An Efficient Gustavson-based Sparse Matrix-matrix Multiplication Accelerator on Embedded FPGAs

Shiqing Li, Shuo Huai, Weichen Liu

***Abstract*—Sparse-matrix sparse-matrix multiplication (SpMM) is an important kernel in multiple areas, e.g., data analytics and machine learning. Due to the low on-chip memory requirement, the consistent data format, and the simplified control logic, the Gustavson's algorithm is a promising backbone algorithm for SpMM on hardware accelerators. However, the off-chip memory traffic still limits the performance of the algorithm, especially on embedded FPGAs. Previous researchers optimize the Gustavson's algorithm targeting high bandwidth memory-based architectures and their solutions cannot be directly applied to embedded FPGAs with traditional DDRs. In this work, we propose an efficient Gustavson-based sparse matrix-matrix multiplication accelerator on embedded FPGAs. The proposed design fully considers the feature of off-chip memory access on embedded FPGAs and the dataflow of the Gustavson's algorithm. At first, we analyze the parallelism of the 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. Moreover, we propose to perform the merge on part of partial results, which removes some redundant merges in the naive implementation and has little postprocessing overhead. 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 1.75x performance speedup compared to the baseline.**

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

I. Introduction

SPARSE-matrix sparse-matrix multiplication (SpMM) is essential 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], Markov clustering [4], finite element simulations based on domain decomposition [5], matching algorithm [6], interior point methods [7], and molecular dynamics [8]. 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. Meanwhile, accessing isolated elements from the off-chip DDR is much more inefficient than accessing sequential elements. Consequently, the off-chip memory traffic becomes the main bottleneck of SpMM.

SpMM mainly consists of three dataflows: the inner-product algorithm, the outer-product algorithm, and the Gustavson's algorithm (i.e., the row-wise product algorithm). Among these algorithms, the Gustavson's algorithm [9] has shown its potential to be the backbone algorithm for SpMM on hardware accelerators [10] [11] [12] [13] 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*. It mainly has three advantages: (1) index matching (i.e., the row index of elements in the first matrix should equal to the column index of elements in the second one) is eliminated; (2) only a buffer of one row is required; (3) *A, B,* and *C* are in row-major order. However, since the results of PEs should be written back in order, the inherent row-wise parallelism leads to uneven workloads among processing elements (PEs). The synchronization among PEs stalls some of them and degrades the overall performance.

For each PE, it performs multiplications between one element of *A* and the corresponding row of *B* and merges partial results. For multiplications, sparse matrices are stored in the off-chip DDR and the off-chip memory traffic affects the fetching of input data. There are two ways to mitigate the off-chip memory traffic issue: streaming off-chip memory accesses and caching data on-chip. Fortunately, the Gustavson's algorithm provides reusability for rows of *B*. However, there are two challenges. On the one side, although a banked cache can serve multiple requests per cycle, the inherent irregularity of sparse matrices incurs bank conflicts. Moreover, the long miss latency which includes the long off-chip memory access latency leads to more cache conflicts. On the other side, each element of *A* is multiplied with a sequence of ordered elements in the corresponding row of *B*. The main challenge is that lengths of rows of *B* are unpredictable. Given a fixed cacheline size, each row usually occupies multiple cachelines belonging to multiple banks. Thus, multiple cache requests are required. If some of them miss the cache, multiple off-chip memory accesses are issued which results in a long off-chip memory access latency compared to one streaming off-chip memory access for the row. Besides, these requests lead to more cache conflicts. For merges, we should keep partial results ordered to enable an efficient merge. However, the naive implementation traverses all the partial results, which results in redundant merges.

S. Li, S. Huai, and W. Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. W. Liu is the corresponding author. Email: shiqing.li@ntu.edu.sg, shuo.huai@ntu.edu.sg, liu@ntu.edu.sg.

To solve these challenges, in this work, we propose an efficient Gustavson-based SpMM accelerator on embedded FPGAs. 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.
- We propose to perform the merge on part of partial results, which removes some redundant merges in the naive implementation and has little postprocessing overhead.
- We conduct experiments on the Xilinx ZCU106 platform with a set of benchmarks from the SuiteSparse matrix collection [14]. The results show that our proposed design achieves an average 1.75x performance speedup. The implementation is available at https://github.com/lsq314/SpMM_TCAD.

The paper is organized as follows. In Section II, we introduce the background of SpMM, its three dataflows, and off-chip memory access features on embedded FPGAs and discuss related work. We analyze the parallelism and show our motivational example in Section III. Section IV details the hardware design. In Section V, we analyze the experimental results. Finally, Section VI concludes this work.

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* arrays and then the corresponding values and column indices are accessed. For example, we access $[rptr[1], rptr[2])$ (i.e., [2, 4)) and then access *val*[2:4] and *cid*[2:4] to get value and column indices of 1_{th} row of *B*. Note that elements in each row are sorted by column indices.

