2023 Design, Automation & Test in Europe Conference (DATE 2023) – Best Paper Award Candidate

Accelerating Gustavson-based SpMM on Embedded FPGAs with Element-wise Parallelism and Access Pattern-aware Caches

Shiqing Li and Weichen Liu
School of Computer Science and Engineering, Nanyang Technological University, Singapore
{shiqing.li, liu}@ntu.edu.sg

***Abstract*—The Gustavson's algorithm (i.e., the row-wise product algorithm) shows its potential as the backbone algorithm for sparse matrix-matrix multiplication (SpMM) on hardware accelerators. However, it still suffers from irregular memory accesses and thus its performance is bounded by the off-chip memory traffic. Previous works mainly focus on high bandwidth memory-based architectures and are not suitable for embedded FPGAs with traditional DDR. In this work, we propose an efficient Gustavson-based SpMM accelerator on embedded FPGAs with element-wise parallelism and access pattern-aware caches. First of all, we analyze the parallelism of the Gustavson's algorithm and propose to perform the algorithm with element-wise parallelism, which reduces the idle time of processing elements caused by synchronization. Further, we show a counter-intuitive example that the traditional cache leads to worse performance. Then, we propose a novel access pattern-aware cache scheme called SpCache, which provides quick responses to reduce bank conflicts caused by irregular memory accesses and combines streaming and caching to handle requests that access ordered elements of unpredictable length. Finally, we conduct experiments on the Xilinx Zynq-UltraScale ZCU106 platform with a set of benchmarks from the SuiteSparse matrix collection. The experimental results show that the proposed design achieves an average 1.62x performance speedup compared to the baseline.**

***Index Terms*—SpMM, FPGA, Gustavson, Cache**

I. Introduction

Sparse matrix-matrix multiplication (SpMM) is of paramount importance in multiple domains such as data analytics, graph processing, and scientific computing. Specifically, SpMM is a vital kernel in graph contraction [1], recursive formulations of all-pairs shortest-paths algorithms [2], colored intersection searching [3], and molecular dynamics [4]. SpMM operates on sparse matrices which are stored using sparse formats. Although sparse formats reduce the memory footprint by omitting all the zeros, these formats incur indirect and hence irregular memory accesses. Consequently, the off-chip memory traffic becomes the main bottleneck of SpMM especially on embedded FPGAs.

Different algorithms of SpMM have different on-chip memory requirements and off-chip memory traffic. Recently, the Gustavson's algorithm [5] has shown its potential to be the backbone algorithm for SpMM on hardware accelerators [6] [7] [8] [9] with low on-chip memory requirement and less off-chip memory traffic. In the algorithm, each element A_{ik} in the i_{th} row of A (we refer to the first input matrix as A) is multiplied with the k_{th} row of B (we refer to the second input matrix as B) to generate one partial result of the i_{th} row of C (we refer to the output matrix as C) and all the partial results are merged to get the final i_{th} row of C. An important advantage of the Gustavson's algorithm is that A, B, and C are in row-major order. However, the inherent row-wise parallelism leads to uneven workloads among processing elements (PEs) but the results of PEs shall be written back in order. The synchronization among PEs stalls some of them and degrades the performance. Besides, the performance is still bounded by off-chip memory traffic.

Streaming off-chip memory accesses and caching data on-chip are widely used to reduce off-chip memory traffic. Streaming is the most efficient way to transfer data between the off-chip DDR and the FPGA chip, which sequentially accesses contiguous elements. Besides, caching data on-chip can avoid long-delay off-chip memory accesses. Following the Gustavson's algorithm, rows of B can be reused for elements of A with the same column indices. However, there are two challenges. The first one is that irregular memory accesses lead to bank conflicts and the long off-chip memory access latency further aggravates the conflicts when requests miss the cache. Secondly, a sequence of ordered elements in each row of B is accessed. However, the length of them is unpredictable. These elements usually fall into different cache banks and multiple sequential requests are required. However, these standalone requests issue multiple standalone off-chip memory accesses if they miss the cache, which fails to utilize the streaming fashion and increases the off-chip memory access latency instead. In addition, they also aggravate bank conflicts.

