MEMORY EFFICIENCY IMPLICATIONS ON SPARSE MATRIX OPERATIONS

by

Shweta Jain

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2014

ABSTRACT

SHWETA JAIN. Memory efficiency implications on sparse matrix operations
(Under the direction of DR. RON SASS)

Sparse Matrices are very large matrices with very few nonzero elements and operations on sparse matrices are central to many numerical and graph algorithms. The fundamental bottleneck in these operations is the usage of specialized storage formats which only store the NonZero (NZ) elements and the indirect memory references required to access those elements. This makes the operations very sensitive to memory latency and bandwidth. Unfortunately, microprocessor trends are not encouraging for sparse matrix operations: latency is increasing and bandwidth is becoming more scarce. This results in many important applications having very poor computation performance.

This dissertation describes a new sparse matrix format called Variable Dual Compressed Blocks (VDCB) that divides a matrix into a number of smaller, variable-sized submatrices with a bitmap to indicate the presence of NZ values. When used in conjunction with customized memory subsystem, this converts the memory reference pattern from random look-ups to a serial access pattern. To quantify how detrimental the legacy sparse matrix storage formats are, the proposed system has been implemented on an FPGA device and two common sparse matrix operations, Sparse Matrix Vector Multiplication (SMVM) and Sparse Matrix Matrix Multiplication (SMMM), were evaluated. These two operations represent a number of challenges for the memory and computation subsystems. Results demonstrate gains in bandwidth efficiency, significant impact on the performance of the SMVM and SMMM operations, and the scalability of the approach.

ACKNOWLEDGMENTS

A sincere thanks to several individuals who have guided me academically, professionally and personally towards the successful completion of this dissertation.

I would like to extend my thanks to my advisor Dr. Ron Sass for his patience and critical feedback. His guidance has helped me to solve difficult problems creatively and approach new research problems enthusiastically.

My parents who had firm belief in my abilities when I had none and who supported me throughout with their unconditional love and encouragement. Anita and Madhu for being my academic role models and I hope to follow their example in my professional life as well.

Ashwin you made it all possible. None of this would have ever happened without you and I will be forever grateful for your love in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIREN	Architecture independent reconfigurable network
BCSR	Block compressed sparse rows
COO	Coordinate format
CSR	Compressed sparse rows
DMA	Direct memory access
DRAM	Dynamic random access memory
EOF	End of frame
FPGA	Field programmable gate array
FSM	Finite state machine
HPC	High performance computing
IP	Intellectual propoerty
MGT	Multi Gigabit Transceiver
MPI	Message passing interface
NPI	Native port interface
SOF	Start of frame
SMVM	Sparse matrix vector multiplication
SMMM	Sparse matrix matrix multiplication
VDCB	Variable dual compressed blocks

CHAPTER 1: INTRODUCTION

The sustained projection of Moore's Law has had a significant impact on processor and memory architectures. Both processor and memory systems have seen an improvement in terms of speed, but the rates have been dramatically diverging. Processors have seen an exponential growth in performance (speed and bandwidth) as a result of reduced feature size and increased number of transistors giving rise to the multi-core and many-core architectures. In case of memories the additional transistor count has provided increased capacities at a cheaper cost. But unlike the processors the increasing transistor count does not provide a significant speed improvement for the memories. In fact memory latency in terms of processor clock cycles has *increased* (memory cell speeds have remained constant over the past decade) as shown in Figure 1.1. In context of processor architectures where the main memory is implemented using the Dynamic Random Access Memory (DRAM) on seperate chips the dormant nature of memory speeds has resulted in a "Memory Wall". The "Memory Wall" was first identified by Wulf and Mckee in [1]. The authors predicted that the disparity between memory speeds and processor speeds will eventually result in performance degradation as the processor will be always waiting for the data from the memory.

The "Memory Wall" has seen different manifestations for uniprocessor and multiprocessor systems. In case of uniprocessor system the critical performance impediment was the memory access latency which was alleviated using caches and latency hiding techniques like hardware/software prefetching and Out-of-Order execution of instructions. But as we moved towards the multi-core and many-core architectures the increased transistor counts provided the capability of adding larger caches in order to hide the memory latency. Ideally this should have provided a performance

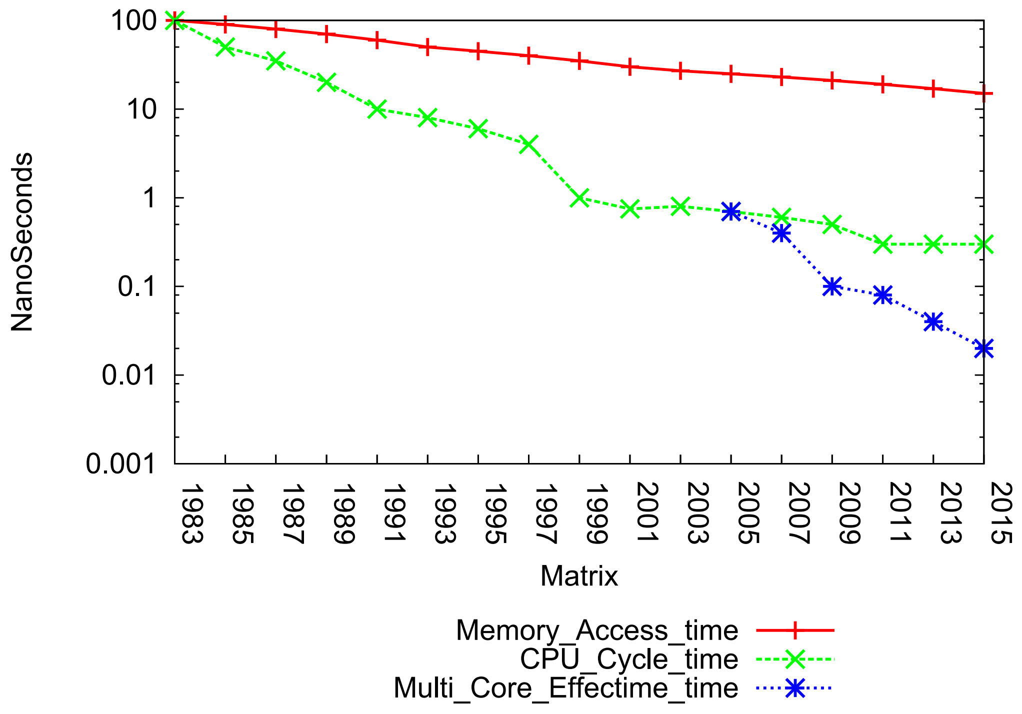

Figure 1.1: Changes in memory access times

improvement in the order of the increased number of cores, but this was not the case. As the number of cores and size of caches increased, the memory traffic generated also grew proportionally proving the off-chip memory bandwidth to be the performance bottleneck.

An increased bus width can be a viable option to offset the increasing demands placed on the memory bandwidth providing higher throughput for the off-chip communication in turn increasing the memory bandwidth. The I/O pins available on the processor chip can be increased to provide wider bus widths. This solution is fundamentally limited by the fact that with every processor architecture generation number of cores are going to grow proportional to the area of the chip (C^2, where C is the length of one side of the chip)whereas the number of pins are going to be proportional to the perimeter of the chip ($4 \times C$). The significant difference between the rate at which the core counts and pin counts are going to improve has made memory

$$\begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ \hline 0 & 10 & 0 & 0 & 0 \\ 1 & 3 & 9 & 0 & 0 \\ 2 & 0 & 0 & 7 & 8 \\ 3 & 3 & 0 & 8 & 0 \end{array}$$

`val`	10	3	9	7	8	3	8
`col_ind`	0	0	1	2	3	0	2
`row_ptr`	0	1	3	5	7	×	×

`val`	10	3	9	7	8	3	8
`col_ind`	0	0	1	2	3	0	2
`row_ind`	0	1	1	2	2	3	3

Figure 1.2: Example of CSR and COO format

bandwidth a critical resource on multi-core and many-core processor architectures. In this thesis we are going to investigate the *Sparse Matrix Operations* which perfectly exemplify the performance ramifications faced due to the memory bandwidth limitations.

1.1 Sparse Matrix Operations

The Sparse Matrices are matrices with a large number of zero elements. They arise in a number of scientific applications like Finite Element Method, Economic Modeling, Page Rank, Graph Algorithms et al. The sparse matrix due to large number of zero elements employ specialized storage formats which only store the Non Zero (NZ) elements and associated metadata to indicate the location information (row column positions) of the NZ elements.

The two most commonly used storage format for sparse matrices are Compressed Sparse Row (CSR) and COOrdinate (COO) format. An example of these formats is shown in Figures 1.2. These formats result in 1-D partitioning of sparse matrices as the matrices are partitioned either by row or columns. We also have block based storage formats like Block CSR and Block COO. The block based formats rely on 2-D partitioning as the matrix is partitioned both by rows and columns.

An important operation involving sparse matrices is the Sparse Matrix-Vector

Multiplication (SMVM). The SMVM operation perform $y = A \times x$ where A is a sparse matrix, x and y are dense vectors. The SMVM operation is of considerable importance due to its low floating point performance (measured in MFLOPS, or million floating-point operations per second) on modern computing platforms. The poor performance of the algorithms using the SMVM operation (varies between 10% to 33% theoretical peak rate of computation [2]) can be attributed to two factors: the basic matrix-vector multiplication operation and sparse nature of the matrix. If we consider general matrix-vector multiplication, we are performing $O(n^2)$ floating point operations on $O(n^2)$ elements. The ratio of floating-point operations to memory transactions is $O(1)$. With the widening gap between between processor clocks (increased computation speeds) and memory speeds this ratio of $O(1)$ cannot hide the latency of fetching data from slow memory to the fast computational unit, making matrix-vector multiplication memory bound in nature. The SMVM performance further deteriorates due to the sparsity of matrix involved. If we consider a NZ element represented using a single precision floating point we are performing two mathematical operations (a multiply and an add) on four bytes of data, resulting in a flop:byte ratio of 0.5. Now if we consider the specialized storage format we are using for storing the NZ elements we are also going to have some associated metadata in form of indexing information (location information) with the NZ element. Assuming we need two integers (eight bytes) to indicate the row and column to which the NZ element belongs we are now generating 12 bytes of memory traffic for performing two floating point operations, thereby further driving down the flop:byte ratio. Thus we are not only having low arithmetic intensity but we also have an increased traffic on the memory subsystem due to the index information associated with each NZ element. These two factors make the SMVM problem *memory bandwidth bound* in nature.

It is also important to consider the issue of memory latency in case of the SMVM operation. Although not memory latency bound, the memory access latency still has a significant impact on the SMVM operation. It has been reported by Buluç in [3] that the Intel Nehalem processor takes four clock-cycles to perform two mathematical operations and around 24 clock-cycles to fetch the 12 bytes of the data associated with the NZ operand, resulting in 20 wasted clock-cycles when processor is idle. The gap is going to increase further over the next processor generations. In order to resolve the SMVM performance issues we need a solution which is not only capable of efficient utilization of memory bandwidth but also is agnostic to the memory latency.

1.2 Thesis Statement

The research efforts for the sparse matrix operations have been focussed on improving data resusability for operands involved other than the sparse matrix. This is a reliable approach considering sparse matrix does not offer any temporal locality (due to the usage of sparse matrix storage format) and hence limits the data reusability. The modest performance improvements obtained using the data reusability does warrant us to look at the problem from a different performance. The researchers have been continuously modifying storage formats to be capable of extracting maximum performance from the underlying memory subsystem which is intrinsic to a processor architecture.

We investigate the problem from the point of view of storage formats and how they have evolved over the decades. If we refer to Figure 1.3 we see that for almost three decades (1960s to late 1990s) the storage formats have been exclusively based on 1-D decomposition of sparse matrices. This resulted in formats which were essentially some variation of CSR or COO format based on the NZ distribution present within the matrix. But in late 1990s we see the first block based storage format in form of Block CSR [4] almost after three decades of development of the CSR format. The sudden need of block based storage format has its roots in the widening gap between processor and memory performance. If we refer to the Figure 1.1 we see that by the late 1980s the memory access time had significantly increased when compared to processor clock-

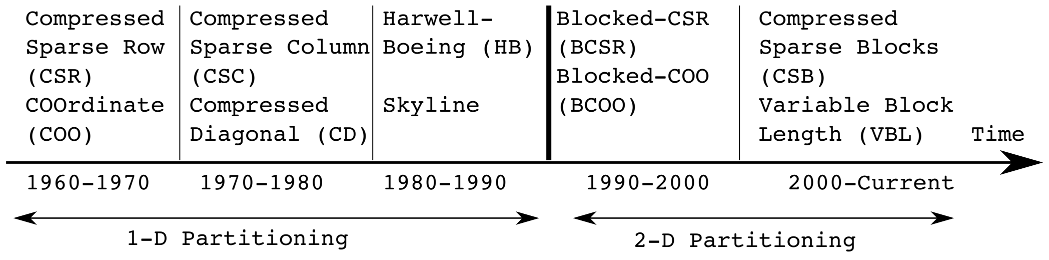

Figure 1.3: Evolution of sparse matrix storage formats

cycles, indicating the memory access latency to be a critical performance impediment. This resulted in development of processors with on-chip caches in late 1980s and addition of more level of caches by early 1990s. As the researchers recognized the performance gains provided by caching the focus shifted towards enhancing data reusability and hiding the cache miss latency.

In case of the BCSR format the reordering of the NZ elements was used in order to provide dense blocks which improved the reusability of vectors x and y. The block based formats only needed to store the location of the block and the relative position of the NZ elements was deduced from the block location information. This reduced the memory requirement for the index information of the block based approach which resulted in reducing the number of load operations. The newer block based formats developed after BCSR, like BCOO and VBL also continued to focus on memory latency issue and provided solutions accordingly. These formats were basically developed to address the performance needs of that time. As discussed before they predominantly focused on memory access latency and improving reusability of vector x,y and looked at any improvement in memory bandwidth utilization as an ancillary benefit, although the SMVM operation is memory bandwidth bound in nature. This approach essentially created a major disconnect between the needs of today's processor architectures and the original requirements for which these storage formats were designed. All the storage formats shown in Figure 1.3 (except CSB) were designed keeping memory latency as the key performance impediment. As the paradigm shift towards multi-core/many-core architectures provided an every increasing core count the focus moved towards the memory bandwidth. The increasing complexity of the memory subsystems and aggressive memory hierarchy designs to hide memory latencies resulted in contention of aggregate memory bandwidth available to the processor, making memory bandwidth a valuable resource.

This strengthened our hypothesis that the current storage formats are not suitable for the current and future architectures and this basically motivates the central question of this thesis:
As the memory bandwidth remains limiting issue on current and future processor architectures, will the usage of legacy sparse matrix storage formats prove detrimental for sparse matrix operations?

In order to answer this question we have developed a new storage format called *Variable Dual Compressed Blocks* [5]. Based on this format we will validate our hypothesis by considering the following key questions:

- Is it possible to design a new storage format for the sparse matrices which focuses on memory bandwidth efficiency?
  By comparing the different storage formats and their shortcomings we can assess the requirements of a storage format which will be exclusively based around memory bandwidth. If this storage format can alleviate the factors that affect the memory bandwidth negatively then we will consider it to be a successful design. We will also evaluate if the storage format solely can provide an average memory bandwidth efficiency of at least 60% for sparse matrix operations.

- Can developing a memory hierarchy for sparse matrix operations which works in conjunction with new storage format provide high memory bandwidth efficiency and result in performance improvement for sparse matrix operations?
  If the storage format on its own cannot provide an average 60% memory bandwidth efficiency then if using a customized memory hierarchy result in an aver-

age memory bandwidth efficiency of at least 60%. If we are able to achieve the projected target using the customized memory hierarchy we will consider it to be a successful design.

- Can this storage format be extended to the Sparse Matrix-Matrix Multiplication (SMMM) Operation?

  If this storage format can be extended to design a new technique for the SMMM without using unordered merge operation then we will consider it to be a successful implementation. We will further discuss this metric in detail in Chapters 2.

- Can this storage format be extended to a scalable implementation of the sparse matrix operations?

  If a parallel implementation of the sparse matrix operations using the new storage format can achieve at the minimum a 2X improvement in computation time over a sequential implementation for a parallel system then we will consider it to be a successful implementation.

The rest of this thesis is organized as follows.Chapter 2 provides the background knowledge on Field Programmable Gate Arrays (FPGAs), Xilinx Tool Chain, IEEE-754 floating point format and Unordered merge. We also discuss the performance impediments faced by the sparse matrix operations due to storage formats and a survey of the current state of research for the sparse matrix operations. In Chapter 3 we will discuss in detail the VDCB format we have developed and the customized memory hierarchies for SMVM and SMMM operation. Chapter 4 presents the experimental setup and the evaluation of performance metrics to validate the efficacy of our solution. Chapter 5 concludes with a brief summary of the research.

CHAPTER 2: BACKGROUND

This chapter provides an overview of the background knowledge used as groundwork for this research. We discuss the Field Programmable Gate Arrays in detail in Section 2.1. The capability of designing and implementing different functionalities on the FPGA is provided by the Xilinx tool chain. A brief overview of the Xilinx tool chain is provided in Section 2.2. The IEEE 754 floating point format and the multiplication/addition operations involving the format are used extensively throughout this work. An overview of the IEEE 754 floating point format and the mathematical operations is presented in Section 2.3. The problem of unordered merge which is relevant to the Sparse Matrix-Matrix Multiply operation is presented in Section 2.4. A custom high speed network which is used to study the scalability of the design presented in the later chapters is presented in Section 2.5. We discuss the Sparse Matrix Vector Multiplication (SMVM) operation and the various performance issues and current research efforts related to the SMVM operation in Section 2.6. The Sparse Matrix-Matrix Multiplication (SMMM) operation and the related performance impediments and a brief survey of the current state of the art for the SMMM operation are presented in Section 2.7

2.1 Field-Programmable Gate Arrays

In order to design a customized memory hierarchy as part of the this research, we look at Field Programmable Gate Arrays (FPGAs). A Field Programmable Gate Array (FPGA) provides an Integrated Circuit (IC) which consists of a hardware fabric which can be configured for the needed functionality after it has been manufactured. The FPGAs can be programmed using the Hardware Description Language (HDL) to describe the functionality. They consist of a large number of logical resources and

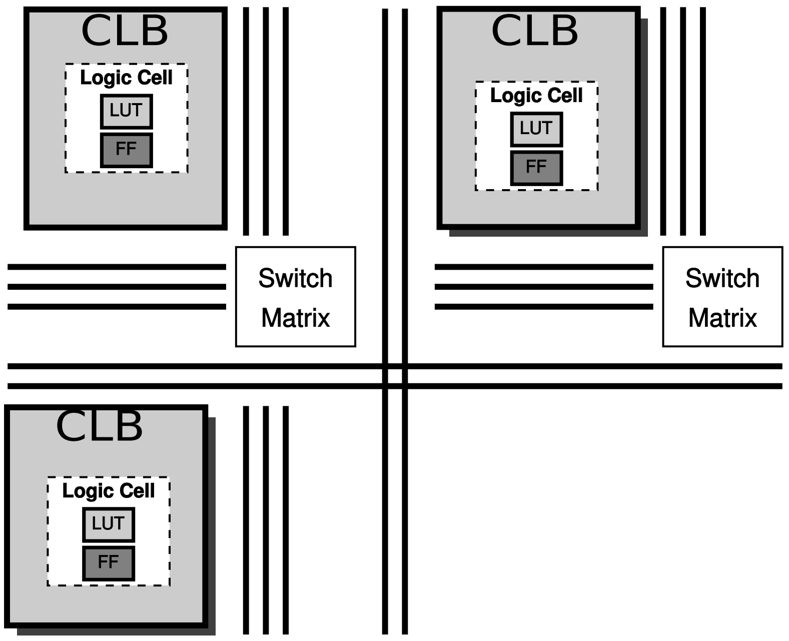

Figure 2.1: High level view of FPGA device

Block RAMs (BRAMs) to implement complex designs. A vendor specific toolchain is used to synthesize the HDL into a bitstream which can be used to configure the FPGA. The flexibility and enormous amount of computational capacity offered by an FPGA device makes it a natural fit for designing custom memory hierarchy that matches the memory access patterns of the applications for which it is used.

The FPGA consists of arrays of Configurable Logic Blocks (CLB), I/O Blocks, routing networks and special purpose blocks as shown in Figure 2.1. The CLBs are composed of LookUp Tables (LUTs) which are used as a function generator, flip-flops which are used to hold states and special purpose circuitry for interconnection. The routing network consists of switch boxes which ensures connection between the various components of a design and the I/O blocks are capable of supporting a large number of I/O standards including Low Voltage Differential Signaling (LVDC), Low Voltage CMOS (LVCMOS).

A number of special purpose design blocks are already made available by the

FPGA vendors which can be used without any modifications. These are generally known as Intellectual Properties (IPs). There are two type of IPs: Soft IP and Hard IP. The Soft IPs are implemented using the FPGA logic resources and the user needs to explicitly instantiate these IPs in the HDL design. The Hard IPs are IPs which are already implemented within the FPGA fabric They generally consist of Processor Cores, DSP blocks, High Speed transceivers, Block RAMs. A detailed description of the inner workings of an FPGA device can be found in [6].

2.2 Xilinx Integrated Software Environment

The Xilinx Integrated Software Environment (ISE) is the front-end GUI of the Xilinx tools which are used to program the FPGA devices with the user-defined functionality. The user describes the design in a Hardware Description Language (HDL) like VHDL or Verilog and using netlists. The *netlist* is a colloection of logic units and the intermediate connections between the units. The Xilinx tools use a set of commands to convert the HDL description of a user design and netlists into a configuration file for the FPGA. The configuration file for the FPGA is known as a *bitstream* and it is used to place the various parts of a user design into the FPGA design components. We briefly describe the various steps it takes for the Xilinx ISE to convert an HDL design into a bitstream for the FPGA device. A more detailed description of the design flow is available in [7].

- Xilinx Synthesis Tool (XST)

  The Xilinx Synthesis Tool (XST) is used to convert an HDL design into a netlist. The XST tool performs HDL code parsing for checking the syntax and reports errors if present. The XST tool is able to perform FSM extraction and macro recognition for in-built logical units like Flip-Flops, logic gates and memory. It applies low level optimizations when available for timing, area and technology. Some of the optimizations can be selected by the user and some are recognized by the tool from HDL design description.

- NGCBuild

  The NGCBuild compiles different netlists into one common netlist in the Xilinx proprietary format of *.ngc*. The NGCBuild opens the design hierarchy and traverses it recursively to find the netlists associated with different IPs and also applying any user constraints specified within the User Constraint File (UCF).

- NGDBuild

  The common netlist generated in the previous step of NGCBuild is converted into a Xilinx Native Generic Database (NGD) by NGDBuild. The NGD file contains the description of the netlist in terms of Xilinx primitives of LUTs, OR AND gates, memory and Flip-Flops. The design can now be mapped to a specific Xilinx device technology.

- Map

  The Map program is used for mapping a NGD file to a specific Xilinx device. The program first performs a Design Rule Check (DRC) on the design presented within the NGD file and then maps the design to the components of the specific Xilinx device technology. The output of Map is a Native Circuit Description (NCD) file which is used for placement and routing. An initial timing information for the design is available at this point and Setup checks can be performed. The Hold checks cannot be performed till the design has been routed by the tool.

- Place And Route (PAR)

  The PAR accepts the NCD generated as output of Map and uses it for placement on the FPGA device. During placement the physical constraints are applied to the design using the specification provided in Physical Constraints File (PCF). The placement of the various design components is followed by the routing which is used to use the interconnection network present on the FPGA device to connect the physically placed design components. The routing step is the most time consuming step of the entire design flow. The complete timing information for the design is available at this point and a final NCD file is made available.

- BitGen

  The NCD file available after PAR is used for generating the bitstream using BitGen.

A number of tool specific optimizations for area, power, performance and timing are available at each step of the design flow to cater to specific needs of the user defined design. The details of these options can be found in [7]

2.3 IEEE 754 Floating Point Format

The IEEE 754 floating point format is a binary representation for floating point numbers. It is a common standard established for representing floating point numbers across various architectures and providing portability for scientific code. The format provides two forms of representation : Single Precision (32-bit) and Double Precision (64-bit). The format has three components associated with it: *Sign* (S), *Exponent* (E) and *Fraction* (F). In general the IEEE 754 format can be represented using the following form:

$$(-1)^S \times F \times 2^E \tag{2.1}$$

The *sign* value can be '0' to represent a positive floating point number or a '1' to represent the negative floating point number. The IEEE 754 format uses a concept similar to the normalized scientific binary floating point representation where no leading zeros are present. In order to use this form of representation the format relies on *exponent* and *fraction*. The *exponent* part of the format represents the amount of decimal point shift to the left in order to have only a leading one. The *fraction* part of the format represents the trailing part after the decimal point once the left

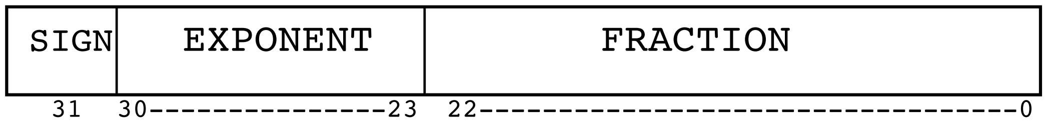

Figure 2.2: Single precision floating point representation

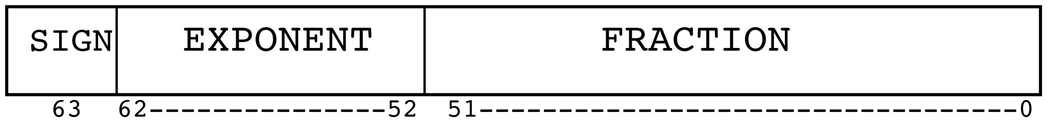

Figure 2.3: Double precision floating point representation

shift operation has been performed to have only a leading one. In case of the single precision representation the *exponent* part can vary from -128 to 127 for signed values and 0 to 255 for unsigned values. The *exponent* for double precision representation varies from -1024 to 1023 for signed values and 0 to 2047 for unsigned values. An example of single and double precision representation is shown in Figures 2.2 and 2.3. The format also has reserved bit patterns for representing zero, Not a Number (NaN), positive and negative infinity.

The selection between single and double precision formats is based on the requirement of the application. The double precision format can be used over single precision when a better precision is required (increased fraction bits) and the chances of underflow/overflow have to be reduced. The double precision format increase the memory requirement and can reduce the speed of operation due to higher number of bits needed for its representation.

2.3.1 IEEE 754 Floating Point Multiplication

The multiplication operation is heavily used in this research for the different sparse matrix operations we have performed. In this section we will discuss the floating point multiplication operation when using the IEEE 754 floating point format.

The floating point multiplication is performed by adding the exponents of the two operands and multiplying the fractions together. Before the actual operation begins a check is performed to see if any of the operands is zero. If we consider the two operands: x represented in IEEE 754 format as $-1^{S_x} \times F_x \times 2^{E_x}$ and y represented using $-1^{S_y} \times F_y \times 2^{E_y}$, then the product $z = x \times y$ is calculated using the following steps:

- $S_z = S_x \oplus S_y$
- $E_z = E_x + E_y$
- $F_z = F_x \times F_y$
- $z = -1^{S_z} \times F_z \times 2^{E_z}$

The final result is checked for overflow which can occur quite frequently in case of the multiplication due to increased bit requirement (48-bits for the fraction in case of single precision and 106-bits for the fraction in case of double precision). In case of no overflow the correct rounding scheme is applied to ensure the result is within the precision limit. In case of overflow a suitable flag is set along with the result indicating the overflow.

2.3.2 IEEE 754 Floating Point Addition

The addition operation is used for the implementation of the accumulator (3.2.1.1) which is part of the hardware design implemented in this research. The addition operation is more complex than the multiplication operation due to the need of comparison operation between the exponents of two operands and aligning the fraction components accordingly.

If we consider the two operands: a represented in IEEE 754 format as $-1^{S_a} \times$

$F_a \times 2^{E_a}$ and b represented using $-1^{S_b} \times F_b \times 2^{E_b}$, then the sum $c = a + b$ is calculated using the following steps:

- Align the fraction part of the operands based on the exponents
  - If $E_a > E_b$ perform right shift on F_b until F_b equals to $F_b \times 2^{E_b - E_a}$
  - If $E_b > E_a$ perform right shift on F_a until F_a equals to $F_a \times 2^{E_a - E_b}$
- Compute sum of the aligned fractions
  $F_c = F_a + F_b$
- $E_c = E_a$
- $S_c = S_a$
- $c = -1^{S_c} \times F_c \times 2^{E_c}$

2.4 Unordered Merge

The merge operation is equivalent of an *AND* operation. When performing a merge operation between two lists the resultant list will consist of elements from the two list if and only if the element belongs to both the operand lists. An example of the merge operation can be seen in Figure 2.4, where *List A* and *List B* are the input lists for the merge operation and *List C* is the new resulting from the merge operation.

`List A`	1	3	2	14	19

`List B`	10	3	9	7	8	14	1

`List C`	3	14	1

Figure 2.4: Example of unordered merge

It can be seen from the example presented in Figure 2.4 that performing the merge requires a search and compare between the two input lists (*List A* and *List*

B) making it a fairly expensive operation. The merge operation when used with the sparse matrix storage formats has to parse the list of columns and rows in order to perform the required sparse matrix mathematical operation. A lot of times the column indices associated with a row are not in the increasing order in which they occur within the matrix resulting in an unordered list. This makes the search and comparison operations more complex. Lets consider an example of two lists : list 1 and list 2 consisting of column indices arranged in an increasing order and used for merge operation. If list 1 provides an element A larger than element B provided by list 2, then all the elements preceding B are not used for the search operation as they are going to be smaller than the element A (due to increasing order of column indices) and this will reduce the number of elements over which a search and compare has to be performed. Thus a merge operation over an unordered list (*unordered merge*) becomes more expensive as every time a search operation has to be performed over all the elements of the two lists, making unordered merge an expensive operation.

