Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

BRAM	Block Random Access Memory	9, 13, III
CNN	Convolutional Neural Network	3
COO	Coordinate List	5
CSC	Compressed Sparse Column	5
CSR	Compressed Sparse Row	5, 12, 14
DRAM	Dynamic Random Access Memory	1
DSP	Digital Signal Processing	6, 9
EDDO	Explicit Decoupled Data Orchestration	10, 13
FIFO	First-In-First-Out	12, 13, 15–17, 20, IV
FPGA	Field Programmable Gate Array	1, 2, 20
HBM	High-Bandwidth Memory	1, 2, 6, 7, 9
HDL	Hardware Description Language	2
HLS	High Level Synthesis	2, 12, 17, 18, 20
LUT	Look-Up-Table	9
MAC	Multiply-Accumulate	3
NoC	Network-on-Chip	20
OoO	Out-of-Order	7, 13
PE	Processing Element	10, 12, 15, IV
PU	Processing Unit	1, 2, 4, 6–17, 19, 20
RAW	Read-After-Write	1, 2, 8
SIMD	Single Instruction Multiple Data	13
SpGEMM	Sparse-Sparse General Matrix Multiplication	1, 3, 6, 20
WAR	Write-After-Read	2
WAW	Write-After-Write	2

1 Introduction

Sparse-Sparse General Matrix Multiplication (SpGEMM) plays a key role in graph analytics [1], machine learning [7] and scientific computation. Many problems, such as ranking and recommendation systems, can be modelled as Markov chains. Each node representing an entity and the edge representing the probability of transition between entities. Most entities present no transitional relation between them, hence may be represented as sparse matrices. Therefore, SpGEMM can be used for page ranking [11] and recommendation networks [6].

In the Amazon co-purchase network [5], each product is represented as a node of the graph and the edge representing the probability of them being bought together. The network consists of more than 100K nodes and approximately 3M edges giving rise to a sparse matrix of density 0.002%. For low density matrices, the SpGEMM computation becomes highly memory-bound requiring efficient utilization of memory bandwidth to achieve better performance.

1.1 Challenges

Parallel computing puts another degree of constraint on memory access due to the overlapping memory access requests from different Processing Units (PUs), resulting in memory bank conflicts and increased latency. This forces many SpGEMM problems to require in general either, a high memory bandwidth, very fast memory or heavy pre-processing to facilitate better scheduling and load balancing among the PUs.

- **Scheduling and Pre-Processing –** Can heavily influence load balancing and memory access due to memory dependencies such Read-After-Write (RAW). Improper scheduling could lead to memory bank conflicts, expose memory access latency and decrease the overall throughput of the system. Proper scheduling of the system facilitates and improves hardware flexibility by enabling the system to handle a wider range of problem sizes without altering the design flow template and improves latency masking, at the cost of higher pre-processing and resource utilization. Contemporary designs implement more sophisticated scheduling and pre-processing methods such as non-zero based Out-of-Order (OoO) scheduling [13] and adaptive tiling [4] to maximize resource utilization.
- **Memory Bandwidth –** Large problem sets have an inherent pre-requisite for high memory bandwidth to transfer data to and from the PUs. Modern Field Programmable Gate Arrays (FPGAs) provide High-Bandwidth Memory (HBM) (figure 1.1), a 3-D stacked memory die on a Dynamic Random Access Memory (DRAM) chip, that can provide wide bit-vector access to memory. To provide scale, an HBM stack of four DRAM dies (4-Hi) has two 128-bit channels per die for a total of 8 channels and a width of 1024 bits in

total. HBM based designs [12] can work on larger matrices but must mask the high access latency incurred by employing more sophisticated pre-processing.

- **Conflicts and Caching –** Although large memory banks can provide the necessary bandwidth, high port fan-out expected from parallel computing forces designs to implement port-conflict resolution by access scheduling to avoid RAW, Write-After-Read (WAR), Write-After-Write (WAW) hazards and keep the Iteration Interval (II) of the pipeline to a minimum. Data caching in-addition to bridging the memory access latency gap, can improve port-conflicts by providing a multi-bank solution.

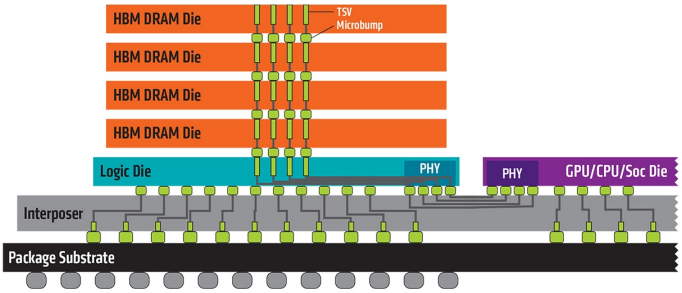


Figure 1.1: High-Bandwidth Memory structure (https://www.amd.com/)

1.2 Motivation and Goal

Contemporary works use a mixture of sophisticated pre-processing, scheduling, HBMs, and caching to leverage as much as possible from modern FPGA devices. Both vendors Xilinx and Intel provide FPGAs with HBM dies for high performance vector computing and a plethora of on-chip storage and resources, scaling well for problems with large data-sets that can maximize the memory usage effectively. In contrast, such solutions involving HBMs and high radix vector processing [14, 13, 16] cannot be realized on smaller development boards for smaller problem sets.

The motivation for this work is to explore the possibility to reduce memory latency by reducing the number of memory access requests. Following the approach of employing caching structures along with a suitable network framework to effectively utilize the available resources on a smaller FPGA. Thus, freeing up the memory ports for other designs or modules present on the board.

The goal is to implement a computing solution that can be scaled in terms of, number of PUs, problem matrix size and resources. These three criteria form the basis for the design process, along with the added design constraint to use Xilinx Vitis High Level Synthesis (HLS) workflow to explore the complex system designing capabilities of HLS, beyond its ability to realize modular algorithms into RTL IP and ease of use over conventional Hardware Description Language (HDL) such as Verilog and VHDL.

2 Background

This chapter explores the most widely used sparse matrix multiplication algorithms, their advantages and disadvantages in section 2.1 and describes briefly the major storage formats used in tandem with said algorithms to represent sparse matrices in section 2.2.

2.1 Multiplication Algorithms

As SpGEMM is widely used in many applications, multiple accelerator architectures have been proposed. Commonly used architectures, depending on their constraints towards information re-use and resource utilization, select one of the three major algorithms described below for implementation.

2.1.1 Inner-product

Inner-product (figure 2.1), also known as the dot product, multiplies a row vector with a column vector to produce a scalar. This method suffers from wasteful index matching operations, resulting from the situation shown in figure 2.4 (a), produces a null output at the end resulting in wasted operations. But, dot-product provides effective input reuse, banking on the high utilization factor of Multiply-Accumulate (MAC) operation. In the case of Convolutional Neural Network (CNN), Gondimalla *et al* [3] take the inner-product approach to achieve high input reuse.

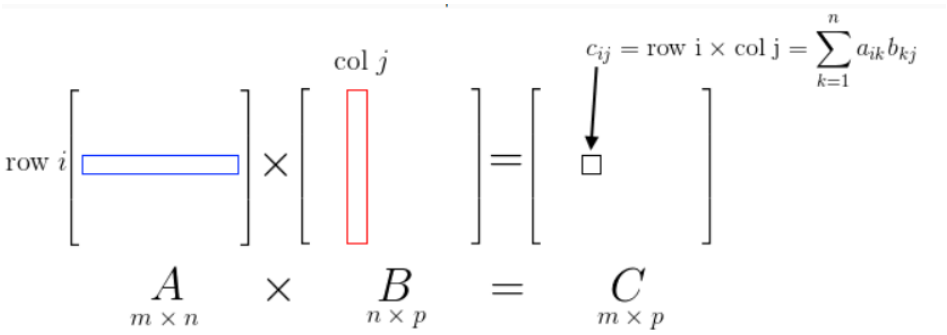


Figure 2.1: Dot-product multiplication (http://mlwiki.org/)

2.1.2 Outer-product

Outer-product (figure 2.2) produces a matrix partial product by the vector multiplication of a column with its corresponding row. This approach requires a large intermediary memory to store the partial product matrix and extensive synchronisation between PUs (figure 2.4 (b)) to avoid memory access hazards. Outer-product based approaches have better output reuse from the generated partial matrices. Works like OuterSPACE [8] and SpARCH [16] leverage the the output re-use of the outer-product approach by applying varies optimization techniques.