To solve these challenges, in this work, we propose an efficient Gustavson-based SpMM accelerator on embedded FPGAs with element-wise parallelism and access pattern-aware caches. The main contributions are as follows:

- We analyze the parallelism of the Gustavson's algorithm and propose to perform the algorithm with element-wise parallelism. Further, we show a counter-intuitive example that the traditional cache leads to worse performance.
- We propose an efficient accelerator for SpMM on embedded FPGAs. A novel access pattern-aware cache scheme called SpCache is proposed to reduce bank conflicts caused by irregular memory accesses and handle requests that access ordered elements of unpredictable length.

978-3-9819263-7-8/DATE23/© 2023 EDAA

- We conduct experiments on the Xilinx ZCU106 platform with a set of benchmarks from the SuiteSparse matrix collection [10]. The results show that our proposed design achieves an average 1.62x performance speedup.

II. Background and Related Work

A. Sparse Matrix-matrix Multiplication

Sparse matrix-matrix multiplication (SpMM) refers to the multiplication between a sparse matrix A and a sparse matrix B to generate a sparse matrix C. In general, sparse matrices are stored in compressed formats to lower the memory requirement. Compressed sparse row (CSR) is one of the most widely used formats and sparse matrices are in CSR format in this work. As shown in Fig. 1, CSR format mainly includes three arrays *rptr*, *val*, and *cid*. Instead of holding all the row indices, the CSR format only holds the index of each row's first element in *rptr*. For example, the index of the third row's first element (i.e., element 5) is 4 in Fig. 1 and thus $rptr[2] = 4$. In CSR, elements in the i_{th} row are fetched in two steps. $rptr[i]$ and $rptr[i{+}1]$ are accessed first to get the index range (i.e., $[rptr[i], rptr[i{+}1])$) in *val* and *cid* and then the corresponding values and column indices are accessed. Note that elements in each row are sorted by column indices.

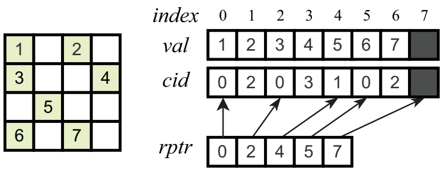
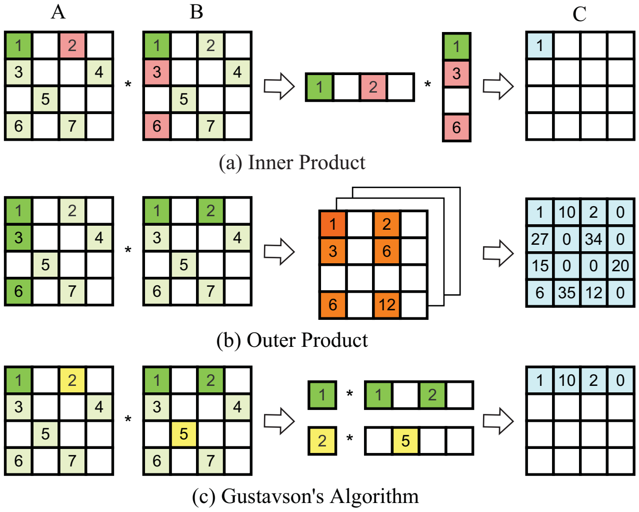


Fig. 1: CSR Format

In the following paper, we will use A_i to represent the i_{th} row of the matrix A and A_{ij} to represent the nonzero element of the matrix A whose row index and column index is *i* and *j* separately. The same notation is also applied to B and C.