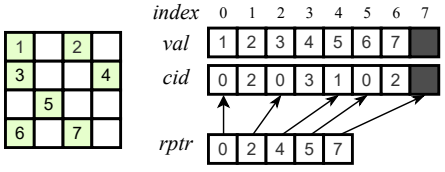


Fig. 1: CSR Format

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

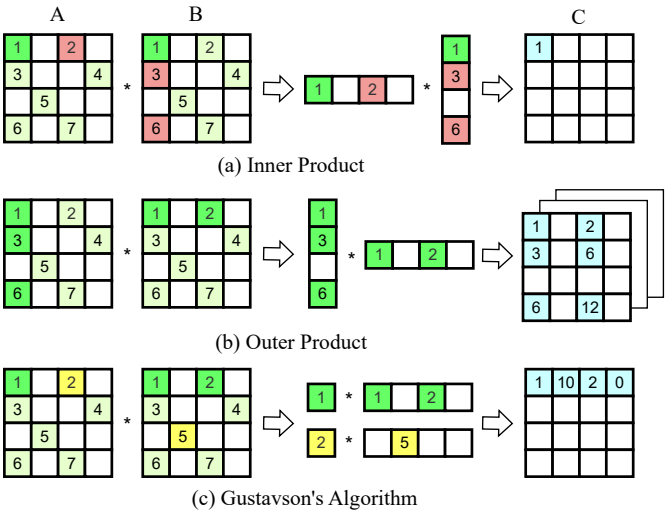


Fig. 2: SpMM's Algorithms

Three algorithms are typically applied to SpMM shown in Fig. 2. Note that SpMM follows the rule that an element A_{ij} is multiplied with an element of *B* whose row index is *j*. The inner-product algorithm which is widely used in general matrix-matrix multiplications multiplies A_i and the j_{th} column of *B* to generate the value of C_{ij}. However, index matching lowers the bandwidth efficiency and thus degrades the performance. For example, elements 2, 3, and 6 in Fig. 2(a) are mismatched. Only one pair of elements (i.e., A_{00} and B_{00}) is multiplied. To eliminate index matching, the outer-product algorithm is utilized to improve the hardware efficiency as shown in Fig. 2(b). In the algorithm, the i_{th} column of *A* is multiplied with the i_{th} row of *B* to generate a partial result matrix. At last, all the partial result matrices are merged to get the output matrix *C*. Although index matching is not required, massive partial results lead to heavy off-chip memory traffic which significantly degrades the performance, especially on embedded FPGAs. Moreover, the previous two algorithms have a common disadvantage which is that two input matrices are in different formats. Specifically, one matrix is in row-major order and the other matrix is in column-major order. As a result, format conversion is required if the output matrix is reused to be *A* or *B*. To further improve the hardware efficiency, the Gustavson's algorithm which processes *A* row by row is exploited to be the backbone algorithm on hardware accelerators. For element A_{ij} in A_i, it is multiplied by the elements in B_j. For example, element A_{00} whose value is 1 is multiplied with B_0 (i.e., [1, 2]) in Fig. 2(c). Then, all the products in format (value, column index) are merged to get the final result of C_i. In Fig. 2(c), [(1, 0), (2, 2)] and [(10, 1)] are merged to get C_0 (i.e., [(1, 0), (2, 2), (10, 1)]). Overall, its advantages are threefold. At first, index matching is eliminated. Then, we only need to buffer one row of *C*. At last, three matrices can use a consistent format since they are in row-major order.

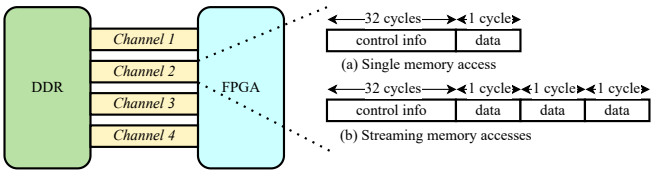


Fig. 3: ZCU106's Off-chip Memory Access

B. Off-chip Memory Access on Embedded FPGAs

Embedded FPGAs mainly include limited on-chip memories and a relatively large off-chip DDR. For instance, the Xilinx Zynq-UltraScale ZCU106 platform has around 4.7MB on-chip memories and sparse matrices are stored in the off-chip DDR. Data in the off-chip DDR is fetched via multiple memory channels and transactions in different channels are independent. For transactions in one channel, they follow the bus protocol to transfer data. Specifically, Advanced eXtensible Interface (AXI) protocol is deployed in Xilinx devices. We show the two most important fashions in data transactions of AXI protocol in Fig. 3. The basic one is single memory access. It first exchanges the control information (e.g., read address) which takes around 32 cycles and then gets the target data (the bitwidth varies across devices) in the next cycle. When a sequence of contiguous elements is accessed, streaming memory access which is the most efficient way to transfer data between the off-chip DDR and the FPGA chip only requires one control information exchange phase.

C. Related Work

Sparse matrix operations have been well studied in the literature. Sparse matrix-vector multiplication (SpMV) is the simplest operation in which the biggest challenge is to efficiently fetch vector elements of the randomly distributed matrix elements. In [15] proposes a framework that can generate a specific accelerator for each matrix. In [16], the authors propose a high-performance dataflow engine for CPU-FPGA heterogeneous platform. They utilize the CPU part to fetch vector elements and the FPGA part performs computations in parallel. However, this design relies on a high-speed CPU counterpart that matches the parallel FPGA design. In addition, they transfer repeating vector elements instead of their column indices which wastes lots of memory bandwidth. In [17], the authors propose a new data locality-aware compressed format targeting multi-FPGAs environment. However, these work did not efficiently utilize the memory bandwidth on embedded FPGAs. [18] targets embedded FPGAs and proposes to explore data reuse via reordering which achieves a higher bandwidth utilization rate.

Although SpMM can be performed by splitting it into multiple SpMVs, data reuse of both two matrices is not fully considered. For SpMM, [19] conducts design space exploration for SpMM using the inner product algorithm. It successfully finds the trade-off between performance and energy consumption. The OuterSPACE [20] and the SpArch [21] targets the outer product algorithm. [21] further reduces the number of partial result matrices by condensing matrices. However, these works suffer from the drawbacks of the algorithms. Besides, the MapRaptor [10] proposes an accelerator targeting the Gustavson's algorithm. With the inherent row-wise parallelism, PEs fetch rows of B via multiple off-chip memory channels and merge products as shown in Fig. 9. Besides, it proposes a dedicated format to enable vectorized and streaming memory reads. However, it fails to explore the reuse of B's rows. The InnerSP [11] and the GAMMA [13] propose cache-based architectures to explore the reuse of B's rows. With row-wise parallelism, a PE in GAMMA first fetches multiple rows of B to perform the merge operation and then performs multiplications, which requires a pretty large on-/off-chip memory bandwidth to fetch data and thus stalls PEs. Besides, they target the high bandwidth memory-based ASIC accelerators and are unsuitable for embedded FPGAs with traditional DDR.

Sparse-matrix dense-matrix multiplication is also a hot topic. Sextans [22] targets sparse-matrix dense-matrix multiplication and proposes a hardware accelerator that enables prototyping once to support all the target operations as a general-purpose accelerator. SDMA [23] proposes equal-value partition and vertex-clustering optimization to solve the load imbalance and irregular memory accesses. However, these works cannot efficiently handle SpMM where both input matrices are sparse. Besides, model pruning [24] [25] in deep learning converts MV and MM into SpMV and SpMM. The pruning strategy considers the hardware efficiency and allows us to manually decide the positions of nonzero elements.

