Algorithm Architecture Co-Design

for

Dense and Sparse Matrix Computations

by

Saurabh Animesh

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2018 by the
Graduate Supervisory Committee:

Chaitali Chakrabarti, Chair
John Brunhaver
Fengbo Ren

ARIZONA STATE UNIVERSITY

December 2018

ABSTRACT

With the end of Dennard scaling and Moore's law, architects have moved towards heterogeneous designs consisting of specialized cores to achieve higher performance and energy efficiency for a target application domain. Applications of linear algebra are ubiquitous in the field of scientific computing, machine learning, statistics, etc. with matrix computations being fundamental to these linear algebra based solutions. Design of multiple dense (or sparse) matrix computation routines on the same platform is quite challenging. Added to the complexity is the fact that dense and sparse matrix computations have large differences in their storage and access patterns and are difficult to optimize on the same architecture. This thesis addresses this challenge and introduces a reconfigurable accelerator that supports both dense and sparse matrix computations efficiently.

The reconfigurable architecture has been optimized to execute the following linear algebra routines: GEMV (Dense General Matrix Vector Multiplication), GEMM (Dense General Matrix Matrix Multiplication), TRSM (Triangular Matrix Solver), LU Decomposition, Matrix Inverse, SpMV (Sparse Matrix Vector Multiplication), SpMM (Sparse Matrix Matrix Multiplication). It is a multicore architecture where each core consists of a 2D array of processing elements (PE).

The 2D array of PEs is of size 4×4 and is scheduled to perform 4×4 sized matrix updates efficiently. A sequence of such updates is used to solve a larger problem inside a core. A novel partitioned block compressed sparse data structure (PBCSC/PBCSR) is used to perform sparse kernel updates. Scalable partitioning and mapping schemes are presented that map input matrices of any given size to the multicore architecture. Design trade-offs related to the PE array dimension, size of local memory inside a core and the bandwidth between on-chip memories and the cores have been presented. An optimal core configuration is developed from this analysis. Synthesis results using a

$7nm$ PDK show that the proposed accelerator can achieve a performance of upto 32 GOPS using a single core.

ACKNOWLEDGMENTS

I would first like to express my sincere gratitude to my thesis advisor Dr. Chaitali Chakrabarti, for her continuous guidance, motivation and patience throughout my thesis work. I am also thankful to my commitee members Dr. John Brunhaver and Dr. Fengbo Ren for their time and for providing me critical feedback on my research.

Lastly but most importantly, I would like to thank my parents, sister and friends for their unconditional love and support throughout my masters studies.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Page

Chapter 1

INTRODUCTION

1.1 Application Specific Hardware Accelerators

Starting with the first commercially available microprocessor Intel 4004 in 1971 up until the early 2000s, achieving better performance for an application required less effort from the programmers. Moore's law provided enough transistors to implement complex hardware logic and software developers mostly had a free lunch. Clock frequencies kept rising until the last decade and software automatically became faster on newer generation processors. Dennard Scaling ensured that the power dissipation per unit area on chip did not increase with the shrinking transistors. So, hardware designers had an abundance of faster transistors at their disposal.

In those three decades, several innovative micro-architectural techniques such as branch prediction, out-of-order execution, etc. were employed. Instruction Level parallelism (ILP) was exploited and higher instructions per cycle became synonymous with better performance. But soon Dennard scaling stopped and processors hit the power wall. When clock frequencies for a single core stopped to scale, multicores came to the rescue. Multiple cores increased the performance by exploiting Thread Level Parallelism (TLP).

Today, we see the emergence of heterogeneous systems where processors exist with specialized accelerators for achieving higher performance. Tensor Processing Unit (Google), NVDLA (Nvidia), Neural Engine (Apple), etc. represent alternative techniques for achieving higher performance. The path forward is to create specialized hardware that targets a specific application domain. Recent research has shown that

achieving better performance of 10-100X and higher energy efficiency is possible only through application specific hardware design with custom compute units, memory, interconnect and datapath. In this work we sought to build specialized hardware that accelerates dense and sparse matrix computations that are common in Basic Linear Algebra Subprograms (BLAS) used in linear algebra libraries.

1.2 The Application

Linear algebra is the branch of mathematics that deals with linear functions and linear equations such as: $a_1x_1 + a_2x_2 + a_3x_3 + ... + a_nx_n = b$ and their representations through matrices and vectors. Linear algebraic computations are used in scientific computing, engineering simulations, image processing, statistical analysis, machine learning, etc.

The performance of linear algebra subroutines is highly dependent on the implementation techniques and also the platform chosen for running the programs. This platform dependence of linear algebra routines gave rise to a standard application for performance-critical linear algebra kernels, called the Basic Linear Algebra Subprograms (BLAS)[22]. These are software routines that provide standard building blocks for performing basic vector and matrix operations. Level 1 BLAS performs scalar-vector and vector-vector operations, Level 2 BLAS performs matrix-vector operations, and Level 3 BLAS performs matrix-matrix operations.

The BLAS routines are efficient, portable, and widely available and thus are commonly used in the development of high quality linear algebra software. The earliest mathematical software packages such as LAPACK[2], ScaLAPACK[3] have relied on the underlying BLAS to provide portability and scalability[8]. With the development of deeper levels of memory hierarchies, computer vendors had to come up with their own finely tuned implementations, e.g MKL (Math Kernel Library)[18] for Intel pro-

cessors and CuBLAS[26] for Nvidia based GPU cards. These are hand-optimized for target architectures and hence fail to provide portability.

The development of ATLAS (Automatically Tuned Linear Algebra Software)[45] provided a new approach where the open source software automatically generated optimized implementations for any given architecture based on generic BLAS and LAPACK subroutines. This technique provides portability but tends to trail behind its platform dependent variants when it comes to performance.

Another successful open source implementation was the GotoBLAS[15] which has been optimized to obtain peak performances on specific processors. Other examples of this category are MAGMA[7], BLASX[44], PLASMA[4], etc. which are specifically targeted for Multicores and GPGPUs but they still fail to achieve high performance because of lack of support from underlying platforms. Certain specialized libraries like the LAPACKrc[14] have also been developed for FPGAs which provide high performance but pose a problem in terms of scalability.

Many scientific computing and engineering problems have to deal with sparse matrices. Sparse BLAS interface was developed along with the traditional BLAS (dense) to provide routines for such unstructured sparse matrix computations. Vendor specific libraries such as Intel MKL, CuSparse[27] (Nvidia), support sparse computations through sparse BLAS interfaces. However they fail to achieve maximum performance as they are bandwidth limited on these architectures. ClSPARSE[16] (AMD) is an example of open source sparse library which has its performance benefits but is optimized for GPUs and lacks interoperability.

A challenging problem is implementing both sparse and dense computations on a single platform efficiently. Many complexities arise from non obvious and often unpredictable interactions between the compute engine (the processors) and the data upon which they perform calculations[13]. Techniques that prove helpful for dense

matrix based computations do not provide the same performance benefits when applied to sparse matrices. This is largely because of the irregular data structures that are used to represent sparse matrices in compressed form.

1.3 Motivation

Scientists have traditionally aimed to increase the performance of a single kernel such as Matrix Matrix Multiplication or Matrix Decomposition through algorithm-aware architecture design. However there is new push towards architectures that could support multiple linear algebra routines. Several attempts have been made to implement matrix computations on reconfigurable cores[32][25] which provide high performance and power efficiency with programmability. Architectures that could support both dense and sparse matrix computations have not been explored with the exception being a reconfigurable Sparse/Dense Matrix Vector Multiply unit[5] developed at TU Delft.