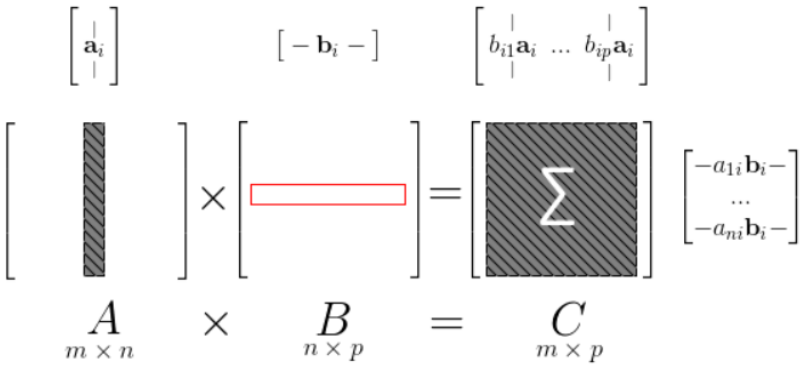


Figure 2.2: Outer-product multiplication (http://mlwiki.org/)

2.1.3 Row-Wise product

The Gustavson's method (figure 2.3) performs row-wise multiplication resulting in a row partial products, that are accumulated to produce the resultant row. This algorithm provides a good balance of MAC reuse and small intermediary memory usage while removing the need for inter PU synchronisation by one-to-one mapping matrix rows to PUs, resulting in a simpler yet faster architecture for general purpose applications. Both the inner and outer product work best for dense matrices, but have limitations when compared to row-wise product approach for very sparse matrices.

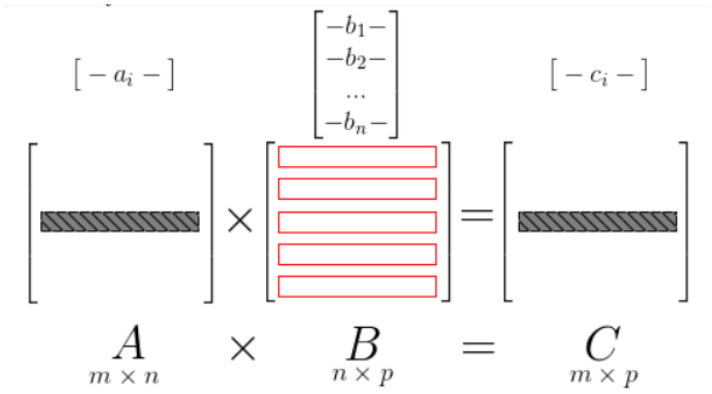


Figure 2.3: Row-wise Product multiplication (http://mlwiki.org/)

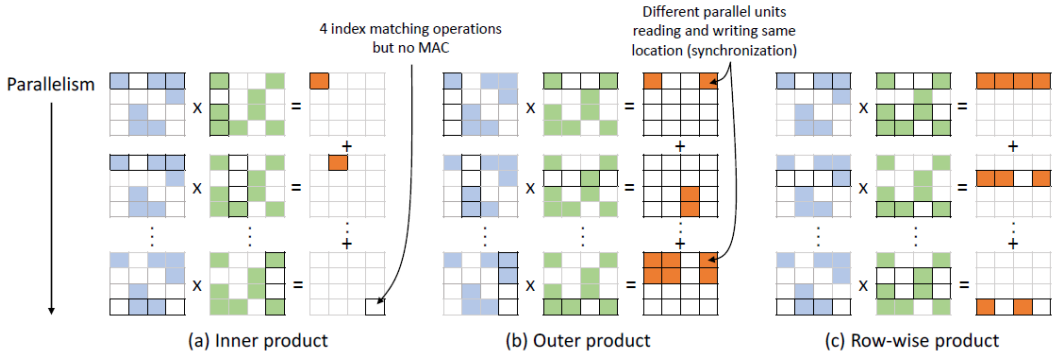


Figure 2.4: The non-zero elements in the two input matrices are shown in blue and green colors, the non-zero elements in the output matrix are shown in orange color, and the elements of the matrices involved in the computation are shown with dark borders. [14]

2.2 Matrix Storage Formats

Sparse matrices by virtue of their nature, can be compressed by removing zero elements, producing a resultant matrix consisting of only non-zero elements. This opens up the possibility to store sparse matrices in multiple formats (figure 2.5), both uncompressed and compressed.

2.2.1 Co-Ordinate Format

Coordinate List (COO) is an uncompressed storage format, widely used for inner-product multiplication, since the existence of both co-ordinates make index matching easier. The matrix elements are represented by their Cartesian co-ordinates and corresponding values, requiring three arrays 'value', 'row' and 'column' for representation.

2.2.2 Compressed Sparse Row and Column Formats

Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats are complementary and are the most preferred storage format due to their ease of use, depending on the approach taken on SpGEMM. The outer-product uses both CSR and CSC formats to store matrices, while the row-wise approach can be implemented using CSR format.

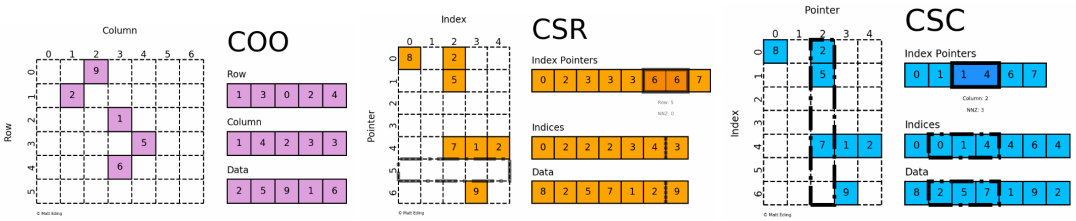


Figure 2.5: Sparse Matrix Storage formats: (a) Co-ordinate format. (b) Compressed Sparse Row format. (c) Compressed Sparse Column format (https://matteding.github.io/)

3 Related Work

This chapter explores works based on the row-wise product approach, how they target the simplicity of scalar multiplication and partial product merge stage, to attain a boost in performance. MatRaptor [14] (section 3.1), GAMMA [15] (section 3.2) and SEXTANS [13] (section 3.3) are works that showcase the simplicity and effectiveness of implementing the Gustavson's approach coupled with HBM for SpGEMM accelerator design.

3.1 MatRaptor

MatRaptor (figure 3.1) proposes a light-weight PU design, coupled with its custom sparse matrix storage format Circular Compressed Sparse Row (C^2SR) to efficiently schedule PUs while avoiding HBM channel conflicts. The simplicity of the PU can be implemented, but the memory access method presents a bottle neck without the presence of an HBM.

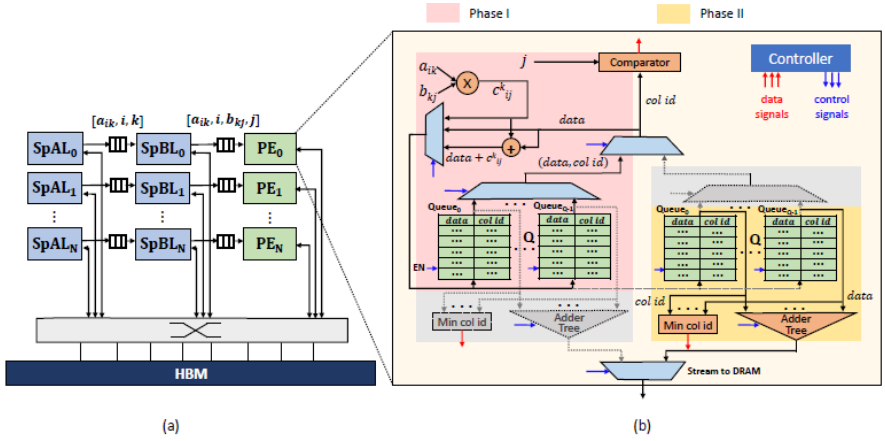


Figure 3.1: MatRaptor Architecture: (a) Microarchitecture consisting of Sparse A Loaders (SpALs), Sparse B Loaders (SpBLs), Processing Elements (PEs), system crossbar and high-bandwidth memory (HBM). (b) Microarchitecture of a single PE consisting of multipliers, adders, comparator and queues on the left performing phase I of the multiply and merge operations, and the queues on the right, adder tree and minimum column index logic performing phase II of the merge operations [14].