(a) Inner Product

(b) Outer Product

(c) Gustavson's Algorithm

Fig. 2: SpMM's Algorithms

There are mainly three algorithms to perform SpMM as shown in Fig. 2. First of all, the inner product algorithm multiplies one row of A and one column of B to get one element of C. Since the sparse matrices are stored using compressed formats, index matching is required. However, the memory bandwidth is wasted by transferring mismatched elements. As shown in Fig. 2(a), only one multiplication is performed and element 2, 3, and 5 are transferred but not used. Secondly, the outer product algorithm shown in Fig. 2(b) multiplies the i_{th} column of A and the i_{th} row of B to get one partial result matrix and merge all the partial matrices to get the final result. Compared to the inner product algorithm, the outer product algorithm doesn't require index matching. However, partial results are written to the off-chip DDR and then read to the FPGA chip which consumes lots of memory bandwidth and on-chip memories. Consequently, it's not suitable for embedded FPGAs. Besides, A and B use different formats in both the inner product algorithm and the outer product algorithm. For example, A is in row-major order and B is in column-major order in the inner product algorithm. The third algorithm is called the Gustavson's algorithm [5] (i.e., the row-wise product algorithm). It multiplies each element A_{ik} in A_i with ordered elements of B_k to generate a partial result of C_i and merges all the partial results to get the final result. As shown in the figure, element A_{00} (i.e., the element whose value is 1 in Fig. 2(c)) is multiplied with elements in B_0 and the partial results $[(1, 0), (2, 2)]$ and $[(10, 1)]$ are merged to get C_0. The Gustavson's algorithm has three advantages: (1) index matching is not required, (2) the on-chip memory requirement is low since it only buffers one row of C, and 3) A, B, and C are in row-major order and use a consistent format. With these three advantages, the Gustavson's algorithm is a promising backbone for SpMM on embedded FPGAs.

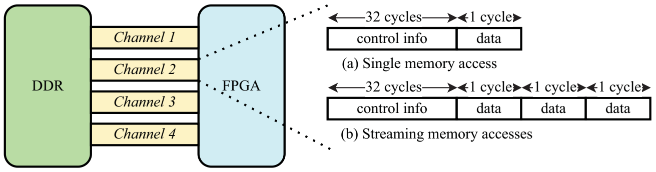


Fig. 3: ZCU106's Off-chip Memory Access

B. Off-chip Memory Access on Embedded FPGAs

The on-chip memory is usually limited on embedded FPGAs and large-volume data (e.g., sparse matrices) are stored in the off-chip DDR. For example, the Xilinx Zynq-UltraScale ZCU106 platform only has around 4.7MB on-chip memories and four channels between the off-chip DDR and the FPGA chip as shown in Fig. 3. Data transactions in channels follow the Advanced eXtensible Interface (AXI) protocol and there are two important manners. As shown in Fig. 3(a), the first one is single memory access. It includes a 32-cycle control information exchange phase and a 1-cycle data (up to 128 bits) exchange phase. However, if the target elements are contiguous, only one control information exchange phase is required. This transaction mode is called streaming memory access as shown in Fig. 3(b), which is the most efficient way to transfer data from the off-chip DDR to the FPGA chip. Besides, data transactions in different channels are independent.

C. Related Work

SpMM gains much attention on hardware designs. [11] conducts design space exploration for SpMM using the inner product algorithm. It successfully finds the trade-off between performance and energy consumption. The OuterSPACE [12] and the SpArch [13] targets the outer product algorithm. [13] further reduces the number of partial result matrices by condensing matrices. However, these works suffer from the drawbacks of the algorithms. Besides, the MapRaptor [6] proposes an accelerator targeting the Gustavson's algorithm. However, it fails to explore the reuse of B's rows. The InnerSP [7] and the GAMMA [9] propose cache-based architectures to explore the reuse of B's rows. However, they target the high bandwidth memory-based ASIC accelerators and are not suitable for embedded FPGAs with traditional DDR.

To the best of our knowledge, this work is the first work which accelerates Gustavson-based sparse matrix-matrix multiplication on embedded FPGAs with element-wise parallelism and access pattern-aware caches. Although this work targets the Xilinx ZCU106 platform, the proposed ideas can be applied to other platforms with similar memory access features.

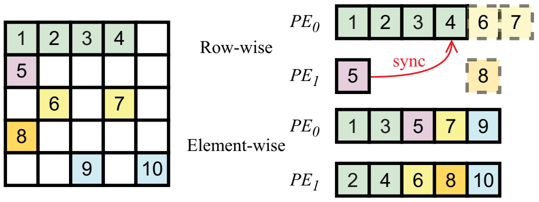


Fig. 4: Gustavson's Parallelism Analysis

III. Parallelism analysis & Motivation

In this section, we first analyze the parallelism of the Gustavson's algorithm and propose to perform it with element-wise parallelism. Then, we show our motivational example that traditional cache leads to worse performance.