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

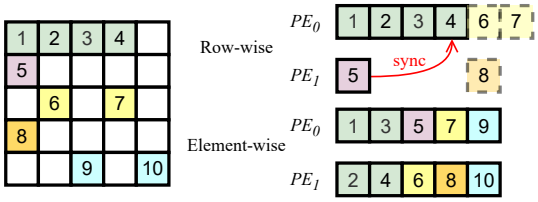


Fig. 4: Gustavson's Parallelism Analysis

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 performs SpMM with row-wise parallelism. For the i_{th} row of A, the algorithm performs multiplications between elements of A and the corresponding rows of B and then merges all the products to get the i_{th} row of C. With row-wise parallelism, PEs process rows of A and they only accept new rows if they finish the current ones. Since

A, *B*, and *C* are stored in CSR format, PEs are supposed to write results to ensure the correctness of the result. Thus, the synchronization of PEs stalls some of them and leads to a bad performance. We first use a simple example in Fig. 4 to show the drawback of row-wise parallelism. In the beginning, the two PEs accept the first two rows of *A*. However, since A_0 is much longer than A_1, PE_1 stalls to wait for PE_0. Moreover, the workload is also uneven in the following rows. A possible solution to mitigate the synchronization is deploying a buffer to hold the partial results. However, it is hard to determine its size to hold all the partial results.

In this work, we propose to perform the Gustavson's algorithm with element-wise parallelism. As shown in Fig. 4, PEs accept elements of *A* rather than rows of *A*. As a result, the workload is evenly distributed to PEs compared to row-wise parallelism. If the current row finishes, PEs stream the buffered results to a final merger which merges all the partial results from PEs. The latency of the final merger can be overlapped by computations of the following rows. In some rare cases (e.g., A_1 in Fig. 4) where a row is too short to fully utilize all the PEs, the synchronization issue may still exist. However, it is significantly mitigated compared to row-wise parallelism. Moreover, since PEs sequentially process elements of *A*, *val*, *cid*, and *rptr* of *A* are accessed in the streaming mode which is the most efficient way to access the off-chip DDR. Overall, element-wise parallelism is more efficient for SpMM on embedded FPGAs.

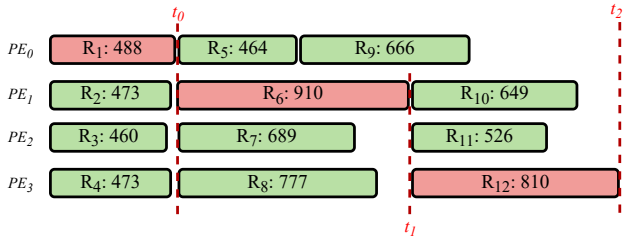


Fig. 5: Idle time using row-wise parallelism in `poisson3Da`

To help understand the drawback of row-wise parallelism, we show the workload of the first twelve rows due to the space limitation in Fig. 5. In the figure, we use the number of multiplications in each row of *A* to represent its execution time in PEs and assume that there are four PEs. As shown in Fig. 5, the synchronization appears at three timestamps. At t_0, the workload of PE_0 is bigger than the other three PEs and thus stalls them. Specifically, PE_2 stalls for around 28 cycles. Since results should be written in order, synchronization occurs when PEs write results. After processing R_5 and writing the result to DDR, PE_0 can accept a new task and thus R_9 is processed by it without synchronization. When R_9 is processed, the result is written to DDR since results of R_1 to R_8 are written. At t_1, PE_1 processes the 6_{th} row which has a workload of 910 multiplications and it stalls PE_2 for around 221 cycles and PE_3 for around 133 cycles. At t_2, the process of the 12_{th} row stalls PE_0 for around 600 cycles. These stalled PEs would then affect others while processing the remaining rows.

The synchronization issue among PEs is mitigated using element-wise parallelism. For each PE, it performs multiplications between elements of *A* and the corresponding rows of *B* and then merges the products to get the final result. For multiplications, input data are fetched from the off-chip DDR and the off-chip memory traffic affects the performance. Following the Gustavson's algorithm, rows of *B* are reused for elements in a column of *A*. Thus, caching data on-chip is a promising solution for the algorithm. However, the irregularity of sparse matrices brings challenges to the cache design. On the one side, irregular memory accesses incur cache conflicts and the long miss latency which includes the long off-chip memory access latency leads to more conflicts. On the other hand, lengths of rows of *B* are unpredictable. One row of *B* usually falls into multiple cache banks and multiple requests are required. If some of these requests miss, the off-chip memory latency increases because one streaming memory access is split into multiple single memory accesses. Meanwhile, these requests also aggravate bank conflicts. For merges, although we need to keep partial results ordered, we do not need to generate a final result for each element of *A*.

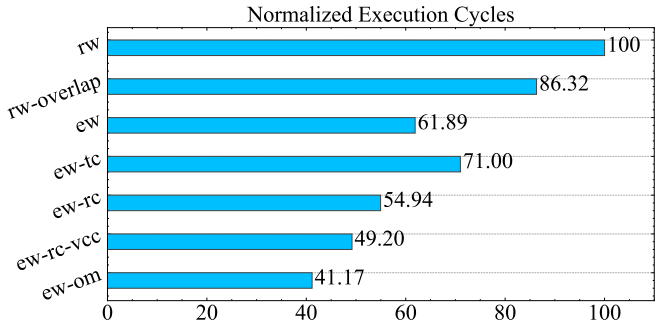


Fig. 6: The Motivational Example

B. Motivational Example

In this subsection, we show the improvement of proposed optimizations with an example. We conduct the experiment using `poisson3Da` from the SuiteSparse matrix collection [14] on the Xilinx Zynq-Ultrascale ZCU106 platform. `poisson3Da` is a 13514×13514 matrix with 352762 nonzero elements. We process four elements of *A* in parallel and deploy caches with the least-recently-used (LRU) replacement strategy. All the caches are 16-way and are split into 4 banks. In addition, only two elements of *rptr* of *B* are accessed and an unpredictable number of elements of *cid* and *val* is accessed. Thus, we propose different designs for these arrays. Caches for *rptr* and caches for *val* and *cid* are referred to as *rcache* and *vccache*. Moreover, *rw, ew, tc, rc, vcc, om* denotes row-wise parallelism, element-wise parallelism, traditional cache, rcache, vccache, and optimized merger. Overall, the proposed design (i.e., *ew-om*) achieves a 2.42x performance speedup. A counter-intuitive example is that the traditional cache (i.e., *ew-tc*) degrades the performance compared to *ew*. Instead, the proposed *rcache* (i.e., *ew-rc*) achieves a 1.12x performance speedup. Further, the proposed *vccache* achieves a 1.12x more performance speedup. We analyze the experimental results in Section V-B. We also deploy a buffer to hold partial results of several rows and thus the synchronization issue is mitigated compared to pure row-wise parallelism. As shown in Fig. 6,