The design performs scalar multiplications on non-zero indices of matrix A and B, which can be vectorized to take advantage of the on-board Digital Signal Processing (DSP) modules and

employs a simple, ping-pong buffer solution for the merge phase. This presents us with good insight into the advantages of system partitioning, with task-level autonomy, resulting in a simplified control structure and independent scheduling of tasks.

3.2 Gamma

The GAMMA architecture (figure 3.2) achieves high performance using high radix PUs in conjunction with HBM and intermittent data caching. The architecture depends heavily on the caching framework *'FIBERCACHE'* and its novel pre-processing technique to stream blocks of data in bulk, from matrix B corresponding to the rows of matrix A into the cache.

An affinity based pre-processing technique is used to determine a permutation of rows of matrix B, which share the highest affinity among multiple rows of matrix A. The corresponding rows of matrix B, based on the caching window size, are brought into the cache. In addition, a priority counter is implemented within the cache to keep it updated, to minimize the probability of a cache miss. The fibercache is also used to store the partial products produced, thus improving the output reuse. From Gamma, we get a feeling for the importance of data-orchestration within the system to improve throughput and reduce memory traffic.

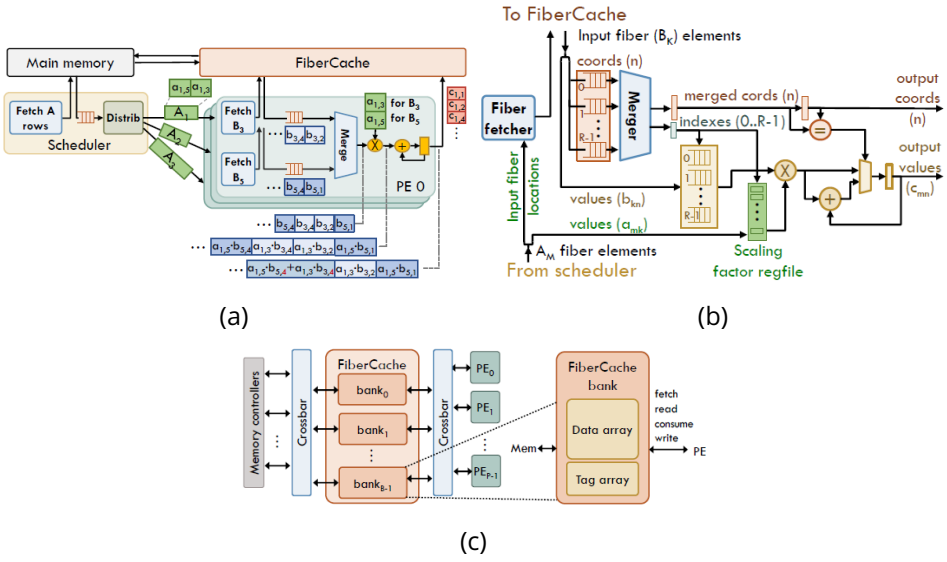


Figure 3.2: GAMMA System Overview: (a) System architecture showcasing control-data decoupling by fetch and read stage prior to merger. (b) Processing Element internal architecture. (c) Fiber Cache architecture realized using a multi-bank solution with relevant networking framework [15].

3.3 Sextans

Compared to previously discussed works, Sextans (figure 3.3) tackles the problem from a scheduling perspective. It switches from a row based parallelization into a non-zero based parallelization. Essentially, it implements an *'Out-of-Order (OoO)'* schedule by assigning the

non-zero elements to PUs based on the RAW distance 'D' between elements. This improves load balancing by reducing PU idle times and removes memory conflicts at the root.

Further, Sextans partitions the matrix A into sub-matrices for higher granularity during scheduling. This finer control coupled with pointer based book-keeping allows Sextans to achieve hardware flexibility to cater for multiple matrices, vectoring the whole system towards general purpose utilization.

The grouping of multiple processing elements into groups allows the system to better share scratchpad memory resources within the group, enabling the scratchpads to act as a pseudo cache, improving memory access traffic and overall memory resource utilization.

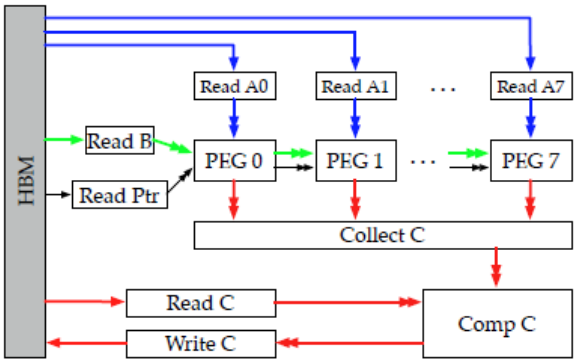


Figure 3.3: SEXTANS System architecture [13].

3.4 Design Inferences and Similarities

Based on the understanding attained from the prior works discussed above, some system characteristics were extracted and abstractly used as the ground work to implement certain design choices (described in detail in chapter 4 and chapter 5):

- ***Task-level Autonomy*** **–** The concept of autonomous tasks was extracted from MatRaptor to implement a three phase system with *'Setup'*, *'Multiply'* and *'Compare'* phases which are autonomous with respect to each other and only operate upon their streamed data inputs, producing streamed data outputs consumed by the autonomous successors.
- ***Scratchpad Memory (Traps)*** **–** The cache concept from Gamma was inferred and implemented using a distributed local scratchpad memory, similar to the shared memory concept of Sextans. To mask the memory access latency and reduce memory traffic while minimizing memory resource utilization.
- ***Interconnect*** **–** A suitable packet switching framework is developed to benefit from the distributed scratchpad memory architecture by sharing local rows of matrix B.
- ***Sliding Window and Row Lifetime*** **–** A sliding window is used to refresh the scratchpad memories with new rows of matrix B. The new rows brought in from memory depends upon the lifetime of a row in the scratchpad.

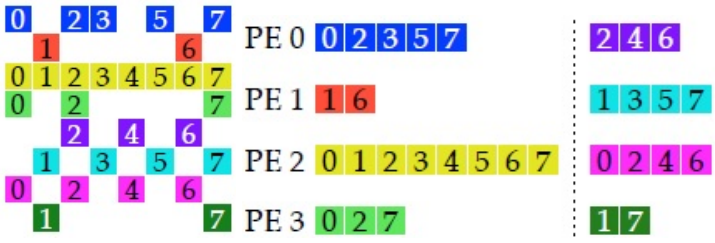
4 Design Methodology

In this chapter we will discuss the base constructs, assumptions and the factors backing design decisions. First, we re-iterate over the goal for this project, it is to reduce the memory traffic between the system accelerating matrix computation and main memory holding the required data. Due to the constraints on logic resources and absence of HBM, an intermittent (local) storage based solution is adopted while keeping the design modular and scalable. Following the bottom-up paradigm, the problem is divided into parts and addressed individually in the upcoming sections while system integration and implementation is explored in chapter 5.

4.1 Batch Processing

Due to limited resources on the target board ZYNQ-7 ZC702 Evaluation Board, unlike other sophisticated accelerator designs, we must take a humble approach towards matrix processing. We must fit our resource consumption within 140 BRAMs, 220 DSPs, 53200 Look-Up-Tables (LUTs). We limit the number of PUs to P and perform 1-D partitioning on the sparse matrix *A* consisting of N rows by assigning each non-zero row to an individual PU. This results in processing the matrix in 'U' batches, with each batch processing 'P' rows per batch. For example, a 20×20 matrix will be processed by 4 PUs in 5 batches. Figure 4.1 gives a pictorial representation of a 8×8 matrix split into 2 batches.

$$Number\ of\ Batches\ (U) = \left\lceil \frac{Number\ of\ Rows\ in\ Matrix\ A\ (N)}{Number\ of\ PUs\ (P)} \right\rceil$$



(a) (b)

Figure 4.1: Batch Processing: (a) 8×8 matrix with each row colour coded. (b) Assignment of rows to individual PUs and matrix split into 2 Batches [13].