A. Gustavson's Parallelism

The Gustavson's algorithm is also known as the row-wise product algorithm. For each row i of A, it mainly consists of multiplications between A_{ij}'s value and B_j's values and merging of ordered partial results. Following the algorithm, rows of A are assigned to available processing elements (PEs) and each PE accepts the next row after finishing the current row. As discussed in Section. II-A, one of the advantages of the Gustavson's algorithm is that A, B, and C are in a consistent format (i.e., CSR). To ensure the correctness of C, PEs shall return rows in order. As shown in Fig. 4, assume that there are two PEs. At the beginning, PE_0 and PE_1 accept A_0 and A_1 separately. However, the workload of PE_0 is much bigger than PE_1 and thus PE_1 doesn't accept a new row until PE_0 finishes. Although we can allocate buffers to hold results and eliminate synchronizations among PEs, it's hard to determine the buffer size and big buffers waste the limited on-chip memories.

In this work, we propose to perform the Gustavson's algorithm with element-wise parallelism as shown in Fig. 4. With element-wise parallelism, A's elements are distributed to available PEs. For elements in the current row, PEs merge the partial results in their local buffer like using row-wise parallelism. When PEs receive elements of the next row, they push buffered partial results to a final merger. The final merger merges partial results of PEs and its latency can be covered by the processing of the next row. If the next row is pretty short which is a rare case in real benchmarks, the final merger's latency may not be fully covered by it. However, compared to row-wise parallelism, element-wise parallelism can significantly reduce PEs' idle time caused by synchronization. Besides, since we sequentially access A's elements in row-major order, memory accesses to A's three arrays are in the streaming memory access fashion. As shown in Fig. 4, PEs are much busier than using row-wise parallelism.

With the element-wise parallelism, PEs' idle time caused by synchronization is significantly reduced. However, it still suffers from off-chip memory traffic. The Gustavson's algorithm provides reusability for rows of B and caches can be deployed to avoid long-delay off-chip memory accesses. However, access patterns in the algorithm shall be considered. First of all, irregular memory accesses of rows of B lead to bank conflicts and the long cache miss delay aggravates the conflicts. Secondly, a sequence of ordered elements of unpredictable length (i.e., B_j's elements) is required to ensure a smooth merge between ordered lists. However, it's hard to hold these elements in one cacheline. In addition, multiple sequential cache requests aggravate bank conflicts and issue multiple single off-chip accesses if they miss the cache.

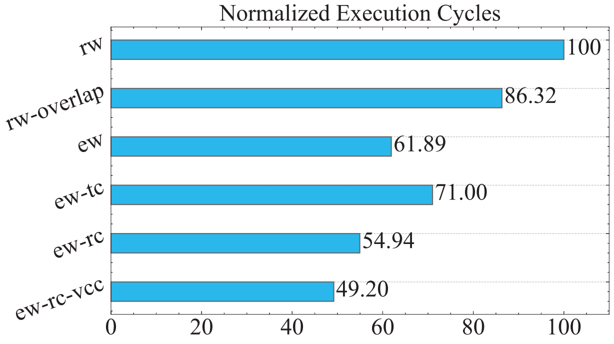


Fig. 5: The Motivational Example

B. Motivational Example

We use a motivational example to show the performance speedup of the proposed design. We use `poisson3Da` from the SuiteSparse matrix collection [10] and conduct experiments on the Xilinx ZCU106. `poisson3Da` is a 13514×13514 matrix with 352762 nonzero elements. Four elements of A are processed simultaneously and caches use the same configuration, which is a 16-way 4-bank cache with the least-recently-used (LRU) replacement strategy. Since the access pattern of B's *val* and *cid* is different from that of B's *rptr*, we refer to caches for *rptr* as *rcache* and caches for *val* and *cid* as *vccache*. Overall, we refer to row-wise parallelism, element-wise parallelism, traditional cache, rcache, and vccache as *rw, ew, tc, rc, vcc* in short. Here, the traditional cache is deployed for *rptr*. As shown in Fig. 5, our proposed design (i.e., *ew-rc-vcc*) achieves 2.03x performance speedup. Meanwhile, a counter-intuitive example is that the performance degrades if we use traditional cache (i.e., *ew-tc* compared to *ew*). Instead, the rcache using the proposed SpCache scheme achieves 1.12x performance speedup (i.e., *ew-rc* compared to *ew*). Further, we achieve 1.12x more performance speedup with the *vccache*. The more detailed performance analysis is conducted in Section V-B. Besides, we also perform one additional experiment (*rw-overlap*) that a buffer is deployed to hold results for PEs using row-wise parallelism. The buffer can hold the results of several rows in the experiment. As shown in Fig. 5, although it achieves 1.15x speedup compared to the pure row-wise parallelism (i.e., *rw*), it still takes 39.47% more cycles compared to the *ew*.