although it (i.e., *rw-ovelap*) 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*. Finally, we achieve a 1.19x performance speedup with the optimized merger (i.e., *ew-om*) which removes some redundant merges.

IV. HARDWARE DESIGN

In this section, we first show the overview of our proposed hardware design. Then, we detail each component. Recall that only elements of *B* are reused and elements of *A* are directly read from the off-chip DDR.

A. Overview

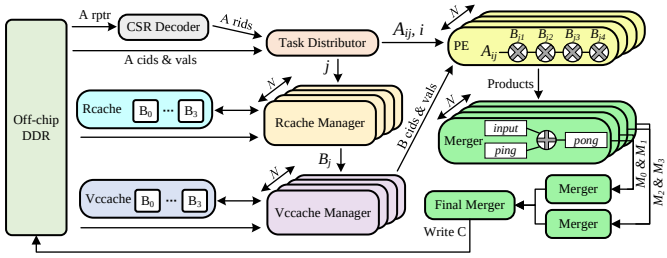
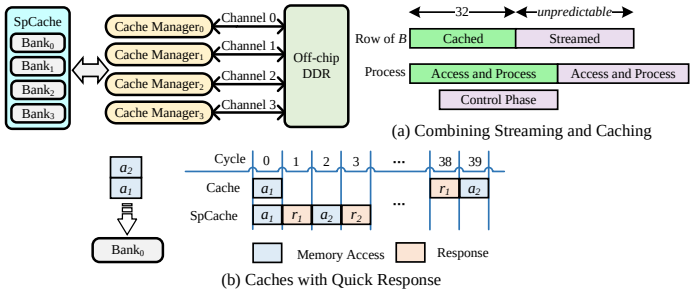


Fig. 7: Hardware Overview

As shown in Fig. 7, the hardware mainly consists of a SpCache for *rptr*, a SpCache for *val* and *cid*, *N* PEs, and *N* mergers. To avoid the memory channel conflict, we process four elements of *A* simultaneously (i.e., *N* in Fig. 7 is 4) and one memory channel on Xilinx ZCU106 is used to serve transactions of one PE. As discussed in Section III, three arrays of *A* are accessed in the streaming mode and the task distributor assigns elements to idle PEs. For each A_{ij}, it sends the value and the row index to a PE and sends the column index to the corresponding *rcache* manager. The manager is an important component detailed in Section IV-B and it fetches the required elements of *rptr* of *B* and then transfers [*rptr*[*j*], *rptr*[*j*+1]) of *B* to the corresponding vccache manager. Then, values and column indices of B_j are fetched via *vccache* and stream into the corresponding PE. After that, the PE performs multiplications between the value of A_{ij} and values of B_j and transfers products to the corresponding merger. For the current row of *A*, mergers buffer the partial results in local buffers and PEs cannot accept a new element until the corresponding mergers finish. When mergers receive elements of the next row, they first stream buffered results to the final merger and then perform the merge of products belonging to the new row. Finally, the final merger merges the four partial results with two helper mergers and writes the result to the DDR.

B. SpCache

Caches are split into multiple banks to serve multiple requests in parallel [11] [13] [26]. However, the inherent irregularity of sparse matrices incurs bank conflicts. In addition, the long cache miss latency which includes the long off-chip memory access latency further aggravates bank conflicts. In this subsection, we detail the access pattern-aware cache design called SpCache, which mitigates bank conflicts by extracting off-chip memory accesses from caches and thus quickly responds to irregular requests. As shown in Fig. 8, there is a banked cache and four cache managers in a SpCache. When a request arrives, the manager issues a request to the target bank. Then, the SpCache bank only takes 1 cycle to check whether the request hits and then responds to the cache manager. If the request misses, the cache manager directly performs the off-chip memory access and sends the fetched data to the cache bank and the target component. For example, the rcache manager transfers *rptr*[*j*] and *rptr*[*j*+1] to the rcache bank and the vccache manager. We use an example to illustrate the benefit in Fig. 8(b). Assume that there are two cache requests a_1 and a_2 targeting the Bank_0 and a_1 misses. a_2 waits for the processing of a_1 until cycle 39 due to the off-chip memory access latency in conventional cache design. Instead, we take only one cycle to process a_1 with the proposed SpCache. As a result, a_2 is processed in cycle 2 and the target data can be fetched either if it misses or hits. Besides, the replacement of SpCache can be overlapped by the following computations.



(a) Combining Streaming and Caching

(b) Caches with Quick Response

Fig. 8: Access Pattern-aware Caches

The access pattern of requests should be also considered. For each element A_{ij}, we first access *rptr*[*j*] and *rptr*[*j*+1] of *B* and then access the corresponding elements of *val* and *cid*. Consequently, two elements of *rptr* are accessed and an unpredictable number of elements of *val* and *cid* is accessed. For *rptr*, the two elements may fall into two cachelines (i.e., two banks). Thus, we include an additional element (i.e., the next element of the last element) in each cacheline. For *val* and *cid*, an ordered sequence of elements is required to ensure a smooth merge. However, the length is unpredictable. On the one hand, we cannot use a cacheline to hold a sequence with unpredictable length. On the other hand, the overhead is not negligible if we issue multiple cache requests for each row of *B*. For example, the off-chip memory access latency increases if some of them miss the cache. In this subsection, we propose to cache some elements of one row and utilize the streaming mode to fetch other elements. Note that the time-consuming part of the streaming memory access is the control information phase. Thus, we propose to cache some elements to overlap the control phase. If the cache request misses, the manager issues one streaming access. Otherwise, the manager sends the cached data to the corresponding PE and issues one streaming access to the remaining elements. In this work, the first 32 elements are cached. As shown in Fig. 8(a), the

proposed design can effectively process requests which access a sequence of ordered elements with unpredictable lengths.