2.5 Architecture Independent REconfigurable Network

The Architecture Independent REconfigurable Network (AIREN) is an integrated on-chip/off-chip network that supports node-to-node communication. The AIREN interface has enabled us to implement and study the scalability of our design presented in Sections 3.2.2.1, 3.5.4. The AIREN interface consists of an AIREN Router supporting the Xilinx LocalLink Interface [8]. The router provides the ability to connect compute cores to a network including both on-chip and off-chip compute cores. The routing module present within the router is used to make the routing decisions based on the interconnection network used. The router uses the dimensional order routing for the routing decision.The router can be configured to support various network topologies. In order to support node-to-node communication AIREN interface uses the high speed transceivers present on the FPGA. The AIREN interface also uses the locallink interface to assemble the packets for the router. A packet consists of a Start of Frame (SOF) and End of Frame (EOF) along with the payload. The locallink interface enables flow control to be incorporated for the transaction made on the AIREN network. The locallink interface uses Source ReaDY (SRDY) and Destination ReaDY (DRDY) to implement flow control. The locallink interface is a light weight protocol and incurs a very small amount of overhead. A more detailed description for AIREN can be found in [9, 10].

2.6 Sparse Matrix Vector Multiplication

The Sparse Matrix Vector Multiplication is used in a number of scientific and engineering problems (e.g. Finite Element Method, Conjugate Gradient, Page Rank). The operation performs $\vec{y} = A \times \vec{x}$ where, A is a sparse matrix and $\vec{x}$ is a dense vector.

2.6.1 Performance Issues of Sparse Matrix Vector Multiplication

In order to develop a Sparse Matrix Storage format which is centered around memory bandwidth we need to understand the shortcomings of the pre-existing storage formats. We use the CSR format which is the oldest and most commonly used sparse matrix storage format to highlight the performance limitations incurred by the memory subsystem when performing the SMVM operation. We look at an example presented in Algorithm 1 for performing the SMVM operation using the CSR format.

$$\begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ 0 & 10 & 0 & 0 & 0 \\ 1 & 3 & 9 & 0 & 0 \\ 2 & 0 & 0 & 7 & 8 \\ 3 & 3 & 0 & 8 & 0 \end{array}$$

`val`	10	3	9	7	8	3	8
`col_ind`	0	0	1	2	3	0	2
`row_ptr`	0	1	3	5	7		

Figure 2.5: Example of CSR format

We list the memory subsystem impediments that will arise using this particular implementation (Algorithm 1) of the SMVM operation as follows:

```
Input: Number of rows, row_ptr, col_ind, val, $x$
Output: $y = A \times x$
for $i \rightarrow 0$ to $rows$ do
    for $j \rightarrow row\_ptr[i]$ to $row\_ptr[i+1]-1$ do
        $y[i] \mathrel{+}= val[j] \times x[col\_ind[j]]$;
    end
end
```

Algorithm 1: SMVM using CSR storage format

- Additional load operations are incurred for the index information of the NZ element in form of `row_ptr` and `col_ind`. These particular operations do not contribute towards the actual SMVM computation
- The indirect memory access takes place via `row_ptr` for `col_ind` and NZ values
- Indirect and irregular memory access on x

We can see all the three performance impediments are related to the storage format and are going to affect the available memory bandwidth negatively. If we look at the first two impediments they are directly related to the storage format. The problem here is two-fold: firstly we have additional load operations in form of the indirect memory access that takes place for the `row_ptr` and `col_ind`. Secondly, these load operations are going to be used only for the purpose of correct indexing of `val` array and not for any useful computation, driving down the flop:byte ratio.

The third impediment is due to the sparsity pattern of the matrix involved and not so much related to the storage format. If we have matrix in which a large number of NZ elements are present in the same column then all of them will access the same value of x and result in improving reusability of x. This might require reordering of the NZ elements of the matrix and inclusion of zero-padding in order to improve temporal locality on x. As the performance gains using this particular approach will be highly dependent on the NZ element distribution within the matrix and up to what extent can these elements can be rearranged, we will not address this particular aspect when developing our storage format.

Based on this discussion we can summarize the two main issues that need to be addressed by a new storage format as follows:

- Can we minimize the number of additional load operations that take place for the index information of the NZ elements ?
- Can we minimize or possibly eliminate the indirect memory access that are present within the storage formats ?

2.6.2 Related Work for the SMVM Operation

There has been a significant interest in implementing the SMVM operation on an FPGA and other compute accelerators (such as IBM Cell Broadband Engine, GPGPUs, and others). Below we explain how this work fits within the context of prior efforts.

2.6.2.1 FPGA Implementations

The FPGAs have been actively pursued over the past decade for SMVM kernel. The main premise in a lot of these research advances have been essentially to increase the computation speed to compensate for the poor memory utilization.

The work done by Zhou et al. in [11] is one of the first research efforts on performing floating point SMVM on FPGAs. The sparse matrices used are stored in traditional CSR format and the FLOPS are improved by parallelizing the multiplication and addition of non-zero elements of a row. The paper proposes a tree-based architecture comprising of floating point adders and multipliers to achieve this. Although innovative, the splitting of rows requires padding of zeros or merging of rows together to provide the required number of operands to the multiplier nodes of tree. The zero-padding is a wasteful operation and degrades the total floating point performance by increasing the number of idle cycles and merging sub-rows from two different rows subsequently increases the complexity of the accumulation circuit.

A seminal work presented in [12] discusses the need of using off-chip memory for storage of matrices and addresses the issue of increased latency due to off-chip memory requirements of larger matrices. The design presented, focuses on matrix reordering and providing a cache based memory structure for improving the overall performance of SMVM kernel.

2.6.2.2 Impact on Multi-Core Platforms

A comprehensive and detailed study on latest multi-core platforms has been performed in [13] for SMVM kernel. An exhaustive set of optimizations based on matrices and underlying architecture are used for improving performance. The results presented show *Cell Blade* (one of the platforms studied) provides a consistently high floating-point performance when compared to other state of the art architectures used. This seems contrary to popular approach for speedup, as Cell Blade has a relatively slower floating-point unit. But an essential factor on achieving speedup is the fact that Cell Blade due to its memory organization effectively utilizes the available memory bandwidth.

A number of GPU implementations of SMVM are also available. The work presented in [14] provides optimization strategies to efficiently map tasks to the GPU threads. Also, a thorough implementation of SMVM using different storage formats on a GPU is presented in [15].

2.6.2.3 Algorithmic Advances

An active area in terms of algorithms regarding SMVM has been the storage format used for sparse matrices. A blocked representation of sparse matrix using CSR called BCSR format was proposed by Pinar et al. in [4]. One of the most recent developments in storage format has been Compressed Sparse Block (CSB). It has shown promising performance for multi-core platforms. The researchers involved in developing CSB have also proposed a bitmasked implementation of CSB in [3]. Although, the premise is similar to our storage format, there are some significant differences. The bitmasked implementation of CSB does not have a concept of *Block*

Header, which necessitates an offline analysis of the entire matrix to determine the number of zeros within a block and the parallelization decisions are made based on this analysis. Also, VDCB tries to represent matrix as a group of variable-sized dense blocks unlike CSB, which envisions matrix as a group of constant sized sparse blocks.

2.7 Sparse Matrix Matrix Multiplication

The Sparse Matrix-Matrix Multiplication (SMMM) operation is used to compute $C = A \times B$ where both A and B are sparse matrices. The SMMM operation is used frequently in graph algorithms such as Breadth First Search, Cycle Detection, Peer-Pressure Clustering etc. A significant amount of research effort has been invested towards the Dense Matrix-Matrix Multiplication (DMMM) and has resulted in a number of cache friendly optimizations like software-prefetching, register-blocking etc. These performance optimizations have been implemented to hide memory latency and to increase the data reuse. Although applicable towards the SMMM operation to a certain extent, the performance gains using these techniques are not significant when compared to the DMMM operation.

2.7.1 Performance Issues of SMMM Operation

The naïve approach for matrix-matrix multiplications uses $O(n^3)$ operations, where $n \times n$ is the size of matrix . To reduce the number of operations, fast matrix multiplication algorithms such as Strassen and Coppersmith-Winograd are widely used. The complexity for these algorithms varies from $O(n^{2.78})$ to $O(n^{2.375})$. This indicates the number of multiplication operations are dependent on the size of the matrix and not on the Number of Non-Zero (NNZ) elements present within the matrix. This is a desirable feature in case of dense matrices where the NNZ is $O(n^2)$. It indicates that the NNZ elements will grow proportionally with the size of the matrix and hence having an algorithm where complexity is a function of the size of matrix $(n \times n)$ instead of NNZ elements is more suitable. But in case of sparse matrices these algorithms provide an over-estimation of the number of multiplication operations that are actually

$$\begin{pmatrix} 10 & 0 & 0 & 0 \\ 3 & 9 & 0 & 0 \\ 0 & 0 & 7 & 8 \\ 3 & 0 & 8 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 20 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 7 & 0 & 32 & 0 \\ 8 & 6 & 0 & 0 \end{pmatrix}$$

Figure 2.6: Basic matrix multiplication

`val`	10	3	9	7	8	3	8
`col_ind`	0	0	1	2	3	0	2
`row_ptr`	0	1	3	5	7		

`val`	2	1	4		
`col_ind`	1	0	2		
`row_ptr`	0	1	1	2	3

Figure 2.7: CSR Multiplication Example

needed. For sparse matrices NNZ is $o(n^2)$ and the general trend is that as the size of matrix increases the NNZ elements reduce. If we have two sparse matrices with a and b NNZ elements respectively, then the number of multiplications operation required are around $O(ab)$. Hence the available fast matrix multiplication algorithms do not utilize the sparse nature of matrices involved and end up performing more number of multiplication operations than are actually needed. It can be seen from Figure 2.6 that only six multiplications are needed (due to large number of zero elements) for the resultant matrix C. But if a naïve implementation is used we are still performing 64 multiplications in order to calculate the final result.

Another layer of complexity is added to this problem due to the usage of sparse matrix storage formats. In order to determine the NZ elements from the matrices which are going to multiply an unordered merge has to be performed between the indexing elements of the two matrices. If we assume both matrix A and B are represented in the CSR format then the multiplication will take place as shown in Figure 2.7.

In order to perform multiplication using the CSR format the `row_ptr` of matrix B has to be decoded in order to find out the row positions of its NZ value. Then each decoded row positions have to be compared with each `col_ind` of matrix A.

This process has to be repeated with every row position of matrix B. This essentially results in performing an *unordered merge* (Section 2.4) between rows of matrix B and columns of matrix A. Although this method provides a means of avoiding unnecessary multiplication which take place in the naïve implementation; the unordered merge that needs to be performed is a very expensive operation and results in performance deterioration.

Hence another requirement that the new storage needs to address is:
Can the new storage format perform the SMMM operation without the unordered merge and unnecessary multiplications with the zero elements?

2.7.2 Related Work for the SMMM Operation

The classic SMMM algorithm developed in [16] is one of the seminal works for this problem. The algorithm uses the traditional CSR format for computing the product of two sparse matrices. The MATLAB CSparse operation is based on this particular algorithm. A fast sparse matrix multiplication has been proposed by Yuster and Zwick in [17]. The proposed algorithm is not specifically used in conjunction with a format. It uses fast rectangular dense matrix multiplication for performing the multiplication for permutation matrix. The complexity of this algorithm is around $O(m^0.7n^1.2 + n^{(2+o(1))})$ where m and n represent the number of rows and columns of the resultant matrix. The work done by Sulatycke and Ghose in [18] discusses the impact of indirect memory accesses on the performance of the SMMM operation. They also propose a loop-interchange technique for improving the performance of the SMMM operation and demonstrate a multi-threaded implementation of the proposed technique. The work done by Buluç and Gilbert in [19] discusses the scalability issues of the SMMM operation. They also use a new storage format called Doubly Compressed Sparse Columns (DCSC) which is a modification of the CSC format for the implementation of the SMMM operation.

The research efforts for SMMM operation on the FGPAs is still in nascent state.

One of the initial work on the SMMM is done by Lin et al. in [20]. The work deals with the energy efficiency of implementing the SMMM operation on FGPAs and uses the CSR format for the matrices. This work is further extended to design an analytical model for matrix-multiplication on FPGAs focusing on the SMVM and the SMMM operation are suggested in [21].

CHAPTER 3: DESIGN

The premise of this research is that the currently available storage formats are not memory bandwidth friendly and in turn result in performance deterioration for sparse matrix operations. In order to validate this argument we looked at the shortcomings of the currently available sparse matrix formats and develop a new storage format known as the Variable Dual Compressed Blocks (VDCB). Our work focuses not only on the development of the storage format but also on the feasibility of this format to perform the sparse matrix operations in a computationally efficient manner. We have hypothesized in Chapter 1 that the inefficient utilization of the memory bandwidth when performing sparse matrix operations is not solely due to the shortcomings of the storage formats but also the inherent processor memory hierarchy.

We evaluate our hypothesis by examining if the VDCB format independently can serve the performance deficits suffered by the sparse matrix operations or the memory hierarchy present in the processor architectures is also responsible for performance degradation. In order to examine our argument we must have a two-fold approach when developing the experimental setup. Firstly we need to use the VDCB format by itself to perform the sparse matrix operations in software. This will help in understanding if the performance shortcomings are only due to the storage format and independent of the conventional memory subsystem. Secondly we need to develop a memory hierarchy to work in conjunction with the VDCB format to perform specific sparse matrix operations. The comparison of the performance from these two approaches will help us to answer our hypothesis. We have selected two sparse matrix operations for the purpose of design development: Sparse Matrix Vector Multiplication (SMVM) and Sparse Matrix Matrix Multiplication (SMMM). Based on the discussion above we classify the high level design into two categories:

- *Hardware/Software Co-Design* Solution

  In this solution a memory hierarchy is developed using the FPGAs to work in conjunction with the VDCB format to perform the SMVM and SMMM operations

- *Software Only* Solution

  In this solution a software code is developed to use a VDCB encoded sparse matrix to perform the SMVM and SMMM operation on a conventional processor

3.1 Variable Dual Compressed Blocks

Based on our discussion presented in Sections 2.6.1 and 2.7.1 we list out our expectations from an ideal storage format.

- Limits the number of indirect memory accesses
- Provides a low overhead for adding the location information of non-zero element
- Agnostic to the sparsity structure

The various formats available for sparse matrix storage differ from each other in how the index information for a NZ element is stored. The indirect memory access happening in CSR, is also present for all the currently available storage formats. An optimization proposed for reducing indirect memory access is to minimize the amount of index information needed to determine the NZ element position. This reduction in index overhead is used in block based storage formats like BCSR[4] and Compressed Sparse Blocks (CSB)[22]. Relevant information required for determining block position within a matrix is only stored for these formats. Also, blocking improves the cache reusability of the vector for SMVM[23].

An ideal storage format should limit the number of indirect memory accesses and have a low memory overhead for adding the location information of NZ element. To

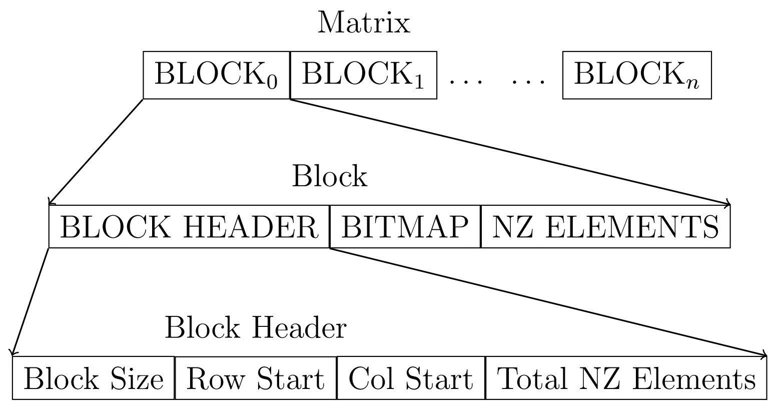

Figure 3.1: VDCB format

achieve this we have developed a storage format called *Variable Dual Compressed Blocks* (VDCB). The VDCB format works by dividing a matrix into a number of smaller variable sized sub-matrices. These sub-matrices are referred to as BLOCKS. Each block has three components associated with it as shown in Figure 3.1. The first component is a Block Header; it consists of all the parameters needed to define the location of a block within a matrix. The second component is a Bitmap and it is used to store relevant index (location) information of NZ elements associated with a block. The bitmap sets a one to indicate the presence of a NZ element within a block and zero otherwise. The last component of the format is the double precision NZ elements present in a block.

3.1.1 VDCB Encoder Software Design

The sparse matrices are only available in the commonly used storage formats like CSR and COO. This makes it essential to develop an encoding software which is able to accept a sparse matrix encoded in CSR/COO format and generate the corresponding VDCB encoded sparse matrix. We use a simple heuristic for generating the VDCB storage format from a COO encoded sparse matrix, as shown in Algorithm 2. The heuristic selects blocks based on their densities. Currently we are only using multiples of eight for block sizes and the largest block size we can support is 64x64. We have developed the search code using *C++ Standard Template Library* (STL).

```
**Input**: Number of block_rows, block_row_count
**Output**: Generate VDCB format for each block_row
while $block\_row \neq empty$ do
    set_vector_x = block_row_begin;
    set_vector_y = block_row_end;
    ldim = set_vector_y - set_vector_x + 1;
    for $j \rightarrow set\_vector\_x$ to $set\_vector\_y + ldim$ do
        Push all row, column elements of block_row in temporary block $\rightarrow$ $temp\_block$;
        Push all non-zero elements of a block_row in temporary block $\rightarrow$ $nnz\_search\_block$ ;
    end
    while $temp\_block \neq empty$ do
        Determine the starting search coordinates of temp_block;
        for $i \rightarrow 1$ to 8 do
            Search blocks of sizes in multiple of 8 using starting search coordinates;
            Choose the block with highest density $\rightarrow$ final_block;
            Select larger block if multiple blocks have same density;
            Remove row, col from temp_block that correspond to final_block;
            Generate block header for final_block;
            Generate bitmap for final_block;
            Remove non-zero elements corresponding to final_block from nnz_search_block;
        end
    end
end
```

Algorithm 2: Search heuristic

The STL provides a rich set of generic algorithmic solutions for search, sort and insertion that can be applied to user-defined data structures easily. We have used matrices from University of Florida Matrix Market Place[24], for testing our software and hardware design.

3.1.2 Definitions

Definition 1. If the blocks or NZ elements are arranged in order of increasing rows, the storage scheme is referred as *Row Major Ordering.*

Definition 2. If the blocks or NZ elements are arranged in order of increasing columns, the storage scheme is referred as *Column Major Ordering.*

Definition 3. A *Block Row* is used to represent a collection of consecutive rows of a matrix when constant block sizes are used. The number of block rows for a matrix is given by equation:

$$\beta = \frac{n}{b} \tag{3.1}$$

where $n \times n$ is the size of the matrix, $b \times b$ is the constant block size and β is the total number of block rows. Similarly, a set of consecutive columns of a matrix when constant block size is used for a *Block Column.* The number of block columns of a matrix is given by Equation 3.1.

Definition 4. The collection of consecutive block rows is referred as *Super Block.* The number of super blocks present within a matrix is given by equation

$$S_B = \frac{\beta}{S} \tag{3.2}$$

where S_B is the number of Super Blocks present within a matrix, β represents the total number of block rows present within a matrix and S is the size of each Super block.

3.1.3 Notations Used

Table 3.1: Notations used

Symbol	**Description**
V	Total Number of Blocks encoded in the VDCB format for matrix A
γ	Size of a block encoded in the VDCB format
A_V	Represents the array of all the V blocks encoded in the VDCB format of matrix A
$A_V[i]$	Represents the i^{th} block from array $A_V$of matrix A when encoded in the VDCB format
$A_V[i]_{XY}$	Represents the block-header of i^{th} block of matrix A, where X, Y are Row-Start and Col-Start fields
$A_V[i]_{BMP}$	Represents the bitmap associated with the i^{th} block of matrix A
$A_V[i]_{NZ}$	Represents the NZ-array associated with the i^{th} block of matrix A
A_{XY}	Block of matrix-A encoded in the VDCB format with X,Y denoting the Row-Start, Col-Start field of the block header
B_{UV}	Block of matrix-B encoded in the VDCB format with U,V denoting the Row-Start, Col-Start field of the block header
BMP_{XY}	Row-Major bitmap of A_{XY}
BMP_{UV}	Column-Major bitmap of B_{UV}
NZ^A_{XY}	NZ elements present in A_{XY}
NZ^B_{UV}	NZ elements present in B_{UV}
bmp^A_m	Bitmap associated with the m-th row of A_{XY}
bmp^B_n	Bitmap associated with the n-th column of B_{UV}
nz^A_m	NZ-element array associated with $bmp - A_m$
nz^B_n	NZ-element array associated with $bmp - B_n$

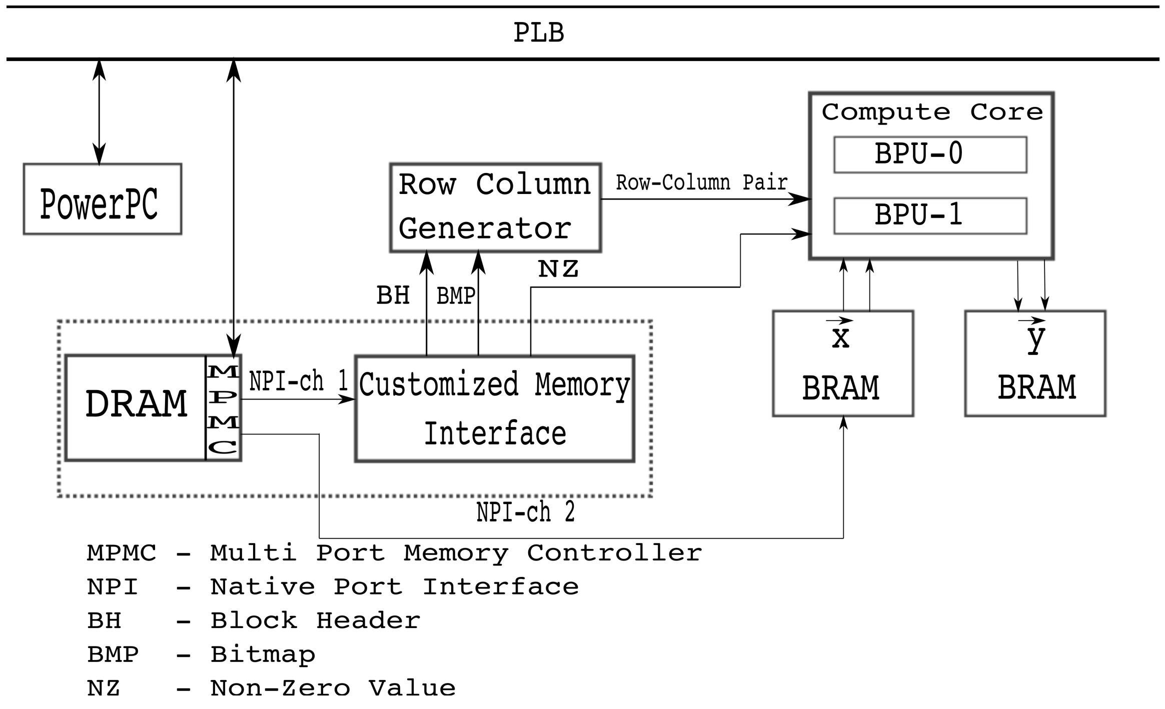

Figure 3.2: High level architecture for SMVM operation

3.2 Hardware Design for Sparse Matrix Vector Multiplication

In this section we will discuss the customized memory subsystem and the computation core design used to perform the SMVM operation on a matrix encoded in the VDCB format. This hardware design will provide us the evaluation platform for the SMVM operation when Hardware/Software Co-Design approach is used with the VDCB format to perform the operation.

3.2.1 Sequential Hardware Design

The top level sequential hardware design consists of three subsystems: Customized Memory Interface (CMI), Row Column Generator (RCG) and Block Processing Unit (BPU) as shown in Figure 3.2.

3.2.1.1 Block Processing Unit

When performing SMVM operation using VDCB format, matrix vector multiply operation takes place for each block. This generates a partial result vector for each block. The final resultant vector $\vec{y}$ is a sum of all the partial results computed.

In our previous work we identified the high latency of accumulation operation to be a major performance deterrent [5]. We alleviate this problem by implementing single cycle accumulation loop floating point accumulator (based on the work presented in [25]) for the purpose of calculating partial results for each block. The single cycle accumulation loop ensures, that every time a row within a block is switched, we have to wait only for a clock-cycle before applying new sets of inputs. The BPU trigger is controlled by a Finite State Machine (FSM) which starts all the computation operations only when the $\vec{x}$ has been read into the BRAM. The results generated by the accumulation loop need to be normalized to the standard IEEE-754 floating point format. We have modified the partial result accumulator to function as a simple loop back adder for the final stage of accumulation. In the final stage of accumulation we perform the normalization operation which is skipped in the partial result accumulation. The normalization operation takes about four clock cycles and is not implemented in the partial result accumulator, as it will be a redundant step. The blocks are interleaved in software in such a way that two consecutive blocks do not have any common rows. This avoids race conditions when the results have to be written to $\vec{y}$. The BPU operates at 100 MHz. The Computation Unit (CU) supports two BPUs (Figure 3.3) enabling us to perform matrix vector multiplication on two blocks in parallel. The vectors $\vec{x}$ and $\vec{y}$ are shared between the two BPUs. Both the vectors are stored in true dual-port Block-RAMs (BRAM) providing us the capability of issuing two read requests in parallel for $\vec{x}$ and $\vec{y}$.

The inclusion of normalization step for final result accumulator provides a total latency of six clock cycles for the final stage adder. This latency might cause a data hazard if the operand from result BRAM ($\vec{y}$) is needed before it has been written to it. This happens if partial results corresponding to the same row are applied to final stage accumulator in an interval smaller than the final stage adder latency. To avoid this we interleave the blocks in software in such a way that no two consecutive blocks

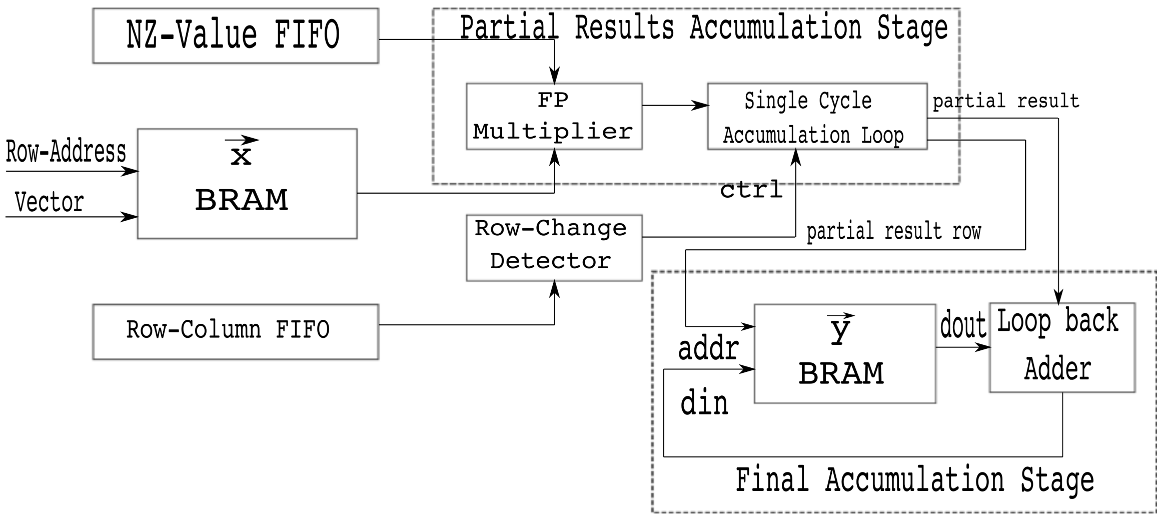

Figure 3.3: Block processing unit

have any common rows. This satisfies the latency constraint placed on partial results corresponding to the same rows.

3.2.1.2 Customized Memory Interface

The memory interface is designed to manage data coming from main memory. The key component of our Memory Interface is a Data Management Unit (DMU). The DMU controls a 1:4 De-Multiplexer managed by the DMU FSM. The input is connected to Native Port Interface (NPI) channel of memory controller as shown in Figure 3.4. The NPI channel provides the VDCB encoded matrix A stored in the main memory to the CMI. The four outputs are connected to FIFOs (Figure 3.4), these FIFOs are referred as VDCB component FIFOs. To prevent overflow of data from VDCB component FIFOs in case of larger matrices we incorporate a flow control strategy in the DMU. If any one of the VDCB component FIFOs is about to get full, we pause the NPI memory channel connected to De-Multiplexer. This prevents new data being written to the VDCB component FIFOs. Once sufficient amount of data has been read out we resume the transaction on the memory channel. The DMU operates at 200 MHz.