IV. HARDWARE DESIGN

A. *Overview*

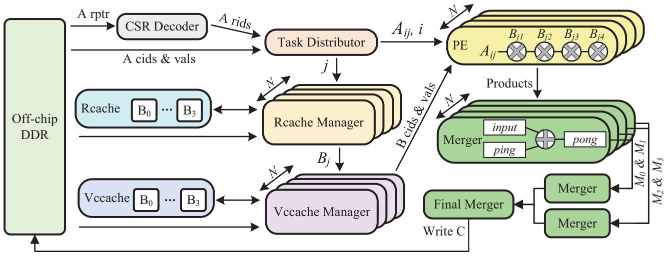
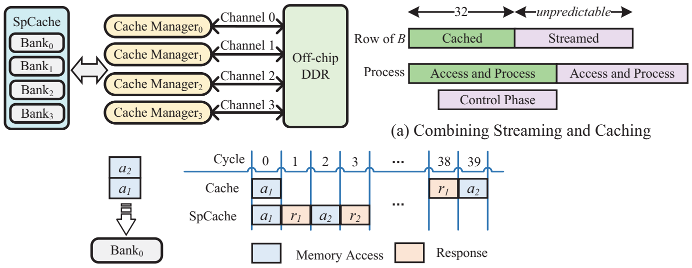


Fig. 6: Hardware Overview

Processing elements of A requires a SpCache of *rptr*, a SpCache of *val* and *cid*, PEs, and mergers. Since these components and interconnects between them consume lots of hardware resources, this design processes four elements simultaneously (i.e., N in Fig. 6 is 4) and each element occupies one memory channel on the Xilinx ZCU106. As shown in Fig. 6, thanks to element-wise parallelism, the task distributor accesses A's three arrays in the streaming fashion and distributes elements to available PEs. Specifically, it transfers A_{ij}'s value and row index *i* to a PE and transfers the column index *j* to the corresponding rcache manager. The rcache manager follows the SpCache scheme mentioned in Section IV-B and transfers [*rptr*[*j*], *rptr*[*j*+1]) of B to the corresponding vccache manager. Then, the vccache manager fetches B_j's values and column indices and streams them to the corresponding PE. After that, the PE performs multiplications between A_{ij}'s value and B_j's values and transfers products to the corresponding merger. For elements in A_i, mergers merge partial results in their local buffers. PEs accept new elements after the corresponding mergers merge partial results. When the next row begins, mergers push buffered results to the final merger. The final merger merges the four partial results with two helper mergers and writes the result to the DDR.

B. *SpCache*

A banked cache is widely utilized to process parallel requests [7] [9] [14]. However, the inherent sparsity of sparse matrices leads to irregular memory accesses and hence incurs bank conflicts. Further, the long off-chip memory access latency (i.e., part of the cache miss's latency) aggravates bank conflicts. In this paper, we propose an access pattern-aware cache scheme called SpCache, which enables quick responses. As shown in Fig. 7, the SpCache consists of a banked cache and four cache managers. When a cache manager receives a request, the cache manager accesses the target SpCache bank. Then, the bank only checks whether the request hits the cache which takes only 1 cycle and sends a response to the cache manager. If the request misses, the cache manager accesses the off-chip DDR and writes the fetched data to the cache and the target component. We deploy two SpCaches (i.e., one for *rptr* and one for *val* and *cid*) for PEs and the proposed SpCache can significantly reduce bank conflicts by extracting off-chip memory accesses. For example, as shown in Fig. 7(b), if two cache requests a_1 and a_2 access the Bank_0 in parallel and the a_1 misses, a_2 is processed in cycle 39 due to the long miss latency following traditional cache management. Instead, the proposed SpCache can respond to the two requests in four cycles. The cache manager that issues a_2 can get the result earlier either a_2 misses or hits. Meanwhile, the delay of updating the SpCache can be covered by computations.