The objective of this work is to design a reconfigurable architecture that supports a wide range of kernels from both Sparse and Dense BLAS and to study the design trade-offs in such a unified architecture. Our work builds upon the Linear Algebra Core[31] designed at UT Austin and the REDEFINE[1] CGRA based Linear Algebra Accelerator[25] designed at IISc Bangalore. Both of these reconfigurable architectures provide evidence that an optimized accelerator for dense linear algebra can be designed with careful attention to the algorithms. These designs were shown to achieve higher performance and energy efficiency compared to implementations on state of the art multicores and GPUs.

1.4 Contributions

The main contribution of this thesis is the design of a unified architecture for performing multiple dense and sparse matrix computations.

- A reconfigurable multicore architecture is proposed which performs the following matrix computations: GEMV (Dense General Matrix Vector Multiplication), GEMM (Dense General Matrix Matrix Multiplication), TRSM (Triangular Matrix Solver), LU Decomposition, Matrix Inverse, SpMV (Sparse Matrix Vector Multiplication), SpMM (Sparse Matrix Matrix Multiplication). The accelerator is scalable in terms of number of cores. It is designed to work with matrices of any given size and sparsity.

- Each core in the architecture consists of multiple processing elements and performs kernel updates on small matrix blocks. A large problem size is distributed into multiple cores and further into multiple processing elements inside each core. The partitioning and mapping schemes for each of the kernels is presented. Multiple levels of memory hierarchy and distributed control are proposed to alleviate the need for synchronization between the individual cores.

- Design trade-offs related to the number of processing elements inside a core, the size of local memory in the core and bandwidth between the on-chip memory and the core are presented. The utilization of a single core is maximized based on these trade-offs and an optimal core configuration is derived.

- A core with 4×4 PEs capable of performing updates on 4×4 sized matrices or 16×1 sized vectors has been implemented at the RTL level in SystemVerilog. Synthesis results using a $7nm$ PDK have been used to show that a performance of upto 32 GOPS can be achieved using a single core.

1.5 Methodology and Organization

The research strategy adopted in this thesis can be categorized into 3 phases. The first phase presented in Chapter 2 describes each of the selected kernels. We go through a brief overview of the algorithms, followed by a comparison drawn between the existing works related to each of the kernels. The second phase presented in Chapter 3 describes the unified architecture that can support all the algorithms. We explain the updates performed to a small matrix block on the core for each of these algorithms. We also propose our mapping schemes for folding larger problems onto our core. Finally in Chapter 4, various design trade-offs are observed for such an architecture, followed by the implementation results of a single core. We conclude the thesis with possible future extensions to our design.

Chapter 2

BACKGROUND: ALGORITHMS AND IMPLEMENTATIONS

In this chapter we first give brief descriptions of a few selected BLAS programs. These include GEMV (Dense General Matrix Vector Multiplication), GEMM (Dense General Matrix Matrix Multiplication), TRSM (Triangular Matrix Solver), LU Decomposition, Matrix Inverse, SpMV (Sparse Matrix Vector Multiplication), SpMM (Sparse Matrix Matrix Multiplication). This is followed by a summary of existing hardware implementations of these algorithms.

2.1 Algorithm Descriptions

2.1.1 GEneral Matrix Vector Multiplication (GEMV)

The product of a dense matrix and a dense vector is calculated by using the GEMV subroutine, which is a part of Level-2 BLAS, represented by $c = \alpha Ab + \beta c$ [10]. The input matrix $A_{n\times m}$ and vector $b_{m\times 1}$ are multiplied to generate the output vector $c_{n\times 1}$ [Fig. 2.1]. Specifically, $C[i] = \sum_{k=1}^{m} A[i][k] \times B[k]$, where $1 \leq i \leq n$.

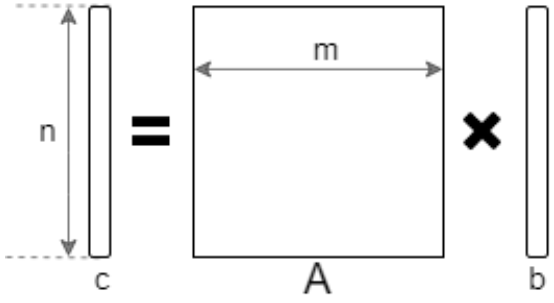

Figure 2.1: Dense Matrix Vector Multiplication: $c_{n\times 1} = A_{n\times m} \times b_{m\times 1}$

Algorithm 1: GEMV

for $i = 1 \rightarrow n$ **do**

$c_i = 0$;

for $j = 1 \rightarrow m$ **do**

$c_i = c_i + A_{i,j} \times b_j$

end

end

The sequential algorithm for GEMV requires $O(n^2)$ computations and $O(n^2)$ data movements. There is a high data level and instruction level parallelism in this subroutine. It has low arithmetic intensity and its performance is memory bandwidth limited. Reducing memory bandwidth requirement is important and it is achieved through blocking data.

The above algorithm is the inner product variant where vector b is accessed repeatedly. To allow for complete reuse of a block of b, the outer product variant can be used. Interchanging the loops gives an outer product form for GEMV where vector c is updated in each iteration.

2.1.2 GEneral Matrix Matrix Multiplication (GEMM)

The product of two dense matrices is calculated using the GEMM subroutine, which is part of Level-3 BLAS, represented by $C = \alpha AB + \beta C$ [9]. The two input matrices $A_{m \times p}$ and $B_{p \times n}$ are multiplied to generate the output matrix $C_{m \times n}$[Fig. 2.2]. Specifically, $C[i][j] = \sum_{k=1}^{p} A[i][k] \times B[k][j]$; where $1 \leq i \leq m$, $1 \leq j \leq n$.

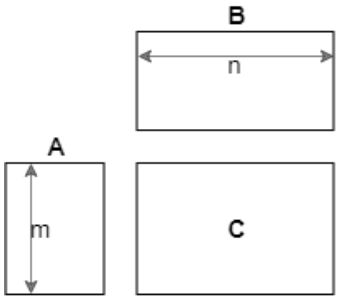

Figure 2.2: Dense Matrix Matrix Multiplication: $C = A_{m \times p} \times B_{p \times n}$

Algorithm 2: GEMM

Input: matrices A and B; Output: matrix C;

for $i = 1 \rightarrow m$ **do**

for $j = 1 \rightarrow n$ **do**

$sum = 0;$

for $k = 1 \rightarrow p$ **do**

$sum = sum + A_{i,k} \times B_{k,j}$

end

$C_{i,j} = sum;$

end

end

The sequential implementation of GEMM involves n^3 multiplications and $n^3 - n^2$ additions and has a time complexity of $O(n^3)$. GEMM exhibits both instruction and data level parallelism and hence it is important that the parallelism be fully exploited. Since it requires $O(n^3)$ computations and $O(n^2)$ data movements, it is desirable to overlap computations with data transfers. GEMM based Level-3 BLAS approach [20] is widely used to implement all Level-3 BLAS functions. Thus its performance is a representative of dense BLAS performance on a given architecture.

2.1.3 TRiangular Matrix Solver (TRSM)

The Level-3 BLAS subroutine TRSM [9] solves a system of linear equations of the form $AX = B$, where A is a nonsingular upper or lower triangular matrix; X and B are dense matrices. Solving triangular linear systems is an important step in implementing Linear Algebra operations such as LU Decomposition, Cholesky factorization, Matrix Inversion, etc.

Algorithm 3: TRSM

$A_{n\times n}X_{n\times m} = B_{n\times m}$ (A is a lower triangular matrix)

for $k = 1 \rightarrow m$ **do**

  $X_{1,k} = B_{1,k}/A_{1,1}$;

  for $i = 2 \rightarrow n$ **do**

    $s = B_{i,k}$;

    for $j = 1 \rightarrow (i\text{-}1)$ **do**
      $s = s - A_{i,j} \times X_{j,k}$

    end

    $X_{i,k} = s/A_{i,i}$

  end

end

The sequential algorithm for solving such a system uses forward or backward substitution depending on whether the matrix A is lower or upper triangular. This approach has a time complexity of $O(n^3)$. Here division is the most expensive computation and is done m times. There is a sequential dependency in the two inner loops as seen from the pseudo-code. However, the outermost loop can be unrolled to obtain parallel solutions along the columns as shown in Figure 2.3. The *white* and *gray* column blocks of B can be processed in parallel to produce the corresponding column blocks of X.