The DMU is responsible for providing necessary data to other two subsystems

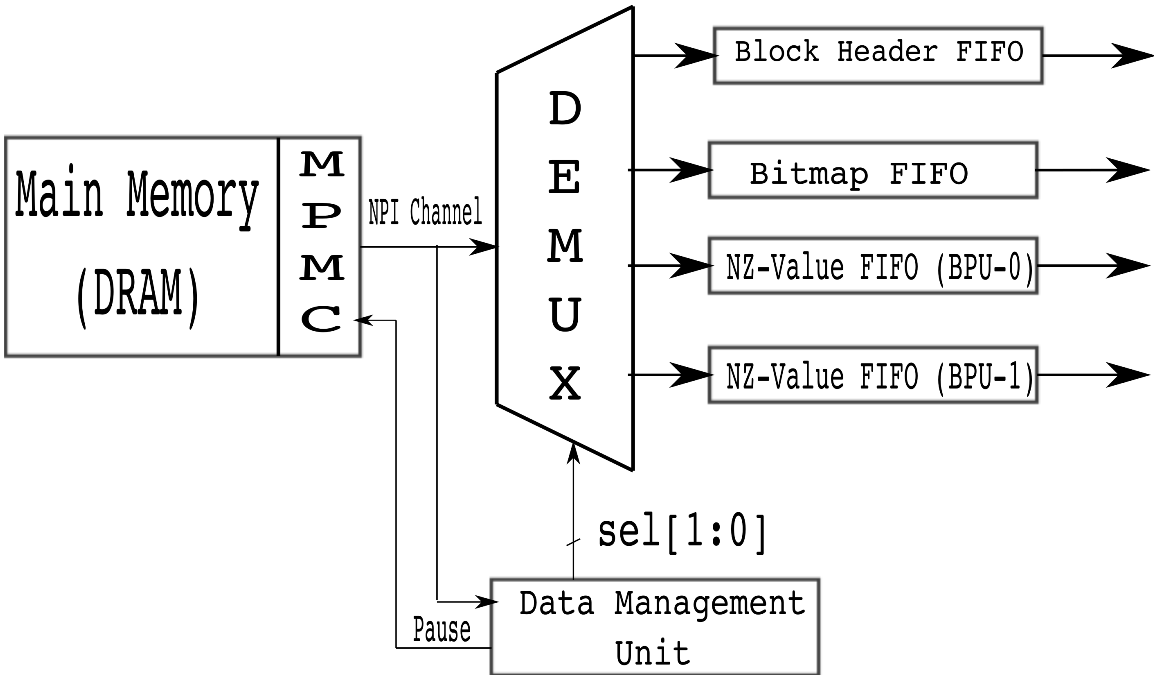

Figure 3.4: Customized memory interface

(Row Column Generator and Block Processing Unit). The FSM (shown in Figure 3.5)controlling DMU is aware of how the VDCB format is arranged in the memory. The FSM knows that the blocks are arranged in memory sequentially and the very first component present within a block is a block header. As soon as the VDCB format is started to be read out from the memory the FSM present within the DMU interprets the very first datum it receives as a block header. The FSM decodes the block header in the "decode block header" state and stores it in the block header FIFO by asserting the relevant select lines for the 1:4 De-Multiplexer. The block header is decoded to determine the size of the block, based on which it determines the number of bitmaps that will be needed to represent the complete index information of a block (Section 4.1.1, Equation 4.1). It also stores the subsequent fields of block header (i.e., row_start, col_start, number of NZ elements) in registers for future use.

After the "decode block header" state the FSM moves to the "write bitmap" state.

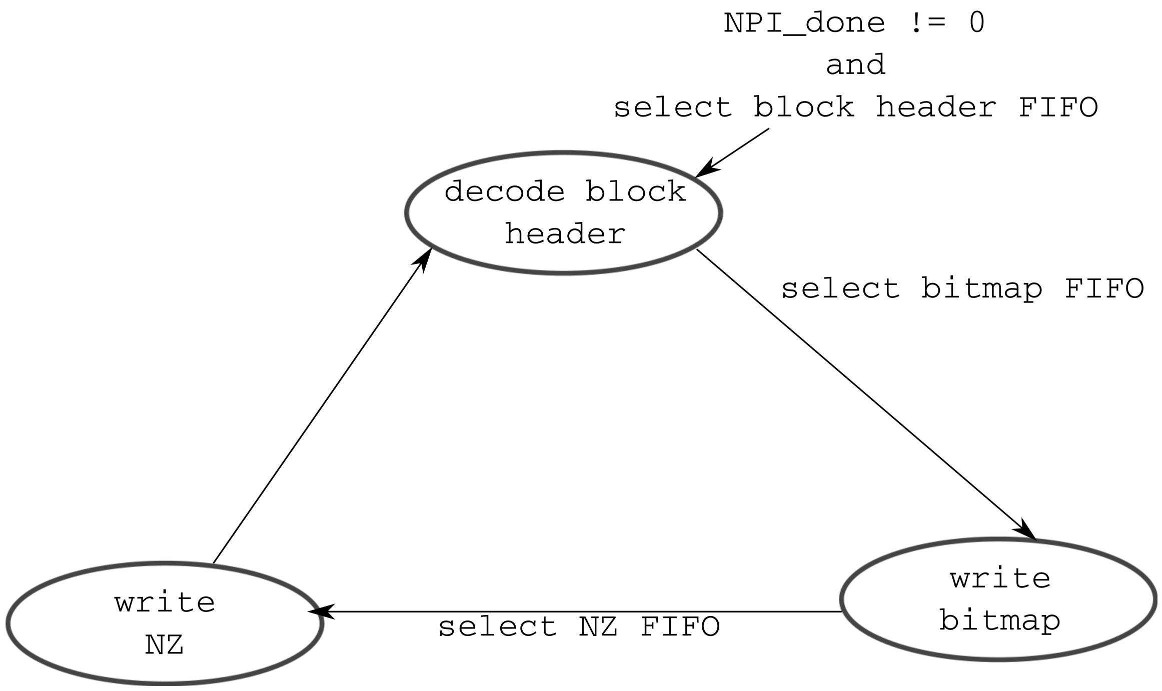

Figure 3.5: DMU control FSM

If it was determined in the "decode block header" state that k number of bitmaps would be needed for representing the index information of the NZ elements within a block, then k number of words which follow the block header (read in the previous "decode block header" state which was used to determine the number of bitmaps) are consider to be bitmaps and are stored in the bitmap FIFO. The reading of bitmaps coming in from the memory and writing the bitmaps to the bitmap FIFO (selecting the bitmap FIFO through 1:4 De-Multiplexer) takes place in the "write bitmap" state.

After writing the bitmaps to the bitmap FIFO the FSM moves to the "write NZ" state. This state is similar to the "write bitmap" state and instead of writing bitmaps the NZ elements that follow the bitmaps are written into the NZ FIFO. The number of NZ elements that will be following the bitmaps is determined by the element stored in the NNZ register in the "decode block header" state. After the "write NZ" state the FSM moves again to the "decode block header" state as the decoding operation for the next block header (indicating the beginning of a new block) which follows the NZ elements (written to the NZ FIFO in “write NZ”) begins. The FSM remains active till the NPI channel on the memory controller asserts a *NPI_done* signal indicating the entire VDCB encoded matrix A has been read from the memory.

We can see from Figure 3.4 that there are two NZ FIFOs present for supporting two BPUs. The FSM pushes the NZ elements from all even numbered blocks (block 0, block 2 and so on) in NZ FIFO 0 and the NZ elements from all odd number blocks (block 1, block 3 and so on) in NZ FIFO 1. The FSM is capable of doing so by maintaining a block counter register in the “decode block header” state which counts the incoming blocks. In the “write NZ” state the block counter is referred and if the block counter is even then the select line for NZ FIFO 0 is asserted, otherwise NZ FIFO 1 is selected. The even or odd count is determined by performing a modulo-2 operation in the “write NZ” state.

3.2.1.3 Row Column Generator

We wanted to provide a hardware design which could efficiently decode bitmaps for necessary index information. In our design we wanted to avoid introducing decoding latency which might be substantial if not greater than the cost of indirect memory access. To achieve this we have implemented a Row Column Generator (RCG). The main component of RCG is a Decoder Unit, which is a modified implementation of priority encoder.

The FSM which implements the Decoder Unit is shown in Figure 3.6. The main job of this FSM is to work in conjunction with DMU to detect all the bits which are set to “one” within a byte of bitmap and provide the corresponding row-column positions. The RCG FSM controls the read operation from the Block Header and Bitmap FIFOs which are present within the DMU. The RCG operates at 100MHz.

- Read Bitmaps

  In the “read bitmaps” state the FSM reads a block header and the corresponding bitmap from the Block Header and Bitmap FIFO present within the DMU. The

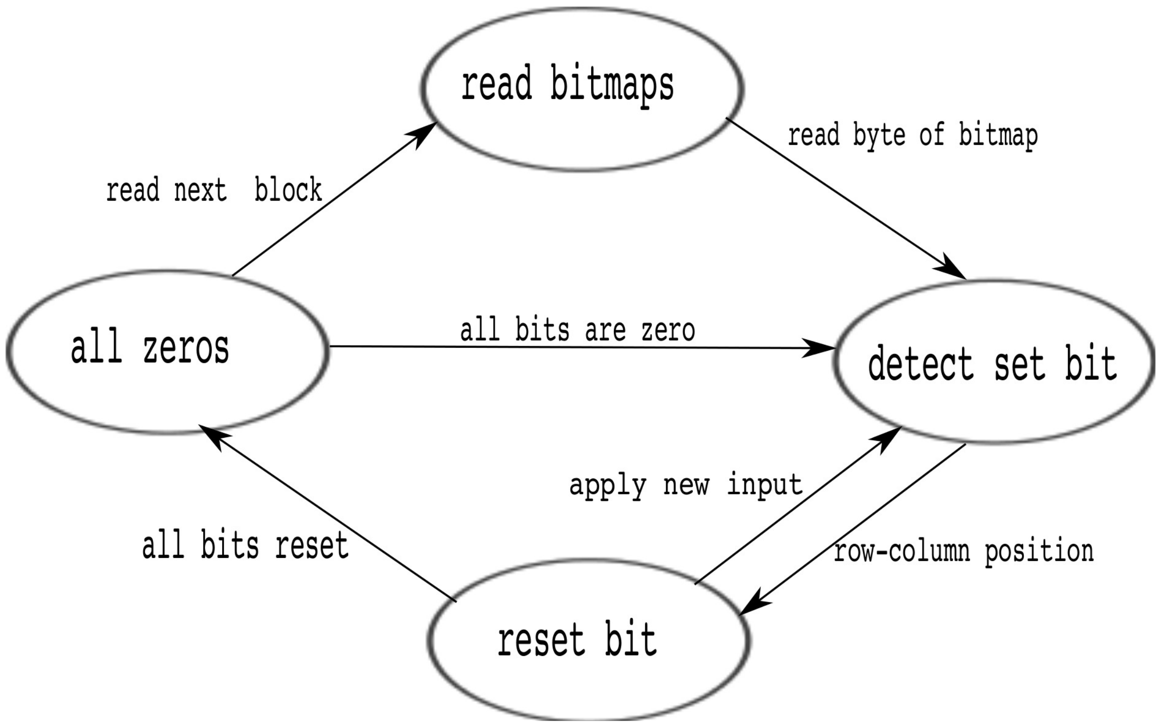

Figure 3.6: Row-Column generator FSM

"read bitmaps" state determines if more than one bitmap is needed to represent the index information of the block (Section 4.1.1, Equation 4.1). This is done by examining the block header. Once the number of bitmaps required for a given block header are determined, the block header read from the FIFO is stored in a register to be used by the subsequent states of the RCG FSM for providing row-column positions.

- Detect Set Bit

  The "detect set bit" state provides the position of the bits set within a bitmap. It operates on a byte of a bitmap at a given time. The "detect set bit" state consists of modified priority encoder which determines the position of bits set to "one" within a byte. If we take for example 00100001 representing a byte of a bitmap. Then the "detect set bit" will provide a position value of *two* and *seven* indicating the presence of a NZ element at these location within a block.

- Reset Bit

  The “detect set bit” does not generate all the positions values together, instead after generating a position value it transitions to “reset bit” state which resets the bit for which position value has been generated. So if we consider the example discussed earlier for a byte value of 00100001 then the “detect set bit” state will generate a position *two* indicating the presence of the NZ element at that particular position. Then it will transition to the “reset bit” state which will set the “one” present at position *two* to “zero” and provide a new byte value of 00000001 before transitioning back to “detect set bit” state. Now the “detect bit state” will generate a position value of *seven* and transition to “reset bit”. The “reset bit” state ensures that a position value for the same bit is not generated twice by the “detect bit state”.

- All Zero

  If the positions for all the bits set to “one” have been determined by the “detect set bit” state then the “reset bit” state will generate a byte value of 00000000. In the example discussed previously for a byte value of 00100001 once the “detect bit state” receives a byte value for 00000001 it will generate a position value of *seven*. After which it will transition to “reset bit” state. In this state “reset bit” state will change the bit value from “one” to “zero” for the position *seven* in the byte. This will result in a byte value of 00000000, then the FSM transitions from “reset bit” state to “all zero” state. In this state it is determined if all the bytes representing a bitmap have been read out. In case there are more bytes remaining for the bitmap the “all zero” state transitions to the “detect set bit” state and decoding of the next byte of the bitmap begins. Otherwise, the “all zero” transitions to “read bitmaps” state to read the block header and bitmap values for the next block.

- Generating Index Information

  As discussed before the "detect bit state" generates the position values for the bits set to one within a bitmap. These position values are used to generate the relevant index information for the NZ elements present within a block. If we consider a block of size 8×8 then we need 64 bits to completely represent the position of all the NZ elements present within the block. An 8×8 block will consist of eight rows and eight columns. We will require a byte (eight bits) for each row, where each bit represents the column present within a row. If a bit is set to one within a byte it will indicate the presence of the NZ element at that column position. In the "detect set bit" state we are generating the positions at which a "one" is set within a byte, that means we are generating a column position. We can add this column position value to the col_start field present within the block header to provide exact column position. Also every time a new byte is read we increment the row_start field of the block header as each byte represents a row in case of an 8×8 block. In case of blocks larger than 8×8 we will have to determine how many bytes are needed to represent a row and modify the increment operation for row and column positions accordingly. For example in case of a 16×16 block we will require 256 bits to completely represent the index information. This will result in incrementing the row_start field after reading every two bytes and decoding two bytes for determining the column positions where a NZ element is present.The number of bytes need for incrementing the row_start field and decoding/incrementing the col_start field is determined in the "read bitmaps" state when the number of bitmaps needed for a block is calculated using the block header.

  As the RCG provides the row-column information to two BPUs (Figure 3.2) and in turn to the two Row-Column FIFO present within the BPU (Figure 3.3), it should be capable of determining which BPU should be provided with the decoded row-column positions. In order to do so the RCG FSM employes a strategy identical to the one used by the DMU FSM for writing the NZ elements to the two BPUs. In the “read bitmaps” state a block counter is used and if the count is even then BPU 0 is selected, otherwise BPU 1 is selected. The block counter is examined in the “detect set bit” state and a modulo-2 operation is performed to select the BPU to which the decoded row-column positions will be provided.

3.2.2 Performing Parallel Sparse Matrix Vector Multiplication using VDCB Format

When performing matrix-vector multiplication operation each row of matrix A is multiplying with the vector $\vec{x}$ to produce a row of resultant vector y (Equation 3.3). We can see from Equation 3.3 that each row of the resultant vector $\vec{y}$ can be calculated independently and this provides an ample amount of row-level parallelism for the SMVM problem.

$$y_i = \sum_{i=0}^{n} A_{ij} \times x_j \tag{3.3}$$

The CSR format presented in Section 1.1 represents the NZ elements present in a matrix on a per-row basis; this makes the format well suited for utilizing row-level parallelism and is widely used for the same. The VDCB format due to its block-based structure and the usage of variable sized blocks cannot be easily be used to implement the row-level parallelism strategy. Thus we need to investigate alternate techniques to introduce parallelism.

If we refer to Section 3.2.1.1 we can see that the Computation Unit is capable of supporting up to two Block Processing Units and this allows computation of product of two blocks simultaneously. Thus we are able to introduce parallelism at a low level in form of block-level parallelism. We can try to extend this further to row-level parallelism within a block to increase performance gains. In case of the VDCB format row-level parallelism is more complicated when compared to the CSR format. This is due to the decoding of the bitmaps to provide row-column values of a block *on-the-fly* by the Row-Column Generator (Section 3.2.1.3). Hence an initial knowledge of the row position of a particular NZ element which is inherently present within the CSR format is unavailable for the VDCB format. Instead of focusing on introducing parallelism at the individual row-level we try to increase the potential of block-level parallelism. In order to increase block-level parallelism we will need to increase the number of BPUs that area available for computation and the data providing resources (DMU, CMI and RCG) for the BPUs to avoid under utilization due to lack of data. With the current design set up we are already reaching almost the maximum utilization (about 85%) of the BRAMs available (Table 4.5). Hence in order to accommodate an increased number of design components (BPU, CMI, RCG) which are heavily reliant on FIFOs (internally implemented using BRAMs) and BRAMs we need to move from a single FPGA node to multiple FPGA nodes.

We first need to decide how the blocks of matrix A and vector $\vec{x}$ can be provided to the additional nodes for computation. Based on this choice we will be able to partition the design functionality when we move to multiple nodes. If we consider a system with "n" memory channels where each channel is connected to a node which stores the entire vector $\vec{x}$ within its main memory, then we can provide the blocks of matrix A to these nodes via the memory channel for computation. This will make the nodes capable of computing the part of resultant vector $\vec{y}$ in parallel. The main issue with this strategy is the feasibility of providing individual blocks to each node; as each node will be only calculating a partial result for vector $\vec{y}$. This will make it difficult to update the vector $\vec{y}$ across all the nodes if a change has been made by one node and it will result in race conditions to write the final resultant vector $\vec{y}$.

In order to overcome this issue we introduce a new technique for increasing block-level parallelism called *Block Row Level Parallelism.* Before we continue this discussion further we will introduce definitions and modifications made to the VDCB format which are relevant for the understanding of this new technique.

Modifications made to the VDCB Format In order to implement the Block Row level parallelism technique we need to introduce some modifications to the VDCB format as listed below:

- For sparse matrix A use a fixed block-size of $b \times b$
- Arrange blocks rows of matrix A in Row-Major Order

The modifications made to the VDCB format results in a block arrangement as shown in Figure 3.7. The gray highlighted area shown in Figure 3.7 represents a block row. Each block row consists of a series of blocks of matrix A encoded in the VDCB format. With this block arrangement (Figure 3.7) we can see that each block row (consisting of the VDCB encoded blocks of matrix A) can multiply with the vector $\vec{x}$ independently in order to provide a part of resultant vector $\vec{y}$ as shown in Figure 3.7. As each node is operating on an entire block row where the blocks are arranged in a row major order the possibility of a race condition on vector $\vec{y}$ is avoided. Also each node is going to calculate a portion of vector $\vec{y}$ without requiring an update from another nodes.

Based on this discussion we can modify the initial system proposed to accommodate Block Row Level Parallelism. We will now consider a system with "n" memory channels where each channel is connected to a node which store vector $\vec{x}$ and each node is provided with a *Block Row* of matrix A via the memory channels, enabling the nodes to compute the part of resultant vector $\vec{y}$ in parallel.

Implementation of the Block Row Level Parallelism In order to incorporate the block row level parallelism for the SMVM operation, we develop a star network topology as shown in Figure 3.8. In this setup each node is an FPGA node and the head node consists of the entire matrix A encoded in the VDCB format. The head node provides the block rows of matrix A to the worker nodes via the networking

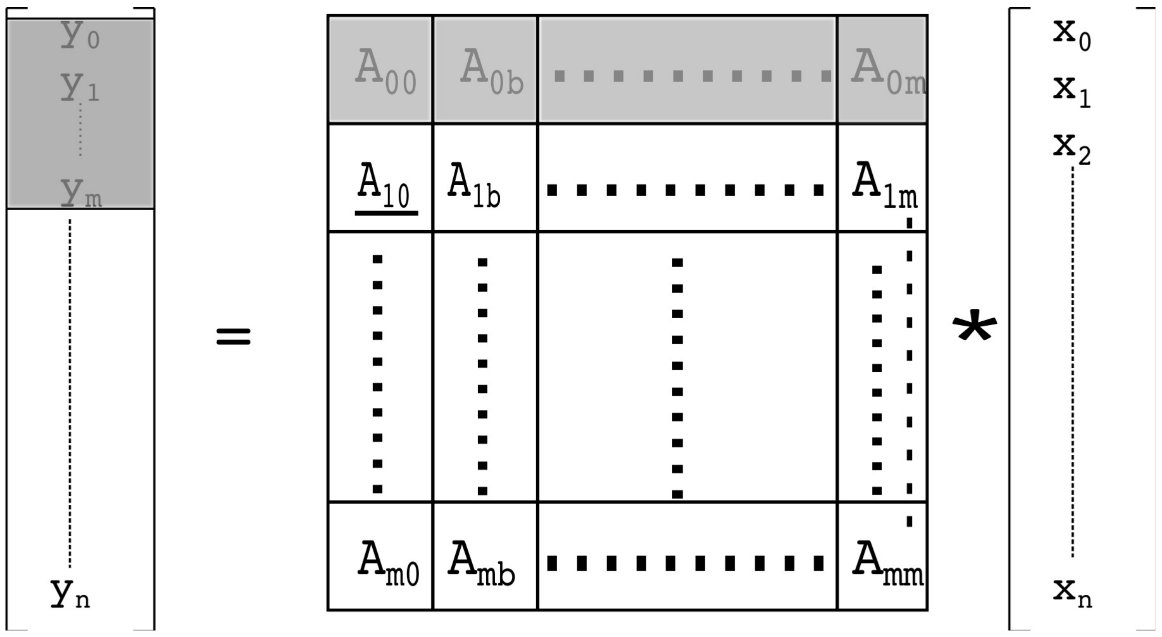

Figure 3.7: Parallel implementation of SMVM operation using Block Rows

channels. The worker nodes store vector $\vec{x}$ within their main-memory and can start the computation of resultant vector $\vec{y}$ once they start receiving the block rows of matrix A.

In the system considered in Figure 3.8 the number of worker nodes will have to be increased as the number of block rows for matrix A increase in order to obtain maximum block row level parallelism. This would not only be an expensive but also an impractical solution. The number of block rows depends upon the constant block-size "b" that has been chosen during encoding. This implies we can use large block sizes and have fewer number of block rows (Equation 3.1) to reduce the number of worker-nodes. This solution is not very effective as the selection of block size will rely on a number of parameters specific not only to the matrix but also to the underlying hardware design. Thus we need to re-evaluate our strategy to create a system which still provides the benefits of block row level parallelism without being extremely expensive.

In order to overcome the high resource requirements (in terms of FPGA worker

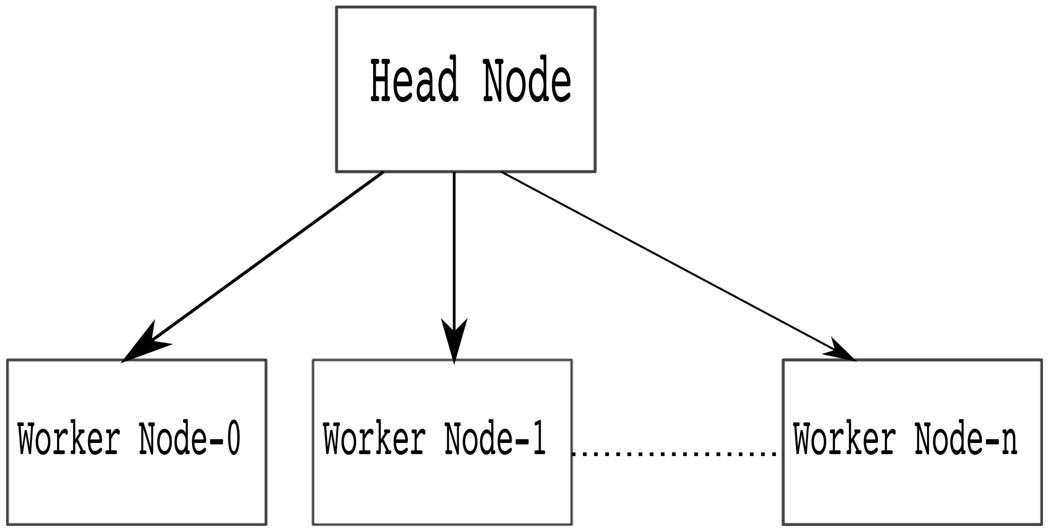

Figure 3.8: Network topology for parallel SMVM

nodes)for block row level parallelism strategy, we constrain our parallel system to have only fixed number of worker-nodes represented by w. As the system is now limited to a fixed number of worker-nodes w, we need to modify the block row level parallelization scheme. There are two approaches we can use to provide block rows to the w number of worker-nodes:

- The head-node can provide individual block rows to each worker-node and keep doing so till all the block rows have been exhausted. We will refer to this as *Round-Robin Approach*

- The head-node can provide a *Super-Block* to each worker-node. In this case we can modify the Equation 3.1 as follows:

$$S_B = \frac{\beta}{w} \tag{3.4}$$

Thus we can now have a matrix with the number of Super blocks of given by Equation 3.4 and we can provide one super-block to each worker-node.

3.2.2.1 Parallel Hardware Design

Based on the earlier discussion of using block row level parallelism to perform the SMVM operation we are able to identify the high level functionality of head node and worker node. A head node will be storing the VDCB encoded matrix A within its main memory and will provide the block row of matrix A the worker nodes via networking channels. The worker nodes will be receiving the block rows of matrix A through the networking channels; they will perform a decode operation on the VDCB encoded blocks present within the block row and perform the SMVM operation using the vector $\vec{x}$ stored in their main-memory. Thus the parallel hardware design can be broadly classified into two components: Networking Core and Computation Core. The Networking Core is used for the purpose of communication between the Head Node and Worker Nodes. The Computation Core is responsible for decoding the VDCB encoded blocks, fetching the vector $\vec{x}$ from the main memory of the worker node and performing the SMVM operation, thereby providing the portion of resultant vector $\vec{y}$.

At the heart of the Networking Core is the Architecture Independent REconfigurable Network card which is an integrated on-chip/ off-chip network that support node-to-node communication. The AIREN card uses the high speed FPGA Multi-Gigabit Transceivers (MGT) for communication between the FPGA nodes. A detailed description of the AIREN interface can be found in [26]. The AIREN interface used for networking in our design is identical to the implementation presented in [26]; for brevity we are not presenting the AIREN functionality details here.

Worker Node The parallel system as discussed in Section 3.2.2 has the matrix A stored on the Head Node and it is provided via networking channels to the Worker Nodes. Thus from the design perspective we can see that the SMVM operation is only performed on the Worker Nodes and hence the Computation-Core should be only needed on the Worker Nodes. As the communication will be needed between the

Head Node and the Worker Node the Networking Core has to be present on both of them.

The Worker Nodes will still need to use the CMI (Section 3.2.1.2) in order to provide the block row data coming in via the Networking Core to the Computation-Core. The only difference in this case is the CMI instead of being connected to an NPI-channel (Figure 3.4) is connected to the Local-Link (LL)Interface core as shown in Figure 3.9. The AIREN interface present within the Networking Core uses the Xilinx Local-Link standard for communication purpose and inserts an associated header-footer information [26]; also the Local-Link standard has a different data-width than the one used by the CMI. To overcome these differences the LL Interface is used. It removes the header/footer from the data coming in from the Networking Core and uses a differential width FIFO to provide the CMI the data-width it expects. The LL-Interface is also used to provide the resultant vector $\vec{y}$ once it has been calculated to the networking core which in turn transmits it back to the Head Node.

It can be also seen from the Figure 3.9 that a single NPI-Channel is used for vector $\vec{x}$ on the Worker Node. As soon as the the LL-Interface starts providing the data (block row) to the CMI the request for vector $\vec{x}$ is placed on the NPI channel and it is written to vector $\vec{x}$ BRAM. It might be considered suitable to store the vector $\vec{x}$ on the BRAM and not in the main-memory as it is being reused for the multiplication operation. The size of the matrix A and in turn the size of vector $\vec{x}$ will change for different matrices. This means we have to allocate very large amount of BRAM for vector $\vec{x}$ in order to accommodate different problem sizes. This will cause a problem to fit other components of our design and is not a practical solution. Thus we use a fixed size of BRAM for vector $\vec{x}$ and store it within the main-memory.

Head Node The Head Node stores the block rows of matrix A within its main-memory. It uses a Block Row Req Core to read the block rows from the main-memory via the NPI-Channel (Figure 3.10). The LL-Interface core on Head Node is used to

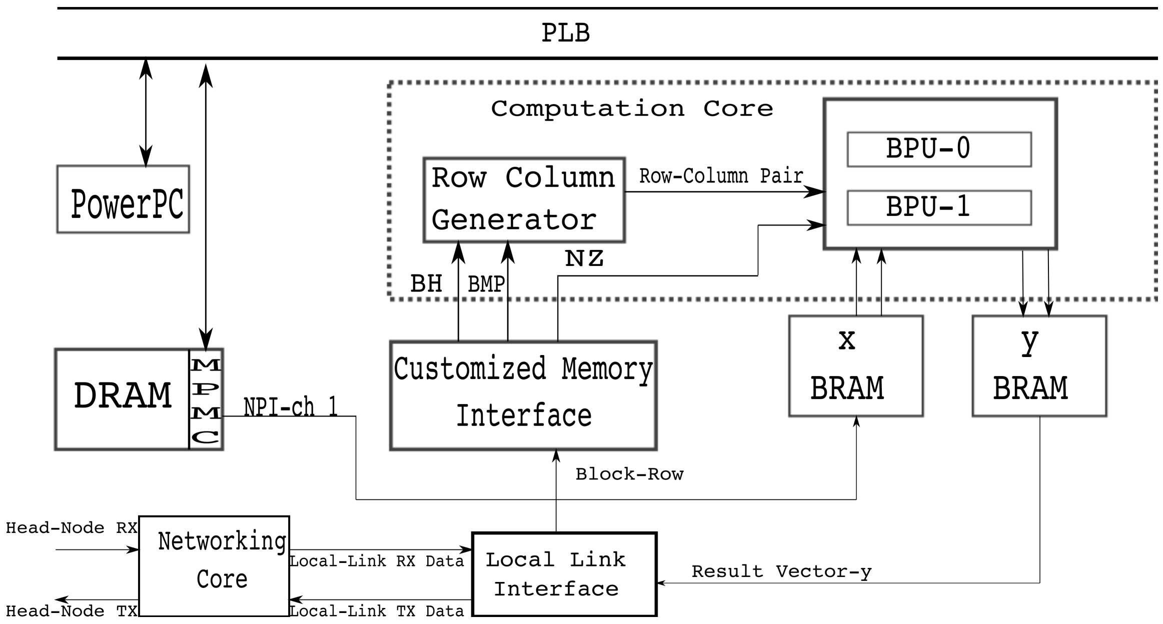

Figure 3.9: High level architecture of worker node

convert the block rows to the Local-Link data standard expected by the AIREN Interface. The Head Node provides the block row to each Worker Node sequentially (similar to an MPI_Send), so the Head Node will first provide block rows to Worker Node 0, then Worker Node 1 and so on. The Head Node also uses the Networking Core to read-back the individual components of the resultant vector calculated by each Worker Node and writes back these results to the main-memory (via NPI-Ch 2, Figure 3.10).