(a) Combining Streaming and Caching

(b) Caches with Quick Response

Fig. 7: Access Pattern-aware Caches

Besides the overall irregular memory accesses, we also need to consider the access pattern of each request. For each A_{ij}, *rptr*[*j*] and *rptr*[*j*+1] of B are accessed and B_j's values and column indices of unpredictable length are accessed. For *rptr*, since *rptr*[*j*] and *rptr*[*j*+1] may belong to different banks, each cacheline of rcache includes an extra 32-bit data to avoid cross-bank accesses. For *val* and *cid*, elements shall arrive at PEs in order (i.e., the column indices are monotonically increasing) to ensure a smooth merge but the length is unpredictable. On the one hand, it's hard to hold these elements in one cacheline. On the other hand, issuing multiple cache requests incurs additional overhead, e.g., more bank conflicts and standalone off-chip memory accesses if some requests miss the cache. To solve this, we propose combining streaming and caching.

Since the streaming memory access fetches elements in order after the long-delay control information exchange phase, we cache some elements to cover the control phase and then access the remaining elements from the streaming. In this work, we cache the first 32 elements of each B_j in a cacheline. As shown in Fig. 7(a), the vccache manager fetches the cached elements and issues the streaming access towards the remaining elements. The control phase of the streaming access is covered by the processing of the cached elements and the remaining elements can stream into PEs in order. In addition, if requests miss, the manager accesses all the elements in the streaming fashion and updates the cache.

C. PEs and Mergers

PEs mainly perform multiplications between A_{ij}'s value and B_j's values. Since the bitwidth of each memory channel is 128 bits, up to four *B*'s values can be fed into one PE in a cycle. Thus, we deploy four single-precision floating-point multipliers in each PE.

With element-wise parallelism, each merger merges partial results from the corresponding PE. When a new row begins, mergers push the buffered partial results into the final merger. The final merger merges the four partial results with two helper mergers and writes the result to DDR. All the mergers perform the merge algorithm shown in Algorithm 1. Thanks to ordered column indices, the algorithm doesn't require repeated traverses of inputs. Further, we deploy a ping-pong buffer to enable a pipelined implementation of the Algorithm 1.

Algorithm 1 Merge Algorithm

```
Input:
    The input stream of products, $P_i$;
    The buffered stream of partial results, $P_b$;
Output:
    The output stream of merged partial results, $P_m$;
$E_{pi}$ = $P_i$.read(); $E_{pb}$ = $P_b$.read();
while $P_i$ or $P_b$ is not empty do
    if $E_{pi}.cid > E_{pb}.cid$ then
        $P_m$.write($E_{pb}$);
        $E_{pb}$ = $P_b$.read();
    else if $E_{pi}.cid < E_{pb}.cid$ then
        $P_m$.write($E_{pi}$);
        $E_{pi}$ = $P_i$.read();
    else if $E_{pi}.cid == E_{pb}.cid$ then
        Merge the value of $E_{pb}$ and $E_{pi}$; $P_m$.write($E_{pm}$);
        $E_{pi}$ = $P_i$.read(); $E_{pb}$ = $P_b$.read();
    end if
end while
```

V. Experimental Results

A. Experimental Setup

The experimental results are obtained on the Xilinx Zynq UltraScale ZCU106 platform, which integrates a quad-core ARM Cortex-A53 application processor, a dual-core Cortex-R5 real-time processor, and an XCZU7EV-2FFVC1156 FPGA chip. The accelerator is written in C++, which is converted to Verilog using the Vitis HLS. We use Vitis HLS version 2022.1 [15] as the High-level Synthesis (HLS) tool and Vivado version 2022.1 to generate the final bitstream. We run all the experiments under a 100 MHz clock. Table I shows the total resource consumption of this design. Caches are 16-way 4-bank caches and the size of rcache and vccache is 40KB and 2MB separately.