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;
1: E_{pi} = P_i.read(); E_{pb} = P_b.read();
2: **while** P_i or P_b is not empty **do**
3: **if** $E_{pi}.cid > E_{pb}.cid$ **then**
4: P_m.write(E_{pb});
5: E_{pb} = P_b.read();
6: **else if** $E_{pi}.cid < E_{pb}.cid$ **then**
7: P_m.write(E_{pi});
8: E_{pi} = P_i.read();
9: **else if** $E_{pi}.cid == E_{pb}.cid$ **then**
10: Merge the value of E_{pb} and E_{pi}; P_m.write(E_{pm});
11: E_{pi} = P_i.read(); E_{pb} = P_b.read();
12: **end if**
13: **end while**

C. PEs

In this design, PEs are single-precision floating-point multipliers. Note that each memory channel is 128-bit in the Xilinx ZCU106 platform. We can transfer up to four values of *B* to a PE per cycle. Overall, up to 16 multiplications can be performed using four PEs in parallel.

D. Mergers

With element-wise parallelism, we process matrix *A* row by row and mergers merge input streams of products from the corresponding PEs. Each merger buffers partial results in its local buffer and pushes the buffered partial result into the final merger when a new row begins. The final merger merges four partial results with two helper mergers and writes the result to the DDR. All the mergers perform the merge algorithm shown in Algorithm 1. In the algorithm, we compare column indices of the first elements of the two input streams. If the two column indices are equal, we add values of the two elements and push the result with the column index to the output stream (line 10-11). Otherwise, we push the element with a smaller column index to the output stream (line 4-8). Thanks to ordered column indices, the algorithm only traverses two input streams once. Further, we deploy a ping-pong buffer to enable a pipelined implementation of the Algorithm 1.

Although this naive implementation pipelines the merge, it traverses all the buffered results and thus performs some redundant merges. In this subsection, we follow the divide and conquer paradigm and perform the merge on part of buffered results. We first show our motivational example. In the example, we assume that there are six input streams and each stream has 70 products. We show the process in two cases. The first case (referred to as *CASE1*) is that all the products are not merged with items in buffered results and the second case (referred to as *CASE2*) is that all the products are merged. Although these two cases are special, they are easy to understand and the execution time is easy to predict. In *CASE1*, the execution time equals the number of two input streams. In *CASE2*, the execution time equals the number of one input stream.

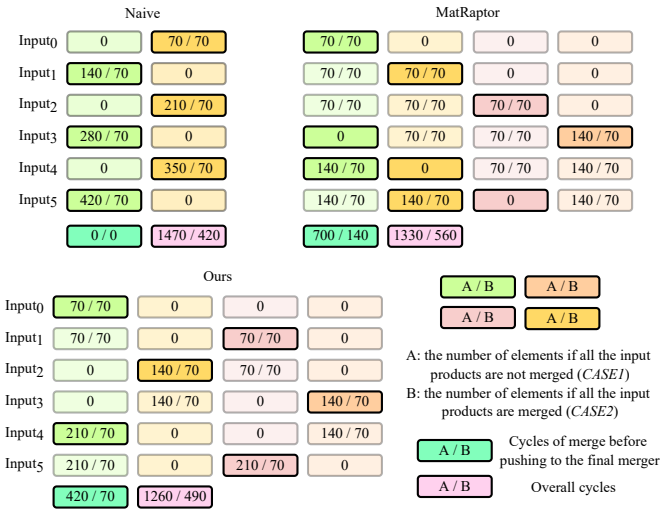


Fig. 9: Motivational Example of Mergers

As shown in Fig. 9, we compare the naive implementation, the design in MatRaptor [10], and our proposed design. The naive implementation traverses all the buffered results and generates a final version after processing each input. When it receives an element of the next row, it does not need to merge the buffered streams. Thus, it takes 0 cycles to post-process in both cases. The main drawback is that it performs several redundant merges in cases like *CASE1*. For example, products in input_0 in Fig. 9 do not need to be merged in *CASE1* and hence we only need to traverse it once. However, the naive implementation traverses them around five times (i.e., processing input_1 to input_5). Overall, it takes around 1470 cycles in *CASE1*. In *CASE2*, it takes around 420 cycles. In MatRaptor [10], the authors propose a merger with Q FIFOs. For the first $Q-1$ inputs, they stream into the first $Q-1$ FIFOs. For instance, input_0, input_1, and input_2 stream into FIFO1, FIFO2, and FIFO3. Then, the following inputs are merged with the shortest stream and the output streams into the empty FIFO. In Fig. 9, input_3 is merged with the stream in FIFO1 and then the output streams into FIFO4. When a new row begins, the MatRaptor [10] merges the buffered ones. Compared to the naive implementation, it takes cycles to post-process the buffered streams before pushing to the final merger. Specifically, it takes 700 cycles in *CASE1* and 140 cycles in *CASE2*. Overall, it takes 1330 cycles and 560 cycles. Especially in *CASE2*, the MatRaptor [10] does not skip any merge and takes cycles to post-process the buffered FIFOs instead. In this work, we propose to use multiple ping-pong buffer blocks to eliminate the drawback. In the motivational example, we have two ping-pong blocks. When the merger receives an input, it selects the block with minimum elements and then performs the merge. Overall, it takes 1260 cycles in *CASE1* and 490 cycles in *CASE2*. The proposed design can

skip some redundant merges in *CASE1* compared to the naive implementation and reduces the cycles used to post-process the buffered streams compared to the MatRaptor [10].