3.3 Software Design for Sparse Matrix Vector Multiplication

As described earlier our design approach consists of designing a custom memory hierarchy that works in conjunction with the VDCB format to perform sparse matrix operations and also providing a solution which is purely software based using the underlying processor memory subsystem to perform the sparse matrix operations. In this section we describe the design for performing the Software Only SMVM operation for a VDCB encoded matrix.

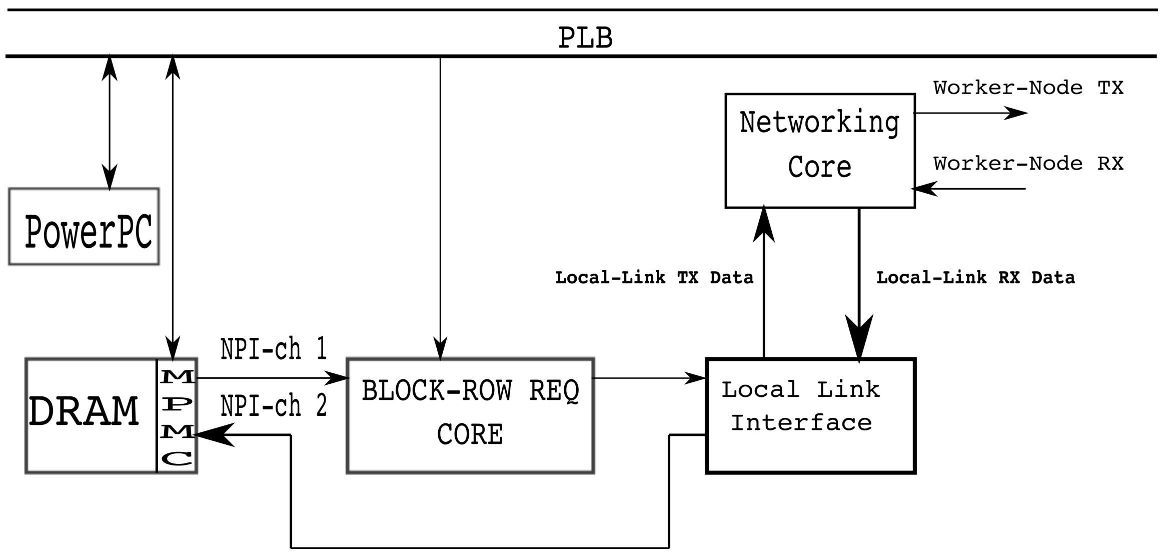

Figure 3.10: High level architecture of Head Node

Sequential Software Implementation We describe the sequential algorithm to implement the SMVM operation using the VDCB format in Algorithm 3, the various notations used to describe the algorithm are presented in Table 3.1. We can see from Algorithm 3 that although we are using the VDCB format to perform the SMVM operation we are still using array based lookups to retrieve relevant block informations from the memory and the decoding of the bitmaps is not happening "on the fly" unlike the hardware design. Thus in the Software Only approach the desired characteristic of the VDCB format to eliminate the indirect memory access is absent due to the lack of customized memory hierarchy provided by the Hardware/Software Co-Design approach.

Parallel Software Implementation In order to implement the parallelized software we use the Message Passing Interface (MPI) library. The parallelization strategy for the software design is identical to the hardware implementation presented in Section 3.2.2. The algorithm implementing the parallelized SMVM operation using the VDCB format is presented in Algorithm 4.

```
Input: $A_V$, $V$, $\vec{x}$, $\gamma$
Output: $\vec{y}$
for $i \rightarrow 0$ to $V - 1$ do
    Read Row-Start, Col-Start fields from $A_V[i]_{XY}$;
    Assign block-size $\gamma$ from $A_V[i]_{XY}$;
    Assign the $NNZ$ values present in $A_V[i]$ from $A_V[i]_{XY}$;
    Read locations $x[Y]$ to $x[Y + \gamma]$ ;
    Decode $A_V[i]_{BMP}$;
    Push the decoded Row-Column values to temporary array temp_row and
    temp_col ;
    for $j \rightarrow 0$ to $NNZ - 1$ do
        $y[temp\_row[j]] += A_V[i]_{NZ} \times x[temp\_col[j]]$;
    end
    Update the resultant vector $\vec{y}$;
end
```

Algorithm 3: SMVM operation using the VDCB format

```
Input: $SB$, $RR$, $\beta$, $w$
Output: $\vec{y}$
if SB == 1 then
    if Head Node then
        MPI_Send $\rightarrow$ Super Blocks to worker nodes;
    end
end
if RR == 1 then
    if Head Node then
        for $i \rightarrow$ to $\beta$ do
            MPI_Send $\rightarrow$ block row to worker nodes;
        end
    end
end
if Worker Node then
    MPI_Recv $\leftarrow$ Super Blocks from head node;
    Read vector $\vec{x}$ from memory;
    Perform matrix vector multiplication using Algorithm 3;
    Update vector $\vec{y}$;
end
```

Algorithm 4: Parallel SMVM operation using the VDCB format

3.4 Performing Sparse Matrix-Matrix Multiplication using VDCB Format

If we consider two square sparse matrices A and B of size $n \times n$ we can use a naïve approach to compute their product using Algorithm 5 ($a[i][k]$,$b[k][j]$ and $c[i][j]$ represents the elements of the matrix A, B, C respectively).

```
**Input**: $A$, $B$, $n$
**Output**: $C$
for $i \to 0$ to $n-1$ do
    for $j \to 0$ to $n-1$ do
        $c[i][j] = 0$
        for $k \to 0$ to $n-1$ do
            $c[i][j] += a[i][k] \times b[k][j]$
        end
    end
end
```

Algorithm 5: Basic matrix multiplication

The algorithm presented in Algorithm 5 performs multiplication even when a_{ik} or b_{kj} are equal to zero. In case of sparse matrices if we symbolically associate a "0" with each zero element and a "$\star$" with each NZ element then we can have four cases for multiplication as follows:
$0 \times 0, 0 \times \star, \star \times 0, \star \times \star$. We can see that for the first three cases the multiplication will result in a zero and due to identity property of addition these cases will not contribute towards the calculation of c_{ij}. In order to avoid these redundant multiplication operations due to presence of zero elements the SMMM algorithm should only perform computation when $a_{ik} = b_{kj} = \star$. We refer to this particular computation as *Required-Multiplication* (RM). We implement the SMMM operation by modifying the VDCB storage format described earlier in Section 3.1.The modifications made to the format in order to perform the SMMM operation are as follows:

- Only used fixed block sizes of $b \times b$ for sparse matrices A and B
- Generate row major bitmaps for all blocks of A and column major bitmaps for

all blocks of B

- Arrange blocks of matrix A in row major order and blocks of matrix B in column major order
- Arrange NZ elements of blocks of matrix A in row major order and NZ elements of blocks of matrix B in column major order

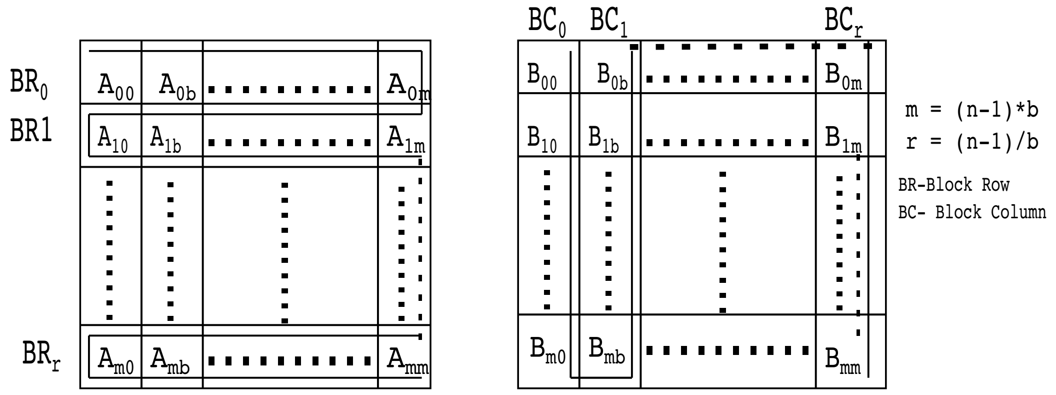

Figure 3.11: Block arrangement for the SMMM operation

3.4.1 Block Selection and Multiplication

If we refer to Figure 3.11 we can see the arrangement of blocks for matrix A and matrix B. We can thus represent the SMMM operation using Algorithm 6 ($m = (n-1) \times b$ in Algorithm 6). The computation of C when A and B are encoded in

```
Input: $A$, $B$, $m$
Output: $C$
for $i \rightarrow 0$ to $m$ do
    for $j \rightarrow 0$ to $m$ do
        for $k \rightarrow 0$ to $m$ do
            $C_{ij} \mathrel{+}= A_{ik} \times B_{kj}$
        end
    end
end
```

Algorithm 6: Matrix multiplication using the VDCB format

the VDCB format is almost identical to naive approach. The only difference is that instead of multiplying individual elements we are multiplying blocks of matrix A and matrix B. This particular distinguishing factor is actually useful in identifying the possible RMs.

Property 1. The condition of multiplication for two elements present at locations a_{xy} and b_{uv} belonging to A and B is that multiplication can only take place if $y = u$.

Instead of checking for the column and row value of every element in order to satisfy Property 1, we can check the block-header fields of the two blocks. Based on this comparison we can deduct if elements present in these blocks can be multiplied or not.

Lemma 1. When two blocks A_{XY} and B_{UV} are encoded in the VDCB format then A_{XY} can only multiply with B_{UV} if $Y = U$.

Proof. Consider two matrices A and B of size $n \times n$ divided into fixed block sizes of $b \times b$. The block A_{XY} can have maximum of b^2 elements with a set of possible indices $x_i \cup y_i$ such that:

$$x_i = \{X + 0, X + 1, X + 2, ...X + b - 1\}$$

$$y_i = \{Y + 0, Y + 1, Y + 2, ...Y + b - 1\}$$

Similarly for B_{UV} we will have elements with a set of indices $u_i \cup v_i$ given by:

$$u_i = \{U + 0, U + 1, U + 2......U + b - 1\}$$

$$v_i = \{V + 0, V + 1, V + 2......V + b - 1\}$$

Now if $u_i \cap y_i = \emptyset$ we can deduct that $Y \neq U$. This means that block B_{UV} covers a different range of rows than the once expected by columns of A_{XY} and hence Property 1 is not satisfied.

Another possibility is when $u_i \subseteq y_i$ or vice-versa. In both cases $u_i \cap y_i \neq \emptyset$. This means multiplication is possible between the two blocks and only checking for inequality between the first element of u_i and first element of y_i is insufficient. This basically implies, only checking for the row-start field and col-start field (that is only comparing Y and U) will result in skipping the RMs that would have taken place for the two blocks.

A subset between u_i and y_i can arise if $Y = U + l$ or vice-versa where $0 \leq l \leq b-1$. If $l = 0$ then $u_i \equiv y_i$ and it satisfies the condition of coverage of rows and columns for the blocks stated before (Property 1). But we will still have to consider the case where $1 \leq l \leq b-1$.

We know that the number of block-rows and block-columns for A and B is given by n/b. So the possible values of row-start field of block header; in our case U, can be given $m \times b$ where $0 \leq m \leq (n-1)$ as shown in Figure 3.11. Similarly the possible values for col-start field of block header Y can be given by $k \times b$ where $0 \leq k \leq (n-1)$. So the values of U and Y (row-start and col-start fields) for blocks of matrices B and A is going to increment by a *constant amount "b"*. Thus the possibility of having a row-start field $U = Y + l$ or col-start field of $Y = U + l$ where, $1 \leq l \leq b-1$ is not going to exist. Thus comparing the row-start field U of B_{UV} with col-start field Y of A_{XY} will be sufficient to conclude if the two blocks can be multiplied or not. $\square$

The selection of blocks based on Lemma 1 is the first step in identifying potential RMs. The second step is to isolate the RMs that are present when multiplying the two selected blocks. If we refer to Algorithm 5 we know that every row of block A_{XY} will multiply with every column of block B_{UV}. We can represent the problem of finding RMs for the two blocks using the code presented in Algorithm 7. The example of block multiplication shown in Algorithm 7 indicates that a dot-product has to be computed between each row of A_{XY} with each column of B_{UV}.

As the blocks A_{XY} and B_{UV} are encoded in the VDCB format we will need to

```
**Input**: $A_{XY}$, $B_{UV}$,$b$
**Output**: $C_{XV}$
**if** $Y == U$ **then**
    **for** $i \to 0$ **to** $b-1$ **do**
        **for** $j \to 0$ **to** $b-1$ **do**
            $c[X+i][V+j] = 0$
            **for** $k \to 0$ **to** $b$ **do**
                **if** $a[X+i][Y+k]$ *and* $b[U+k][V+j] \neq 0$ **then**
                    $c[X+i][V+j]$+=$a[X+i][Y+k] \times b[U+k][V+j]$
                **end**
            **end**
        **end**
    **end**
**end**
```

Algorithm 7: Example of block multiplication

decode the corresponding bitmaps BMP_{XY} and BMP_{UV} to find the NZ elements associated with $a[X+i][Y+k]$ and $b[U+k][V+j]$. This is an inefficient approach and more optimized method can be employed by using the bitmaps of A_{XY} and B_{UV}. Consider that the row-m of block A_{XY} and column-n of B_{UV} have to be used for a dot-product computation. As bmp_m^A is a row-major bitmap the bits set to one indicates the positions of NZ elements present in row m. Similarly bmp_n^B is a column major bitmap and indicates the NZ elements present in the column n. If we perform an AND operation between the two bitmaps we can basically find out if there are any NZ elements for RMs.

$$mult\_bmp = bmp_m^A \ AND \ bmp_n^B \tag{3.5}$$

If the *mult_bmp* is zero then no RMs are present for the dot-product. In case *mult_bmp* is not equal to zero, then we know that row-m and column-n have a point of overlap where they both have a NZ element which has to be utilized for the RM and which contributes towards the dot-product. An example of the dot-product operation using the the bitmaps associated with the row and column, along with the corresponding NZ arrays is shown in Figure 3.12. A block of size $b \times b$ will require a

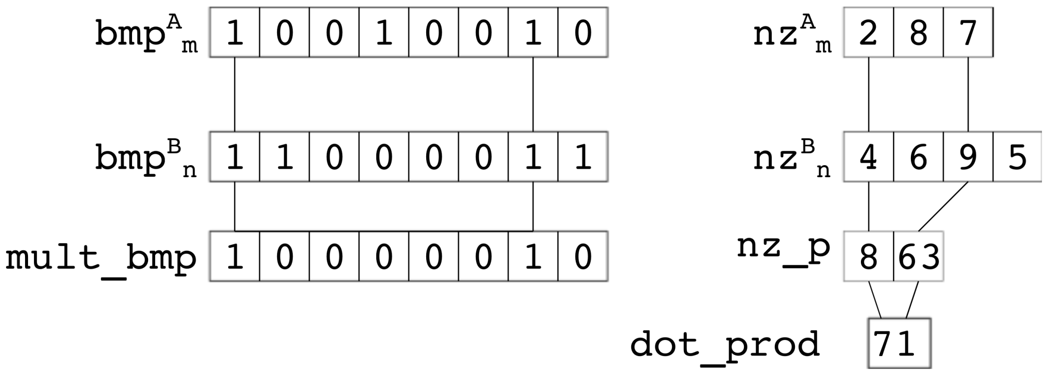

Figure 3.12: Dot product using bmps

bitmap which is b^2 bits wide to completely represent its index information . In other words, for a row-major bitmap for a block of size $b \times b$, each row's index information is represented by "b" number of bits. Similarly for a column-major bitmap each column's information is represented by "b" number of bits.

In the example shown in Figure 3.12 we have used a block size of 8×8 so the size of bmp^A_m and bmp^B_n is eight bits. We can also see from the example in Figure 3.12 that the size of NZ array is equal to the number of bits that are set to one in the corresponding bitmap. It also implies that we cannot simply multiply the two NZ arrays to obtain the dot-product. If we look at the example of dot-product (Figure 3.12); if we multiply the two NZ arrays we will be performing the following operation:

$dot\_prod = (2 * 4) + (8 * 6) + (7 * 9) + (5 * ?)$

Firstly we see that we are performing more number of RMs than actually needed and secondly we don't know what the last value of nz^B_n is going to be multiplied with. This means we need to determine the elements belonging to arrays nz^A_m and nz^B_n which are going to participate in the dot-product computation.

The number of bits which are set in *mult_bmp* indicates the number of RMs that need to take place. In our case the *mult_bmp* (Figure 3.12) has two bits set to "one" and thus two RMs need to take place. Now we have to find out which two elements from nz^A_m and nz^B_n have to be used. This can be done by decoding the bitmaps of a block and determining the location information of the elements present in their NZ array. The location information basically provides the row-column position of each element in NZ array. Thus the decoding operation on bmp_m^A will give us the location of each NZ element in nz_m^A as follows: $(m,0),(m,3),(m,6)$. This basically indicates that NZ values are present within the mth-row at column positions $0,3,6$ Similarly the decoding operation of bmp_n^B which is a column major bitmap will give us locations $(0,n),(1,n),(6,n),(7,n)$, which tells us the row positions within nth-column which have NZ elements. Once the decoding operation has taken place we can easily identify the elements from the NZ arrays that will be used for RMs by using the Property 1.

3.4.2 Count Ones Technique

The decoding of bitmaps requires that the multiplication operation cannot start till the location information for all NZ elements represented by bmp_m^A and bmp_n^B is generated. This additional latency introduced for the multiplication operation by decoding of bitmaps can be eliminated by using the *Count Ones Technique*.

In this method instead of decoding the bmp_m^A and bmp_n^B we directly calculate the offsets for the NZ arrays to find the NZ elements which are going to be used for performing RM. If we revisit the example shown in Figure 3.12 we can see that for nz_m^A array, elements $nz_m^A[0]$ and $nz_m^A[2]$ are used for the RMs.

To start calculating the offsets of nz_m^A array we have to account for the NZ elements present in the array which are not contributing towards RM, but which are needed for correct indexing of the NZ array. We first look at the *mult_bmp* and determine the positions of bits that are set to one. The first bit that is set to one in our case is *mult_bmp*(0). Now for calculation of the offset; as we know that bit-position zero is set to one, we can conclude that $nz_m^A[0] = 2$ is used for RM. After we have performed specified lookup from the NZ array we set *mult_bmp*$(0) = 0$ and the new value of *mult_bmp* $= 00000010$. The next bit which is set to one is at *mult_bmp*(6). To determine the next offset for NZ array we count the number of "ones" that are present in bmp_m^A before the bit position $bmp_m^A(6)$. These "ones" account for the previous NZ elements which have to be skipped in order to obtain the next NZ element from the array which is going to contribute towards the RM. We can see that before $bmp_m^A(6)$ two more bits have been set to one which indicate NZ elements at $nz_m^A[0]$ and $nz_m^A[1]$. Thus the next NZ element from the NZ array which is going to be used for RM is $nz_m^A[2] = 7$. Once the required value has been provided for RM we set the $mult\_bmp(6) = 0$. There are no longer any bits set to one in $mult\_bmp$ thus all the look-ups for RMs have been performed on nz_m^A. This same technique is used for the bmp^B and nz_n^B to determine the NZ elements for the RM. Once all the lookups for the RMs have been performed for both the arrays a new set of dot-product computation can begin. An example demonstrating this particular approach is shown in Algorithm 8 (the variable b indicates the block-size).

```
Input: $b$, $bmp_m^A$, $nz_m^A$,$mult\_bmp$
Output: $RM\_operand$
for $i \to 0$ to $b - 1$ do
    if $mult\_bmp(i) == 1$ then
        $j = i$;
        $count\_ones = 0$;
    end
    for $k \to 0$ to $j - 1$ do
        if $bmp_m^A(i) == 1$ then
            $count\_ones = count\_ones + 1$;
        end
    end
    $RM\_operand = nz_m^A[count\_ones]$ ;
    $mult\_bmp(i) = 0$;
    $count\_ones = 0$;
end
```

Algorithm 8: Count Ones technique

3.5 Hardware Design for the SMMM Operation

In this section we present the hardware design to perform the SMMM operation when using the Hardware/Software Co-Design approach to use the VDCB format to implement the SMMM operation. We have divided our high level architecture

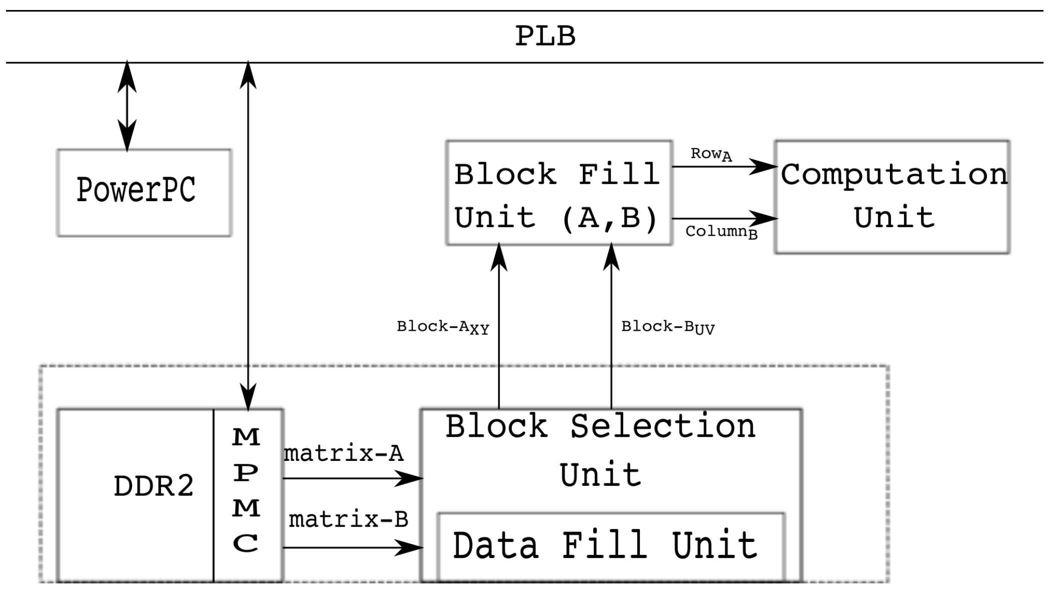

Figure 3.13: High level architecture for SMMM operation

(Figure 3.13) into three subsystems: Computation Unit (CU), Block Selection Unit (BSU) and Block Fill Unit (BFU).

3.5.1 Block Selection Unit

The Block Selection Unit (BSU) is used to identify the blocks which are going to be used for RMs. The BSU also has a Data Fetch Unit (DFU) which is used for fetching data from the memory using Native-Port Interface (NPI) channel of Multi-Port Memory Controller (MPMC). The DFU is aware of the block arrangement of matrix A and matrix B in the memory and based on that it fetches the block rows of A and block columns of B. To obtain a block row of C a block row of A has to be multiplied with each block column of B (entire matrix B encoded in the VDCB format). Thus the DFU stores a block row of A on the on-chip memory called Block RAM (BRAM) and stores all the block columns of B in FIFO. After finishing the calculation of one block row of C the DFU fetches the next block row of A and all the block columns of B.

The BSU has a Finite State Machine (FSM) which uses Lemma 1 to make the selection of the blocks which can be used for RMs. The BSU FSM uses the Table 3.2

Table 3.2: BSU FSM action

Comparison	Pop Block of A	Pop Block of B	Perform Multiplication
col_startA = row_startB	✓	✓	✓
col_startA < row_startB	✓		
col_startA > row_startB		✓	

to determine the actions that it needs to perform based on the comparison of the col_start field of matrix A and row_start field of matrix B.

It can be seen from the Table 3.2 that the inequality between the col_start and row_start field results in a pop operation either from matrix A or matrix B. This is an important step in order to provide a solution which is free of unordered merge. As the blocks of matrix A are stored within a BRAM and blocks of matrix B are stored within a FIFO the comparison operation can be done in two ways.

A simple solution is every time a block of matrix B is popped from the FIFO compare it against all the blocks of matrix A present within the BRAM to find a possible match for multiplication. This is a highly inefficient method and forces us to perform a merge operation at block level. Instead if we look at the way the blocks are arranged in Figure 3.11 we know that if col_startA of block A is less than the row_startB of block B then all the blocks present before block A will not be multiplying with block B as there col_start field is going to be lesser than the col_startA. Hence the address for the BRAM storing blocks of matrix A should be kept incrementing till the col_startA < row_startB condition becomes false. Similarly when the col_startA is greater than row_startB then the blocks from the FIFO need to be kept popping till the col_startA > row_startB condition becomes false.

The popping of blocks of matrix A from the BRAM is easy as it only requires an address increment based on the block headers. The FSM reads the number of NZ field from the block header and it already knows the number of bitmaps that are present for a block (based on the block size) and it increments the address by an amount given by:

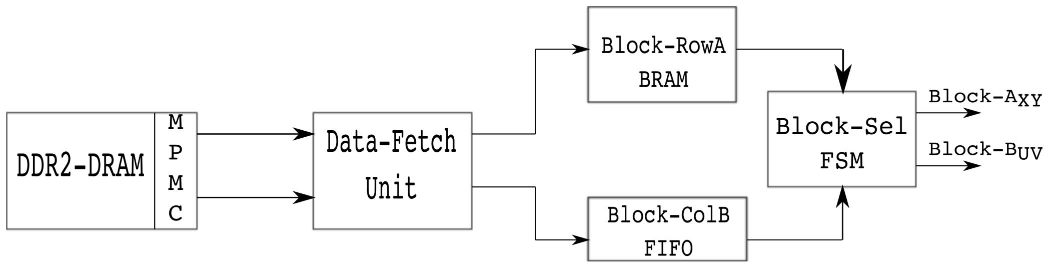

Figure 3.14: Block selection unit

New Block Addr = Current Block Addr + # of bitmaps + # of NZ
This increment operation keeps taking place till the condition for popping blocks of matrix A becomes false.

In case of popping the blocks of matrix B a FIFO is used and this makes the popping blocks of matrix B more time consuming. Unlike the case of matrix A where the address increment was made to move to next block header (beginning of a new block) and all the intermediate bitmaps and NZ elements were skipped; the FIFO storage does not provide us the option of skipping the read operation on bitmaps and NZ elements. The FIFO is a queue based operation and the concept of address is absent for it. This makes the FSM state of popping blocks of matrix B the most time consuming state of the BSU FSM. As soon as the first element of B is fetched from the memory and written into the FIFO the FSM starts comparison and it does not wait for the entire matrix to be written into the FIFO before it starts selection. This is because as the VDCB format is a streaming based storage format and the blocks of matrix B are arranged in column-major order, thus the blocks are fetched from the memory in order in which they are needed for multiplication. Hence the comparison can start as soon as the first element (the block header of the very first block of matrix B) gets written into the FIFO.

3.5.2 Block Fill Unit

The Block Fill Unit (BFU) is provided by the blocks from the BSU as shown in Figure 3.13. The BFU is responsible for providing the row and column information

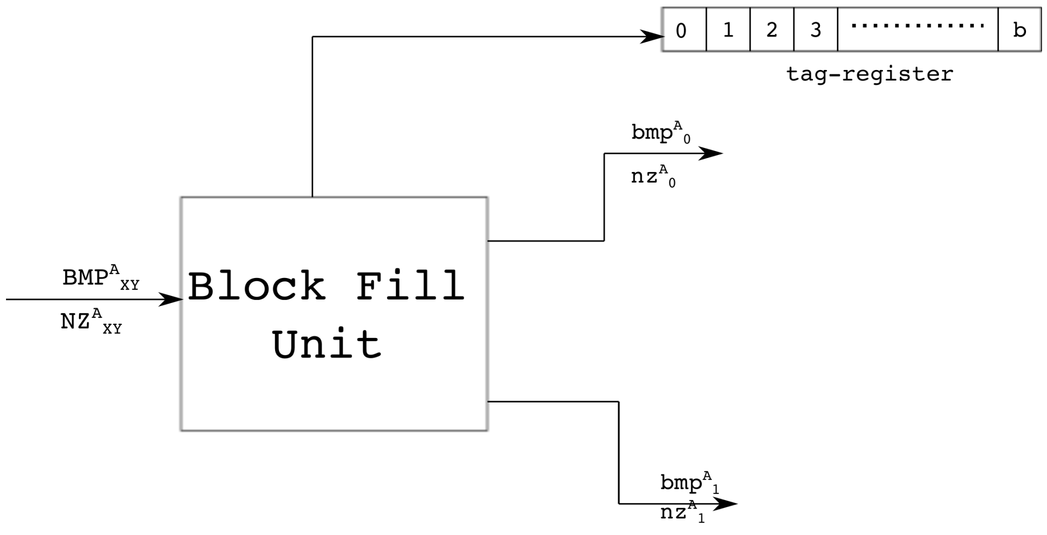

Figure 3.15: Block fill unit

to the PEs.