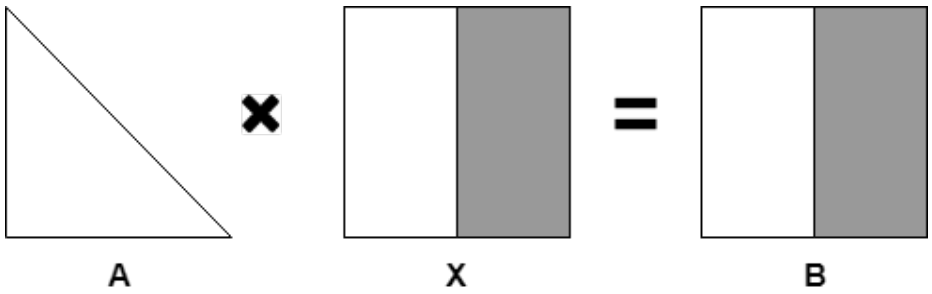

Figure 2.3: Parallel Triangular System Solution: $A \times X = B$

2.1.4 L U Decomposition (LUD)

LU Decomposition of a square matrix A refers to factorization of A into a Lower triangular matrix L and an Upper triangular matrix U and is denoted by $A = LU$ [40]. LUD is useful for solving linear system of equations and inverting matrices.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

The algorithm for solving LU Decomposition is a modified version of the Gaussian elimination technique. Decomposition of A can be done in-place, meaning that matrix A is overwritten with a lower and an upper triangular matrix without using extra memory. For such a solution the diagonal elements l_{ii} are kept as 1 and not stored explicitly.

Algorithm 4: LU Decomposition

for $k = 1 \rightarrow n$ **do**

  for $j = k \rightarrow n$ **do**
    $A_{j,k} = A_{j,k}/A_{k,k}$

  end

  for $j = k+1 \rightarrow n$ **do**

    for $i = k+1 \rightarrow n$ **do**
      $A_{i,j} = A_{i,j} - A_{i,k} * A_{k,j}$

    end

  end

end

The computations are again $O(n^3)$ similar to other Level-3 BLAS but data level parallelism is limited. A larger matrix A, can be divided into blocks and solved for L and U at the block level using LUD, GEMM and TRSM calls.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \times \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

$$\begin{aligned} A_{1,1} &= L_{1,1} \times U_{1,1} && \text{LUD} \\ A_{2,1} &= L_{2,1} \times U_{1,1} && \text{TRSM followed by GEMM} \\ A_{1,2} &= L_{1,1} \times U_{1,2} && \text{TRSM followed by GEMM} \\ A_{2,2} - L_{2,1} \times U_{1,2} &= L_{2,2} \times U_{2,2} && \text{GEMM followed by LUD} \end{aligned}$$

2.1.5 Matrix Inverse

The inverse of a matrix is defined by a matrix that when multiplied by the original matrix results in the identity matrix.

$$A^{-1}_{n\times n} \times A_{n\times n} = I_{n\times n}; \quad A_{n\times n} \times A^{-1}_{n\times n} = I_{n\times n};$$

where A, A^{-1} and I are square matrices and I is the identity matrix. A square matrix may not always be invertible and such a matrix is called a singular matrix or degenerate and its determinant is zero. Matrix inversion is extremely useful in 3-D graphics rendering, transformations and physical simulations. It also plays a crucial role in Multiple Input Multiple Output (MIMO) technology in wireless communications.

There are several methods to invert a matrix. Gauss-Jordan Elimination applies a series of row operations to obtain the identity matrix from the given matrix by serially reducing all non-diagonal elements to zero. When the same operations are applied to an identity matrix we obtain the inverse matrix. Another method uses GEMM, Triangular Matrix Inversion (similar to TRSM) and LU Decomposition to obtain the inverse as follows:

$$\begin{aligned} A &= L \times U && \text{LUD} \\ L \times X &= I && \text{TRSM: } X = L^{-1} \\ U \times Y &= I && \text{TRSM: } Y = U^{-1} \\ A^{-1} &= Y \times X && \text{GEMM} \end{aligned}$$

Other popular methods are based on eigen decomposition, Cholesky decomposition, etc. Analytical solution for matrix inversion uses the formula:

$$A^{-1} = \frac{1}{det(A)} \times adj(A)$$

$adj(A)$ is the adjugate matrix of A and $det(A)$ is the determinant of matrix A.

2.1.6 Sparse Matrix Vector Multiplication (SpMV)

A Sparse Matrix is a matrix in which the majority of the elements is zero. The SpMV[11] kernel calculates the product of a sparse matrix A and a dense vector b to generate a dense vector c and is represented by:

$$c_{n\times 1} = A_{n\times m} \times b_{m\times 1}$$

If the equation is solved using the sequential algorithm for a dense matrix vector product, the efficiency is very low. So only non-zero entries are stored and all operations are done on the non-zero entries.

The conventional implementation of SpMV, that performs dot products across each row in the matrix, is highly irregular. It suffers from branch mispredictions, and is limited by memory bandwidth. Various data structures that compress the non-zero elements are used to reduce the memory bandwidth requirement. The most common data structures used to store sparse matrices are COO (COOrdinate), CSR (Compressed Sparse Row)[Fig. 2.4], CSC (Compressed Sparse Column), ELL (ELLpack), etc. None of these data structures absolutely stand out in terms of performance for the SpMV kernel. In fact, the performance of this kernel depends on how the non-zero elements are distributed in the sparse matrix and also on the hardware architecture being used to perform the SpMV computations. The parallelism exists only along the rows which are used to calculate the dot products.

The CSR format stores all the non-zero elements in a Value array (val) and stores the column indices of the elements from val array in the Index array (ind). A pointer array (ptr) stores pointers to consecutive row beginnings.

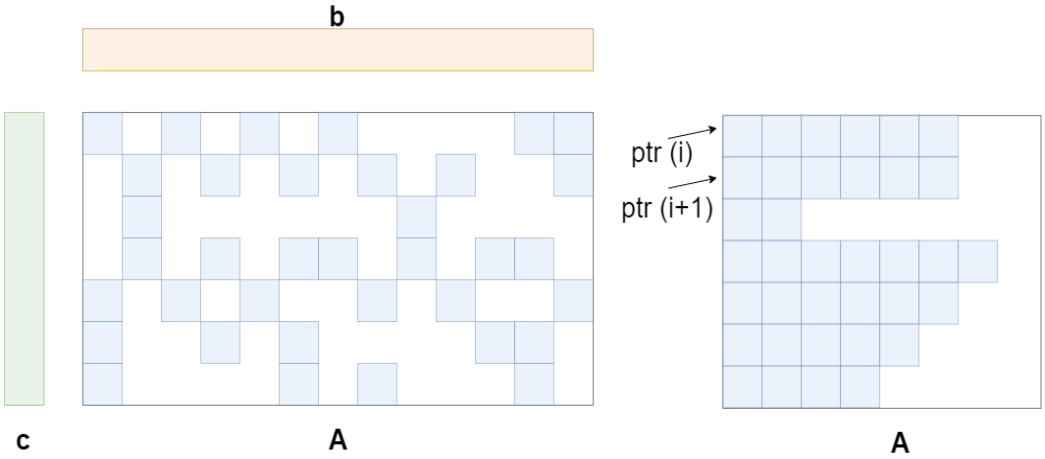

Figure 2.4: Compressed Sparse Rows

Algorithm 5: SpMV

for $i = 1 \rightarrow n$ **do**

$c[i] = 0$;

for $k = ptr[i] \rightarrow ptr[i+1]$ **do**

$c[i] = c[i] + val[k] \times b[ind[k]]$;

end

end

2.1.7 Sparse Matrix Matrix Multiplication (SpMM)

Sparse Matrix Matrix Multiplication is simply the product of two sparse matrices that produces a dense or a sparse matrix as the product. Sparse Matrices are useful in many applications such as modelling, engineering simulations, graph analytics, etc. While the product of two dense matrices shows high level of data and instruction level parallelism, the SpMM[11] subroutine is extremely irregular.