Algorithm 2 Optimized Merger

Input:
The input stream of products, P_i;
Output:
The output stream of merged partial results, P_m;
1: bool ping_pong$_{1,2}$ = true;
2: int length$_{1,2}$ = 0;
3: int min_index = 0;
4: hls::stream ping1, pong1, ping2, pong2;
5: **while** P_i **do**
6: **if** min_index == 0 **then**
7: **if** ping_pong$_1$ **then**
8: merge(P_i, &ping1, pong1, &length$_1$);
9: **else**
10: merge(P_i, &pong1, ping1, &length$_1$);
11: **end if**
12: ping_pong$_1$ = !ping_pong$_1$;
13: **else**
14: The same operation to ping_pong$_2$ as ping_pong$_1$;
15: **end if**
16: min_index = length$_1$ < length$_2$? 0 : 1;
17: **end while**

As shown in Algorithm 2, we first initialize two ping-pong buffer blocks (i.e., four streams) and their lengths. Meanwhile, we use a boolean variable (e.g., ping_pong$_1$) to indicate the output FIFO of each block. After processing one input stream, we find the shortest stream by comparing the two lengths (line 16). If the shortest stream is in the first block (i.e., min_index == 0, line 6-12), we perform the Algorithm 1. If ping_pong_1 is *true*, we write the result to pong1. Otherwise, we write to ping1. After that, we reverse the boolean variable (line 12). Although deploying two merge modules can speed up the post-process of merging the buffered FIFOs in MatRaptor [10], it takes more resources and hinders the scalability to process more elements of *A* (i.e., increasing *N* in Fig. 7) in parallel.

We deploy four 1000-element buffers (i.e., two ping-pong buffers) and each element is 66-bit which includes a 32-bit *float* value, a 32-bit *unsigned int* column index, and two 1-bit control signals. This size is enough to hold partial results for all the selected benchmarks. In the future, we plan to utilize the off-chip DDR to increase the buffer size. Specifically, buffers inside mergers have an on-chip part and an off-chip part. Similar to the proposed vccache, the control phase of streaming the off-chip part is covered by processing the on-chip part. After processing the on-chip part, the off-chip part can stream into the merger unit. Following this way, the merger can hold more partial results and remains fully pipelined.

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 [27] 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. Note that we fail to implement the design in Vivado if we directly optimize the merger on the original design [28]. We try to reduce some logic resources and the optimized design (i.e., this work) consumes fewer LUTs.

TABLE I: Resource Consumption

	LUTs(%)	FFs(%)	BRAM(%)	URAM(%)	DSPs(%)
[28]	73.53	47.13	59.29	62.5	3.13
this work	66.29	42.11	64.42	62.5	2.66

We show selected typical benchmarks from the SuiteSparse matrix collection [14] in Table II. We perform SpMM on two same square matrices following the configuration of [10] [11] [13]. Since this is the first work that studies the Gustavson-based SpMM on embedded FPGAs, we target the pure Gustavson's algorithm with row-wise parallelism as the baseline. 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: Benchmarks

Benchmark	#Cs/Rs[1]	Nonzero	3M[2]	Density	Cycles
poisson3Da	13,514	352,762	6/23/110	0.19%	16,638,816
raefsky1	3,242	294,276	24/108/108	2.80%	25,658,832
crystk01	4,875	315,891	24/54/81	1.33%	20,202,582
s3rmt3m3	5,357	207,695	7/42/48	0.72%	8,411,161
t2dah_a	11,445	176,117	8/13/21	0.13%	5,354,111
nasa2910	2,910	174,296	16/55/175	2.06%	11,731,798
bcsstk24	3,562	159,910	15/51/57	1.26%	7,354,144
cavity26	4,562	138,187	8/26/62	0.66%	6,012,791
ex9	3,363	99,471	10/30/50	0.88%	3,790,519
af23560	23,560	484,256	11/21/21	0.09%	13,979,280

[1] The number of columns or rows in square sparse matrices.
[2] The min/median/max values of the number of non-zero elements per row.

B. Performance Analysis

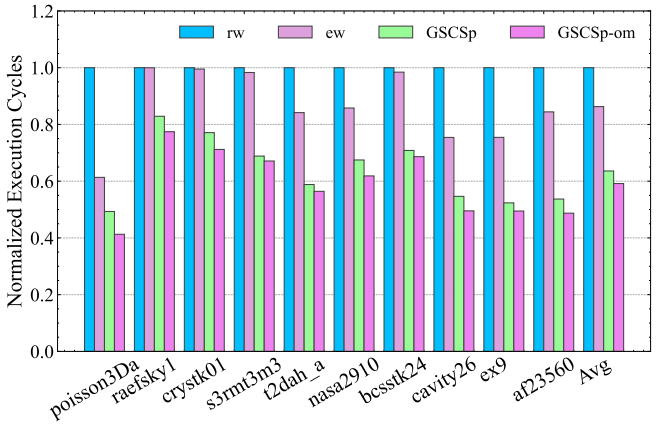


Fig. 10: Performance Comparison

We show the performance comparison with different optimizations in Fig. 10 and label the design with element-wise parallelism, *rcache*, and *vccache* as *GSCSp* in short. We achieve an average 1.75x performance speedup compared to the baseline. The element-wise parallelism reduces the idle time of PEs and achieves an average 1.19x performance speedup. The SpCache further reduces the off-chip memory traffic and achieves an average 1.37x performance speedup. Finally, the optimizer merger removes some redundant merges and brings an average 1.08x performance speedup.