As discussed earlier one row of block of matrix A will multiply with an entire block of matrix B to provide a row of block of resultant matrix C. Thus it is important to consider the potential of reusability the block of matrix B provides. The BFU provides each PE with the block of matrix B to be stored within the BRAM. The BFU then provides individual rows of block of matrix A to the PE. Based on the constant block size b in which the matrices have been divided the BFU determines the size of *bmp* which will represent each row of matrix A.

The BFU accepts BMP^A_{XY} and NZ^A_{XY} from the BSU and parses the BMP^A_{XY} which is b^2 bits wide. The BSU divides the BMP^A_{XY} into b individual bitmaps representing each row of the block of matrix A. The BFU is thus able to provide bmp^A_0, bmp^A_1 and so on till bmp^A_b. Once the BMP^A_{XY} has been divided into the individual bitmaps representing the rows, the next part is to determine the NZ elements from the NZ^A_{XY} array that belong to each row. The BFU evaluates each *bmp* sequentially in an ascending order starting from bmp^A_0 and counts the number of bits that are set to “one” within the *bmp*. It removes the corresponding number of NZ elements from the NZ_{XY}^{A} and provides it to nz_0^A array associated with bmp_0^A. The NZ elements for the block of matrix A are arranged in row major order and the first element of the array NZ_{XY}^{A} will represent the first element present within the first row in which a NZ element is present. The BFU also maintains a tag register which is “b” bits wide and is used to indicate when a valid row information is available for the PE. Every time the BFU generates a pair of *bmp* and *nz* array associated with a row it sets the tag register bit to one. Thus for example if tag register(1) (tag register’s second bit) is set to one, indicating bmp_1^A and nz_1^A are available for computation. The BFU sets a one for tag register even if *bmp* is equal to zero and is not going to result in any computation. This is done in order to keep the BFU tag functionality simple and making the PE responsible for discarding the *bmp* value if it is set to zero.

In the current design we support up to two Processing Elements (PE) which are used to perform the multiplication operation. We use a modulo-2 operation in the BFU and provide the even numbered rows to PE(0) and odd numbered rows to PE(1). The BFU also provides the PEs with the block headers of the blocks selected for multiplication.

3.5.3 Computation Unit

The Computation Unit (CU) (Figure 3.16) is responsible for multiplying the two blocks and assembling the resultant block in the VDCB format. The CU consists of two Processing Elements (PE) as shown in Figure 3.16. The PEs present within the CU consist of a Multiply and Accumulate unit which are implemented using floating point adder and multiplier to calculate dot products.

The CU stores the block headers of the two blocks that were selected for multiplication within a register for later use. The CU stores the bmp_n^A in a register and the corresponding NZ elements nz_0^A in a BRAM. The CU receives the block of matrix B from the BFU and it stores the bitmap of the block BMP_{UV}^{B} in a register and the corresponding NZ elements NZ_{UV}^{B} in a BRAM.

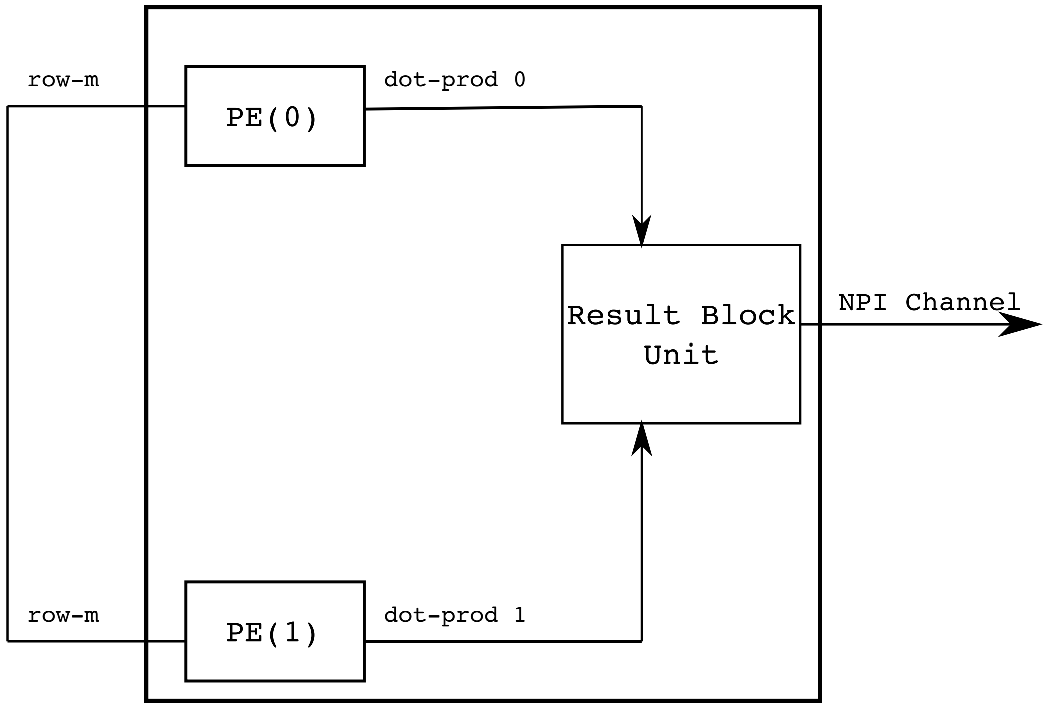

Figure 3.16: Computation unit

The CU FSM shown in Figure 3.17 waits for the row information to be provided by the BFU before it starts the operation. Once the row information has been provided by the BFU the FSM moves out of the “idle” state. The CU FSM does not wait for the Col_BRAM to be populated completely by the block of matrix B before it starts the operation. This is because once the BMP^B_{UV} has been received by the CU it calculates the NZ elements that are going to follow the bitmap based on the number of bits which are set to one within BMP^B_{UV}. Thus the CU FSM knows exactly how many locations are going to be populated by the NZ elements and if a BRAM location where an NZ element is expected is set to zero (the initialization value for the BRAM) the FSM knows it has to wait till the NZ element has been written into the Col_BRAM as the FSM is expecting a NZ element there.

The CU reads the bmp^A_n from Row_BRAM and BMP^B_{UV} from the Col_BRAM and parses the BMP^B_{UV} to provide the individual bitmaps associated with a column. This is similar to how the BFU parses BMP^A_{XY} for the bitmaps associated with a row.

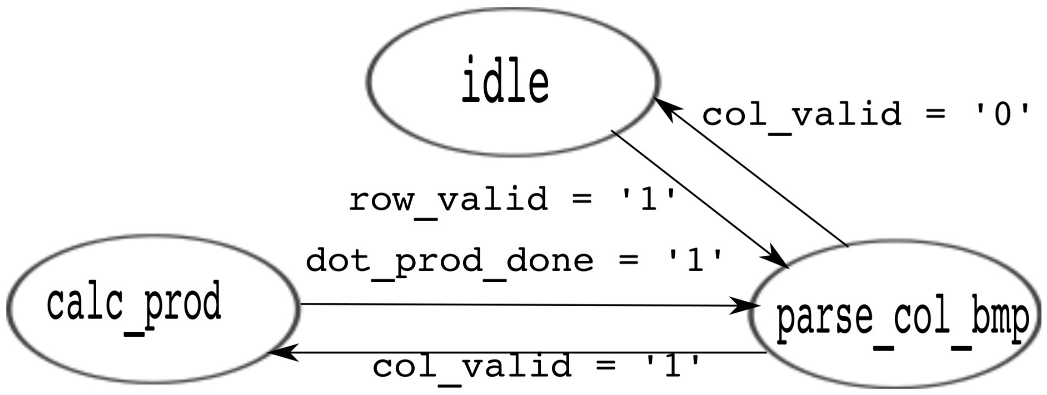

Figure 3.17: Control unit FSM

The "parse_col_bmp" state parses the BMP_{UV}^{B} to provide the bitmap associated with one column at a time. The "parse_col_bmp" state starts from the very first column and determines the NZ elements associated with the column by looking at the column bitmap. The number of bits set in the column bitmaps are used to increment the address of the Col_BRAM and provide the corresponding NZ elements. The first element within the Col_BRAM at address 0x00 will be corresponding to the first element present within the block (column major ordering for NZ elements).

After the column bitmap and the corresponding NZ elements are obtained the FSM moves to the "calc_prod" state. In this state the dot product is calculated by the PE. To compute the RMs between a row and column the PE uses the count-ones technique described in Section 3.4.2. The count-ones unit provides a new RM-operand to the MAcc unit every clock cycle, till all the RMs have been calculated. The FSM transitions from "calc_prod" state back to "parse_col_bmp" state to provide the bitmap and NZ information for the next column. The FSM toggles between these two states till all the columns have been exhausted.

Generating the resultant in the VDCB Format The CU also ensures that the result block C_{XV} is encoded in the VDCB format. The PE maintains a *prod_bmp_reg* which is b bits wide, as the resultant block C_{XV} will be $b \times b$ in size and each bitmap corresponding to a row will be b bits wide. When the count ones technique calculates

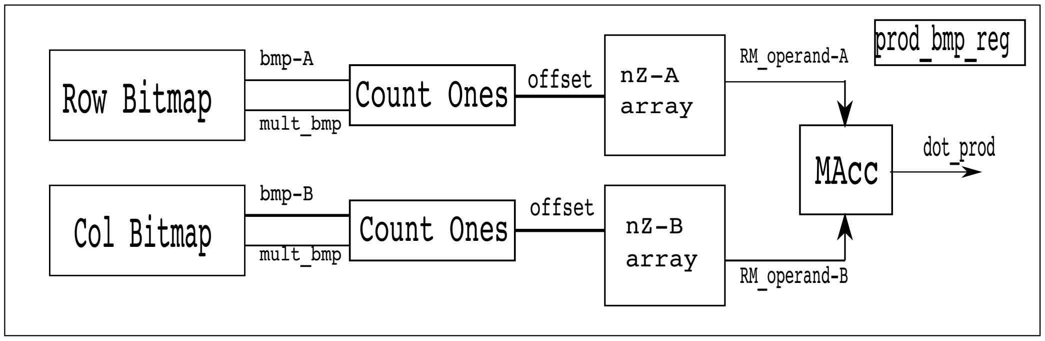

Figure 3.18: Processing element

mult_bmp in the "parse_col_bmp" state the bit is set to one if a *mult_bmp* $\neq 0$ is calculated otherwise the bit is set to zero. As the row and column multiplication takes place sequentially in ascending order the bits are also set in the register *prod_bmp_reg* starting from bit position zero. The dot product computation resulting in the NZ elements belonging to the row of resultant block C_{XV} are stored in a result BRAM in the CU. The result BRAM is b^2 deep as this will be the maximum number of NZ elements that can be present within the resultant block. Each PE maintains a FIFO to store the resultant NZ elements from the computation and then writes these NZ elements to the result BRAM. In writing the NZ elements the FIFO present in PE(0) is used first and then the FIFO present in PE(1); this ensures that the resultant NZ elements are written in a row major order. This is because PE(0) will receive all the even numbered rows, i.e. row-0, row-2 and so on and PE(1) will receive all the odd numbered rows (modulo-2 operation performed by the BFU). Thus alternating the writing from the two FIFOs starting from the one present in PE(0) will ensure that all the NZ elements

The block header for the resultant block is calculated using the block headers provided to the CU by the BFU. Once the computation of one row of C_{XV} is finished the Result Block Unit (RBU) present in CU reads the *prod_bmp_reg* of all the PEs. It combines the one-bit values of all *prod_bmp_reg* to form the bitmap corresponding to that particular row of C_{XV}. It also reads all the dot-products computed by the PEs and stores them within its nz-array. The block header for block C_{XV} is also formed by the RBU. It fills the row-start and col-start field of block header with X and V which it reads from the block headers of the incoming blocks provided for multiplication. It fills the block size to a constant b^2 and continuously keeps track of the number of NZ elements written in the nz-array to determine the value for the Total-NZ elements field of block header. Once it generates the VDCB encoded result block C_{XV} it writes it back to memory using the Native-Port Interface (NPI) channel of Multi-Port Memory Controller (MPMC).

3.5.4 Parallel Hardware Design for SMMM Operation

In order to calculate the block row of resultant matrix C we need a block row of matrix A and entire matrix B. Hence we can use a strategy similar to the one described for the parallelization of the SMVM operation in Section 3.2.2 where we proposed a simple star topology with one head node and multiple worker nodes (Figure 3.8). The only difference in the case of the SMMM operation will be that each worker node will be storing the entire matrix B, instead of the vector $\vec{x}$ which was used in the SMVM operation. We also use the Round-Robin and Super Block communication strategies described in Section 3.2.2 for the head node to worker node communication of block rows. The head node implementation is identical to the one presented in Section 3.2.2.1.

The parallel hardware design can be divided into two major components: The Computation Core which implements the SMMM operation and the Networking Core which enables communication between the nodes within the FPGA cluster. As the SMMM operation is only performed by the worker node, the computation core is only present within the worker node. The head node is used for the distribution of block rows of matrix A to all the worker-nodes and is used to read resultant block rows of matrix C from the worker nodes. The head node and worker node both have the

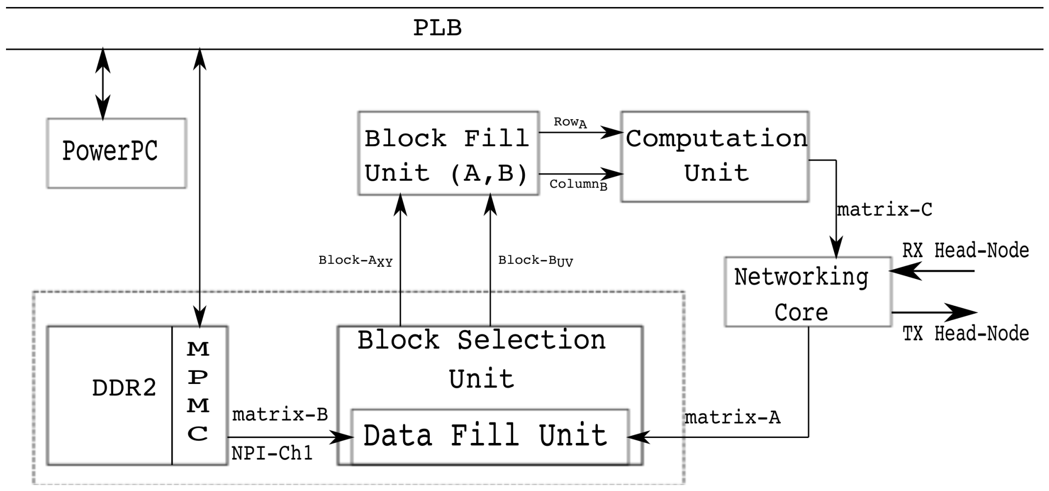

Figure 3.19: High level architecture for parallel SMMM operation

networking core which is used for communication. The high level architecture for performing the parallel SMMM operation is shown in Figure 3.19.

3.6 Software Design for Sparse Matrix-Matrix Multiplication

The software design for the SMMM operation using the VDCB operation is presented in Algorithm 9. The algorithm uses Lemma 1 and Count ones technique to perform the VDCB operation.

The Algorithm 10 presents the implementation of parallel SMMM operation. It can be seen from Algorithm 10 that an approach similar to parallel SMVM operation is used for the parallel SMMM operation. In case of both the operations we use block row level parallelism to perform the matrix computation along with the MPI libraries.

```
Input: $A$, $B$, $m$,$n$
Output: $C$
for $i \rightarrow 0$ to $m$ do
    for $j \rightarrow 0$ to $n$ do
        Read Block_Headers of block $A[i]$ and $B[j]$ ;
        Perform comparison using Lemma 1;
        if $row\_startB == col\_startA$ then
            Perform Count One's Technique;
            Generate the VDCB encoded block of matrix $C$;
        end
        else if $row\_startB$ ¿ $col\_startA$ then
            Keep reading blocks of matrix $B$;
        end
        else if $row\_startB$ ¡ $col\_startA$ then
            Keep reading blocks of matrix $A$;
        end
        Update matrix $C$;
    end
end
```

Algorithm 9: Sequential software implementation of SMMM operation

```
Input: $SB$, $RR$, $\beta$, $w$
Output: $\vec{y}$
if $SB == 1$ then
    if $Head\ Node$ then
        MPI_Send $\rightarrow$ Super Blocks to worker nodes;
    end
end
if $RR == 1$ then
    if $Head\ Node$ then
        for $i \rightarrow$ to $\beta$ do
            MPI_Send $\rightarrow$ block row to worker nodes;
        end
    end
end
if $Worker\ Node$ then
    MPI_Recv $\leftarrow$ block rows from head node;
    Read matrix $B$ from memory;
    Perform matrix multiplication using Count Ones Technique;
    Update vector blocks of matrix $X$;
end
```

Algorithm 10: Parallel SMMM operation using the VDCB format

CHAPTER 4: EVALUATION

The efficacy of the VDCB format will be evaluated for two Sparse Matrix Operations: Sparse Matrix Vector Multiplication (SMVM) and Sparse Matrix Matrix Multiplication (SMMM). As discussed earlier in Chapter 1 the low memory bandwidth utilization of the sparse matrix operations is not just the function of the storage format but also the underlying architecture. In order to test this hypothesis we proposed two different design approaches in Chapter 3 namely: Hardware/Software Co-Design and Software Only Design. These two design approaches are used in order to provide a comparison between a conventional processor based memory hierarchy and customized memory hierarchy implemented using the FPGA when using the VDCB storage format to perform sparse matrix operations. The usage of these two design implementations helps us to evaluate the efficacy in terms of performance gain achieved for sparse matrix operations when using the VDCB storage format exclusively (Software Only) and performance gain achieved when customized memory subsystem working in conjunction with the VDCB format (Hardware/Software Co-Design) is used. The performance metrics which we are going to use to evaluate the two design approaches are presented in Table 4.1.

Table 4.1: Performance metrics for evaluation

Performance Metric	Software Only Solution		Hardware/Software Co-Design Solution	
	SMVM	SMMM	SMVM	SMMM
Index Overhead	✓		✓	
Bandwidth Efficiency	✓	✓	✓	✓
Floating-Point Performance	✓		✓	
Computation Time		✓		✓
Communication time for head node			✓	✓
Speedup	✓	✓	✓	✓
Resource Utilization			✓	✓
Sparsity of matrix B		✓		✓

4.1 Performance Analysis of Hardware/Software Co-Design Approach

The hardware design was implemented using Xilinx ML410 board and developed in VHDL. The Xilinx ML-410 board has a Virtex-4 XC4VFX60 FPGA (with a PowerPC processor) and 512MB of DDR2 SDRAM. The Xilinx Embedded Development Kit (EDK) version 13.2 was used to develop and synthesize the design. The preprocessing for generating VDCB format is done off-line on a conventional server. We have used matrices from University of Florida Matrix Market Place[24] for testing our software and hardware design. The input applied to the Search Heuristic (Algorithm 2) is a matrix in Coordinate format (.coo).

4.1.1 Results and Analysis for Sparse Matrix Vector Multiplication

The structural and domain specific information regarding all the matrices used for testing the SMVM operation are presented in Table 4.2.

The practical implementation of the VDCB format when performing the SMVM

Table 4.2: Test matrices characteristics

Serial Number	**Matrix Name**	**Dimension**	**NNZ**	**Domain**
1	c-38	8127×8127	42908	Optimization
2	fd12	7500×7500	28262	Material problem
3	rajat03	7602×7602	32653	Circuit simulation
4	poli	4004×4004	8188	Economic problem
5	t2dal	4257×4257	20861	Model Reduction
6	rw5151	4008×4008	20199	Statistical Mathematical problem
7	bcssmt26	1922×1922	1922	Structural problem

operation uses double-words (64-bits) to represent the block header and bitmaps. In case of block sizes greater than 8×8 more than one bitmap is used to represent index information of a block, thus the number of bitmaps associated with a block is given by Equation 4.1

$$Number\ of\ Bitmaps = \frac{BlockSize}{64}. \tag{4.1}$$

The peak floating point performance available for the design is calculated using the number of floating point operations and the frequency at which they are being carried out. In our case the floating point computation is being carried out by the BPU and it performs a floating point multiplication and a floating point addition (in the form of accumulation operation) every clock cycle. Both these operations are being carried out at 100MHz frequency and provide a peak floating point performance of 200MFLOPS ($2FLOP \times 100MHz$, where FLOP is FLoating point OPeration). Thus each BPU is capable of providing 200MFLOPs performance. The sequential implementation consists of a Computation Unit with two BPUs (Figure 3.2) resulting in a peak floating point performance of 400 MFLOPS ($2 \times 200MFLOPs$). Also in case of parallel implementation the computation is being carried out by the worker node and the worker node consists of two BPUs providing a peak performance of 400 MFLOPS per worker node.

4.1.1.1 Index Overhead

It is imperative to have index information (location) of NZ elements when performing SMVM, but this information does not contribute towards the actual computation.

Thus from a memory subsystem perspective additional load operations are incurred. If we minimize the number of load operations we effectively maximizes bandwidth utilization as majority of data (NZ elements in this case) being fetched is going to be used for useful computation. To quantify the stress placed on memory subsystem by these additional load operations we introduce a term called *Index Overhead* denoted by α. We define $\alpha = \frac{MS_{ind}}{MS_{tot}}$, where MS_{ind} is the memory needed for storing index and MS_{tot} the total memory needed for storage format. The index components in the VDCB format are Block Headers and Bitmaps. The number of bitmaps associated with a matrix are dependent on the various block sizes a matrix gets decomposed into. We wanted to see the effect of varying block size on α. We generate VDCB format using software code discussed in Section 3.1.1. We refer to this particular VDCB generation scheme as *greedy.* Next we generate fixed block sizes of 8×8 and 64×64; the *Smallest* and the *Largest* blocks supported by the RCG respectively. We will refer these two strategies as *smallest* and *largest.* We also generate VDCB using block sizes picked randomly by the software and remove the density constraint. We refer to this strategy as *random.* We also compare the index overhead from the commonly used CSR and COO formats.

The x-axis in Figure 4.1 indicate the serial number of matrices shown in Table 4.2. In the *greedy* approach, the majority of the blocks were 8×8, thus the index overhead for these two strategies is pretty close. The index overhead for *random* and *largest* is quite high as seen in Figure 4.1. In case of *random* most blocks were of sizes 64×64 or 56×56, resulting in index-overheads close to *largest* strategy. As the block sizes start becoming bigger the number of bitmaps associated with them also increases, thereby increasing the index overhead. Also for larger blocks there is an increase in number of empty rows. In bitmap representation this results in a bitmap with all its bits set to zero. We call such bitmaps as *AllZero* and how much they contribute to the index overhead is shown in Figure 4.1. It can be seen for *smallest*, *AllZero*

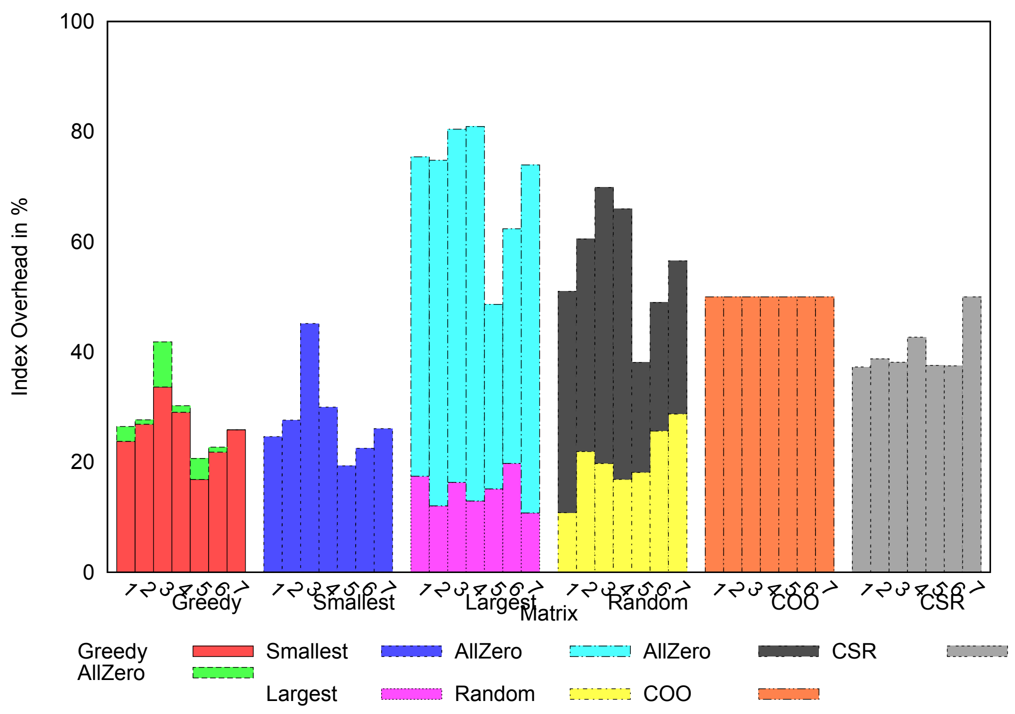

Figure 4.1: Index Overhead

bitmaps are completely absent. An 8 × 8 block will need only one bitmap and for it to be an *AllZero* the block has to be completely empty. If a block is empty it will not be picked by the search software. When compared to CSR and COO formats the index overhead added by VDCB format in *greedy* and *smallest* is significantly less. This is a major advantage of the VDCB format over CSR and COO, not only VDCB format maximizes bandwidth utilization by adding lower index overhead; it also has no indirect memory access. The indirect memory accesses hampers the performance of CSR and COO formats. The index-overhead is largely dependent upon the distribution of NZ elements within a matrix and the efficacy of software code to search for dense blocks. Thus *t2dal* which has a number of dense sub-blocks has lower index-overhead than *rajat03* which comprises of small sparse blocks with one or two NZ elements.

4.1.1.2 Sequential Hardware Design Performance Evaluation

The sequential design consists of a single FPGA node. The FPGA node communicates with the off-chip DDR2-SDRAM via two Native Port Interface (NPI) channels of Xilinx's Multi-Port Memory Controller (MPMC). The two memory channels are used to read the VDCB format and the $\vec{x}$ respectively. The theoretical peak floating point performance for our design is 200MFLOPS with one BPU and 400MFLOPS with two BPUs. The maximum memory bandwidth available is 1GB/sec. Our focus is on the speed of the operation. When timing the operation we do not consider the time it takes to convert sparse matrix to VDCB format nor the time it takes for a standalone C-program running on PowerPC to set up and control the experiments. We do consider the time it takes for the format to be read from the main memory. We have evaluated our design in terms of bandwidth efficiency and floating point performance.

Bandwidth Efficiency In case of sequential design matrix A is stored off-chip within the main-memory of the FPGA node. The speed at which matrix A can be read from the main-memory is going to affect the overall speed of the operation. As the speed at which matrix A can be read from the main-memory depends upon the utilization of available memory-bandwidth, we study the factors that affect the bandwidth-efficiency. We investigate the effect of block-sizes and increased number of BPUs on the bandwidth efficiency. The results are shown in Figure 4.2 and 4.3. The maximum memory bandwidth available for our design is 1GB/sec (based on the measurements made on the NPI channel connected to the main memory).

Our design provides an average bandwidth efficiency of 58% for one BPU when using *greedy* and *smallest*. In case of two BPUs the efficiency increases to 70% for these two strategies. The memory channel has to go through flow control (Section 3.2.1.2) once the VDCB component FIFOs start getting full. The pausing of memory channel prevents us from utilizing the bandwidth to its fullest. We have observed that in the

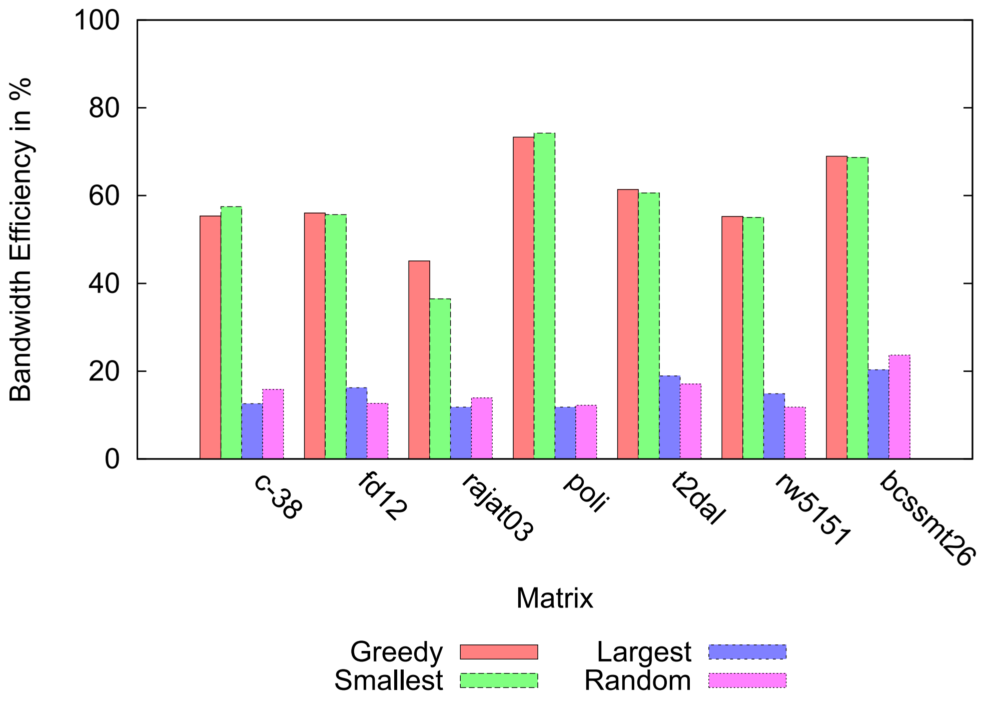

Figure 4.2: Bandwidth efficiency for one BPU

case of *greedy* and *smallest* the NZ-FIFOs start getting full. This means the BPUs are not consuming the data fast enough. It can also be seen for the first three matrices presented in Table 4.2 the bandwidth efficiency does not vary much when the number of BPUs are increased. The main issue here is the number of NZ elements is quite large when compared to the size of the NZ-FIFO and even though we increase the number of BPUs to increase the rate of computation, flow control is still very much needed. A dramatic increase in bandwidth efficiency for *poli* and *t2dal* is seen for two BPUs, providing close to peak performance. The increase in performance in these two cases is due to the fact that increasing the number of BPUs and smaller number of NZ elements help these matrices to completely avoid flow control.