4.2 Memory Traffic Reduction

Memory latency in accelerators can be reduced by using Explicit Decoupled Data Orchestration (EDDO) [9], taking advantage of prior workload knowledge and generating a suitable memory hierarchy. Scratchpad memories (*Traps*) paired with a simple interconnect suit well for accelerator design since they fall under the EDDO paradigm and gap the latency while reducing memory traffic. In order to have a scalable design, a modular approach is adopted by having PEs and their *Traps* packaged into one unit while being spatially distributed and connected with one another through the interconnect.

4.2.1 Sliding Window

Gustavson's algorithm on matrices *A*, *B* and *C* of size $N \times N$, given by the relation:

$$C_i = \sum_{i=0}^{N-1} a_{ij} * B_j \quad for\ j = 0, ..., N-1$$

Gives insight into access ordering during computation. Assuming in-order scheduling of operations, tasks will access every element a_{ij} of *A* and row B_j of *B* in sequence. If row B_j is accessed by PU 'P', the next access will be for row B_{j+1} and the same PU will not access row B_j again. This presents the intuition to create a storage window 'W' that is shared across all PUs of a block and slide the window across all rows of matrix *B*.

To perform effective window sliding with maximum row coverage in a block 'V' (containing V_P number of PUs), no duplicate copies of the same row should exist within the block. Each row B_j in the block is given a *lifetime* '$t_{B_j}|_V$', a counter denoting the total number of times it will be accessed by the block. For example, if block V_0 contains P_0, P_1, P_2 and P_3, with 3 of them accessing row B_1 of matrix *B*, will produce $t_{B_1}|_0 = 3$. If each PU has 'S' *Traps* with each trap holding one row of *B* and its lifetime, the sliding window is realized as shown in algorithm 1.

Algorithm 1 Sliding Window

```
for each S do
    for every B_j access do
        t_{B_j}|_V ← t_{B_j}|_V − 1
    end for
    if t_{B_j}|_V = 0 then
        S ← New B_j from memory
    end if
end for
```

4.2.2 Interconnect

The interconnect essentially provides channels and switching logic for communication of trapped data across PUs and serves as a bridge between PUs and the slow main-memory. The interconnect itself must exhibit the following characteristics:

- ***Light*** – The interconnect should introduce minimum communication overhead in order to not negatively impact the traffic reduction achieved by the caching system.
- ***Flexible –*** The interconnect must be flexible in the number of PUs it can handle and should not be affected by the order in which requests appear. Flexibility is obtained by using streaming constructs for channels connecting PUs, interconnect and main-memory. Streaming constructs in-addition to acting as channels, also double as buffers.
- ***Robust*** – It is paramount to the design, uncertainties and duplicate requests are handled by the interconnect. By virute of the local memory replacement policy, no repeat rows can exist in the traps, hence if two PUs request the same row, only one of them can receive the row from main-memory, while the other PU must request a different row. The PUs do not have a timeout mechanism to perform automated re-requests in-case the request fails at any hierarchy of the system, hence it falls upon the interconnect to manage these uncertainties.

Hence, the system operates on the assumption that every request will be serviced and a reply will reach the requesting PU. In the case a request cannot be serviced, results in a '*failed*' reply to the requesting PU. Notifying the PU to handle the failure accordingly. The details of request failure handling will be addressed in section 4.3.

4.3 Proto-System Design and Simulation

The initial proof of concept is simulated using Python version 3 with the help of '*SciPy*' library for sparse matrix generation and *'multiprocessing'* library to simulate parallel processing by the PUs. The simulated design was later ported over to C++ in Vitis HLS version 2022.1 for further implementation and evaluation. With the software system and testbench complete, the trapping structure is split into three parts, row data, tag (row number) and lifetime. Two methods of handling duplicate requests while maintaining unique copies of rows in the traps were attempted:

- ***Method 1 –*** A shared memory pool was used to simulate a trapping structure (local memory) that can be accessed by all PUs. Unique row requests based on the PU local row lifetime are made after checking all tags in memory pool. But, there can be duplicates as two or more PUs may request the same row. This will result in a failed reply from the interconnect to notify the PU to request a different row. In order to request another unique row, the PU must comb through all tags once again to find a unique row. This transforms into a contention issue with all PUs fighting for a valid request. To make things worse, closer inspection shows each trap (memory pool) would need to broadcast tags to all PUs. On synthesis, this would result in a high fan-out (multi-port) memory resulting in duplicated banks (scales with P) and a spike in resource utilization.
- ***Method 2 –*** Implement trapping structure local to PU and duplicate tag and lifetime information alone into interconnect. Upon request conflict, the interconnect resolves it and makes an auto-request with unique row number to main-memory. This avoids contention issue and reduces resource usage since, only the interconnect needs a copy of tags and lifetime information. This approach is more scalable and is chosen for implementation.

5 Implementation

In this chapter, the system architecture takes form based upon the proto-system design discussed in chapter 4. After a brief description of the architecture as a whole, individual sub-modules are analyzed in-depth including hardware implementation of algorithms and finally integrated into the final design. The process consists of porting design ideas from Python over to Vitis HLS using C/C++ constructs, while integrating support for late stage testing and evaluation. This is achieved by inserting added logic to sub-modules by integrating test ports and scalable logic (realized using pre-processor directives) wherever necessary.

5.1 System Architecture

Based on insights obtained from proto-design, the CSR storage format for matrices is chosen along with the development of a suitable architecture (figure 5.1), that is modular and scalable with respect to resource constraints. The communication channels within the system are modeled using FIFO queues to support data streaming in order to realize task-level autonomy (section 3.4) and provide buffering to mask timing issues. FIFO implementation using '*hls::stream*' class was attempted, but failed to pass testing due to construct incompatibility, variable lifetime issues and C compiler optimizations prior to synthesis. Finally, all FIFOs were realized using arrays, giving the advantage of '*visibility*' into system operation during testing.

5.1.1 Processing Unit

The PU encloses PE and the '*trap*' (scratchpad memory) together. The trap is a structure consisting of '*row*' to hold the row of matrix *B*, *Tag* describing the row number and *Life-Time* showing row validity. The row information is split to allow different routines (sub-tasks) independent access to required type of row information.

The PE consists of three tasks, '*Setup*', '*Multiply*' and '*Compare*', connected through carefully sized FIFOs in that order. The tasks are designed to cause side-effects only if data is available for processing at their inputs, removing the need for a centralized control structure, allowing individual tasks to be autonomous. The individual tasks and their operations are as follows:

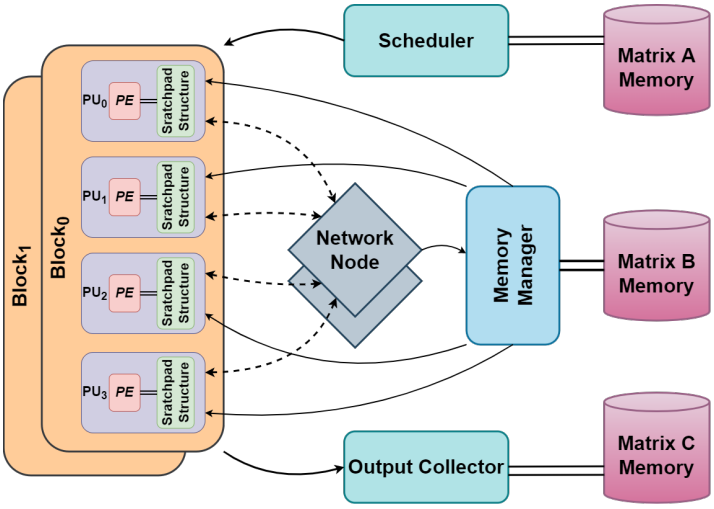


Figure 5.1: Architecture overview: Double block design with 4 PUs per and 1 interconnect node per block. Solid uni-directional lines for External channels, dotted bi-directional lines for communication channels and double lines for memory bus. All memory buses have single port access to reduce BRAM consumption.