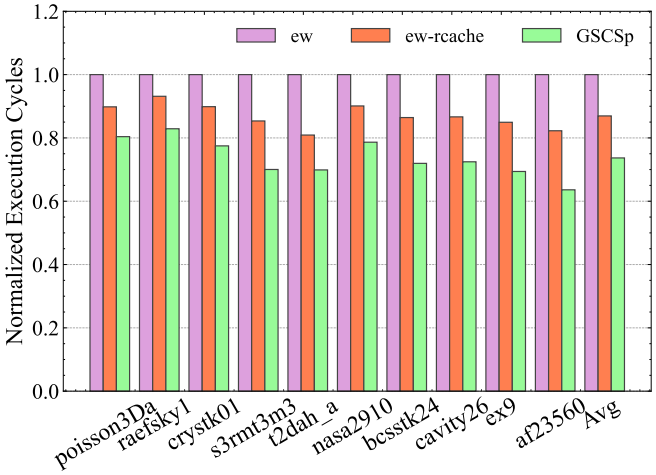


Fig. 11: Performance Speedup of SpCaches

Compared to row-wise parallelism, the proposed element-wise parallelism tries to reduce the idle time of PEs caused by synchronization among them. 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 parallelism. We show the 3M(min/max/median) values of benchmarks in Table II to help understand. Besides, we show changes in the number of non-zero elements per row of `poisson3Da` and `raefsky1` in Fig. 12 and Fig. 13. The figures show that the changes in neighbor rows in `poisson3Da` are more drastic than those in `raefsky1`. As a result, `poisson3Da` gains more improvement with element-wise parallelism. Overall, the average performance speedup of the proposed element-wise parallelism is 1.19x.

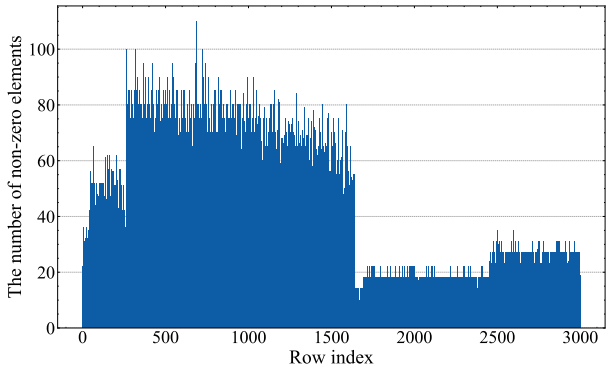


Fig. 12: Variation in Non-zeros per Row of `poisson3Da`

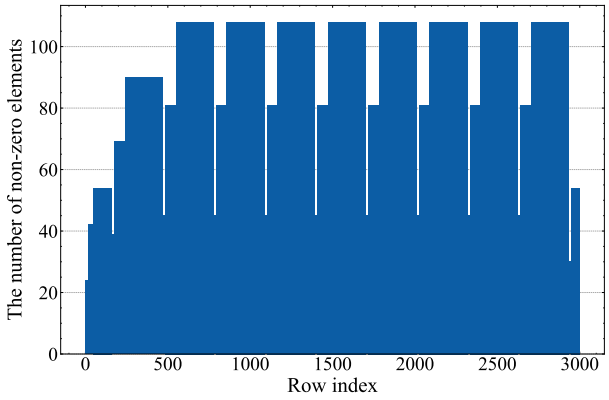


Fig. 13: Variation in Non-zeros per Row of `raefsky1`

Despite the parallelism, the proposed SpCache aims to reduce the overall off-chip memory traffic. As shown in Fig. 11, 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. 14. 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.

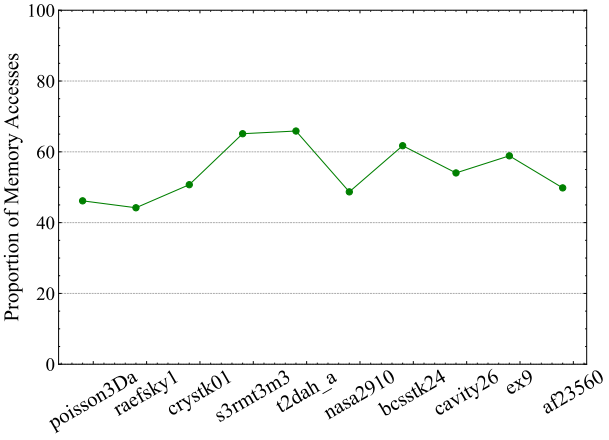


Fig. 14: Proportion of Memory Accesses

As studied in InnerSP [11], the authors find that most sparse matrices in the widely used SuiteSparse matrix collections [14] have relatively good locality. Since the size of *rptr* is much smaller than *val* and *cid*, we can achieve a high hit rate with a small rcache and thus mainly study the hit rate of vccache. The selected benchmarks show good locality using the 2MB vccache. We also test the performance on a 512KB vccache and a 1MB vccache separately. The benchmarks except for `poisson3Da` also show good locality. The performance of `poisson3Da` degrades around 10% if we use a 512KB vccache shown in Table III. Since memory accesses and computations are performed in parallel, the hit rate does not

significantly affect the performance which is not the same as cases in traditional CPU-based contexts.

TABLE III: `poisson3Da` with Different Caches

Vccache Size	Execution Cycles	Hit Rate (%)
512 KB	18,373,486	39.51
1 MB	17,690,833	56.43
2 MB	16,638,816	80.18

Further, we show the performance comparison using different merger designs in Fig. 15. Compared to the MatRaptor [10], our proposed design achieves up to 1.07x performance speedup and an average 1.02x performance speedup. Compared to the naive implementation, our proposed design achieves up to 1.19x performance speedup and an average 1.08x performance speedup. The key idea to optimize the merger is that we try to skip some merges by performing the merge algorithm on buffered partial results. For example, the optimized designs achieve a higher speedup on `poisson3Da`. However, if most partial results are supposed to be merged (i.e., cases like *CASE2*), we actually skip a few merges and we also need to merge the buffered streams. The MatRaptor [10] performs not better than the naive implementation on some benchmarks like `s3rmt3m3` and `bcsstk24`. Since our proposed design takes fewer cycles to post-process the buffered results in FIFOs, it performs better on all the benchmarks than the naive implementation.

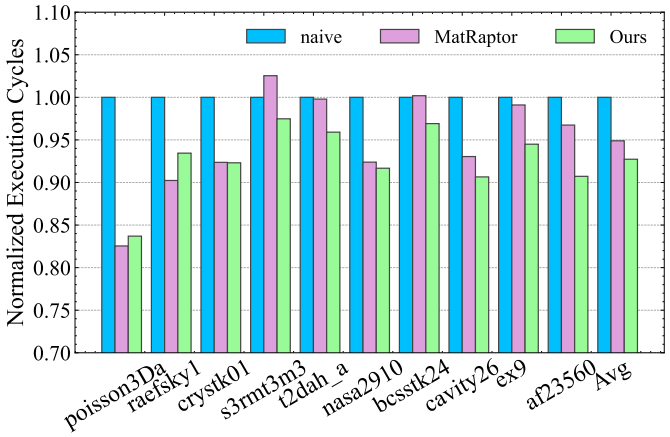


Fig. 15: Performance Comparison of Different Mergers

Finally, since this is the first work that studies the Gustavson's based SpMM on embedded FPGAs, we show the specific execution time (i.e., the number of cycles) in Table IV to help other researchers compare this work with theirs.