In case of *largest* and *random* we obtain average bandwidth efficiency of 17% when using one BPU and about 19% when using two BPUs. The reduction in bandwidth efficiency is due to the increase in index overhead and high percentage of *AllZero* bitmaps (shown in Figure 4.1). The RCG when decoding *AllZero* bitmaps is not

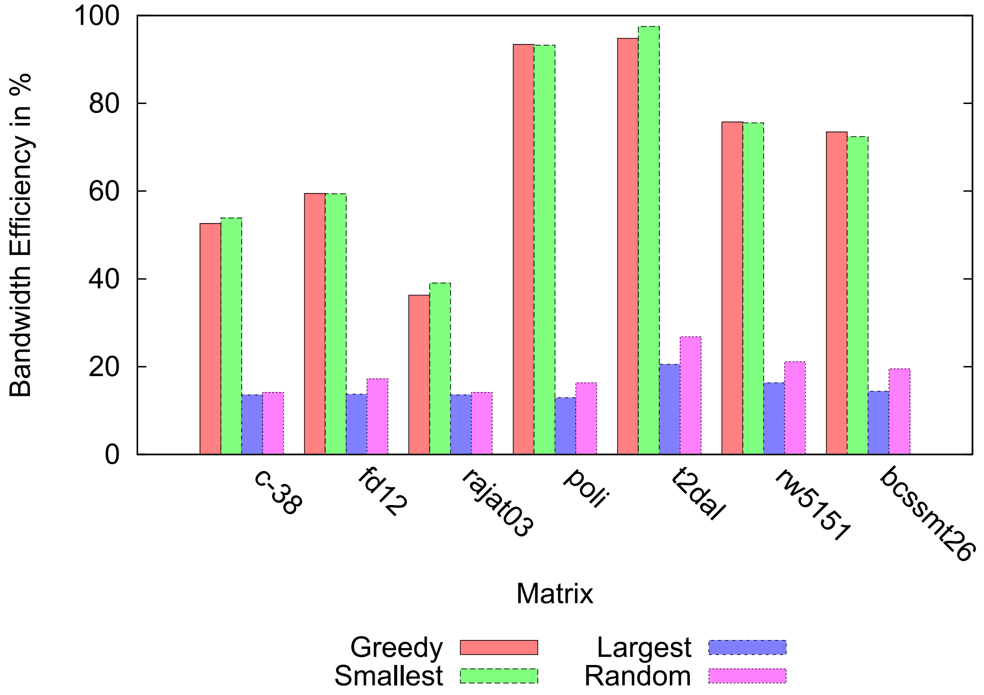

Figure 4.3: Bandwidth efficiency for two BPUs

producing any valid row-column positions, but it still has to read and decode these bitmaps for maintaining correct indexing for a block. The slower, wasteful decoding operation for *AllZero* bitmaps, along with the large index information ends up filling up the Bitmap and Block-Header FIFOs more frequently. This causes the flow-control operation to take place more often and deteriorates the bandwidth efficiency. Even when the number of BPUs are increased the bandwidth efficiency does not improve as the performance bottleneck is from the memory-subsystem side. We can increase the bandwidth efficiency for all the four cases by having deeper VDCB FIFOs which helps us minimize the need of flow control. The size of FIFOs is a trade-off we have to make to fit other components of our design and the resources available on our device.

Floating Point Performance The floating point performance for sequential design is presented in Figure 4.4 for one and two BPUs when using the *greedy* strategy. The floating point performance is closely related to the index overhead. It can be observed that the *t2dal* which has reported the highest performance for our design also has the

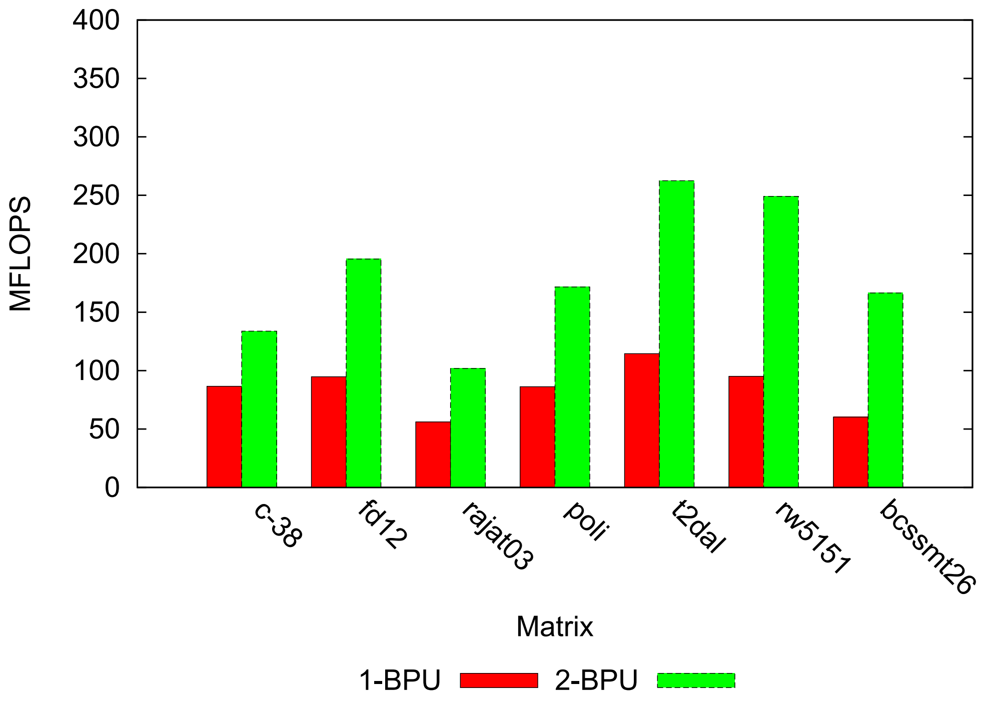

Figure 4.4: Floating point performance

lowest index overhead. However we can see that *bcssmt26* matrix performs poorly for one BPU even though it has a low index overhead. This means that index overhead is not the only factor that affects the performance of our design. The *bcssmt26* matrix is a diagonal matrix and when represented in the VDCB format all the blocks associated with the matrix are also diagonal. Although we have implemented an accumulator capable of handling new set of accumulation operation every other clock cycle, in case of diagonal blocks it is still not sufficient. The one clock cycle bubble in partial result accumulator introduced when the rows are switched impacts the performance negatively. It can be seen from the Figure 4.4 that the performance improves significantly for all the matrices when two BPUs are used, effectively masking the compulsory latency of accumulation operation. The floating point performance for the other three strategies follows the same trend of being related to the index-overhead. For brevity we have reported results with the *greedy* strategy only.

Table 4.3: Sustained performance for sequential SMVM operation

Platform	Matrix Structure	Storage Format	% of Peak Computation	% of Peak Bandwidth
Virtex-4	Sparse	VDCB (Greedy 1-BPU)	47%	58%
	Sparse	VDCB (Greedy 2-BPU)	47%	70%
Virtex-4	Dense	VDCB (Greedy 1-BPU)	53%	68%
	Dense	VDCB (Greedy 2-BPU)	53%	93%
Santa Rosa	Dense	BCSR	30.2%	49.9%
Barcelona	Dense	BCSR	14.7%	50.6%
Clovertown	Dense	BCSR	6.3%	22.1%
Victoria Falls	Dense	BCSR	4.3%	0.9%
Cell Blade	Dense	BCOO	18.6%	62.9%

Overall Performance Evaluation of Sequential Design The discussion we have presented till this point has emphasized the importance of maximum utilization of memory subsystem for SMVM. In terms of memory subsystem an ideal case would be a dense matrix represented in sparse format. In case of general purpose processors it will help in utilizing the memory hierarchies and for the VDCB format it will result in blocks which are completely dense. This will ensure no wasteful decoding operations and maximum throughput from our memory subsystem. Thus the performance of dense matrix in sparse format is an upper threshold for the SMVM operation. In Table 4.3 we compare our implementation with single core performance on various multicore processors reported in [13]. The performance reported in [13] is for SMVM kernel highly tuned for exploiting the underlying architecture characteristics. The dense matrix used for testing in Table 4.3 is an 8Kx8K dense matrix represented in the VDCB format.

4.1.1.3 Block Sizes

It might be suitable to assume at this point to only use fixed block sizes of 8×8 and not even use *greedy* approach. However this strategy will not be beneficial if we use Finite Element Method (FEM) matrices which have dense chunks of NZ elements. Consider a dense block of 64×64 present in a FEM matrix. This block if expressed simply as 64×64 block will have an index requirement of 65 double-words. If this block was divided into 8×8 dense blocks it will result in 128 double-words for indexing. This increase in index requirement will impact performance negatively as seen before.

4.1.1.4 Parallel Hardware Design Performance Evaluation

The parallel design also uses the Xilinx-ML410 boards for implementing the Head Node and Worker Node and the design components are developed in VHDL. Each worker node consists of two BPUs where each BPU is capable of providing 200MFLOPs each and a worker node in all will be able to provide a total floating point performance of 400MFLOPS. The design is evaluated for the following metrics:

- Communication time for the block rows from the head node to the worker node
- Floating point performance
- Speedup
- Scalability

Block Row Communication The block rows can be provided to the worker node via Round-Robin approach or a Super Block approach. We measure the time taken by the head node to provide all the block rows of matrix A to a system consisting of two and four worker nodes respectively when the two approaches are used. The measurements are shown in Figure 4.5 and indicate the total time it takes to communicate all the block rows from the head node to worker nodes. In the Figure 4.5 RR is used to abbreviate Round-Robin and SB is used for Super Block respectively.

It can be seen that the communication time for Round-Robin approach is significantly higher in all cases when compared to the Super Block approach. The main reason for this is the setup latency of the AIREN router. A latency of 0.8μs is added by the AIREN route to the total communication time when performing a transaction between the two FPGA nodes. This latency although small has a significant impact on overall communication time in Round-Robin approach. In the Round-Robin

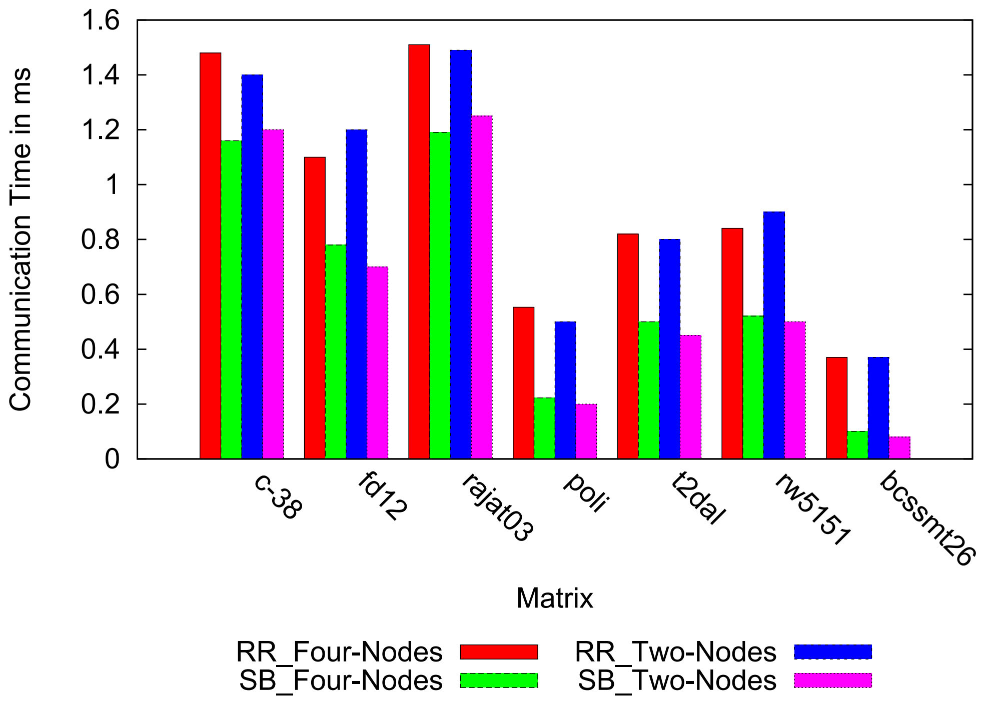

Figure 4.5: Communication time for Block Rows

approach as individual block row is only provided to each node making the size of transaction is significantly small. The total latency added by the router to the total communication time is dependent on the number of transactions carried out by the router. In the case of Round-Robin approach the number of transactions are equal to the number of block rows present within the matrix. The total setup latency for Round-Robin approach can be given by Equation 4.2:

$$Setup\_latency_{RR} = \beta \times setup\ latency \tag{4.2}$$

where *setup latency* is the setup latency for AIREN Interface (0.8μs), *beta* is the total number of block rows that are present and $Setup\_latency_{RR}$ represents the total setup latency for Round-Robin approach.

The Super Block approach also requires a setup latency but in case of Super Block approach the number of transactions is equal to the number of worker nodes that are present (as a collection of block rows is sent to each worker node, Equation 3.4). Thus the total setup latency that will be added to the communication time is given by:

$$Setup\_latency_{SB} = w \times setup\ latency \tag{4.3}$$

where w is the total number of worker nodes and $Setup\_latency_{SB}$ represents the total setup latency for Super Block Approach. It can be seen from Equation 4.2 and 4.3 that setup latency is going to be considerably less in case of Super Block approach. Thus reducing communication time for Super Block Approach even though a significantly larger amount of data is transmitted in case of Super Block approach when compared to the Round-Robin approach.

We can see from the Figure 4.5 that varying the number of worker node does not impact the communication time significantly in both the block row communication strategies. This is an expected outcome because in case of Round-Robin approach where the communication time is dependent on the number of block rows and it is not impacted by the number of worker nodes. In case of Super Block approach where the number of super blocks are dependent on the number of worker nodes selected (Equation 3.2), there is still not a lot of change to the overall communication time. This is due to the fact that the aggregate transaction size (the total request size for all the Super Blocks) remains almost constant. If we change the number of worker nodes from two to four the size of a Super Block required by each worker node will decrease proportionally, as each Super Block will now comprise of lesser number of block rows. This in turn reduces the transaction size when the number of worker nodes are increased. The inverse relationship between the transaction size (size of a Super Block) and the number of worker nodes helps in keeping the communication time fairly constant. We are also not indicating the communication time for a system with one worker node as it is very close to the timing measurements of two and four worker nodes.

It can be seen from the Figure 4.5 that the matrices with higher communication time also had a higher index overhead (Figure 4.1). The higher index overhead increased the overall transaction size (due to more index information present) and resulted in higher communication time.

Floating Point Performance

Impact of Reading Vector $\vec{x}$ The head node provides the block rows to the worker nodes for the computation; but it cannot start immediately as vector $\vec{x}$ is stored in the main-memory and it has to be read into the vector BRAM to begin the computation. It is imperative to consider the time it takes to read the vector$\vec{x}$ from main memory into the BRAM as the performance is going to be dependent on how fast the vector$\vec{x}$ can be read from the main memory. This gives us two options for reading vector$\vec{x}$.

- The worker node can only start reading the vector $\vec{x}$ once the head node starts providing the block row of matrix A. The worker node can examine the block-headers of the block rows and then based on that fetch a portion of vector $\vec{x}$ which is going to be used for computation. This is know as *Vect-PostRead*.

- As soon as the head node starts providing the block row to the first worker node, all the worker nodes can start reading the entire vector $\vec{x}$ from its main-memory. Basically in this setup as soon as the head node starts the block row communication all the worker nodes will know that eventually the SMVM operation is going to take place and hence will already prefetch the entire vector $\vec{x}$. This is known as *Vect-SimRead*.

The *Vect-PostRead* is beneficial in case of matrices where the blocks within the block rows are concentrated only in one region and hence only a portion of vector $\vec{x}$ is required for multiplication. In this case a small-sized read request can be made to the main-memory and the computation can start as soon as that portion of vector

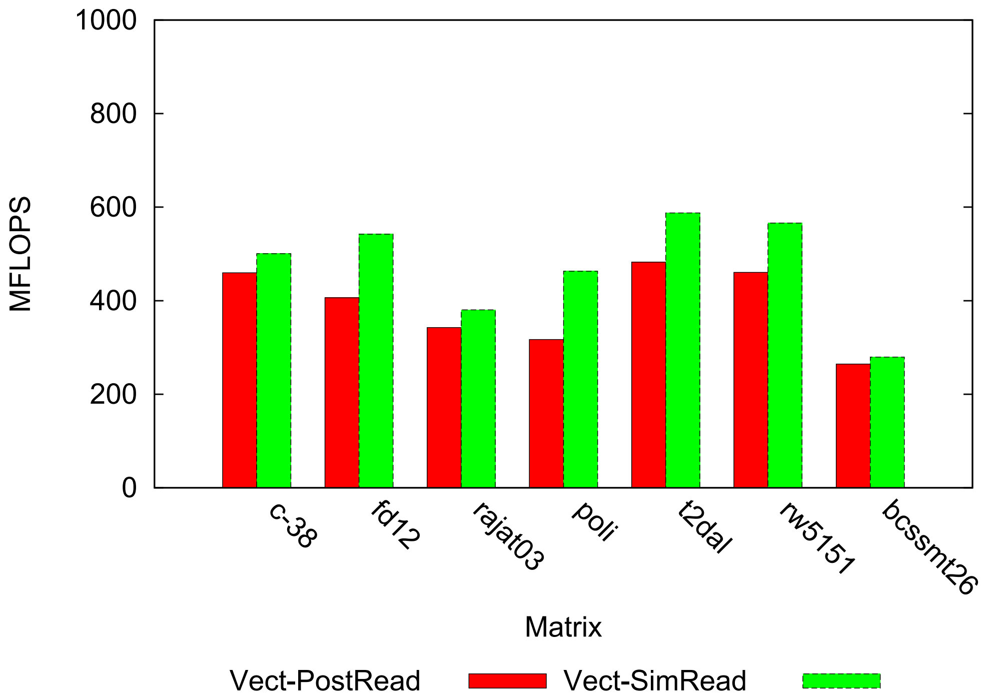

Figure 4.6: Floating point performance

has been read in.

The *Vect-SimRead* is effective in terms of overall speed of operation as all the worker nodes start fetching the vector $\vec{x}$ from their main-memory simultaneously as soon as the communication by the head node begins. This means by the time the block rows reaches a particular worker node the vector $\vec{x}$ is already present and the computation can start immediately.

Based on these two methods of reading the vector $\vec{x}$ we have measured the floating-point performance of a system consisting of four worker nodes which is presented in Figure 4.6. The measurements presented in Figure 4.6 use the Super Block approach due to the lower communication time when compared to the Round Robin approach.

It can be seen from the Figure 4.6 that the floating-point performance is better for *Vect-SimRead* when compared to *Vect-PostRead* in all the cases other than *bc-ssmt26*; where it is almost comparable. The main reason is the distribution of the blocks amongst the block rows of the matrices and also the usage of the Super Block approach. Although in an individual block row the blocks might be concentrated in one region when we have a collection of block rows (Super Block Approach) the distribution of the blocks varies widely. This results in a larger portion of vector $\vec{x}$ which is needed for multiplication and this in turn increases the request size and hence the time it takes to the read the portion from the main-memory. In case of *bcssmt26* the matrix is a diagonal matrix and the VDCB encoding results in blocks which are diagonal as well. When used with the Super Block approach these diagonal blocks only multiply with a very small portion of the vector $\vec{x}$ resulting in a smaller read request sizes and hence faster read times.

Increasing the Number of Worker Nodes We also wanted to see the impact of varying the number of worker nodes on the overall floating point performance. Ideally, the performance should increase with increased number of worker nodes because of the increased BPUs available for computation and in turn an increased amount of block-level parallelism. We performed the measurements provided in Figure 4.7 using one, two and four worker nodes. The experimental setup also employed Super Block Approach (lower communication time) and Vect-SimRead for providing higher speed of operation.

The floating point performance is measured when different number of worker nodes are used. It can be seen that the performance when one worker node is used is very close to the sequential design performance. When the worker node is only one the entire matrix A will be equivalent to one super block and all of that is provided to the worker node. The difference between a parallel design with one worker node and sequential design is how the matrix A is made available. The worker node receives matrix A from the LL Interface from the head node and in case of the sequential design the matrix A is provided by the main-memory via the NPI Channel. The number of blocks over which the single worker node has to operate is the same as the sequential design and in turn the floating point performance remains close to the

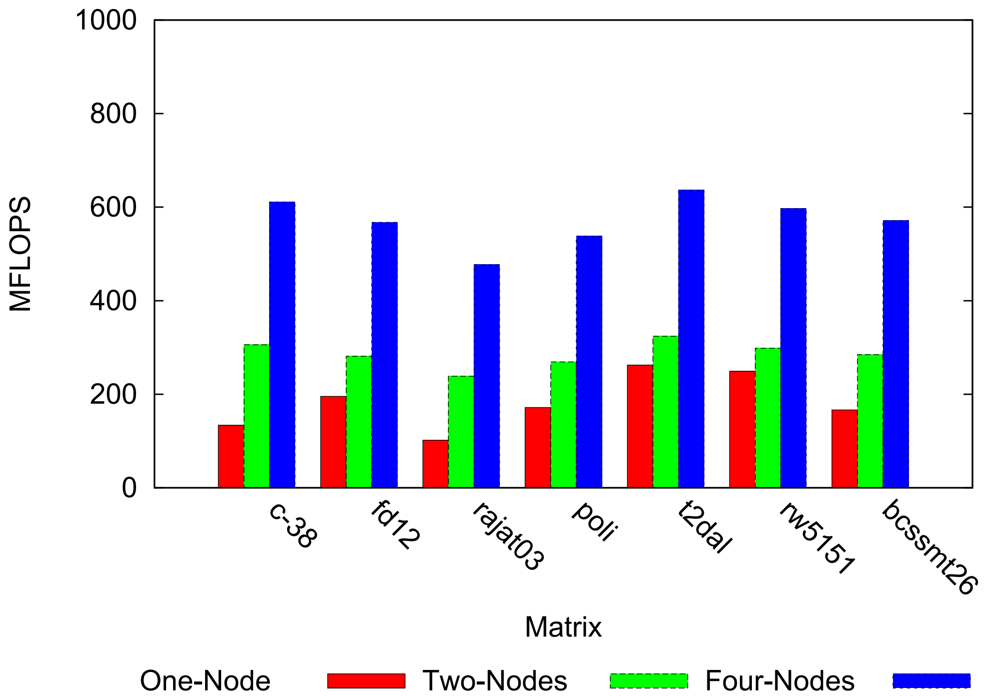

Figure 4.7: Floating point performance on increasing worker nodes

sequential performance.

If we refer to the floating point performance in case of two and four worker nodes we can see the performance is very closely related to the communication time of the block rows in Super Block approach (Figure 4.5). If we refer to the hardware design presented in Section 3.2.2.1 we know that the DMU,RCG and BPUs present on a worker node are operating together and as soon as the very first block is received via the LL Interface the decoding and computation operation begins. Thus a longer communication time translates into a longer computation time to provide the final resultant vector $\vec{y}$, resulting in a lower floating point performance. The communication time is largely dependent upon the transaction size and the transaction size is based on the amount of data present within a block row. The size of a block row depends upon the distribution of the NZ elements within a matrix and the block-size selection.

Speedup We measure the speedup as improvement in floating-point performance as the number of worker nodes are increased when compared to the sequential implementation. The system uses *Super Block* approach for block row communication and *Vect-SimRead* for reading vector $\vec{x}$.

The results are presented in Figure 4.8. Ideally with a system consisting of w worker nodes should provide $w\times$ speedup. But this is not the case as seen from Figure 4.8. In case of a single worker node the performance is very close to the sequential implementation as discussed earlier in Section 4.1.1.4. In case of two worker nodes an average speed up of 1.4x is obtained and in case of four worker nodes a speedup of 2.72x is obtained. We are not indicating the speedup for a one worker node system as the performance is very closely related to the sequential implementation. The main reason is the communication time needed for providing the block rows of matrix A to the worker nodes and the fact that the block rows cannot be provided to all the worker nodes in parallel. The AIREN_Send interface is similar to an MPI_ Send interface and at any given time only one woke node can be provided with the block rows. Only after finishing the transaction with the first worker node the second worker node can be provided with the block rows. If the speed of communication operation can be improved then we will be able to obtain higher speedups.

An interesting observation is the high speedup for *c-38* and *rajat03* when increasing the number of worker nodes. Both the matrices performed poorly on the sequential implementation. The higher speedup is obtained due to the fact that only block rows of matrix A are provided to the worker nodes. As the amount of data being provided to the worker node (in terms of matrix A) is substantially less than the sequential implementation, the associated flow-control operations do not take place for these two matrices which were happening before for the sequential implementation. This results in improving the performance of the parallel implementation substantially over the sequential implementation.

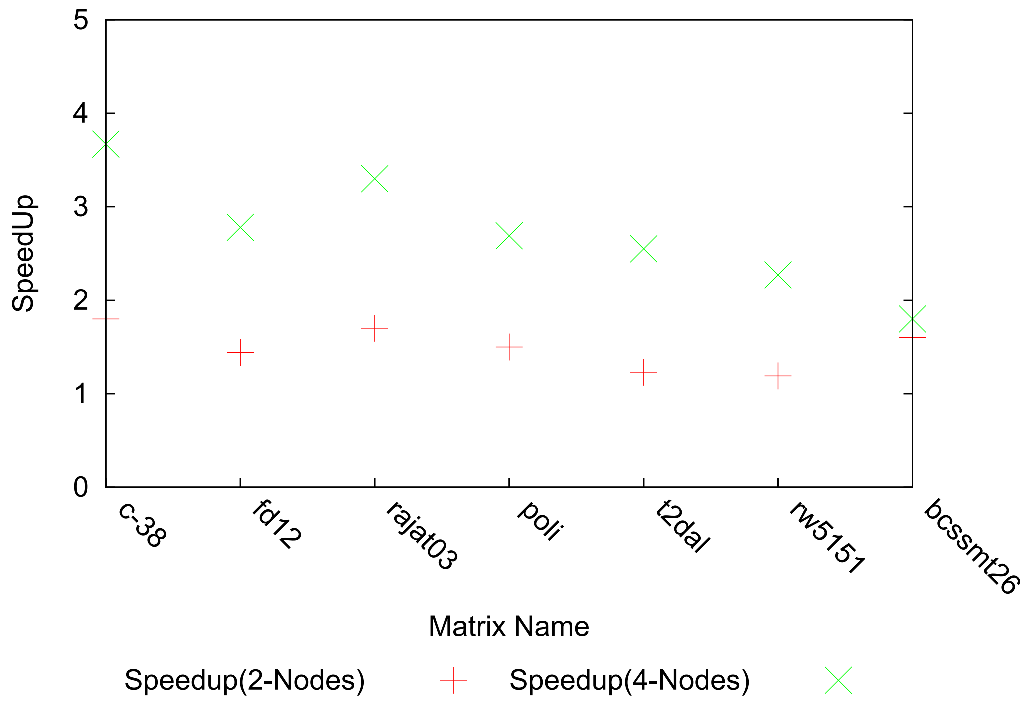

Figure 4.8: Speedup

Scalability We can see from Figure 4.8 as the number of worker nodes increase the speedup also improves for our design. This is because as more worker nodes are added the amount of block-level parallelism also improves. Another important aspect to be considered here is as the number of worker nodes increased the blocking time for a worker node reduces. The blocking time for a node can be considered as the amount of time a node has to wait to receive a block row from a head node while the head node is communicating the block row to another worker node. As discussed earlier in Section 4.1.1.4; as the number of worker nodes increase the transaction size decreases for the Super Block approach. This means a worker node can receive the block rows quicker and the number of blocks it has to compute over is also lesser, thereby decreasing the chances of flow control and increasing the performance. Hence increasing the number of worker nodes will increase the floating point performance.