TABLE I: Resource Consumption

LUTs(%)	FFs(%)	BRAM(%)	URAM(%)	DSPs(%)
73.53	47.13	59.29	62.5	3.13

Table II shows the benchmarks used in this work. All these matrices are available on the SuiteSparse matrix collection [10]. We use the same configuration as [7] [6] [9] that *A* and *B* are the same and both are square. In addition, since there is no previous work targeting the embedded FPGA platform, we compare the performance with the baseline implementation using row-wise parallelism. Besides, the datatype of *val* is 32-bit *float* and the datatype of *cid* and *rptr* is 32-bit *unsigned int*. The matrices are in CSR format without extra processing.

TABLE II: Selected Benchmarks

Benchmark	Cols/Rows	Nonzero	Nz/Col	Density	Cycles
poisson3Da	13514	352762	26.10	0.19%	19878983
raefsky1	3242	294276	90.77	2.80%	27458639
crystk01	4875	315891	64.80	1.33%	21885313
s3rmt3m3	5357	207695	38.77	0.72%	8628745
t2dah_a	11445	176117	15.39	0.13%	5582450
nasa2910	2910	174296	59.90	2.06%	12797230
bcsstk24	3562	159910	44.89	1.26%	7588259
cavity26	4562	138187	30.29	0.66%	6632677
ex9	3363	99471	29.58	0.88%	4011290
af23560	23560	484256	20.55	0.09%	15408694

B. Performance Analysis

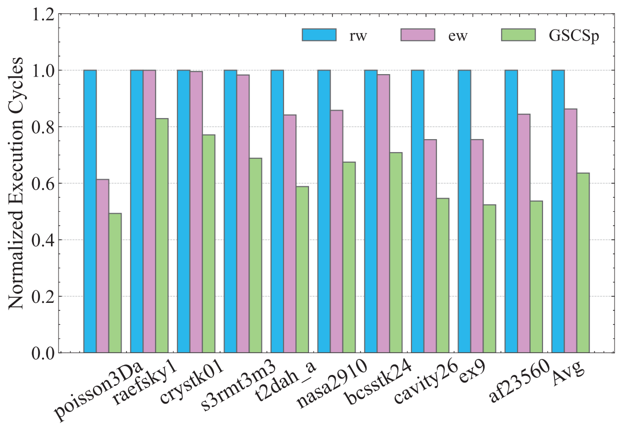


Fig. 8: Performance Comparison

As shown in Fig. 8, our proposed design labelled as *GSCSp* achieves an average 1.62x performance speedup compared to the baseline *rw*. Specifically, the proposed element-wise parallelism *ew* achieves an average 1.19x speedup and the *GSCSp* further achieves an average 1.37x speedup using SpCaches.

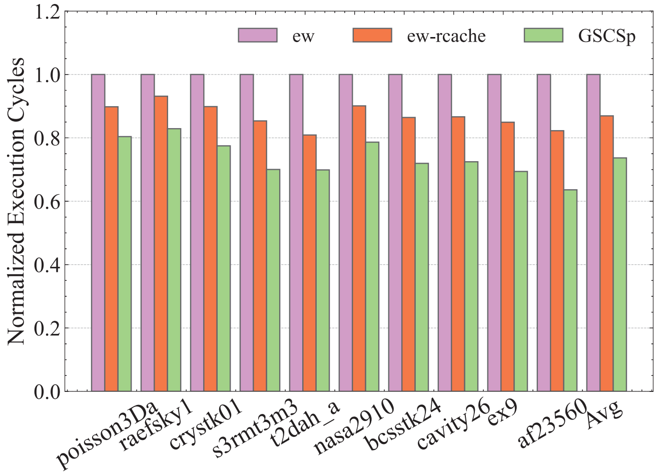


Fig. 9: Performance Speedup of SpCaches

Compared to row-wise parallelism, the proposed element-wise parallelism tries to reduce PEs' idle time caused by synchronization among PEs. However, it may fail to achieve performance speedup. If the workload of each A's row is similar, the overhead of synchronization is little. For example, `raefsky1`, `crystk01`, `s3rmt3m3`, and `bcsstk24` have similar performance with the two parallelisms. Overall, the average performance speedup of the proposed element-wise parallelism is 1.19x.