TABLE IV: Performance of Benchmarks

Benchmark	**rw**	**ew**	**GSCSp**	**GSCSp-om**
poisson3Da	40,309,030	24,725,926	19,878,983	16,638,816
raefsky1	33,132,096	33,126,121	27,458,639	25,658,832
crystk01	28,380,072	28,245,215	21,885,313	20,202,582
s3rmt3m3	12,533,231	12,321,634	8,628,745	8,411,161
t2dah_a	9,490,391	7,987,777	5,582,450	5,354,111
nasa2910	18,965,648	16,272,120	12,797,230	11,731,798
bcsstk24	10,713,286	10,545,166	7,588,259	7,354,144
cavity26	12,137,426	9,154,143	6,632,677	6,012,791
ex9	7,661,489	5,780,135	4,011,290	3,790,519
af23560	28,694,116	24,226,156	15,408,694	13,979,280

C. Discussion

With element-wise parallelism, elements of A are streamingly accessed. For each element, the main bottleneck within the fully-pipelined design is the off-chip memory traffic towards arrays of B. With the help of the proposed SpCache, we reduce the off-chip memory traffic. Consequently, we argue that the reusability of B affects performance as shown in Table III rather than the density of sparse matrices.

VI. Conclusion

In this work, we analyze the parallelism of the Gustavson's algorithm and propose to perform the algorithm with element-wise parallelism. Then, we show a counter-intuitive example that traditional cache leads to worse performance. Further, 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. Besides, we propose to perform the merge on part of partial results. 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 1.75x performance speedup compared to the baseline implementation.

Acknowledgement

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 (M4082282/04INS000515C130).

References

[1] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "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 A. Nicolau, "R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," *Algorithmica*, vol. 47, pp. 203–213, 2007.

[3] H. Kaplan, M. Sharir, and E. Verbin, "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. M. Van Dongen, "Graph clustering by flow simulation," Ph.D. dissertation, 2000.

[5] V. Hapla, D. Horák, and M. Merta, "Use of direct solvers in tfeti massively parallel implementation," in *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers 11*. Springer, 2013, pp. 192–205.

[6] M. O. Rabin and V. V. Vazirani, "Maximum matchings in general graphs through randomization," *Journal of algorithms*, vol. 10, no. 4, pp. 557–567, 1989.

[7] G. Karypis, A. Gupta, and V. Kumar, "A parallel formulation of interior point algorithms," in *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. IEEE, 1994, pp. 204–213.

[8] S. Itoh, P. Ordejón, and R. M. Martin, "Order-n tight-binding molecular dynamics on parallel computers," *Computer physics communications*, vol. 88, no. 2-3, pp. 173–185, 1995.

[9] 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.

[10] 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)*. IEEE, 2020, pp. 766–780.

[11] D. Baek, S. Hwang, T. Heo, D. Kim, and J. Huh, "Innersp: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 116–128.

[12] E. B. Tavakoli, M. Riera, M. H. Quraishi, and F. Ren, "Fspgemm: An opencl-based hpc framework for accelerating general sparse matrix-matrix multiplication on fpgas," *arXiv preprint arXiv:2112.10037*, 2021.

[13] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 687–701.

[14] 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.

[15] P. Grigoras, P. Burovskiy, and W. Luk, "Cask: Open-source custom architectures for sparse kernels," in *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, 2016, pp. 179–184.

[16] K. Lu, Z. Li, L. Liu, J. Wang, S. Yin, and S. Wei, "Redesk: A reconfigurable dataflow engine for sparse kernels on heterogeneous platforms," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[17] S. Li, Y. Wang, W. Wen, Y. Wang, Y. Chen, and H. Li, "A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 2016, pp. 1–6.

[18] S. Li, D. Liu, and W. Liu, "Optimized data reuse via reordering for sparse matrix-vector multiplication on fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[19] C. Y. Lin, N. Wong, and H. K.-H. So, "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.

[20] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[21] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.

[22] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 65–77.

[23] Y. Gao, L. Gong, C. Wang, T. Wang, and X. Zhou, "Sdma: An efficient and flexible sparse-dense matrix-multiplication architecture for gnns," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 307–312.

[24] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 63–72.

[25] S. Huai, L. Zhang, D. Liu, W. Liu, and R. Subramaniam, "Zerobn: Learning compact neural networks for latency-critical edge systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 151–156.

[26] M. Asiatici and P. Ienne, "Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 310–319.

[27] Xilinx, "Vitis Unified Software Platform," https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html, 2023, [Online; last accessed on 19-May-2023].

[28] S. Li and W. Liu, "Accelerating gustavson-based spmm on embedded fpgas with element-wise parallelism and access pattern-aware caches," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023.

Shiqing Li received the B.S. and M.S. degrees in computer science and technology from Shandong University, Jinan, China, in 2016 and 2019, respectively. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering at Nanyang Technological University, Singapore. His current research interests include sparse matrix operations, FPGAs, and high-level synthesis.

Shuo Huai received the B.E. degrees from the School of Computer Science at Shandong University, China, in 2019. He is currently a Ph.D. student at the School of Computer Science and Engineering at Nanyang Technological University, Singapore. His research interests are efficient deep learning algorithms, embedded intelligence and in-memory computing.

Weichen Liu received his BEng and MEng degrees from Harbin Institute of Technology, China, and PhD degree from the Hong Kong University of Science and Technology, Hong Kong SAR. He is currently an Associate Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He authored and co-authored more than 140 research papers in peer-reviewed journals, conferences, and books. His research interests include embedded and real-time systems, multiprocessor systems, network-on-chip, and machine learning acceleration.