Overall Performance Evaluation of Parallel Design We compare the performance of our parallel design against the one socket, all cores, all threads performance

Table 4.4: Sustained performance for parallel SMVM implementation

Platform	Matrix Structure	Storage Format	% of Peak Computation	% of Peak Bandwidth
Virtex-4	Sparse	VDCB	35.64%	52%
Virtex-4	Dense	VDCB	38.2%	57%
Santa Rosa	Dense	BCSR	21.2%	69.8%
Barcelona	Dense	BCSR	6.3%	87.4%
Clovertown	Dense	BCSR	3.5%	56.3%
Victoria Falls	Dense	BCSR	42.3%	74.1%
Cell Blade	Dense	BCOO	40.7%	96.6%

of multicore platforms presented in [13]. We consider the head node and worker node combination of the parallel implementation loosely analogous to the one socket all core implementation for multicore platform. We also consider the BPUs as equivalent of threads which are used in the multicore platforms to perform computation in parallel. We can see from Table 4.4 that the FPGA floating point performance is comparatively better than some platforms (Santa Rose, Barcelona, Clovertown) but is still not close to the performance offered by the Victoria Falls and Cell Blade. This is quite different that the sequential performance presented in Table 4.3 where the FPGA implementation out performed the single core, single thread multicore implementations. The main reason for this is the number of BPUs supported by the FPGA device are quite low when compared to the number of threads that can be spawned in parallel by the multicore platforms. Another issue is the utilization of the peak bandwidth. We have calculated the bandwidth in the Super Block approach using the time it takes to provide all the block rows to all the worker nodes by the head node. The NPI Interface used in the sequential approach for providing the matrix A is a 64-bit interface when compared to the 32-bit LL Interface used in the parallel approach. This reduces the bandwidth significantly in parallel implementation and impacts the floating point performance negatively.

Table 4.5: Hardware resource utilization for SMVM operation

Implementation	**BRAM**	**Slice**	**4 Input LUT**
Sequential	84%	47%	50%
Worker Node	89%	53%	52%
Head Node	12%	8%	17%

4.1.1.5 Resource Utilization for SMVM Operation

We report the FPGA resource utilization for the sequential and parallel implementation of SMVM operation in Table 4.5. The FPGA device used is a Virtex-4 XC4VFX60 consisting of a PowerPC processor. As the parallel design consists of a Head Node and Worker Node we have reported the individual resource utilization for the two nodes. The resource utilization reported in Table 4.5 consists of the BRAM utilization, total slice utilization and 4 input LUT utilization. It can be seen from the Table 4.5 that the BRAM utilization is extremely high in sequential and worker node implementation. This is due to the presence of larger number of FIFOs (implemented using the BRAMs) and BRAMs needed for implementing the various design components like the CMI, RCG and BPUs. The high BRAM utilization for sequential and worker node implementation also limits the number of BPUs which can be made available on these implementations and hence limits the amount of block-level parallelism. In case of the head node as it is only providing the matrices from its main memory and not implementing the various functionalities of the worker node the BRAM utilization results fairly low.

4.1.2 Results and Analysis for Sparse Matrix Matrix Multiplication

We selected sparse matrices which represented graphs (both directed and undirected) from University of Florida Matrix Market place, we used these matrices as matrix A. The information of test matrices is provided in Table 4.2. We generated matrix B synthetically in software and considered four cases for its sparsity as shown in Table 4.2.

Table 4.6: Characteristics of test matrices

Matrix Name	Matrix A		Matrix B	
	Dimension	NNZ_A	NNZ_B	Strategy
gre_512	512×512	2192	$1 \times NNZ_A$	Similar-NZ
delaunay_n10	1024×1024	3056	$0.75 \times NNZ_A$	Reduce-NZ-25
delaunay_n11	2048×2048	6127	$0.5 \times NNZ_A$	Reduce-NZ-50
delaunay_n12	4096×4096	12264	$0.25 \times NNZ_A$	Reduce-NZ-75

Table 4.7: Frequency of operation for BFU and PE

Block Size	Operating Frequency (BFU) (MHz)	Operating Frequency (PE) (MHz)
8×8	200	100
16×16	75	50

4.1.2.1 Block Sizes

The selection of constant block size $b \times b$ is a critical component for the performance of our design. We measure the frequency at which these units can operate for various block sizes. It can be seen from the Table 4.7 that the operating frequencies of BFU and PE deteriorates as the block size increases. The design with block size greater than 16×16 was not synthesized successfully by the tools due to resource limitation. Based on the observations made from the results reported in Table 4.7 we can see that block sizes of 8×8 gave the best operating frequency for BFU and PEs. We ran all our subsequent tests using the block size of 8×8.

4.1.2.2 Effect of Sparsity

As discussed earlier in Section 3.5.1, the FIFO holding matrix B will read all of its entries to determine blocks needed for multiplication. If we assume a case where a block-row of matrix A is completely dense then all of the blocks present within that block row will multiply with all the blocks present within the FIFO holding matrix B (Property 1). This represents the best case scenario in terms of the SMMM operation as all the elements present within the FIFO get used.

Now if we assume that blocks present in block-row of matrix A do not multiply with any of the blocks present within the matrix B; then this will represent the worst case. This means the entire matrix B has to be read out from the FIFO for the purpose of comparison to see if any of the blocks present in matrix A and B satisfy Property 1. Thus no useful operation will take place and it will delay the beginning of computation of new block row. The sparser the matrix B is going to be, lesser number of blocks are going to be associated with it. This is because the VDCB format only stores NZ elements within the blocks (along with one block header and the corresponding bitmaps) and if the number of NZ elements is less the number of blocks needed for them is also going to be lower. This will result in quicker comparison as lesser number of blocks have to be examined. Also the FIFO can be read out much faster as it will be storing less number of elements due to reduction in the number of blocks and its corresponding elements. This particular hypothesis is validated by our test measurements shown in Figure 4.9. It can be seen that the performance for *Reduce-NZ-75* is best in all the four cases. This is because the *Reduce-NZ-75* is the most sparse of the four synthetically generated matrix B and thus results in lowest number of blocks.

We use execution time as a metric to evaluate the computation performance. The main reason for choosing the total computation time and not MFLOPS (the metric used for the SMVM Operation) is the fact that it is difficult to estimate the number of floating point operations that take place for the SMMM operation; unlike the SMVM operation, where $2 \times NZ$ number of floating point operations are going to take place for a matrix. The estimation of number of floating point operations in case of the SMMM operation requires an extensive offline analysis of both matrices A and B to determine the NZ distribution and then a second analysis once the matrices are encoded in the VDCB format due to this reason we avoided the FLOP measurement and used the overall execution time for performance evaluation.

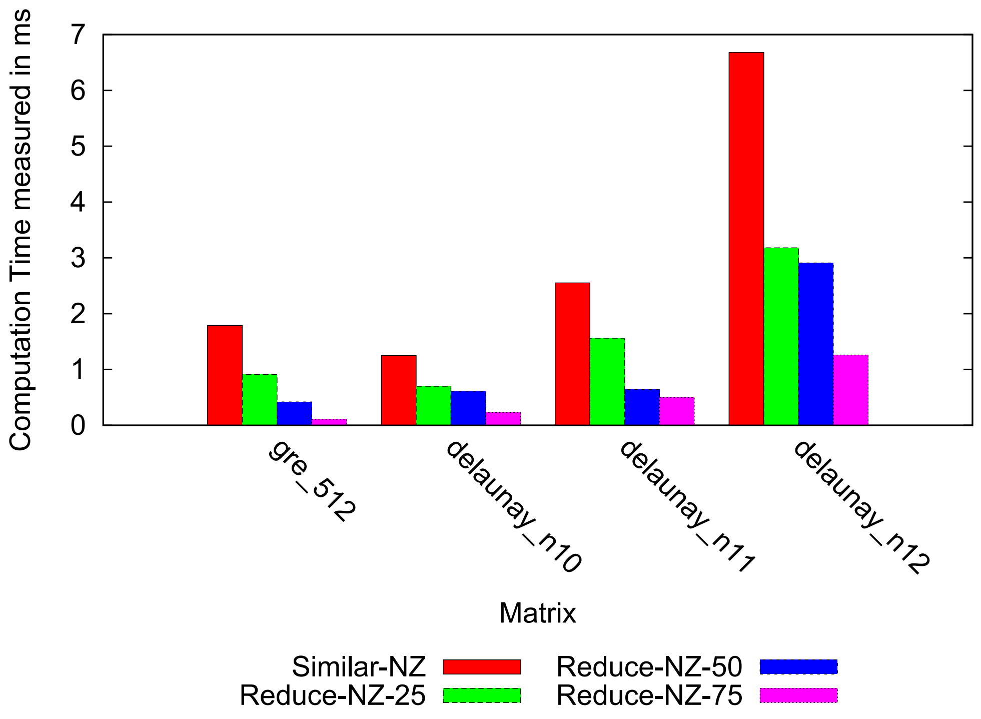

Figure 4.9: Effect of sparsity on computation time

4.1.2.3 Bandwidth Efficiency

In case of the SMMM operation we have implemented; we store the matrix B in FIFO (Section 3.5.1) and for every block row calculation of result matrix C we fetch the entire matrix B from the memory. This becomes the dominant memory transaction because when compared to matrix A where only a block-row has to be fetched, in case of matrix B the entire matrix has to be fetched again (because one complete block row computation of C causes all elements of matrix B to be read out from the FIFO).

We can store the entire matrix B also on the BRAMs (on-chip memory) and avoid this repetitive transaction. But the problem is the lack of available on-chip resources (BRAMs). In case of matrix A we are only fetching a block-row so it is useful to store it in BRAM as it is comparatively smaller and is needed for computation with all the block-columns of matrix B. As the size of matrix B can vary depending upon the number of NZ elements and their distribution it is better to store matrix B in the

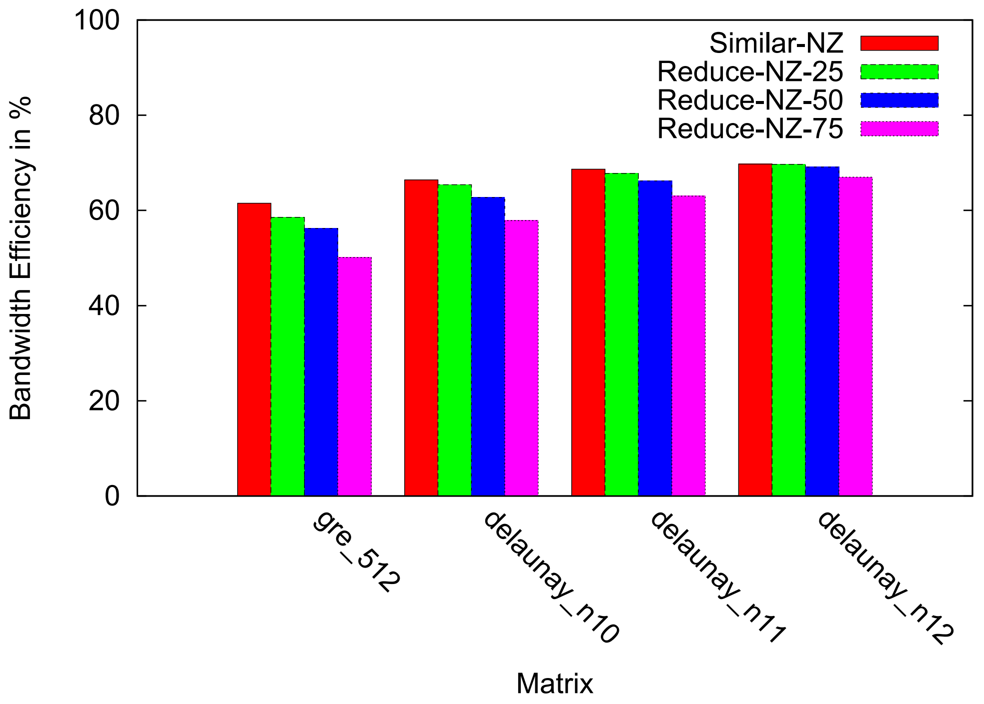

Figure 4.10: Bandwidth efficiency of matrix B

off-chip memory and stream it into the on-chip FIFOs. The bandwidth-efficiency for matrix B over all the transactions needed to finish the SMMM operation is shown in Figure 4.10.

It is interesting to note that as the number of NZ elements are reduced the bandwidth efficiency decreases as can be seen from the Figure 4.10. This particular issue is most pronounced in case of matrix gre_512 which has dimension of 512×512. As the matrix size reduces the corresponding matrix in the VDCB encoded format also reduces in size. This results in smaller request size for matrix B. This should ideally give us higher bandwidth efficiency, but the the problem is we are performing these requests multiple times. It takes around 20 clock-cycles (in 200MHz clock domain) to setup the request and start the transaction. In case of smaller requests which are made multiple times the set-up time overhead starts affecting the overall bandwidth efficiency. As the size of the matrix encoded in the VDCB format depends on the number of NZ elements; it can be seen that for gre_512 the bandwidth efficiency decreases with the reduction in the number of NZ elements (Figure 4.10). In case of delaunay_n12 the bandwidth efficiency reduces slightly as the number of NZ elements is reduced (Figure 4.10). This is because even after the reduction in number of NZ elements the request sizes are still large enough to amortize the set-up time overhead.

We can see from Figure 4.10 that the bandwidth efficiency is on an average is above 63%. This is fairly a good number given the fact that generally sparse matrix operations have bandwidth efficiency less than 30% [2]. The improvement in bandwidth efficiency in our case is due to complete elimination of indirect memory accesses because of using the VDCB format with the customized memory hierarchy. The indirect memory accesses create additional memory subsystem traffic in form of load operations and reduce the memory bandwidth-efficiency.

4.1.2.4 Scalability

The parallel hardware design used to implement the SMMM operation has been presented in Section 3.5.4. We use the results from the SMVM operation block row communication to select the communication method for block rows of matrix A. As can be seen from Figure 4.5 the Super Block approach provides lower communication time and hence improves the overall performance. In the scalability study we do not vary the sparsity of matrix B and only use the Similar-NZ strategy for our measurements. We are doing this to keep our results concise and focus solely on the scalability of our hardware design. We also use the strategy similar to *Vect-SimRead* to read matrix B due to the performance gains provided. As soon as the block rows of matrix A are started to be provided all the worker nodes start reading matrix B.

The results are shown in Figure 4.11, we vary the number of worker nodes from one to four to measure the computation time. We can see from Figure 4.11 that the performance for a single node is very close to the sequential performance. In case of a single node we are providing the block rows of matrix A via the AIREN router network instead of the NPI channel. As the computation of the SMMM operation

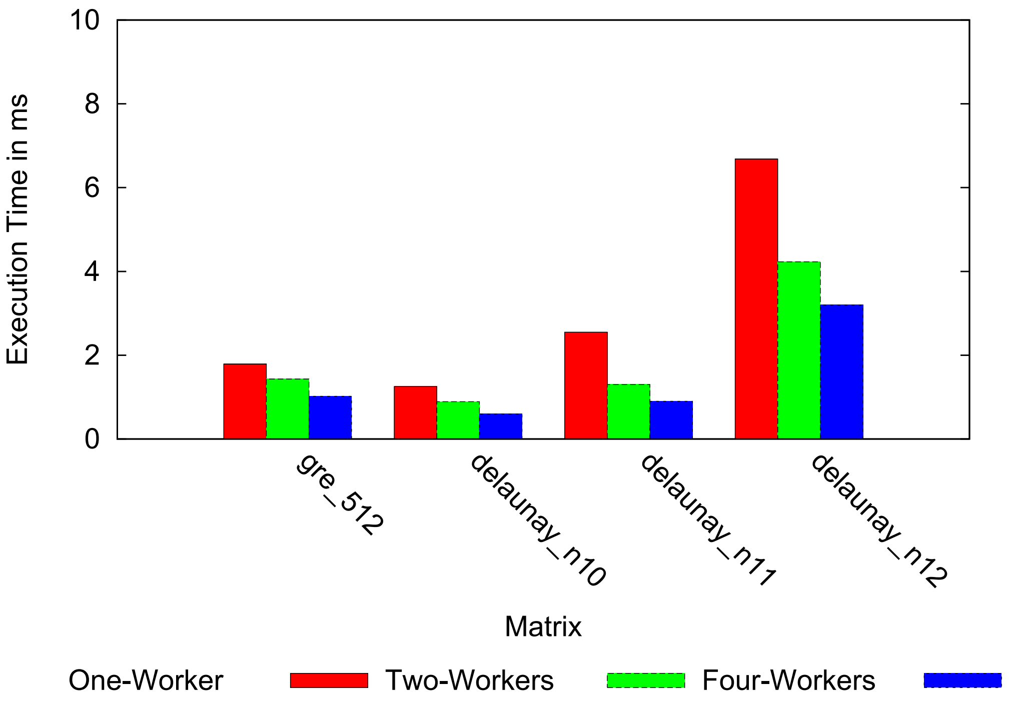

Figure 4.11: Computation time measurement for parallel hardware design

on the worker nodes is completely pipelined the moment very first datum is received the computation begins and the computation core remains oblivious to the mode of delivering the block rows of matrix A. In case of a single worker node the number of block rows on which the node is going to perform the SMMM operation is same as that of the sequential design thus the execution time remains almost identical between the two.

We also evaluate the speedup of our parallel hardware design by comparing the computation time improvement when the number of worker nodes are increased. The speedup measurements are shown in Figure 4.12.

On an average with two worker nodes we achieve a 1.3X performance improvement over sequential design and about 2.2X improvement with four nodes. We are not close to the ideal speedup due to the communication time of block rows of matrix A which dominates the overall computation time. If we are able to minimize the communication time of block rows of matrix A we will able to achieve close to ideal

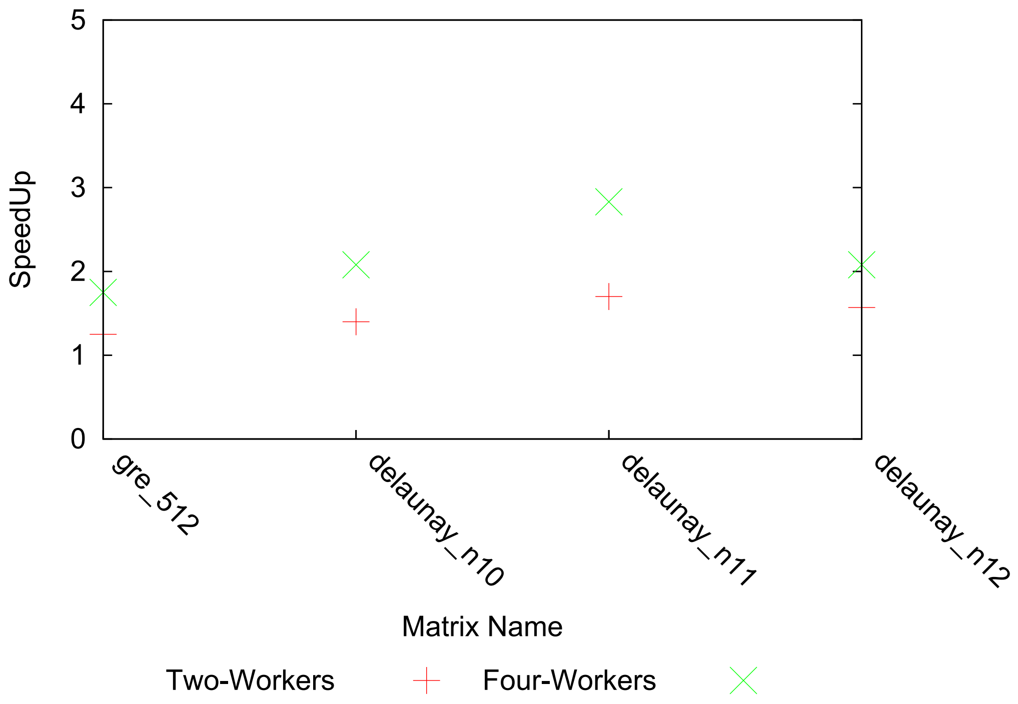

Figure 4.12: Speedup for parallel hardware design

speedup.

4.1.2.5 Resource Utilization for SMMM Operation

We report the FPGA resource utilization for the sequential and parallel implementation of SMMM operation in Table 4.8. The FPGA device used is a Virtex-4 XC4VFX60 consisting of a PowerPC processor. As the parallel design consists of a Head Node and Worker Node we have reported the individual resource utilization for the two nodes. The resource utilization reported in Table 4.8 consists of the BRAM utilization, total slice utilization and 4 input LUT utilization. It can be seen that the slice utilization in case of sequential implementation and worker node implementation is quite high. This is because the SMMM operation is performed on the sequential and worker nodes only. The block selection units and block fill units used for the purpose of the SMMM operation introduce a large amount of logic complexity in the form of FSMs and lookup tables used to perform the block selection and generation of the index information to perform multiplication. On the FPGA device these FSMs

Table 4.8: Hardware resource utilization for SMMM operation

Implementation	**BRAM**	**Slice**	**4 Input LUT**
Sequential	60%	77%	46%
Worker Node	67%	78%	55%
Head Node	12%	10%	19%

and lookup tables get implemented on the logic slices thereby increasing the slice utilization.

4.2 Performance Analysis of Software Only Approach

The software design was implemented on a conventional server consisting of Intel Xeon Phi processor and developed in "C"programming language. The parallel implementation has been developed using the Message Passing Interface (MPI) APIs. The floating point operations have been performed by using the *SoftFloat* library [27] which has performance tuned implementation of IEEE 754 floating point operations (both for single and double precision).

A particularly difficult challenge is presented in the form of implementing performance instrumentation of the software code without impacting the overall performance of the design. We want to measure the performance of the software design without interfering with the matrix operations. In order to do so we use simple counters along with the clock frequency of the processor to calculate the execution time. This provides a non-intrusive instrumentation for the purpose of performance evaluation.

4.2.1 Performance Evaluation of Software Only SMVM Operation

We evaluate the performance of the SMVM operation for the set of matrices presented in Table 4.2. The vector $\vec{x}$ is a dense vector generated in software. The IEEE 754 double precision floating point operations are implemented using the *SoftFloat* library which has processor compatible implementations for floating point operations with specific architecture optimizations. The software implementation of the code uses the *Greedy* strategy for the encoded VDCB format and uses the algorithms presented in 3 and 4 for sequential and parallel implementation of the operation. The floating point operations are performed using the *SoftFloat* library [27].

4.2.1.1 Floating Point Performance

Sequential Software Design The floating point performance result for sequential implementation are shown in Figure 4.13. It can be seen from Figure 4.13 that the performance trends are similar to the corresponding implementation sequential implementation in the Hardware/Software Co-Design Approach. This further validates our analysis discussed in Section 4.1.1.2 that the matrices with lower index overhead will have a better floating point performance due to reduction in unnecessary load operations. We had also discussed in Section 4.1.1.2 that although *bcssmt26* had low index overhead the floating point performance was low due to the one cycle bubble incurred in the accumulation operation during a row change. But in case of software only implementation *bcssmt26* matrix still had a low performance even though the accumulation operation performed using *SoftFloat* library does not introduce a one cycle bubble. The main reason behind the low floating point operation of the *bcssmt26* matrix is due to its diagonal NZ distribution. The *bcssmt26* matrix uses each element of vector $\vec{x}$ only once due to its diagonal nature and hence no temporal locality on vector $\vec{x}$ or $\vec{y}$ can be utilized. Thus resulting in lower memory bandwidth utilization even though *bcssmt26* matrix has a low index overhead.

Parallel Software Design The floating point performance results in case of parallel design was measured for four MPI workers nodes. The floating point performance for all the test matrices for parallel design is presented in Figure 4.14. The performance trends between the sequential and parallel designs are quite similar. The performance improvement can be seen for all matrices in case of parallel implementation, but matrices like *t2dal*, *rw5151* and *poli* which reported highest performance for sequential implement ion only have performance improvement ranging from 1.3X to 1.7X. An interesting case to note is the performance improvement for matrices *c-38*

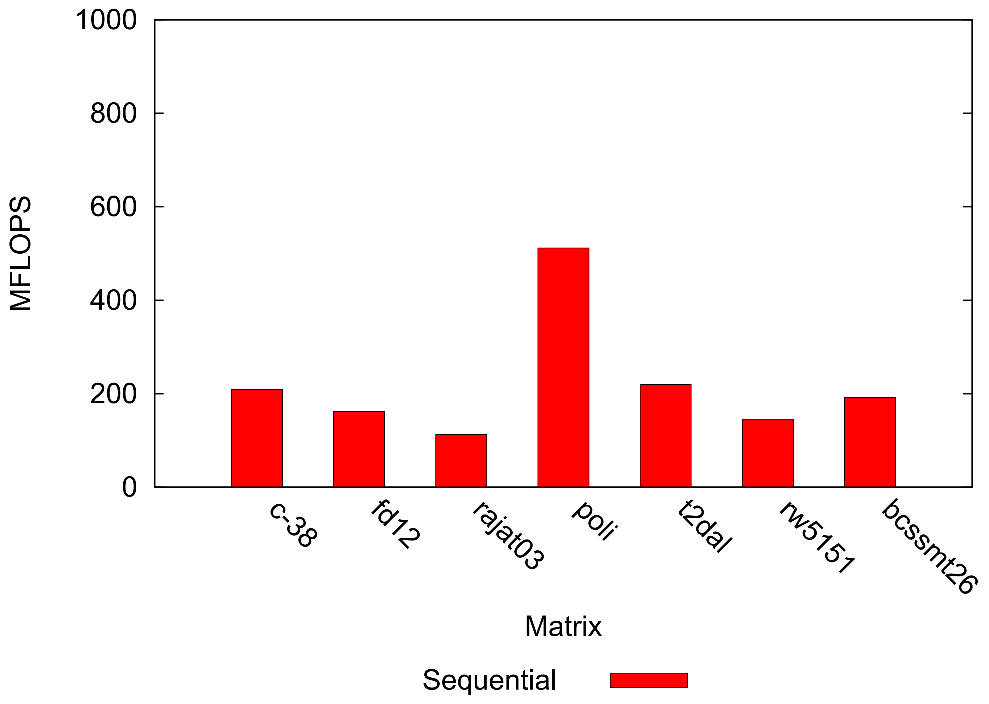

Figure 4.13: Floating point performance of sequential software design

and *fd12*. Both the matrices had a low sequential performance but in case of parallel implementation the performance improves quite substantially. These two matrices have a performance improvement of about 2.4X. This variation in performance can be attributed to the distribution of the NZ elements across the block rows. In case of matrices *c-38* and *fd12* the block rows consists of blocks that span across the same set of columns resulting in reusability of the elements of vector $\vec{x}$ and $\vec{y}$. This results in a higher memory bandwidth utilization for these two matrices in parallel software implementation. Another factor to take into account is that once a matrix has been divided into block rows the distribution of the blocks changes and index overhead for the Super Block received by each worker node will change. In case of *c-38*and *fd12* the index overhead for each worker node reduces significantly due to the clustered distribution of the NZ when decomposition into block rows takes place. This helps in improving the floating point performance in parallel implementation significantly.

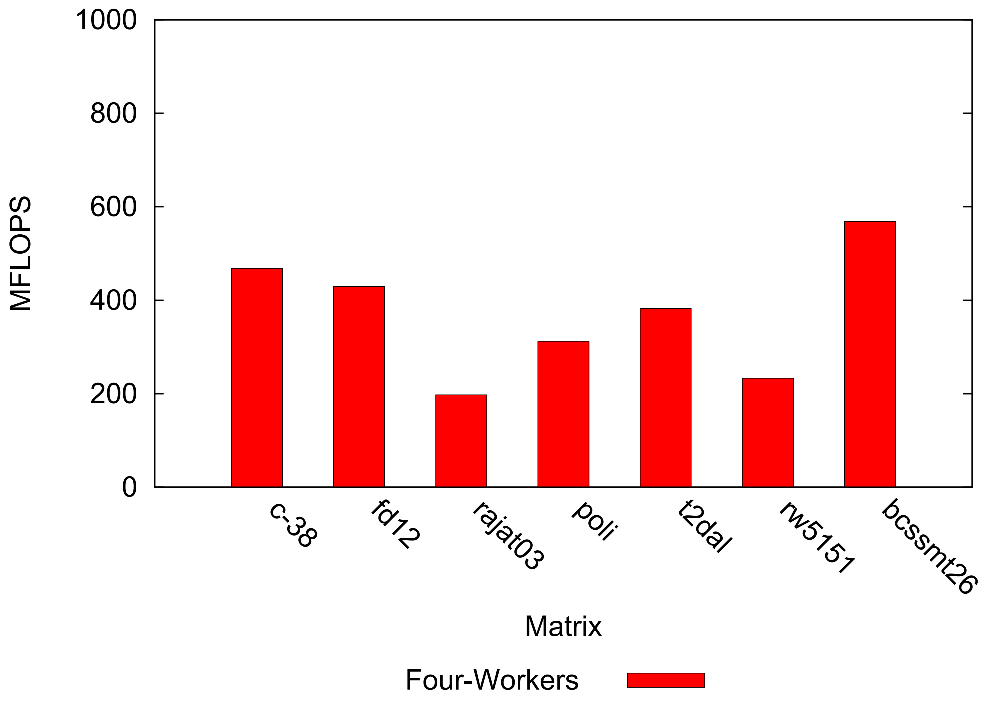

Figure 4.14: Floating point performance for parallel software Design

4.2.2 Performance Evaluation of Software Only SMMM Operation

We evaluate the performance of the SMMM operation using the Software Only approach for matrices presented in Table 4.6. The floating point operations are performed using the *SoftFloat* library. The software uses Algorithms 9 and 10 for sequential and parallel implementations respectively.

4.2.2.1 Computation Performance

The computation performance of the software only approach is calculated by introducing counters and using the processor frequency to estimate the total execution time. We discuss the reason of using the execution time as a metric for computation performance in Section 4.1.2.2.