Consider a CSR representation for matrix A and a CSC representation for matrix B. Let $valA$ and $valB$ store the non-zero elements of A and B, respectively. The col indices of $valA$ are stored in $indA$ while the row indices of $valB$ stored in $indB$. Use

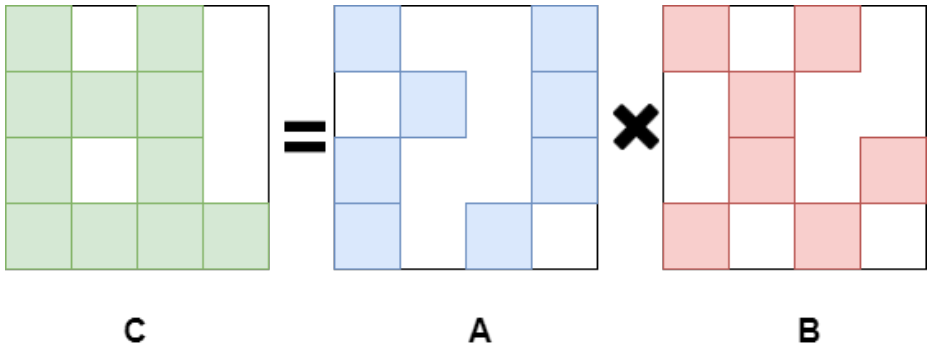

Figure 2.5: Sparse Matrix Matrix Multiplication

$ptrA$ and $ptrB$ to demarcate rows of A and columns of B, respectively:

Algorithm 6: SpMM

for $i = 1 \rightarrow n$ **do**

  for $j = 1 \rightarrow n$ **do**

    $C_{i,j} = 0$;

    for $k = ptrA[i] \rightarrow ptrA[i+1]$ **do**

      If $(indB[p] == indA[k])$*; p varying from ptrB[j] to ptrB[j+1]*

      $C_{i,j} = C_{i,j} + valA[k] \times valB[indB[p]]$;

    end

  end

end

Similar to GEMM and GEMV, the product can be calculated by inner product which produces the product matrix, one row or one column at a time. The inner product method as seen above needs to buffer an entire row of B for generating a single element of C. To reduce the communication computation gap, the outer product variant of the algorithm is used. In the outer product method, a column of A and the corresponding row of B are used to generate partial C matrix which is updated in each iteration to obtain the final matrix C.

2.2 Existing Hardware Solutions

CPU platforms rely heavily on vendor-specific libraries which are not portable to other platforms. On the other hand using open-source libraries is also inefficient because the packages do not fully utilize a given processor in terms of lower bound of CPI (Cycles per Instruction). The peak throughput is only a fraction of what can be obtained by the architectures. Improving performance on general purpose processors requires techniques like cache and register blocking, instruction reordering, loop unrolling and prefetching. Sparse matrix computations on CPUs require separate vendor optimized libraries such as ClSparse (AMD) and Sparse MKL as they do not exhibit predictable behavior like the dense computations. These sparse libraries conform to the Sparse BLAS standards[11].

GPUs have been a popular target for linear algebra problems since they make use of the available parallelism. A number of fundamental dense linear algebra algorithms have been accelerated in the MAGMA[7] library for a single GPU. Libraries such as PLASMA[4] and MAGMA incorporate tiled algorithms that are capable of exploiting the memory hierarchy efficiently. While GPUs perform the computations very efficiently there is a large overhead due to synchronization and data transfer within a heterogeneous (CPU+GPU) system[42]. GPU implementations fail to achieve peak throughput because of insufficient communication/computation overlapping and lower utilizations. Recent implementations of BLAS operations on multi-GPU systems have shown promising results with libraries such as BLASX[44] which minimizes the global communication through two level hierarchical tile cache structures. BLASX also contains better load balancing techniques compared to other GPU based libraries. The huge power consumption in GPUs is a major drawback compared to other architectures. Sparse computation support exists through libraries such as CuSparse[27]

which use compressed data structures to utilize maximum possible bandwidth.

FPGAs have been explored for Linear Algebra Computations either as a dedicated hardware or in combination with a flexible host architecture. Custom Designs on FPGAs provide both high performance and power efficiency. FPGA based linear algebra solvers like LAPACKrc[14] have also been developed to speedup GEMM and other BLAS functions but due to the limited resources available on FPGAs these have poor performance. There are also scalability issues in high performance computing. Large designs lead to routing complexities which lower the achievable clock speeds.

Systolic Arrays serve as an interesting platform for special-purpose processing[43]. Matrix computations can be implemented on a 2-D or 1-D systolic array with either unidirectional or bidirectional data flow[30]. They provide regularity and modularity but suffer a major drawback in terms of scalability[29]. Systolic architectures have been relatively less explored in the past and no commercially successful processor utilized systolic arrays until very recently[19].

CGRAs are very popular as they provide programmability while consuming low power. They have emerged as scalable embedded accelerators for High Performance Computing. The performance gain comes from using a selected number of data-paths from all the possible paths. Several data-paths are realized on a reconfigurable ASIC and thus CGRAs occupy middle ground between ASICs and FPGAs. The general configuration is a tile of processors connected to each other through a Network on Chip (NOC). Some contemporary implementations for solving linear algebra problems have made efficient use of CGRAs such as REDEFINE[1] and Layers[39]. In CGRAs like REDEFINE custom functional units are placed inside the PEs and they are modified to accelerate specific kernels[24] while Layers architecture provides a scalable and parameterized CGRA platform. Both REDEFINE and Layers employ manual mapping of the kernels onto the architecture.

ASICs have been traditionally used to accelerate a single algorithm. Recently architects have started designing efficient programmable accelerators that allow the hardware to solve multiple problems. An interesting example of such an approach is the LAC (Linear Algebra Core)[34] which offers enough flexibility to support multiple matrix computations. Theoretical evaluation of the LAC suggests that using efficient micro-architectural enhancements and mapping techniques, highly effective accelerators can be obtained for a class of operations.

We will now study a few existing hardware implementations for each of the algorithms of Section 2.1 and identify the key features of the architectures.

2.2.1 GEMV

2.2.1.1 CGRA Implementation-1 [25]

GEMV was implemented using a custom Processing Element(PE) in the REDEFINE CGRA architecture [1]. The PE comprises of Floating Point Arithmetic Units alongside a Custom Function Unit (CFU) which orchestrates data flowing in and out of the PEs [Fig. 2.6]. The Register File stores 64 entries in double precision floating point format and the Instruction Memory has a size of 16 KB.