Setup: The task is broken into routines (sub-tasks) that co-ordinate together to realize the EDDO paradigm. The data request and read operations are split into independent routines, controlled by a common predecessor routine as shown in figure 5.3. To parallelize the routines, internal copies of the trap's information are made. Figure 5.2 represents the internal operation of *Setup* task and how data flows among different routines. In addition to reading replies from interconnect, *Update Memory* re-orders replies to maintain determinism and avoid deadlocks, since all external requests from the interconnect processed in-order by the *Service* routine. *Service* is implemented this way to remove overhead caused by OoO servicing. The reason for this will be explained shortly when discussing the '*Compare*' task. Furthermore, request and replies are systolic in nature, they contain information like which PU is requesting ('*Src*'), which trap location the requested row is to be stored at ('*Store Loc*') and some additional information that will be generated and consumed internally by the successive tasks within the *Interconnect*.

Multiply: This task generates partial products from the row-wise multiplication of matrix elements. Matrix information is stored using '*hls::vector*' class to facilitate Single Instruction Multiple Data (SIMD) operations on the input data. The partial product produced is streamed over to the successor '*Compare*' task through a FIFO.

Compare: Consists of two routines '*Merge*' and '*Format*'. This task, like the ones before it is autonomous, it operates purely based on data from predecessors and requires no external control signals.

- ***Merge –*** Accumulates partial products by recursively performing addition and comparison to produce the final product. There exists no re-ordering mechanism within the routine to order elements of the final product. Hence, it is important all preceding stages maintain ordering among the elements. The merge phase follows algorithm 2 by implementing a ping-pong buffer (FIFOs) to accumulate the partial products received

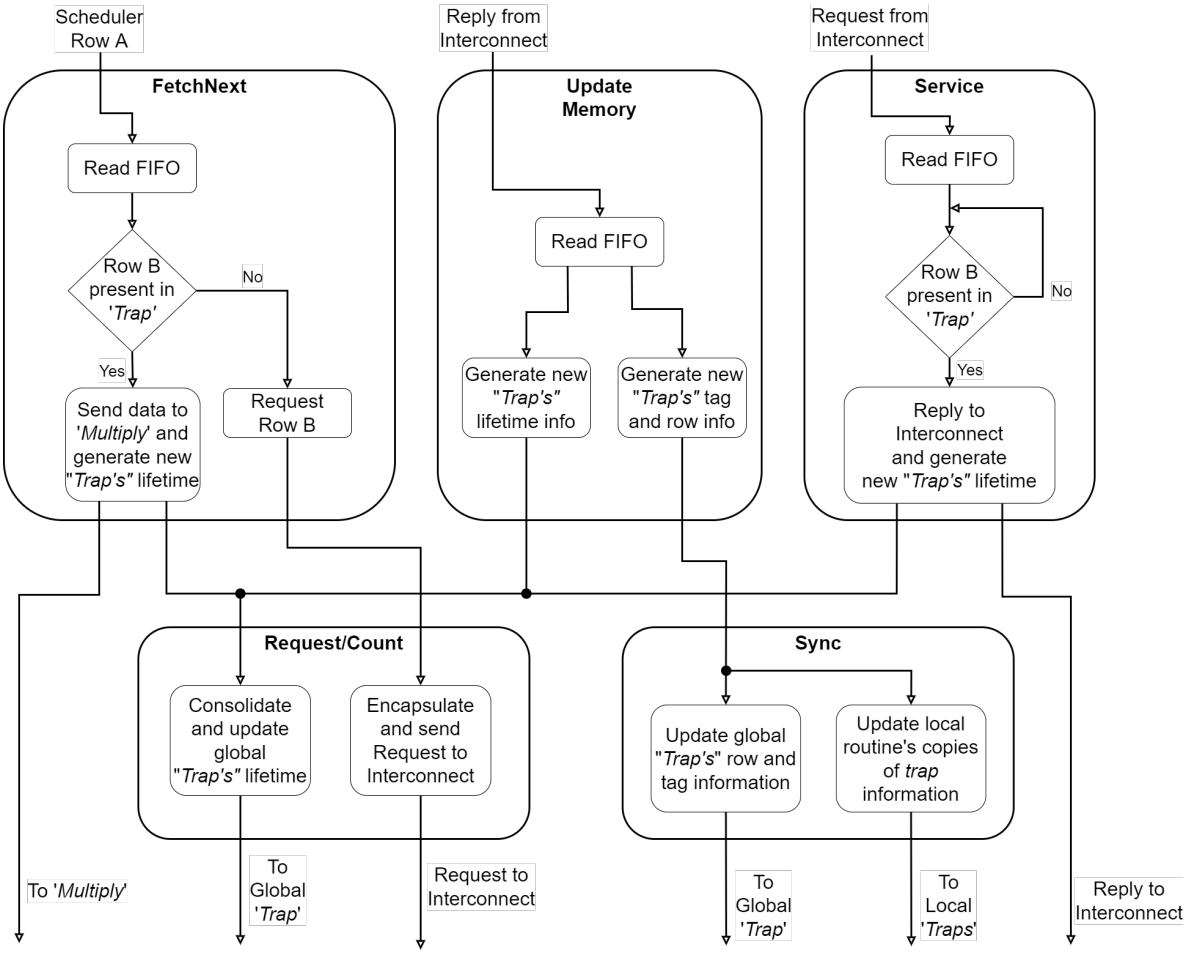


Figure 5.2: Data and control flow among routines within '*Setup*' task.

from '*In-buffer*'. One buffer acts as the main output buffer '*Cur-buffer*', while the other '*Re-buffer*' provides the previously accumulated result of merge as a second input.

- ***Format* –** This routine acts as an adapter to convert internal processing format to match the external storage type on memory (CSR). The routine reads the '*Cur-buffer*', formats and stores the output onto a separate '*Output-buffer*', which can be accessed by the '*Collector*' task for memory write-back.

5.1.2 Interconnect

In addition to routing packets, the interconnect along with PU takes care of the local memory (trap structure) replacement policy. It starts with the PU encapsulating the row request in a packet and transmitting it to interconnect node. The node (figure 5.4), following algorithm 3 routes the request and if necessary, generates an '*auto-request*' to '*Memory-Manager*'. Unlike requests, there is no added processing performed by the node on replies, it simply routes them to their destinations. In-order to perform its functions, the node has its very own copy of row tags from all PUs of the block it handles, formatted into a '*pu_mem*' lookup table along with another '*avail_mem*' table to aid in auto-requesting. Furthermore, to distribute the complexity

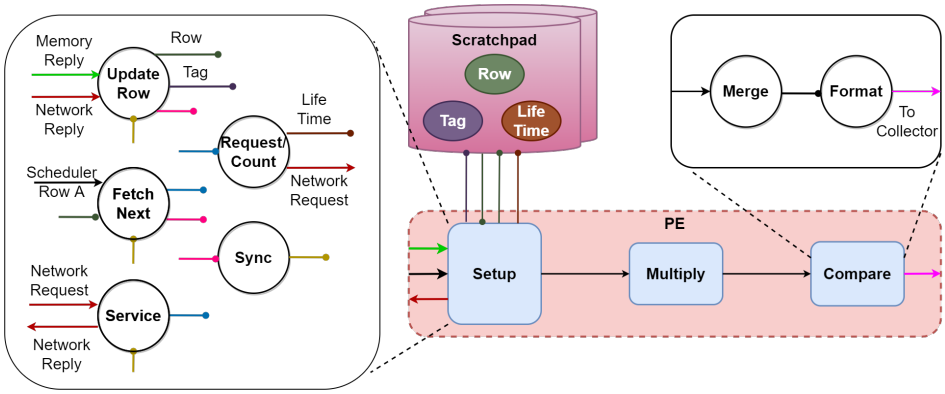


Figure 5.3: PE micro-architecture: Routines are enclosed in circles and all routines occurring in parallel are placed on the same column and the horizontal axis represents sequencing. Scratchpad memory (trap) is shown here only for reference. It sits outside the PE, enclosed by the PU. Lines with arrow-heads represent FIFO channels and circular flat-heads represent registered internal signals.