Sequential Software Only Design The execution time for sequential software design for SMMM operation is presented in Figure 4.15. The computation performance trends for sequential software implementation are identical to the Hardware/Software Co-Design Approach in the sense that as the matrix B becomes sparser the

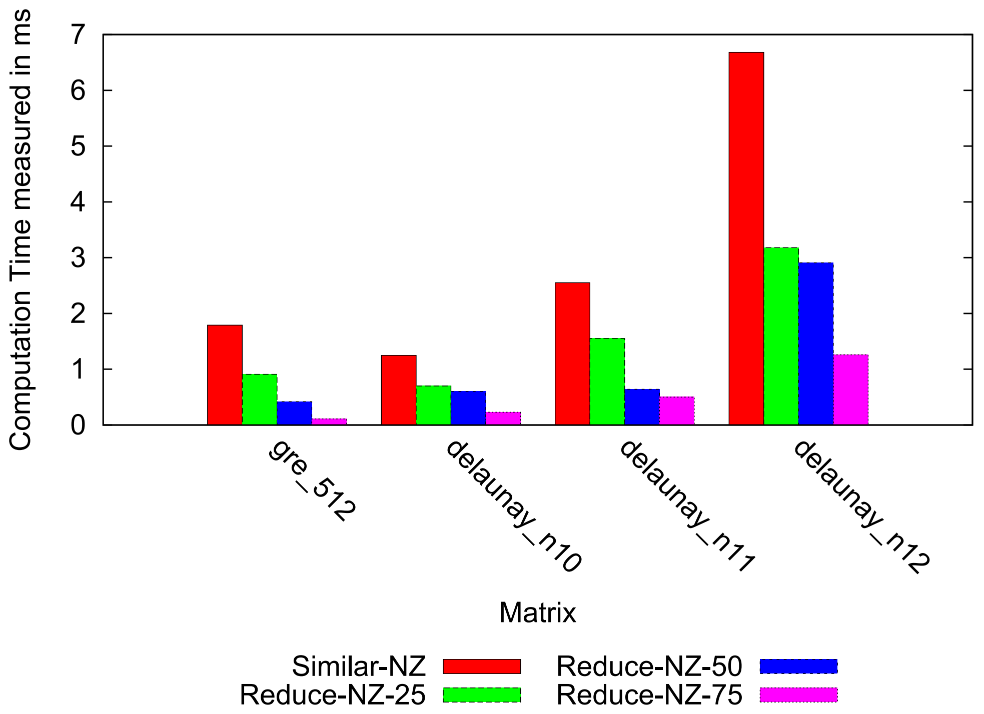

Figure 4.15: Execution time for sequential software design

computation time reduces. Also the computation time for smaller matrices like *gre_512* is significantly less than the bigger matrices like *delaunay_n12*. This is mainly because smaller the matrices lesser will be the number of comparisons that need to be performed (Lemma 1) and lower will be the overall execution time.

Parallel Software Only Design The execution time for parallel software design for SMMM operation is presented in Figure 4.16. We are reporting the execution time for a parallel implementation with four MPI worker nodes and when Similar-NZ strategy for matrix B is used. We can see that the performance trend is similar to the sequential software only design (Figure 4.15). As the matrix A becomes larger the matrix B generated using the Similar-NZ strategy and hence longer it takes to perform the comparisons, affecting the overall execution time. The distribution of the NZ elements across the test matrices is uniform, resulting in block rows which have a distribution of NZ elements close to the original matrices. This results in Super Blocks which have index overheads close to the original matrix. This results in an

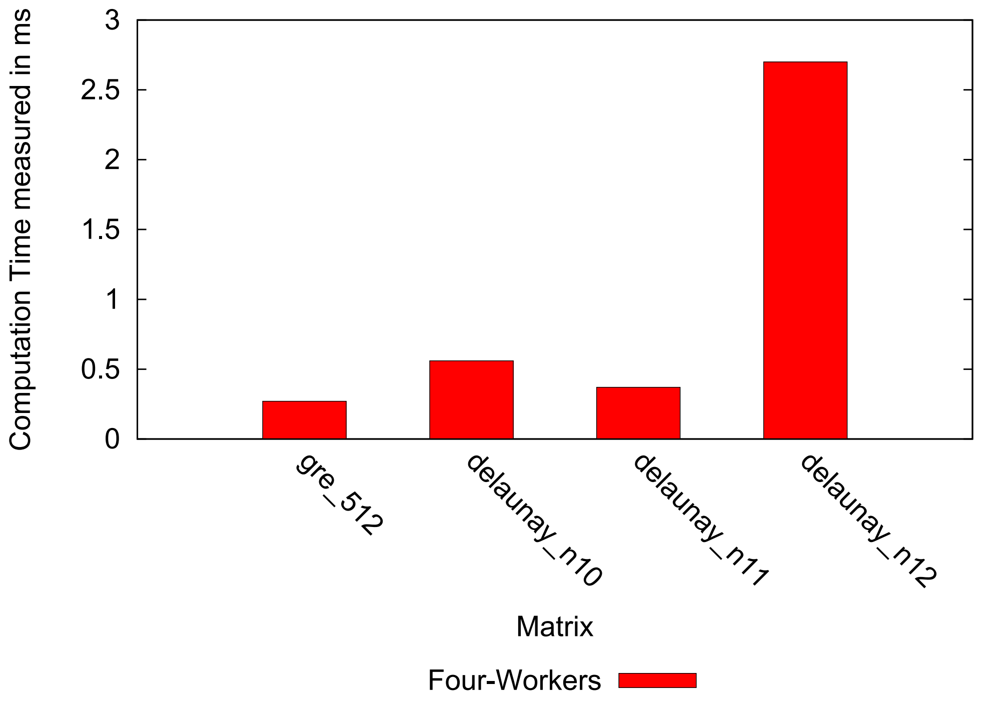

Figure 4.16: Execution time for parallel software design

execution time pattern identical between sequential and parallel implementation of the software only SMMM operation.

4.3 Co-Design Approach versus Software-Only Approach

We compare the performance metrics relevant for the two Sparse Matrix operations when implemented in the two design approaches we have proposed to evaluate their efficacy. The performance metrics that we are using for the purpose of our evaluation are listed in Table 4.9. We use sequential implementation measurements to determine the floating point efficiency, bandwidth efficiency and computation time. This is done to provide a more accurate comparison as the parallel implementation used for the hardware design and software design are significantly different and providing a one-to-one comparison becomes difficult. The software design incurs a lot more overhead due to usage of the MPI APIs for the purpose of communication when compared to the hardware design where the only overhead is the AIREN Router setup latency. We evaluate the parallel implementation using the speedup metric to analyze

Table 4.9: Performance metrics used for comparison

Performance Metric	Sparse Matrix Operation	
	SMVM	SMMM
Floating Point Efficiency	✓	
Bandwidth Efficiency	✓	✓
Computation Time		✓
Speedup	✓	✓

the scalability provided by the two approaches in order to determine the performance improvements.

4.3.1 Floating Point Efficiency

We propose the use of floating point efficiency instead of absolute FLOPS measurement when comparing the floating point performance of the two approaches. The main reason for this is the difference in the operating frequency of the two approaches. The Hardware/Software Co-Design approach is implemented on an FPGA device consisting of a floating point computation core operating at 100MHz. This is a significant difference when compared against the Software Only Design approach which is running on a conventional server consisting of an Intel Xeon Phi processor with a peak operating frequency of 1.6GHz. We calculate floating point efficiency using Equation 4.4 and it estimates the fraction of peak floating point performance available that gets utilized in performing the SMVM operation. In Equation 4.4 $Float_{eff}$ represents the floating point efficiency, $Float_{abs}$ is the absolute floating point performance measured and $Float_{peak}$ is the peak floating point performance available.

$$Float_{eff} = \frac{Float_{actual}}{Float_{peak}} \times 100 \tag{4.4}$$

The peak floating point performance available for the computation unit in Hardware/Software Co-Design approach to perform the matrix vector multiplication is 400 MFLOPS as discussed in Section 4.1.1. As seen from our discussion in Section 4.1.1 the peak floating point performance is based on the operating frequency of the BPUs.

The calculation of peak floating point performance in case of Software Only Design approach is more complicated. We have used *SoftFloat* library [27] to perform the floating point operation for the software implementation and the peak floating point performance can be estimated in two ways:

- An estimated FLOP rating for the double precision multiply and add operation is available from the *SoftFloat* library. These measurements indicates the ideal performance of the floating point operations if performed using the *SoftFloat* library in isolation. The *SoftFloat* library provides 2.7MFLOPS for double precision multiplication and 2.77 MFLOPS for double precision addition. These measurements were obtained by running a performance test program provided with the *SoftFloat* library. The peak floating point performance in this case can be estimated using Equation 4.5:

$$Peak_{sf} = (FLOP_{mult} + FLOP_{add}) \times NZ \tag{4.5}$$

  In Equation 4.5 $Peak_{sf}$ represents the peak floating point performance available if we use the rated FLOP performance listed by the *SoftFloat* library, $FLOP_{mult}$ and $FLOP_{add}$ is the double precision FLOP performance rated by the *SoftFloat* library. We will call this approach of estimating the peak floating point performance as *SF_peak*.

- We can also estimate the peak floating point performance by using the processor frequency of Intel Xeon Phi processor on which the software program is implemented. We will call this approach FP_{peak}. We calculate the FP_{peak} using Equation 4.6, where P_{freq} is the rated processor frequency.

$$FP_{peak} = 2 \times NZ \times P_{freq} \tag{4.6}$$

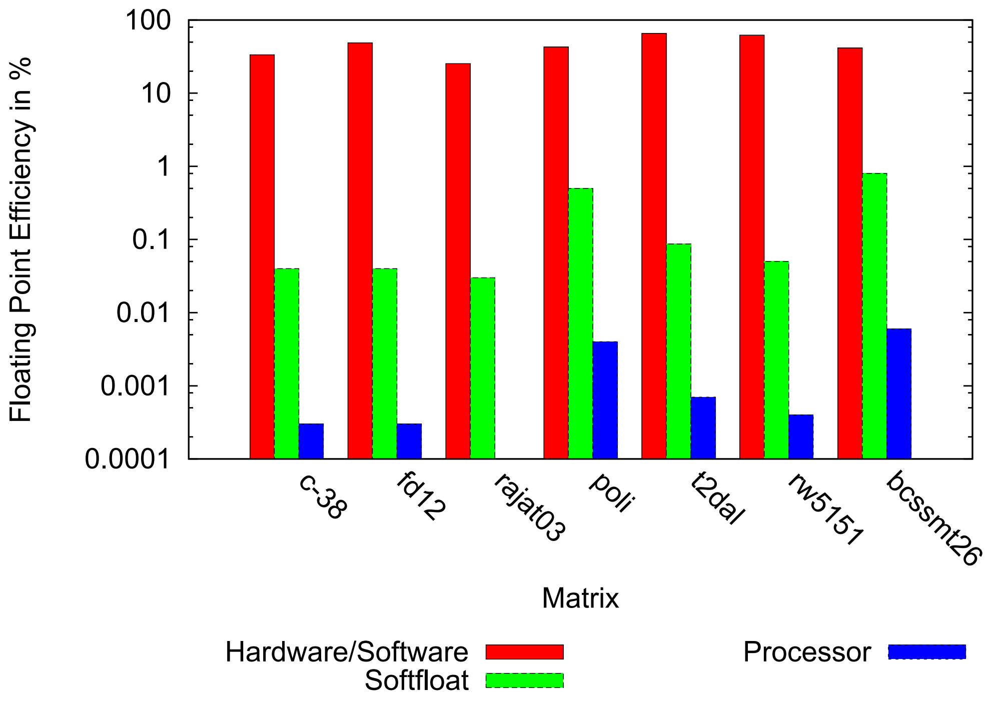

Figure 4.17: Floating point efficiency comparison

We can see from Figure 4.17 that the Hardware/Software Co-Design approach outperforms the Software Only approach. One main reason is the Hardware/Software Co-Design approach presents a completely pipelined design which results in the memory operations and the floating point operations happening simultaneously. In case of the Software Only approach this is not the case; the floating point operations have to wait till the array lookup for the block information has taken place and the relevant decoding operations have been executed. The floating point operations can only start after the memory transactions and decode operations.

4.3.2 Bandwidth Efficiency

Unlike the hardware design where the peak memory bandwidth of the NPI channel could be measured using a read DMA request and hardware counters, in case of software design estimating the peak memory bandwidth is more complex. We have used the *STREAM* benchmark [28] to estimate the peak memory bandwidth for a single Xeon Phi processor and used it to calculate the bandwidth efficiency. The bandwidth efficiency percentage is calculated using Equation 4.7, where BW_{actual} represents the actual throughput for the memory request and BW_{peak} represents peak memory bandwidth obtained using the $STREAM$ benchmark and is around 160GB/sec.

$$BW_{eff} = \frac{BW_{actual}}{BW_{peak}} \times 100 \tag{4.7}$$

In order to measure the BW_{actual} in a non-intrusive manner we rely on a simple assumption that the sequential software implementation performs two types of operations: mathematical (floating point operations to perform matrix-vector multiply) and memory transactions to fetch blocks of the matrix encoded in the VDCB format and to perform look ups on vector $\vec{x}$, $\vec{y}$. Based on this assumption we calculate the time for memory transactions by calculating the difference between the total execution time and time taken to perform the floating point operations. We use this measurement to calculate the BW_{actual} as shown in Equation 4.8, where R represents the size of memory request and t_{mem} indicates the time for memory transaction which is calculated as discussed earlier.

$$BW_{actual} = \frac{R}{t_{mem}} \tag{4.8}$$

The bandwidth efficiency comparison between the two approaches is presented in Figure 4.18 when VDCB format is used to perform the SMVM operation.

It can be seen that the Hardware/Software Co-Design approach outperforms the Software Only approach by a significant amount although it is running on a much slower FPGA device. The main reason for the significant deterioration in the bandwidth efficiency in Software-Only approach is due to the array based lookups for the block information as well as for the decoding operation of the bitmaps. In case of the Software Only operation the decoding is not happening "on the fly" and simultaneously with other operations. Instead all the operations happen sequentially and

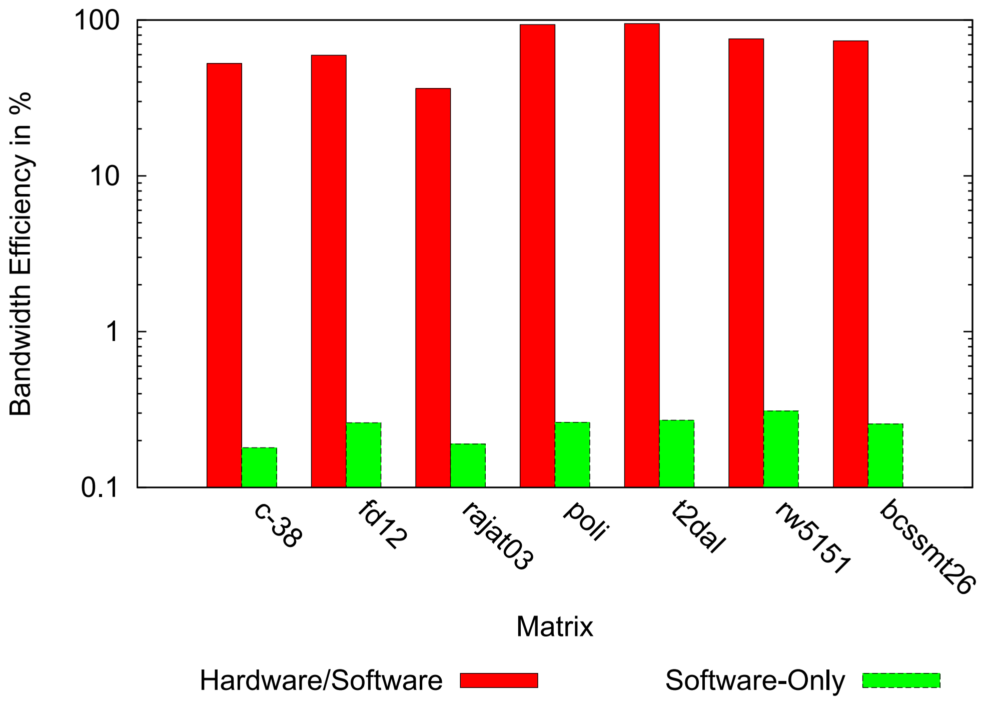

Figure 4.18: Bandwidth efficiency comparison for SMVM operation

require a memory access to fetch relevant information. These factors negatively impact the memory bandwidth efficiency in case of Software Only SMVM Operation as can be seen from Figure 4.18. The bandwidth efficiency comparison for the SMMM operation when using the two approaches is shown in Figure 4.19. It can be seen that the Hardware/Software Co-Design approach in case of the SMMM operation also performs significantly better than the Software Only approach. The Software Only approach again in case of the SMMM operation relies on the array based lookups to obtain relevant block information and introduce memory indirections which reduce the bandwidth efficiency significantly.

4.3.3 Computation Time

As discussed earlier in Section 4.1.2.2 we use execution time to determine the computation efficiency of the SMMM operation. In order to perform a fair execution time comparison between the Hardware/Software Co-Design Approach and Software Only Design Approach we will have to normalize the execution time over a common

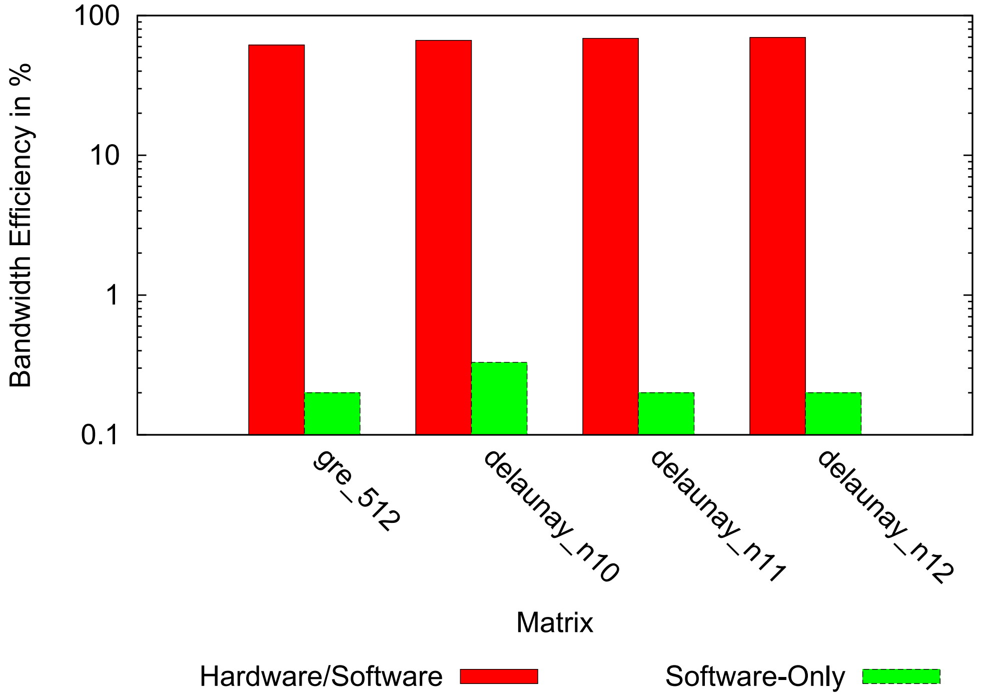

Figure 4.19: Bandwidth efficiency comparison for SMMM operation

frequency as both the implementations are running at different frequencies. The Hardware/Software Co-Design implementation of the SMMM operation uses 200 MHz for memory transactions and 100 MHz for performing the comparisons. In case of the Software Only approach the memory transactions and comparison operations are running at a common processor frequency. We normalize the Software Only approach execution time as we do not need to differentiate between the memory transaction time and computation time as both the operations are taking place at a common frequency. We scale down the execution time of Software Only Design approach with a clock frequency of 200 MHz as that is the highest frequency of operation available on the Hardware/Software Co-Design approach.

We can see from Figure 4.20 that the Hardware/Software Co-Design implementation provides a much lower execution time. This is due to the complete elimination of memory indirection by the customized memory hierarchy in case of Hardware/-Software Co-Design approach. In case of the Software Only approach the memory

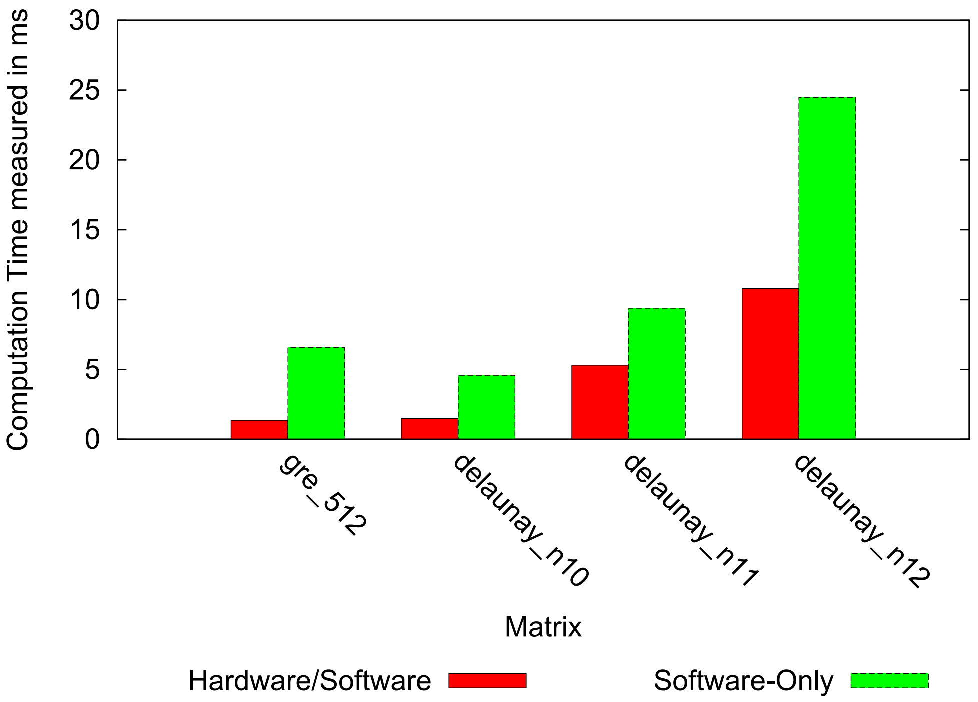

Figure 4.20: Computation time comparison

Table 4.10: Speedup comparison

Matrix Operation	Hardware/Software Co-Design	Software Only
SMVM	2.72X	1.8X
SMMM	2.2X	2.7X

indirections creates long stalls for the software comparison unit resulting in higher execution time.

4.3.4 Speedup

We present a brief comparison of the speedup obtained over sequential performance when four worker nodes are used in case of Hardware/Software Co-Design and Software Only approaches. In case of the SMMM operation Similar NZ technique is used for the generation of matrix B for the test matrices. The speedup results are presented in Table 4.10.

It can be seen in case of the SMVM operation the Hardware/Software Co-design approach provides a higher speedup in comparison to Software Only approach, thereby proving to be a more efficient and scalable solution. In case of the SMMM operation it is seen that the speedup for the Software Only design is higher. The main reason in this case is the way the matrix B is stored in both the approaches. In case of Hardware/Software Co-Design approach matrix B is stored in the FIFOs and when the blocks of matrix B have to be popped each element of the FIFO has to be read till the inequality condition is negated, as seen in Algorithm 9. The Software Only design stores the matrix B in an array and the popping the blocks of matrix B requires an address increment of the array to the next block based on the block header resulting in a much more quicker negation of the inequality. This particular issue is more pronounced in case of the parallel design as each worker node is using a part of the matrix A to perform the operation and hence the number of memory indirections (in case of Software Only) are lesser and hence better execution times are achieved for parallel design, resulting in higher speedup.

CHAPTER 5: CONCLUSION

The currently available sparse matrix storage formats have shown consistently poor performance on various processor architectures and this issue is going to be exacerbated with the future processor architecture. In order to answer the thesis question *As the memory bandwidth remains limiting issue on current and future processor architectures, will the usage of legacy sparse matrix storage formats prove detrimental for sparse matrix operations?*, a new sparse matrix storage format known as Variable Dual Compressed Blocks (VDCB) was designed and was used to implement the Sparse Matrix Vector Multiplication (SMVM) and Sparse Matrix-Matrix Multiplication (SMMM) operation.

We have conjectured in our thesis question that the incompatibility between the legacy storage formats and the memory subsystems available on conventional many-core and multi-core architectures result in performance deterioration for the Sparse Matrix Operations. In order to validate our conjecture we examine if the VDCB format can solely address the performance impediments of the Sparse Matrix Operations or a customized memory hierarchy working in conjunction with the VDCB format will alleviate the performance deterrents of the Sparse Matrix Operations. In order to evaluate this we proposed two design approaches: Hardware/Software Co-Design Approach and Software Only Design Approach. The Hardware/Software Co-Design approach used FPGAs to design customized memory hierarchy to perform the SMVM and SMMM operation using the VDCB format. In case of Software Only Design approach a C code was developed and executed on a conventional Intel Xeon Phi processor to perform the SMVM and SMMM operation using the VDCB format.

We have evaluated the two approaches for bandwidth efficiency, floating point efficiency (only for SMVM operation), computation time (only for SMMM operation) and speedup obtained between parallel implementation with four nodes and sequential implementation. The average bandwidth efficiency for the Hardware/Software Co-Design approach is 70% for the SMVM operation and 63% for the SMMM operation. In case of the Software Only approach the average bandwidth efficiency is 0.2% for the SMVM operation and around 0.25% for the SMMM operation. The average floating point efficiency in case of Hardware/Software Co-Design approach is around 45%. The floating point efficiency for the Software Only approach is evaluated in two ways: performance based on SoftFloat library and performance based on the processor frequency. In case of the Softfloat library the average floating point efficiency is 0.221% and in case of the processor frequency the average floating point efficiency is 0.001%. We have used the computation time as the performance metric to evaluate the SMMM operation is 4.5ms for the Hardware/Software Co-Design approach and 6.2ms for the Software Only approach. We achieve a speedup of 2.72X for the SMVM operation and 2.2X for the SMMM operation in Hardware/Software Co-Design approach when four worker nodes are used. In case of the Software Only approach a speedup of 1.8X for the SMVM operation and a speedup of 2.7X for the SMMM operation is obtained from four worker nodes.

Based on the comparison performed between the two approaches it can be clearly seen that the Hardware/Software Co-Design approach outperforms the Software Only approach for overall computation and memory bandwidth performance for the Sparse Matrix Operations. In short, the Variable Dual Compressed Block storage format working in conjunction with a customized memory hierarchy resolves the performance deterrents associated with the Sparse Matrix Operations and provides a high computation performance.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[2] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *Proceedings of the First international conference on High Performance Computing and Communications*, ser. HPCC'05, 2005, pp. 807–816.

[3] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proc. IPDPS*, 2011.

[4] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '99, 1999.

[5] S. Jain, R. Pottathuparambil, and R. Sass, "Implications of memory-efficiency on sparse matrix-vector multiplication," in *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, ser. SAAHPC '11, 2011, pp. 80–83.

[6] R. Sass and A. Schmidt, *Embedded Systems Design with Platform FPGAs: Principles and Practices*, ser. Morgan Kaufmann.

[7] Xilinx, Inc., "Command line user guide (ug628) v13.2," June 2011.

[8] ——, "LocalLink interface specification (sp006)," July 2011.

[9] A. G. Schmidt, W. V. Kritikos, R. R. Sharma, and R. Sass, "AIREN: A novel integration of on-chip and off-chip FPGA networks," in *Symposium on Field-Programmable Custom Computing Machines.* IEEE Computer Society, April 2009, pp. 271–274.

[10] A. G. Schmidt, B. Huang, and R. Sass, "Checkpoint/restart and beyond: Resilient high performance computing with FPGAs," in *Symposium on Field-Programmable Custom Computing Machines.* IEEE Computer Society, May 2011, pp. 162–169.

[11] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, ser. FPGA '05, 2005, pp. 63–74.

[12] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, "Fpga based sparse matrix vector multiplication using commodity dram memory," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, aug. 2007, pp. 786 –791.

[13] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07, 2007, pp. 38:1–38:12.

[14] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus," *Engineering*, vol. 24704, no. RC24704, 2008.

[15] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[16] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, Sep. 1978.

[17] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Trans. Algorithms*, vol. 1, no. 1, pp. 2–13, Jul. 2005.

[18] P. Sulatycke and K. Ghose, "Caching-efficient multithreaded fast multiplication of sparse matrices," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, mar-3 apr 1998, pp. 117 –123.

[19] A. Buluç and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," *CoRR*, vol. abs/1006.2183, 2010.

[20] C. Lin, Z. Zhang, N. Wong, and H.-H. So, "Design space exploration for sparse matrix-matrix multiplication on fpgas," in *Field-Programmable Technology (FPT), 2010 International Conference on*, dec. 2010, pp. 369 –372.

[21] C. Lin, H. So, and P. Leong, "A model for matrix multiplication performance on fpgas," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, sept. 2011, pp. 305 –310.

[22] A. Bulucc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '09, 2009, pp. 233–244.

[23] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication," *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 711 –725, nov. 1997.

[24] T. A. Davis, "The university of florida sparse matrix collection," *NA DIGEST*, vol. 92, 1994.

[25] T. O. Bachir and J.-P. David, "Performing floating-point accumulation on a modern fpga in single and double precision," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 105–108, 2010.

[26] A. Schmidt, W. Kritikos, S. Gao, and R. Sass, "An evaluation of an integrated on-chip/off-chip network for high-performance reconfigurable computing," in *International Journal of Reconfigurable Computing*, February 2012.

[27] John Hauser, "Softfloat library."

[28] John McCalpin, "Stream benchmark."