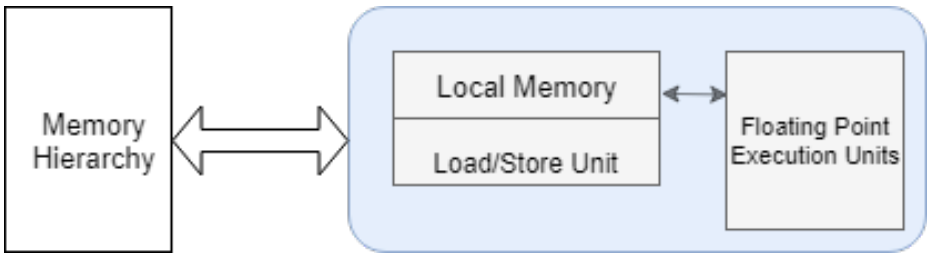

Figure 2.6: PE in the CGRA

The PEs in this architecture executed a series of Dot Product calls in parallel to obtain the product vector. A reconfigurable dot product unit was the key to achieving high performance on this architecture. Starting with a naive implementation, several

micro-architectural enhancements were implemented to increase the throughput of the PEs and eventually these PEs were attached in the CGRA to show the scalability of the approach.

2.2.1.2 CGRA Implementation-2[36]

The Layers CGRA[38] was used to implement GEMV. In this implementation rows are processed in parallel. Individual rows also utilize multiple execution units by running parallel multiplications for a reconfigurable window size and accumulating the results on reaching a row boundary [Fig. 2.7]. The execution window is shown in blue and the accumulation result for a row in yellow.

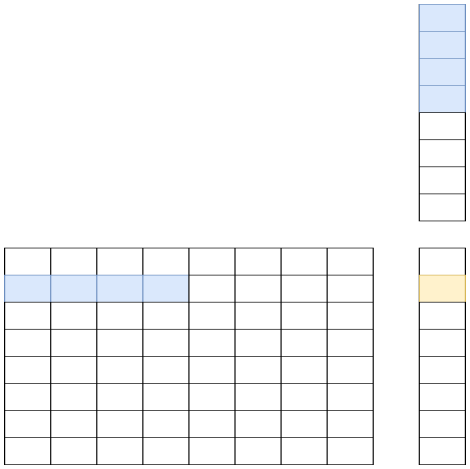

Figure 2.7: GEMV with a reconfigurable window size

Observations:

- GEMV can be easily parallelized using multiple execution units which allows the design of scalable architectures for this kernel.
- Blocking, pre-fetching and pipelining are used to achieve higher performance on custom architectures.

2.2.2 GEMM

GEMM based BLAS realization is a widely used technique for maximum resource utilization, hence we studied two existing GEMM implementations in detail.

2.2.2.1 CGRA Implementation[25]

A REDEFINE CGRA based implementation of GEMM was done together with the GEMV implementation. This architecture was explained in Section 2.2.1.1. The various architectural improvements were:

(a) Overlapping Communication and Computation by running a Load Store Unit in CFU alongside the PEs. Each of the arithmetic units inside the PEs were pipelined.

(b) The PEs were modified further for executing complex instructions such as a vector dot product and also making it reconfigurable to perform the dot product on 2, 3 or 4 elements to support different matrix sizes.

(c) Blocked load and store was adopted to support larger matrices which do not completely fit inside the register files of the PEs.

(d) Loops were restructured which allowed for prefetching data required by a PE in a subsequent iteration to fully exploit the pipelined units.

2.2.2.2 ASIC Implementation[32]

The accelerator core here consists of a 2-D array of Processing Elements (PEs). Each PE has a MAC (Multiply Accumulate) unit with a local accumulator, local storage, simple distributed control, and bus interfaces to communicate data through row and column broadcasts. The MAC unit is fully pipelined and achieves a throughput of

one MAC per cycle. The Linear Algebra Core (LAC) [Fig. 2.8] computes a 2×2 block of the output matrix C with each element of the block residing inside a PEs accumulator. Data reuse inside the LAC is ensured by sharing the input matrix elements among the PEs using column and row broadcast channels.

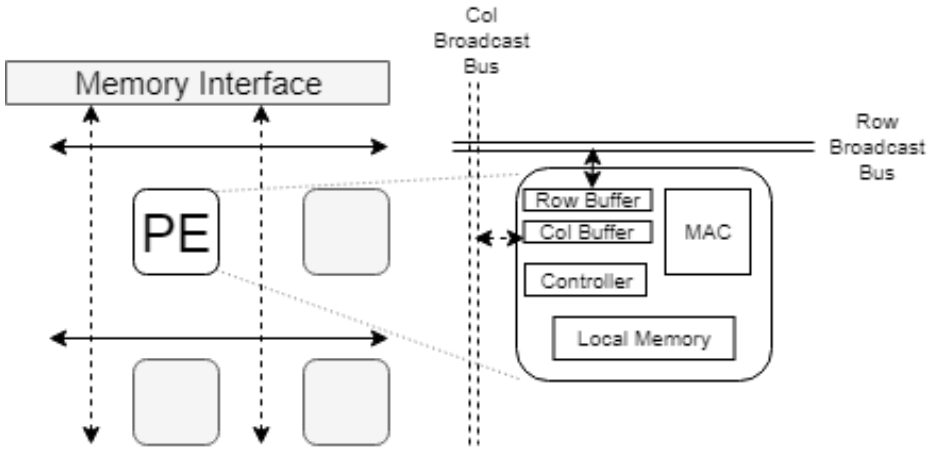

Figure 2.8: LAC with 2×2 PEs

The optimization strategies that enhanced GEMM performance on the LAC are:

(a) Blocking data was a necessity as the size of LAC core was limited. Also the blocking scheme alleviated the need for high bandwidth communication

(b) Broadcasting helped in full utilization of the input data currently inside all PEs

(c) Prefetching was also employed to keep the execution units busy

(d) The MAC units performed delayed normalization in order to achieve a throughput of one MAC per cycle which improved the throughput of the overall system

Observations:

Both implementations [25] and [32] were similar in terms of data blocking, prefetch-

ing, pipelining execution units and overlapping communication with computation. The differences between the two architectures are:

- Mapping : In [25] a block was mapped onto a single PE, thus making the CGRA implementation scalable as each block can be computed independently, while in [32] a single block was mapped onto the entire LAC ($n \times n$ PEs) thus to obtain a scalable solution, multiple LACs would be required.

- Data Reuse: Implementation [25] did not attempt to exploit data reuse in GEMM, hence a block of the input matrix needs to be supplied to multiple PEs. In [32] however the focus was to reuse input matrix blocks completely before fetching a new block. Also data reuse was employed at the local memory and cache memory level.

2.2.3 TRSM

2.2.3.1 CPU-GPU Implementation[23]

The approach was based on a block recursive algorithm that aims to reduce the computations to matrix multiplication (GEMM) in order to benefit from the well-known Strassen's algorithm[41] for Matrix Matrix Multiplications.

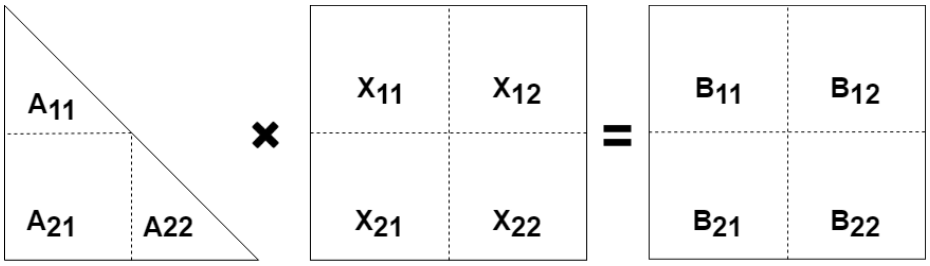

Figure 2.9: Blocked TRSM

Each node of the heterogeneous CPU-GPU system solves multiple columns of X using

forward or backward substitution. Matrix A is replicated for each node while column blocks of matrix B are distributed among the nodes [Fig. 2.9]. The key features of this implementation are:

- No communication is needed between the nodes i.e, they can function in parallel.
- Large memory requirements for each of the nodes as triangular matrix has to be replicated.