Algorithm 2 Merge Routine

for every $parital\text{-}product$ **do**
  while $In\text{-}buffer$ or $Re\text{-}buffer$ not empty **do**
    $in_1 \leftarrow In\text{-}buffer.pop(top - elem)$
    $in_2 \leftarrow Re\text{-}buffer.pop(top - elem)$
    if $in_1.index = in_2.index$ **then**
      $Cur\text{-}buffer.push() \leftarrow in_1 + in_2$
    else if $in_1.index < in_2.index$ **then**
      $Cur\text{-}buffer.push() \leftarrow in_1$
    else
      $Cur\text{-}buffer.push() \leftarrow in_2$
    end if
  end while
  $Cur\text{-}buffer \leftarrow Re\text{-}buffer$
  $Re\text{-}buffer \leftarrow Cur\text{-}buffer$
end for

while providing flexibility, the node is split into two tasks namely '*Handler*', handling internal table reads and '*Crossbar*', performing all internal table writes.

Handler: The handler is responsible for interacting with the PU. It receives a request packet, finds the destination node based on '*pu_mem*' table and provides the next possible auto-request row number to the '*Crossbar*' task which generates the request based on some criteria. The node attains flexibility in the sense that, provided there is availability of resources, the PUs and handlers may be run in parallel by instantiating multiple copies.

Crossbar: As the name suggests, the crossbar performs switching of packets within the interconnect node and performs '*auto-request*' generation (section 4.3). The crossbar performs the quintessential function of discrepancy handling caused by mismatch between *pu_mem*

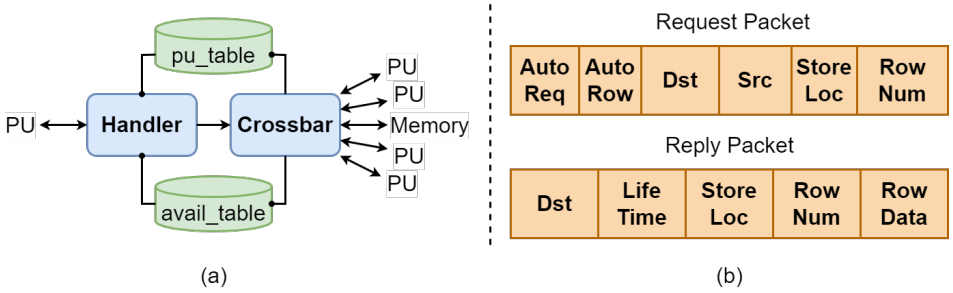


Figure 5.4: (a) Interconnect micro-architecture. (b) Packet structure for request and reply.

table reads in *Handler* task and writes in *Crossbar* task. To elaborate, if PU P_1 request row B_3 which was requested by P_2 in the previous cycle, the *Handler* would not see the change in *pu_table* until the next cycle (write happening by *Crossbar* in current cycle). This results in the *Handler* setting the wrong destination PU in the packet information forwarded to the *Crossbar*. Hence, it falls upon the *Crossbar* to compare current requests with previous requests and switch the packet to the proper destinations FIFO. The *Store Loc* field is also used by the *Crossbar* to control *auto-request* function, *i.e,* if a PU requests a row with temporary storage (when none of its trap locations are available for replacement) for instant use and the requested row must be brought in from main-memory, then the *Interconnect* and *Memory Manager* must update their internal copies of *lifetime* to handle this edge case.

Algorithm 3 Auto-Request Generation

$row_{auto} \leftarrow min(avail\_table)$
if $RequestPacket.row \in Previous\_Requests$ **then**
    $RequestPacket.row \leftarrow row_{auto}$
    $destination \leftarrow Previous\_Requests[RequestPacket.row].P$
else
    $destination \leftarrow Memory$
end if
if $destination = Memory$ **then**
    $Memory_{requestFIFO}.push() \leftarrow RequestPacket$
else
    for each P in block **do**
        if $destination = P$ **then**
            $P_{requestFIFO}.push() \leftarrow RequestPacket$
        end if
    end for
end if

5.1.3 Memory Manager

All accesses to matrix *B* from the system go through the Memory Manager. It dis-aggregates the request packets from interconnect and performs memory access to get rows from matrix *B*. The received rows are encapsulated in reply packets and sent back directly to the PUs. The interconnect is bypassed to avoid communication delays on the reply lines to reduce

idle times of tasks waiting on replies. As an added advantage, direct reply lines reduce the complexity of the interconnect node by removing the crossbar circuitry required for replies.

5.1.4 Scheduler

The Scheduler and Output-collector act as access point to the memories of matrix *A* and matrix *C*. At the beginning of each batch, the Scheduler reads matrix *A*, assigns a PU to every row based on row-scheduling (figure 4.1) and asserts the '*active*' signal to activate every PU that has a valid row scheduled for processing. In addition, it initializes every module at the start of every batch and populates the lifetime values used by both *Memory Manager* and *Interconnect* nodes.

5.1.5 Output-Collector

The Output-collector runs at the end of each batch, reads all output buffers from PUs and performs write-back to main memory. This task is separated from the rest to facilitate pipelining of the system such that another batch may start without having to wait for memory writeback to complete. In comparison to scheduler, the Output-collector performs relatively no extra processing and only exists to act as an adaptor between system and output matrix *C*.

5.2 Integration and Realization

To realize and test the discussed architecture in Vitis HLS using C++, a simple design with a single block, containing 4 PUs and connected together by 1 interconnect node is implemented. A larger replica of the architecture consisting of multiple blocks and interconnect nodes cannot be realized due to resource limitations presented by the dev-board. So, we reproduce the minimum units required to test the fidelity of the system to the theorized design structure.

Individual tasks and routines are realized as functions in C++ and integrated using wrapper functions. The '*class*' data-structure is used to model the PU and Interconnect modules, which are instantiated in the top-level function along with FIFOs and copies of internal registers used by PU and interconnect. The registers are passed on to the class methods upon invocation according to the PU being run, providing higher flexibility during scheduling (figure 5.5) of functions.

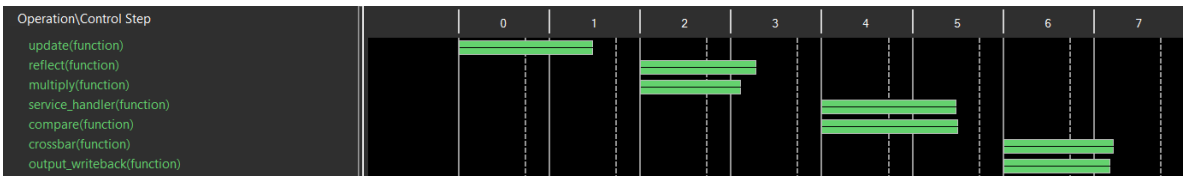


Figure 5.5: Static schedule of tasks. The '*Setup*' task has been split into two sub-tasks '*Update*' and '*Reflect*' for better pipelining.

6 Results and Conclusion

6.1 Evaluation

The design is tested by computing the square of sparse matrix *A* with variable density and size. The matrix *A* is generated using the '*SciPy*' library in python and tested in RTL using a C++ testbench, taking advantage of Vitis's C/RTL Co-Simulation feature. For evaluation the ZYNQ-7 ZC702 Evaluation Board was targeted (table 6.1) and for evaluation, two metrics are selected: '*Memory-Access-Count*', specifying number of times memory is accessed to obtain row B_j and '*Trapping-Count*' measures how many of those requests were stored in scratchpad memory. Another additional metric, the '*Trapping-Efficiency*' is chosen to exhibit the effectiveness of the sliding window. The reduction in memory traffic is measured by benchmarking against baseline values generated by a baseline run of the system, which performs no trapping and sends all requests to main-memory, effectively removing interconnect communication overhead.

