A and a column of B is taken to fully compute one element of C before moving on to the subsequent elements of C. In order to take advantage of the sparsity in this flow and reduce the computation, we note that only nonzero pairs between the row of A and column of B contribute toward the output. With this intuition, the first iteration of the design started from the inner-product Python implementation that used the CSR and CSC data formats.

In principle, these compressed formats allow us to quickly find the nonzero values across a given row (in the case of CSR) or a given column (in the case of CSC) and its corresponding index. However, the index-matching paradigm of the inner-product dataflow makes using these formats challenging. In other words, given that row A has a nonzero at index k, it is hard to determine whether column B has a nonzero value at that same index without performing a linear search, which is expensive. Moreover, because this search is non-trivial with these formats, the parallelization of the dot product also becomes challenging.

Ultimately, we converted the input matrices back to dense formats and used a different method of index matching in the final inner-product design. First, we used bit vectors to quickly highlight the nonzero values across the rows of A and columns of B. Then, given a row of A and a column of B, we performed the AND of the corresponding bit vectors, which quickly identifies the nonzero pairs. We then passed this information to an intersection unit inspired by the Trapezoid design, which takes the row of A and column of B and shifts them such that the nonzero pairs align to one side of the resulting vectors [1]. Finally, based on the number of pairs found, the resulting vectors are streamed to the dense MAC units. With this intersection unit, we simplify the control logic used to perform the index matching in our inner-product accelerator, saving us some cycles.

Moreover, since we use bit vectors to represent nonzero elements in the matrices A and B, we can quickly identify rows of A or columns of B that have all zeroes. Thus, we optimize our inner-product design further by keeping track and iterating through only nonzero rows of A and nonzero columns of B, which saves more cycles as sparsity increases.

These two sparsity-aware optimizations are the main computation optimizations used in the design of our inner-product accelerator. However, we also optimize data movement by using the coordinate format to store the matrix C. Since the inner-product dataflow only works on one element of C per loop of the K index, the outputs can easily be streamed into our coordinate data structure. Moreover, since this data structure only contains nonzero elements, we save cycles in data movement (given that matrix C is sparse) as we write the nonzero results back to DRAM.

With these three optimizations, our final inner-product design performed much better than the initial design. However, it only starts to outperform the baseline in performance at lower densities than 15%, given in figures 6 and 7. This performance was unexpected because inner-product sparse GeMM accelerators from prior works generally had higher performance in the mildly-sparse ranges. But we believe that this lower performance was due to the slightly higher cycle count in the innermost loop due to the control logic responsible for allocating the 8 multipliers and performing the accumulation. Designing this section to be more stream-based rather than

iterative could have simplified the control logic, allowing us to reclaim the extra cycles and regain the performance. However, we were unable to incorporate this update in the final design.

The table below lists the resource utilization by our inner-product design. The design consumes about 2x more BRAMs and 4x more FFs and LUTs compared to the baseline.

TABLE III
INNER-PRODUCT ACCELERATOR RESOURCE UTILIZATION

BRAM	72
DSP	24
FF	7067
LUT	23341

C. Outer Product Accelerator Design

In this section, we discuss the outer-product accelerator as implemented in Vitis HLS. Recall that in the outer-product dataflow, a column of A and a row of B are taken to produce partial results for the full matrix C. In particular, every element in the column of A is crossed with each element in the row of B to produce each element in the partial matrix of C. Thus, unlike in the inner-product dataflow, index matching is not an issue in this design. Therefore, it is easier to incorporate the CSC format for A and CSR format for B as in the Python implementation and directly work with that to produce the final matrix C.

However, while working with both of these formats was trivial in a sequential manner, they were surprisingly difficult to parallelize. While we could quickly determine when a column or row would begin or end in the CSR/CSC formats, it was difficult to effectively partition the data structures that stored these input matrices in these formats, which led to resource utilization conflicts.