Despite the parallelism, the proposed SpCache aims to reduce the overall memory access latency. As shown in Fig. 9, the proposed rcache and vccache achieve an average 1.15x and 1.18x performance speedup separately. Since memory accesses, multiplications, and merges of the four elements of A are processed in parallel, it's hard to accurately analyze the performance. We remove multiplications and merges to get the execution time that only memory accesses are performed and the proportion of memory accesses is shown in Fig. 10. For example, memory accesses account for 44% of the overall execution time in `raefsky1` and the SpCache achieves minor performance speedup. Overall, the proposed SpCaches achieve an average 1.37x performance speedup.

VI. Conclusion

In this work, we analyze the parallelism of the Gustavson's algorithm and propose to perform the algorithm with element-wise parallelism. Further, we show a counter-intuitive example that traditional cache leads to worse performance. In addition, we propose a novel access pattern-aware cache scheme called SpCache, which provides quick responses to reduce bank conflicts caused by irregular memory accesses and combines streaming and caching to handle requests that access ordered elements of unpredictable length. Finally, we conduct experiments on the Xilinx ZCU106 platform with a set of benchmarks from the SuiteSparse matrix collection and the experimental results show an average 1.62x performance speedup compared to the baseline implementation.

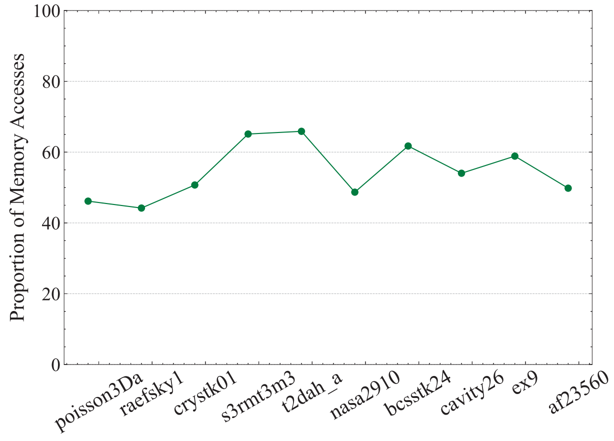


Fig. 10: Proportion of Memory Accesses

Acknowledgment

This work is partially supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-071), and Nanyang Technological University, Singapore, under its NAP (M4082282).

References

[1] J. R. Gilbert and et al., "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008.

[2] P. D'alberto and et al., "R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," *Algorithmica*, vol. 47, no. 2, pp. 203–213, 2007.

[3] H. Kaplan and et al., "Colored intersection searching via sparse rectangular matrix multiplication," in *Proceedings of the twenty-second annual symposium on Computational geometry*, 2006, pp. 52–60.

[4] S. Itoh and et al., "Order-n tight-binding molecular dynamics on parallel computers," *Computer physics communications*, vol. 88, no. 2-3, pp. 173–185, 1995.

[5] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[6] N. Srivastava and et al., "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM MICRO*. IEEE, 2020, pp. 766–780.

[7] D. Baek and et al., "Innersp: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing," in *2021 30th PACT*. IEEE, 2021, pp. 116–128.

[8] E. B. Tavakoli and et al., "Fspgemm: An opencl-based hpc framework for accelerating general sparse matrix-matrix multiplication on fpgas," *arXiv preprint arXiv:2112.10037*, 2021.

[9] G. Zhang and et al., "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the 26th ACM ASPLOS*, 2021, pp. 687–701.

[10] T. A. Davis and et al., "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[11] C. Y. Lin and et al., "Design space exploration for sparse matrix-matrix multiplication on fpgas," *International Journal of Circuit Theory and Applications*, vol. 41, no. 2, pp. 205–219, 2013.

[12] S. Pal and et al., "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE HPCA*. IEEE, 2018, pp. 724–736.

[13] Z. Zhang and et al., "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE HPCA*. IEEE, 2020, pp. 261–274.

[14] M. Asiatici and et al., "Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas," in *FPGA 2019*, 2019, pp. 310–319.

[15] Xilinx. Vivado design suite. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html