Resource	Used	Available	Utilization (%)
BRAM	96	140	68.571
LUT	14033	53200	26.377
FF	21182	106400	19.907
DSP	1	220	0.454

Table 6.1: Resource Utilization on a ZYNQ-7 ZC702 Evaluation Board

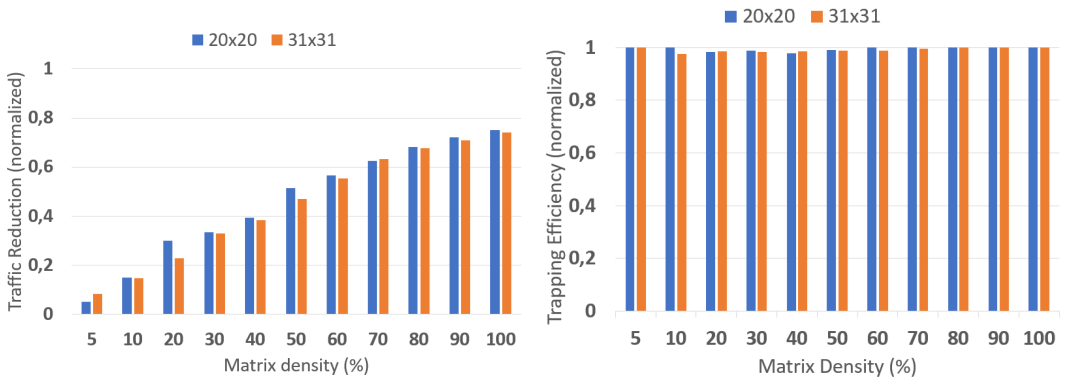
The generated Verilog code from Vitis HLS is exported to '*Synopsis Design Compiler*' (appendix A.1) for synthesis to obtain area and power metrics shown in table 6.2. The synthesis uses the '*hpsnlib_g28_9t_RVT_wc_0d76V_m40C*' library [2] based on 28nm technology node. Majority of the power utilization occurs from the FIFO queues used for data transfer, *Memory-Manager*, *Scheduler* and *Output-Collector* combined. This shows there is more room to improve the data transfer methods and memory request handling.

The '*Traffic Reduction*' (appendix A.2) is computed by measuring the difference between the total *Memory-Access-Count* of base design and that of the implemented design, normalized to values between 0 and 1. The values shown in figure 6.1 give an idea on how different densities affect performance, not shown here is the effect of distribution of non-zero elements on traffic. The '*Trapping Efficiency*' (appendix A.3) gives a measure of how well the sliding window is performing for different matrix densities. There is very negligible reduction in efficiency for the tested matrices since their sizes are small compared to large sparse matrices.

Component	Area (mm^2)		Power (mW)	
	MatRaptor	This work	Matraptor	This work
Processing Logic	1.997	0.326	1013.557	2.338
Loaders/Buffers	0.258	0.175	288.300	3.640
Total	2.255	0.501	1301.857	5.978

Table 6.2: Area and Power consumption compared to Matraptor [14]

Performance is best for higher densities since there exists more number of redundant access to same rows by multiple PUs in a batch. But, for lower densities, even though the *Trapping Efficiency* is higher, there is no significant *Traffic Reduction* since, there is no redundant access (row re-use) made to main-memory due to high sparsity of non-zero elements in a row. In such a case, the performance of a non-enhanced design (baseline) performs similar to a enhanced one (this work).



(a) (b)

Figure 6.1: (a) Memory traffic reduction measured for different matrix densities. (b) Sliding window efficiency measured for different matrix densities

For the sliding window to achieve maximum performance, there is an optimal range of values for number of traps S implemented in the design. In order to provide a frame of reference to figure out the optimal value, another metric 'Tnz' is introduced. This metric is defined as the total number of memory access performed by an ideal design for a matrix of fixed size and density. The frame of reference is provided by the maximum (Tnz_{max}) and minimum (Tnz_{min}) values of Tnz. Optimal performance is achieved as long as the measured *Memory-Access-Count* of the implemented design falls within Tnz_{max} and Tnz_{min} (figure 6.2).

If we assume the total number of non-zero elements per row to be nz_R, then nz_R memory accesses will be performed by the PU handling the row in an ideal system because such a system would access every row only once. This ideal behaviour can be witnessed in a design when traps $S \rightarrow N$, since the whole matrix may be stored in local storage and only one access per row occurs to bring in the row. Accounting for the different types of non-zero element distribution that could occur in the matrix, two boundary conditions arise for the total number of possible memory accesses an ideal system would perform:

- First is the minimum condition (Tnz_{min}) that occurs when the all the PUs access the same set of rows (the case when density reaches 100%), then each row would be requested

from main-memory only once. This is the reason why the system exhibits 100% trapping efficiency when operating on very dense matrices (figure 6.1b).

- Second is the maximum condition (Tnz_{max}) when nz_R distribution in a block V, is such that, intersection of all nz_R in the block maximizes the number of main-memory access. For example, if PU P_1 accesses rows B_1, B_2, B_3 and P_2 accesses B_4, B_5, B_6, then the there is no overlap between the rows and memory access is maximized. This results in the two boundary values for Tnz (methodology described in appendix A.4):

$$Tnz_{min} = nz_R$$

$$Tnz_{max} = \min(V_P * nz_R, N)$$

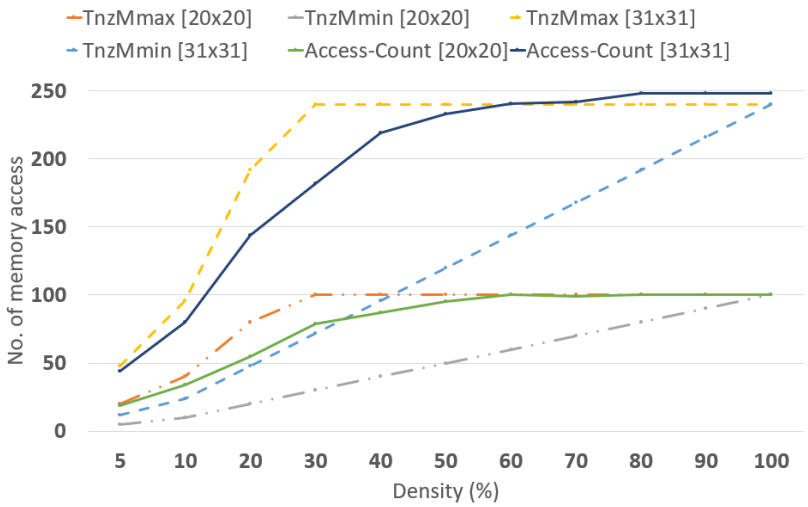


Figure 6.2: Sliding window performance with 2 traps 'S' is shown for two matrices of size 20×20 and 31×31 with varying densities. Sub-optimal performance is witnessed with the larger matrix (31×31) above the density range of 70% (*Memory-Access-Count overshoots* Tnz_{max}).

6.2 Conclusion and Future work

SpGEMM has key applications in a variety of fields and memory traffic is of prime concern when implemented on accelerators. A simple, yet effective method along with a suitable system design is presented to reduce memory traffic while keeping the design lightweight, flexible and robust for implementation on of-the shelf, small scale FPGA development boards. The results from evaluation agree with the hypothesized algorithm while minimizing area, power and resource utilization along with a boost in performance compared to designs from its own class. The resulting implementation shows the advantageous prospects of utilizing HLS tools to accelerate design flow and testing.

In the future, the design could be improved by implementing a better FIFO structure for communication and streaming to realize a systolic array for faster autonomous processing, paving the way to implement a Network-on-Chip (NoC) to integrate multiple processing blocks together, to process larger data sets faster. On the HLS side, '*fine-grained pointer synthesis*' [10] can be performed to synthesize smaller and faster system.

Bibliography

[1] Daniele Buono et al. “Optimizing Sparse Linear Algebra for Large-Scale Graph Analytics”. In: *Computer* 48.8 (Aug. 2015), pp. 26–34. ISSN: 0018-9162. DOI: 10.1109/MC.2015.228.

[2] TU-Dresden. “HPSNLIB: Standard-cell Library for synthesis.” In: *"https://tu-dresden.de/"* (2020).

[3] Ashish Gondimalla et al. “SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 151–165. ISBN: 9781450369381. DOI: 10.1145/3352460.3358291.

[4] Changwan Hong et al. “Adaptive Sparse Tiling for Sparse Matrix Multiplication”. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP ’19. Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 300–314. ISBN: 9781450362252. DOI: 10.1145/3293883.3295712.

[5] L. A. Adamic J. Leskovec and B. A. Huberman. “The Dynamics of Viral Marketing”. In: *ACMTransactions on the Web (TWEB)* (2007). DOI: 10.1145/1232722.1232727.