To overcome this issue, we needed a data format that was regular so that it was easy to partition, but it still needed to take advantage of the sparsity. Thus, we ended up using the ELLPACK format to store the input matrices. Effectively, for a given k index, all nonzero elements in the m direction for A or n direction for B are shifted to one side of the matrix, and the m and n indices are saved. This data format makes it regular to access the input matrices along the k dimension while offering compression in the m and n dimensions. Taking advantage of this data format, we split the 8 MAC units across the k dimension, enabling parallelism in our outer product design. In other words, each MAC unit is responsible for a set of columns of A and row of B, producing partial matrices for each. Once all partial matrices are produced, they are all reduced via accumulation to produce the final matrix C before they are sent to DRAM. This data format optimization and parallelism are the key performance optimizations for our outer product design.

Although the outer-product design should have significantly higher resource utilization due to the large arrays used to store each partial matrix, the resource utilization was surprisingly small. The table below depicts this design's resources. This design only uses 20% more BRAMs and 2x more FFs and LUTs than the baseline, beating our inner-product design. Moreover, this accelerator performs the best of all the designs as depicted in figures 6 and 7. At high density, the accelerator performs on par with the baseline. And at densities lower than 90%, the accelerator starts to beat the baseline. Overall, we are content with the results of this outer-product accelerator.

TABLE IV
OUTER-PRODUCT ACCELERATOR RESOURCE UTILIZATION

BRAM	40
DSP	24
FF	4672
LUT	9122

D. TRGT Accelerator Design

The TRGT (Tile-Reordered Gustavson-Type) accelerator implements the Trapezoid-style sparse matrix multiplication algorithm [1], where the output matrix C is computed tile-by-tile using a fine-grained 5D tiling structure. Each tile corresponds to a partitioned region in $C[M2][M1][M0][N1][N0]$, and each compute unit processes a specific tile in a pipeline of three stages: loading, computing, and storing.

Our HLS implementation incorporates the following key design choices and optimizations:

- **Tiled Matrix Layout and Dataflow:** Matrix A is partitioned into 3D tiles $[M2][M1][M0][K]$, matrix B into $[N1][K][N0]$, and the output C into $[M2][M1][M0][N0]$. We use a `tile_id` loop over $M2 \times N1$, and apply HLS `#pragma DATAFLOW` to overlap the execution of each tile's stages: loading B, computing tile values, and writing results back to memory.
- **Bitvector-based Sparsity Filtering:** We generate bitvectors for each row of A and column of B using `ap_uint<K>`, indicating nonzero locations. An AND operation between these vectors quickly identifies which multiplications are necessary, enabling early exit for entirely sparse positions.
- **Index Compaction via k-list Compression:** For each output index $(m1, m0, n0)$, active k-indices are compacted into a `k_list[]` array, which stores only the positions where both A and B are nonzero. This prevents unnecessary MACs on zero-valued elements.
- **Unrolled MAC Accumulation:** The `mac_accumulate()` function computes partial sums across active k values. It uses an unrolled loop to process up to 8 entries in parallel, supported by a fully partitioned `partial[8]` buffer, effectively simulating a wide SIMD lane.
- **Buffer Partitioning for Parallelism:** Buffers such as `A_buf`, `B_buf[K][N0]`, `C_tile[M1][M0][N0]`, and `k_list` are fully partitioned to support concurrent access across loops. This maximizes instruction-level parallelism and reduces pipeline stalls.
- **Tile-Level Modularity:** The accelerator separates functionality into three HLS modules—`load_B_buf`,

`compute_tile`, and `store_C_tile`—for clarity, reuse, and synthesis friendliness. Each module is designed for pipeline execution with `#pragma HLS INLINE off` and loop-level `#pragma HLS PIPELINE` to ensure throughput is maximized at each stage.