2.2.3.2 ASIC Implementation[32]

The LAC described in Section [2.2.2.2] is augmented with an inverse unit that computes $f(x) = 1/x$ and communicates with all diagonal PEs [Fig. 2.10]. The computation for a small TRSM problem of size 2×2 is performed in place. For a large TRSM problem, the computation is split into GEMM and small TRSM operations.

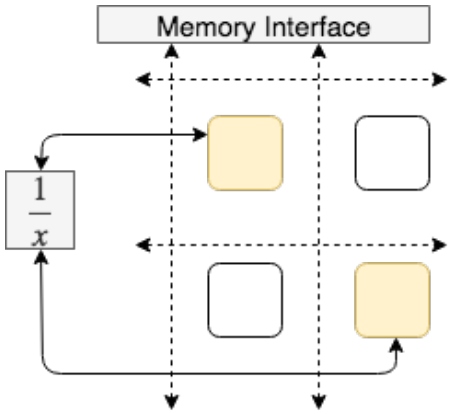

Figure 2.10: LAC with an augmented Reciprocal Unit

Observations:

- Both implementations [23] and [32] use GEMM based TRSM approach by breaking down a large TRSM problem into large GEMM and small TRSM problems.

- Each node in [23] works independently with its own copy of triangular matrix and computes column blocks of X in parallel (size of block is proportional to each node's capacity). Implementation [32] can be made scalable by replicating the core and employing a suitable partitioning scheme for input data.

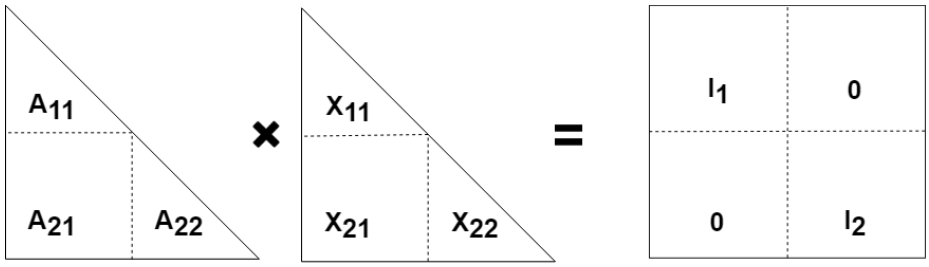

Figure 2.11: Triangular Matrix Inverse

- *Triangular Matrix Inverse*

  Any TRSM implementation can be used to obtain the inverse of a triangular matrix by replacing B with identity matrix I such that the equation $AX = B$ transforms to $AA^{-1} = I$, where X is the inverse matrix. This method was discussed in [23] and Figure 2.11 shows the blocked version.

2.2.4 LU Decomposition

2.2.4.1 FPGA Implementation[21]

An FPGA based parallel architecture was developed based on the sequential LU Decomposition Algorithm [Fig. 2.12].

The PEs comprise of MAC units. Separate data storage units are used to provide inputs to the PEs. The memory banks are accessed by PEs in an entire row or column. (Memory interface for all PEs are not shown in the figure to maintain clarity of image). LUT based dividers were used in $N - 1$ PEs, where N is the number of PEs in a row/column.

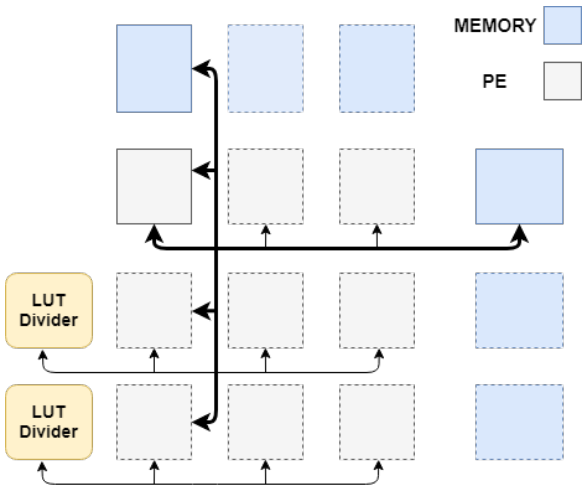

Figure 2.12: 3x3 PE array with Memory and LUT based Divider

2.2.4.2 ASIC Implementation[33]

The LAC architecture used to implement TRSM was shown to be sufficient for implementing LUD. Multiple iterations of row and column broadcast combined with reciprocal and MAC operations were performed following the sequential LUD algorithm [Section 2.1.4] to obtain inplace LU Decomposition.

Observations:

- Implementation [21] uses a divider unit while [33] uses a simpler reciprocal unit.
- PEs in [33] have local data storage memory while those in [21] uses separate data memories for each of the columns.
- The PEs do not communicate with each other in [21] whereas [33] has inter-PE communication through broadcast busses.
- FPGA implementations are not scalable in terms of number of PEs and most of the PEs are under utilized because of the sequential nature of LUD solutions.

2.2.5.1 InvArch Implementation[6]

Gauss-Jordan Elimination was used to perform matrix inversion on a custom hardware [Fig. 2.13]. The datapath consists of multiple normalization and elimination blocks. Each of the blocks has a pipelined floating point multiplier and a pipelined floating point subtractor. It stores a column of the matrix to be inverted. Multiple rows are normalized in parallel. The diagonal elements of the matrix are stored in a separate memory (R-RAM). A single floating point divider is used since only one row is processed every cycle. The data-path control is handled through a dispatch state-machine along with multiple counters and pipelines for the control signals. The counters keep track of the iteration number, row index and the range of columns being processed and help in achieving the overall synchronization.

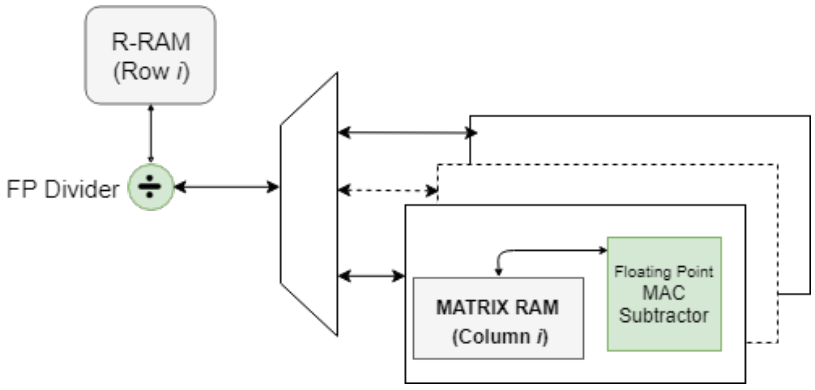

Figure 2.13: Parallel Processing along Columns for Matrix Inverse

Hardware-efficiency was improved by minimizing the number of floating point multiplication units. Floating-point computation blocks were pipelined to increase the throughput and stalls were removed by reordering the operations. Multiple custom units could be used in parallel, making the architecture scalable.

2.2.5.2 FPGA Implementation[37]

A QR decomposition based matrix inversion was implemented in this work. The QR decomposition was performed using Gram-Schmidt method. QR decomposition factorizes a matrix $A = QR$, where Q is an orthogonal such that $Q^T * Q = I$ and R is an upper triangular matrix. The inverse matrix A^{-1} is computed in 3 steps: First the factors Q and R are obtained, followed by a triangular matrix inversion of the upper triangular matrix R. Finally a matrix matrix multiplication is performed to obtain $A^{-1} = Q^T \times R^{-1}$

The datapath consists of LUT based dividers, square root units and vector multipliers. The 3 step algorithm was implemented using state machines and then the separate units for QRD, triangular inverse and matrix multiplication were stitched together to realize the matrix inversion unit. 5 RAM modules were used to store the data and compute resources were time shared whenever possible. The pipelined implementation was shown to have low error, however it could only support very small matrix sizes upto 23×23.