[6] Onur Küçüktunç et al. “Fast recommendation on bibliographic networks with sparse-matrix ordering and partitioning”. In: *Social Network Analysis and Mining* 3 (Dec. 2013). DOI: 10.1007/s13278-013-0106-z.

[7] Hui-Hsin Liao et al. “Support Convolution of CNN with Compression Sparse Matrix Multiplication Flow in TVM”. In: *50th International Conference on Parallel Processing Workshop*. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450384414.

[8] Subhankar Pal et al. “OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018), pp. 724–736.

[9] Michael Pellauer et al. “Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 137–151. ISBN: 9781450362405. DOI: 10.1145/3297858.3304025.

[10] Nadesh Ramanathan, George A. Constantinides, and John Wickerson. “A Case for Precise, Fine-Grained Pointer Synthesis in High-Level Synthesis”. In: *ACM Trans. Des. Autom. Electron. Syst.* 27.4 (2022). ISSN: 1084-4309. DOI: 10.1145/3491430.

[11] Fazle Sadi et al. “PageRank Acceleration for Large Graphs with Scalable Hardware and Two-Step SpMV”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547561.

[12] Linghao Song et al. “Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC ’22. San Francisco, California: Association for Computing Machinery, 2022, pp. 211–216. ISBN: 9781450391429. DOI: 10.1145/3489517.3530420.

[13] Linghao Song et al. “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*. FPGA ’22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 65–77. ISBN: 9781450391498. DOI: 10.1145/3490422.3502357.

[14] Nitish Srivastava et al. “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. DOI: 10.1109/MICRO50266.2020.00068.

[15] Guowei Zhang et al. “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*. ASPLOS ’21. Virtual, USA: Association for Computing Machinery, 2021, pp. 687–701. ISBN: 9781450383172. DOI: 10.1145/3445814.3446702.

[16] Z. Zhang et al. “SpArch: Efficient Architecture for Sparse Matrix Multiplication”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 261–274. DOI: 10.1109/HPCA47549.2020.00030.

A Appendix

Definitions of custom metrics used in evaluation of this work, related software and methodology used to produce said results are described below.

A.1 Synopsis Design Compiler (SDC)

To establish a comparison between the synthesis metrics of the "MatRaptor" [14] architecture and the this work, the generated Verilog files from Vitis HLS are ported over to SDC for synthesis and evaluation on the 28nm technology. This is accomplished using the SDC tool licensed for the course work "VLSI Processor Design" provided for students by the chair of "Highly-Parallel VLSI Systems and Neuro-Microelectronics" for processor design.

The Verilog design files are copied over to the remote machine running the SDC and following the design flow guidelines established throughout the exercises provided in "https://hpsn.et.tu-dresden.de/scpm/icpro/wiki/IcProPractise", a hardware synthesis of this work is performed. Further details regarding the configuration of SDC and library used for synthesis can be obtained by following the exercise link.

A.2 Traffic Reduction

Traffic here refers to the total number of main memory accesses performed by the device-under-test (DUT) during the computation of sparse-sparse matrix multiplication. This arises from the assumption that the main memory is accessed by multiple systems in addition to the DUT and every access request by the DUT causes arbitration on the main memory bus for data transfer. Thus, higher the number of accesses to main memory by the DUT results in higher traffic. Reduction in number of accesses by DUT to main memory results in main memory traffic reduction.

To measure the traffic reduction of the DUT, the total number of main memory access count of the DUT is obtained and normalized with respect to an ideal system (one which has no intermediate storage and must access the main memory for every row of matrix B). The resulting values provide the reduction in the number of main memory access caused by virtue of the DUT's system architecture and algorithm.

A.3 Trapping Efficiency

Trapping efficiency provides insight into how well the sliding window algorithm is performing for a fixed number of traps '*S*' present in the DUT. In an ideal case, the DUT would have unlimited number of traps present resulting in no redundant access to main memory. But, in real world scenario, the number of traps will be limited and determined by the resource availability of the FPGA. In such a case, there may exist a bottle neck caused by the limited traps, manifesting as redundant main memory access. Let us consider the following example to better explain the scenario.

Consider a DUT with 2 PUs and 1 trap per PU and both traps filled with a lifetime of 2 (that is, both traps are accessed by both PUs). Looking at it from the perspective of a single PU, after the PU finishes accessing the row in its trap, the lifetime decreases, but the trap is not replaced with a new row from main memory since the current row still is alive and needs to be accessed by the other PU in order to be replaced. So, the new row will be accessed from main memory instead of the trap. This results in an redundant access to main memory, which could have been avoided if the PU had a larger number of empty traps to store more rows from main memory.

Another scenario that affects performance of the sliding window algorithm is the distribution of values across the sparse matrix. For a matrix that is highly structured, we require fewer traps to perform effective sliding, as seen in the case of a full dense matrix, only 1 trap per PU is required to achieve maximum efficiency from the sliding window algorithm. Whereas, for a highly random and unstructured matrix, the number of traps will greatly affect the effectiveness of the sliding window algorithm.

Thus the trapping efficiency provides an outlook on how the sliding window is performing with respect to the matrix density, given a fixed number of traps to utilize. Trapping efficiency is measured as the difference between the total '*Memory-Access-Count*' of ideal design (by setting the number of traps to a large number) and DUT in percentage.

A.4 Trap size optimization

In order to determine the number of traps required for optimal performance, two boundary conditions Tnz_{min} and Tnz_{max} are established. Tnz represents the ideal access count, i.e, the total number of times an ideal system accesses main memory for rows of matrix B. Here ideal behaviour is defined as a system that accesses the main memory only once per row of matrix B in order to move it into local memory (trap). This ensures an ideal system will not cause redundant main memory access, thus reducing main memory bus traffic as previously discussed in appendix A.2. Thus, any DUT can achieve ideal system behaviour as long as the total number of accesses by the DUT does not exceed Tnz_{max}.

The minimum (Tnz_{min}) and maximum (Tnz_{max}) boundary conditions are obtained by analyzing the sparse matrix distribution and generalizing the main memory access patterns into worst and best case scenarios. To perform said analysis, lets assume an ideal system with 2 PUs per block and the rows of matrix B accessed by PU_1 and PU_2 will be combined and represented as set R_1 and R_2:

1. When both PUs of the block access the same rows of matrix B (figure A.1a), the The total number of rows accessed from main main memory will be the union of R_1 and R_2, resulting in $R_1 = R_2$. This leads to the best case scenario providing the minimum condition Tnz_{min} described in section 6.1.
2. When both PUs access unique rows of matrix B (figure A.1b) i.e, the intersection of R_1 and R_2 is zero, leads to the worst case scenario described in section 6.1. Factoring in the sparse matrix density, the numerical value of the worst case scenario can range from $V_P * nz_R$ to N. Hence, we take the minimum of these two values since for higher densities $V_P * nz_R$ can exceed N.

Utilizing the lower and upper boundary conditions Tnz_{min} and Tnz_{max}, a simple window can be established, which will indicate if the DUT sliding window algorithm is performing optimally. If the total memory access count of the DUT for a given matrix size and density exceeds the roof determined by Tnz_{max}, means the number of traps is sub-optimal and can be increased provided the FPGA has more resources to expand.

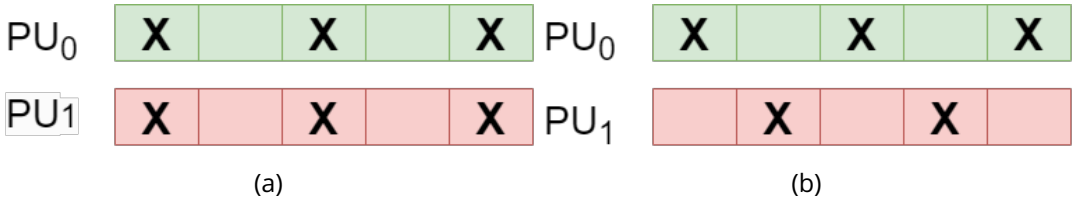


Figure A.1: Two PUs assigned rows of matrix A, where 'X' indicates non-zero values in row of matrix A, i.e, the rows of matrix B that will be accessed by the PU for computation. (a) Best case scenario when both PUs access the same rows of matrix B. (b) Worst case scenario when both PUs access unique rows of matrix B.