This implementation captures the essence of the Trapezoid TRGT flow while enriching it with sparse-aware optimizations and hardware-friendly restructuring. These features enable efficient MAC utilization and minimize computation on zero elements, making it a suitable architecture for high-sparsity workloads.

E. HLS Performance Evaluation

We measured HLS cycle counts under two sparsity regimes:

1) **DxD to SpxSp**: Both A and B are sparse.
2) **DxD to DxSp**: Only B is sparse.

The figures below show the total latency (in cycles) across different sparsity levels.

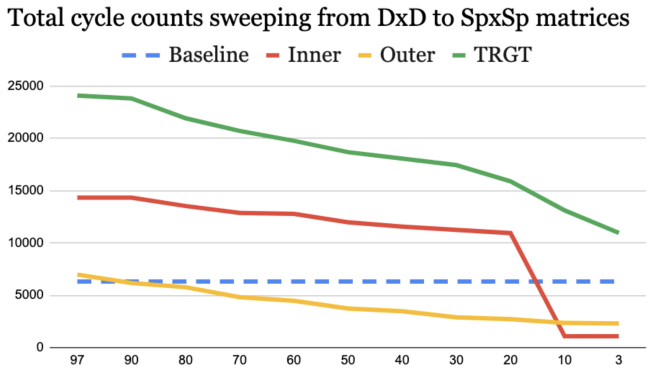

Fig. 6. HLS Cycle Count vs. Sparsity (DxD to SpxSp)

All three sparse accelerators show decreasing cycle counts as matrix density decreases, highlighting the benefit of exploiting sparsity. The baseline dense implementation remains flat at 6317 cycles. Outer Product consistently outperforms both Inner Product and TRGT across all density levels, showing strong sparsity-awareness and better data reuse. Inner Product improves noticeably at high sparsity (10%) due to its bitvector-based filtering, but TRGT remains the slowest across all densities. Despite being designed for sparse workloads, the TRGT implementation suffers from control and data movement overheads that prevent it from achieving competitive performance.

When only matrix B becomes sparse, all three dataflows again benefit from reduced cycle counts. Outer Product remains the best-performing method across all sparsity levels. Inner Product improves steadily as sparsity increases, and TRGT, while showing reduced latency, still underperforms compared to both. At no point does TRGT outperform the other dataflows, reinforcing the presence of inefficiencies in its implementation. These results suggest that the current TRGT design does not fully capitalize on its theoretical advantages, especially in partially sparse workloads.

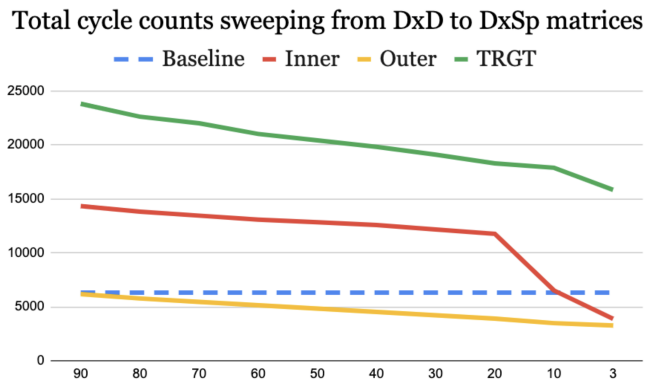

Fig. 7. HLS Cycle Count vs. Sparsity (DxD to DxSp)

X. Analysis

A. Python simulation

A direct performance comparison between Gustavson's algorithm and the inner and outer product approaches may lead to misleading conclusions if the inherent properties of the input formats are not taken into consideration. Gustavson's algorithm and other similar row accumulation based dataflows are typically implemented using CSR format, which enables it to operate exclusively on nonzero elements and to accumulate results efficiently on a row by row basis. This built in sparsity awareness gives Gustavson's method a significant performance advantage in sparse matrix multiplication settings.