Observations:

- Implementation in [6] was computationally cheaper whereas [37] had better accuracy at the cost of complex arithmetic units such as a square root unit.
- Neither of the implementations are scalable in terms of large matrix inversion problems as they both had limited hardware resources.

2.2.6 SpMV

2.2.6.1 Reconfigurable Hardware Implementation[35]

A hardware accelerator unit was designed for SpMV to improve the performance of two iterative algorithms namely, conjugate gradient and Jacobi solver. CSR data structure was used to store the sparse matrix. The accelerator design has a dot-product core and a pipelined adder in addition to local buffers and memory. The main limitation with this design is the number of simultaneous memory reads from the local memory and also the limited compute resources of the FPGA. The accelerator achieves speedup of 2 times.

2.2.6.2 Pipelined SpMV Accelerator[17]

The accelerator designed for SpMV uses a novel compression scheme which partitioned the sparse matrix and stored the submatrices in a two level storage format. Row blocks are parallely processed by the PEs. Each PE [Fig. 2.14] consists of a Column Index Decoder, multiple-input multiple-output MAC and reduction unit. The accelerator was implemented on a Xilinx Virtex-7 FPGA platform and was shown to have high bandwidth utilization. It could handle sparse matrices with arbitrary size and sparsity pattern because of the hardware friendly compression format.

Observations:

- Both the architectures are custom designed for working with their exclusive sparse compression formats.
- Inner product approach requires large dot product units and tree of adders.

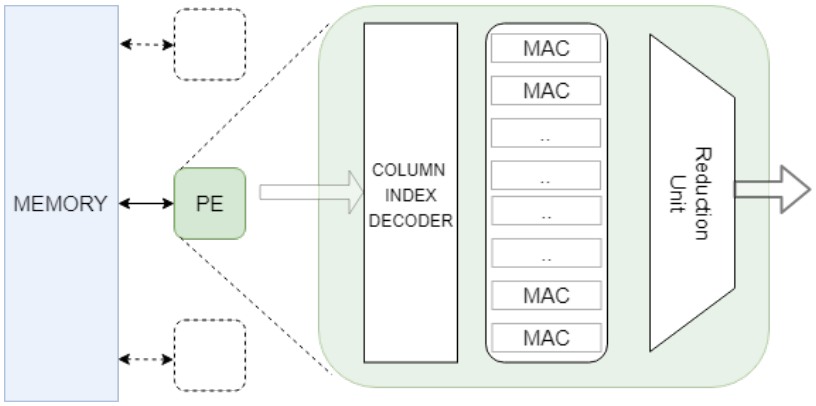

Figure 2.14: PE inside the SpMV accelerator

2.2.7 SpMM

2.2.7.1 Outer-Product based Implementation[28]

An outer product based SpMM was implemented on a reconfigurable multicore architecture [Fig. 2.15], called *Outerspace*.

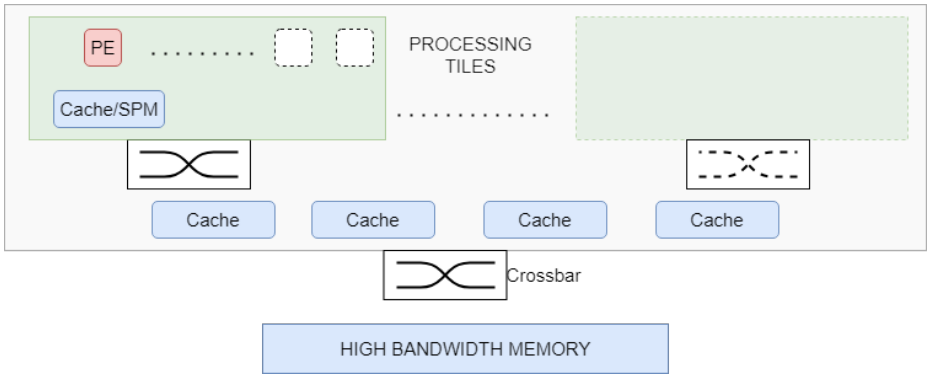

Figure 2.15: OuterSpace Architecture

The outer product operation is broken into a multiply and a merge phase and the entire computation is distributed among multiple PEs in a hierarchical fashion. The architecture consists of multiple tiles, where each tile consists of a linear array of PEs. A reconfigurable cache is used by the PEs which also serves as a Scratchpad memory (SPM). Data reusability and asynchronous workload sharing are the key

features of this architecture. Through the reconfigurable memory hierarchy, access to main memory is limited.

2.2.7.2 3-D stacked LiM Implementation[46]

A 3-D stacked Logic in memory (LiM) architecture [Fig. 2.16] was used to accelerate SpMM. The proposed computing system has logic layers stacked in between DRAM dies which communicate with each other vertically using through silicon vias (TSVs). The novelty lies in a 3D-stacked DRAM which offers high bandwidth and low latency data transfer via TSV and a stacked LiM layer that is customized for sparse matrix multiplication through a fine-grain integration of logic, CAM and SRAM.

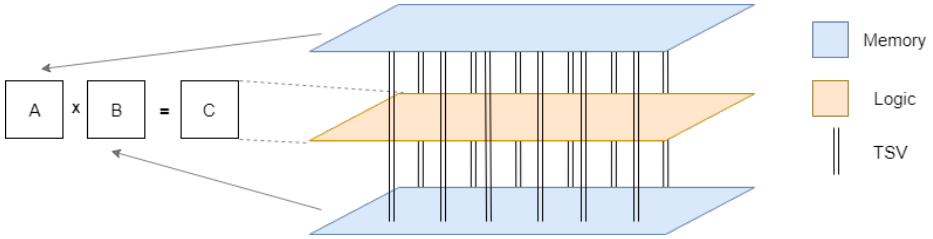

Figure 2.16: 3-D Stacked Logic in Memory

Observations:

- Outer product based implementation for SpMM leads to lesser memory bandwidth requirements and more parallelism.
- Fine grained logic memory integration can be used to achieve high performance using algorithm architecture co-design.

Chapter 3

PROPOSED UNIFIED ARCHITECTURE

In Chapter 2, we analyzed the characteristics of each of the algorithms and the corresponding existing hardware implementations. Several of these designs followed the CGRA approach and introduced some level of programmability so that they could support more than one algorithm. Examples of such an approach were Linear Algebra Core[32], REDEFINE[1] and Layers[39]. In this chapter, we introduce a unified architecture that supports a selected set of dense and sparse linear algebra programs, namely, GEMM, TRSM, LUD, Inverse, GEMV, SpMV and SpMM. The overview of the proposed architecture is presented in Section 3.1, followed by mappings for dense kernels in Section 3.2 and mappings for sparse kernels in Section 3.3

3.1 The Architecture: Overview

The proposed architecture, shown in Figure 3.1 is a multicore design, where the cores work asynchronously. A large problem is partitioned into smaller subproblems and each subproblem is assigned to a core. Blocks of data are sequentially mapped from the on-chip memory of a core to its local memories. Each core is designed to perform kernel updates on an $N_r \times N_r$ sized matrix or an $N_r^2 \times 1$ sized vector; the corresponding computation is referred to as a panel update. Multiple panel updates are executed sequentially to perform a kernel update on the entire block of data residing in the local memory of a core; the corresponding computation is referred to as a block update.