In contrast, the inner and outer product dataflows are conventionally designed for dense matrix inputs, lacking inherent mechanisms to exploit sparsity. Their basic formulations assume that every element, including zeros, is processed during the multiplication and reduction phases. If these approaches are used without modifications, the comparisons with Gustavson's algorithm become skewed, as much of Gustavson's efficiency is attributable to the sparse format rather than differences in the underlying multiplication strategy.

Moreover, when the inner and outer product methods are adapted to accept sparsity aware inputs (CSC and CSR), their performance improves considerably. In such implementations, the methods can similarly bypass unnecessary operations associated with zero entries. This adaptation brings their efficiency closer to that of Gustavson's algorithm in terms of throughput and resource utilization. Consequently, it becomes clear that the quality of the input representation can have a profound impact on performance.

The key takeaway from this is that properly formatted inputs, which exploit inherent sparsity, are perhaps more critical to performance than the choice of dataflow alone. While Gustavson's algorithm naturally capitalizes on sparse input representations, the inner and outer product approaches can achieve comparable gains when extended with CSR/CSC

handling. In hardware accelerator design for sparse matrix multiplication, emphasis should therefore be placed on optimizing input data formats and ensuring that the processing architecture effectively leverages the sparsity properties, regardless of the specific dataflow method used.

B. HLS

1) **inner**: While the inner-product accelerator should have outperformed the baseline for the mildly sparse matrices, it ended up having a higher latency overall. This was likely due to the slightly higher cycle count in the innermost loop because of the control logic. To reduce this count, a dataflow approach could be taken, which could save a few cycles and yield the expected results.
2) **outer**: The outer-product accelerator yielded the best performance out of all the designs, consistently beating the baseline. But optimizations could be made to this accelerator to reduce BRAM consumption. Rather than storing all the partial matrices before the merge, prior works often have merging units to reduce the number of partial matrices needed. An optimization like this could improve the area of this design.
3) **trgt**: The TRGT accelerator, based on a tile-reordered Gustavson-style sparse matrix multiplication, was expected to outperform Inner and Outer Product in high-sparsity regimes due to its structured tiling, selective computation, and bitvector-based filtering. However, our HLS evaluation shows that TRGT consistently underperforms both Inner and Outer Product designs across all tested sparsity levels. To reduce latency, there exist opportunity for further optimizations:
    - Introduce double buffering between tile stages to enable pipelined tile processing.
    - Dynamically allocate MAC units based on active nonzero count.
    - Parallelize across multiple tiles using shared on-chip buffers.

XI. Conclusion

This project has provided valuable insights into the design and implementation of Sparse GeMM accelerators. By evaluating and comparing various dataflow approaches (IP, OP, Gustavson), we were able to recognize important considerations to increase accelerator performance. Most important are the format of the input matrices. Our analysis demonstrates that while Gustavson's algorithm inherently benefits from sparse representations, similar gains can be realized in IP and OP when they adapt to leverage the same structured input formats. Furthermore, our HLS evaluation across multiple sparsity levels confirms that properly formatted inputs and effective memory management are critical to balancing compute throughput and data movement, regardless of the underlying dataflow.

The results of our project highlights the importance of aligning algorithmic strategies with hardware constraints and input data characteristics to optimize both speed and energy efficiency. In summary, while different dataflows are most suitable for different sparsities, a careful integration of sparsity-aware representations and flexible dataflows is key to an efficient and performant accelerator.

References

[1] Y. Yang, J. S. Emer, and D. Sanchez, "Trapezoid: A versatile accelerator for dense and sparse matrix multiplications," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 931–945.

[2] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 766–780.

[3] Y. Nagahara, J. Yan, K. Kawamura, M. Motomura, and T. Van Chu, "Sparse-sparse matrix multiplication accelerator on fpga featuring distribute-merge product dataflow," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 785–791.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[5] J. S. E. Guowei Zhang, Nithya Attaluri and D. Sanchez, "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *in Porceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021, pp. 687–701.