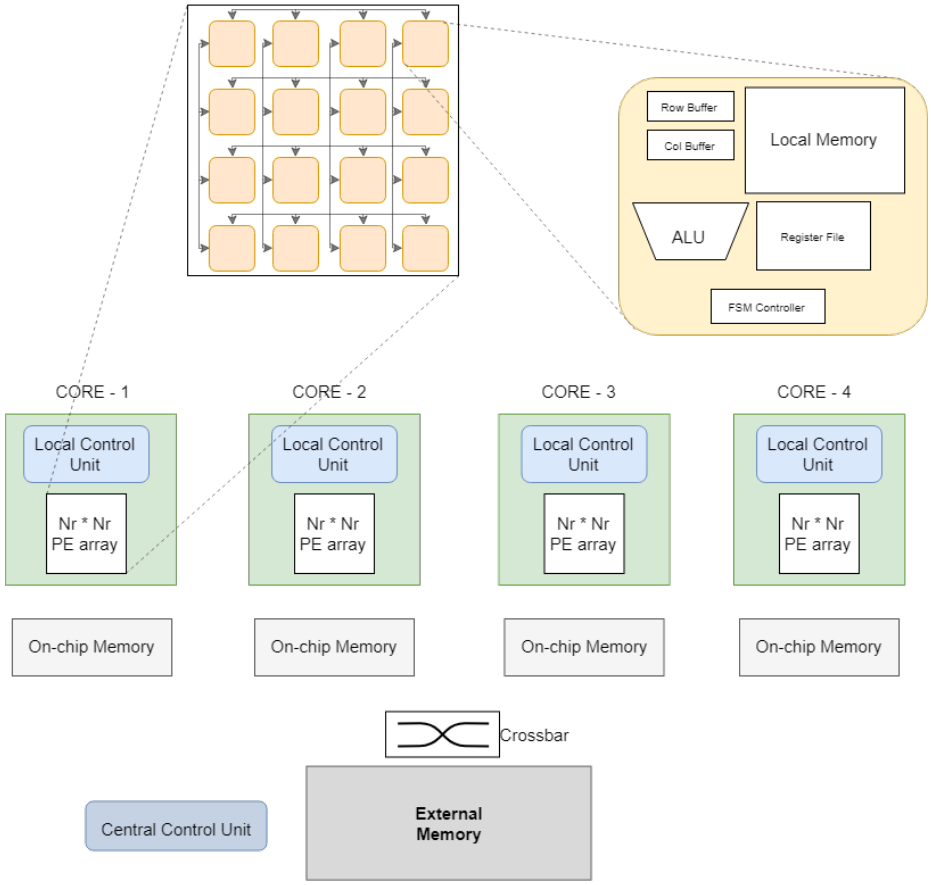

Figure 3.1: Block Diagram of the unified architecture ($N_r = 4$)

PE array with Broadcast Buses: Each core consists of a 2-D array ($N_r \times N_r$) of Processing Elements (PE). The PEs share data with each other through row and column broadcast busses. All the PEs are identical except the diagonal PEs which have an additional reciprocal unit. The row broadcast busses are also used to transfer data in and out of the core.

Memory Hierarchy: There are 4 levels of memory in this architecture. The external memory is the shared off-chip memory. The on-chip memory for each core is connected to the external memory through crossbars. The local memory inside a PE is a single ported SRAM which is essential for blocking. A register file, with 2 read

ports and 1 write port, is also present inside the PEs to provide inputs to the ALU.

Central Control Unit: This unit is responsible for partitioning a large problem into subproblems and assigning them to the local controllers.

Local Control Unit: The Local Control Unit schedules panel update instructions and data flow from on-chip memory to the local memory of a core. It is also used to synchronize the PEs inside a core while running sparse computations.

PE Design: Each PE consists of an ALU, local memory, row and column broadcast buffers, register file and an FSM controller. All the PEs have a MAC unit. The diagonal PEs also have a divider implemented by a reciprocal unit and a multiplier. Each PE consists of a Finite State Machine controller that coordinates computations for panel updates. It translates a panel update instruction provided by the local controller into a unique sequence of load-store and ALU operations for each PE.

3.2 Mapping Dense Linear Algebra Programs

We shall now describe the operations in panel update and in block update for each of the algorithms. We fix the bandwidth between on-chip memory and core as N_r elements per cycle.

3.2.1 GEMV

Partitioning: In matrix-vector multiplication (GEMV), the dense matrix A and the dense output vector c are partitioned into blocks and distributed among the cores. Dense vector b is replicated in each core as shown in Figure 3.2. There is equal load distribution and no synchronization is required between the cores.

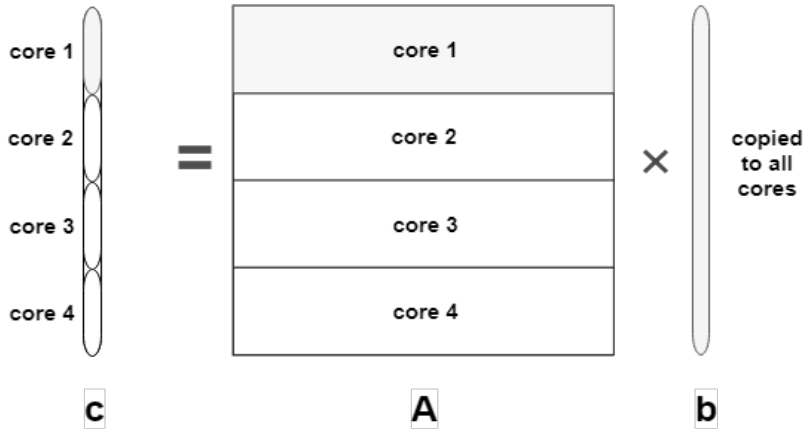

Figure 3.2: Partitioning GEMV input to multiple cores

Panel Update: The $N_r^2 \times 1$ vector update for GEMV is achieved by N_r columns of PEs inside the core, each column working in parallel. A column of PEs updates $N_r \times 1$ elements of c and hence the N_r columns together update the $N_r^2 \times 1$ sized vector c. To explain the update, we demonstrate how a single column of PEs operates.

Figure 3.3 shows the operations taking place in a single column of the PE array, where $N_r = 4$. A 4×4 sized matrix A and a 4×1 sized vector b are mapped to a column of PEs such that each PE stores 4 elements of A and 1 element of b. In the 1^{st} cycle, b_1 is broadcast by the PE in row 1 to all the PEs in the column. In the 2^{nd} cycle, all the PEs perform MAC operations using b_1: $c_1 += a_{11}b_1$, $c_2 += a_{21}b_1$, $c_3 += a_{31}b_1$ and $c_4 += a_{41}b_1$. In the same cycle, b_2 is broadcast, by the PE in row 2. In subsequent cycles, MAC operations and column broadcasts are overlapped. After the 4^{th} cycle all the 4 elements of vector b and all the 4×4 elements of matrix A have been used. Thus, it takes $N_r + 1$ cycles to update the $N_r \times 1$ vector c in a column.

During these $N_r + 1$ cycles, each column of PEs works on a different 4×4 sized matrix A but on the same 4×1 sized vector b which is replicated to each column of PEs. Thus, $N_r \times N_r$ array of PEs update $N_r^2 \times 1$ elements of vector c in $N_r + 1$ cycles.

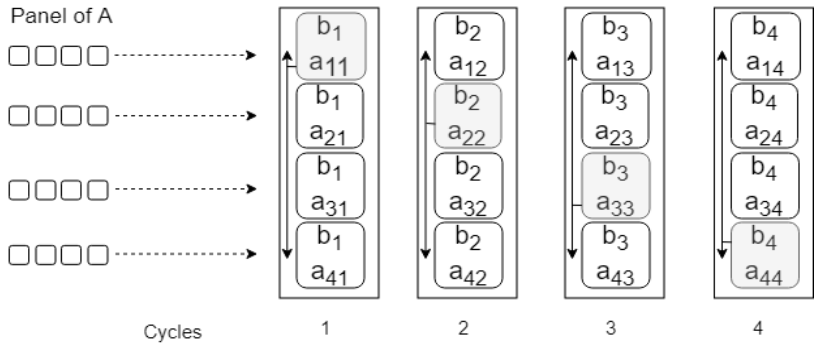

Figure 3.3: GEMV Panel Update in a column of PEs ($N_r = 4$)

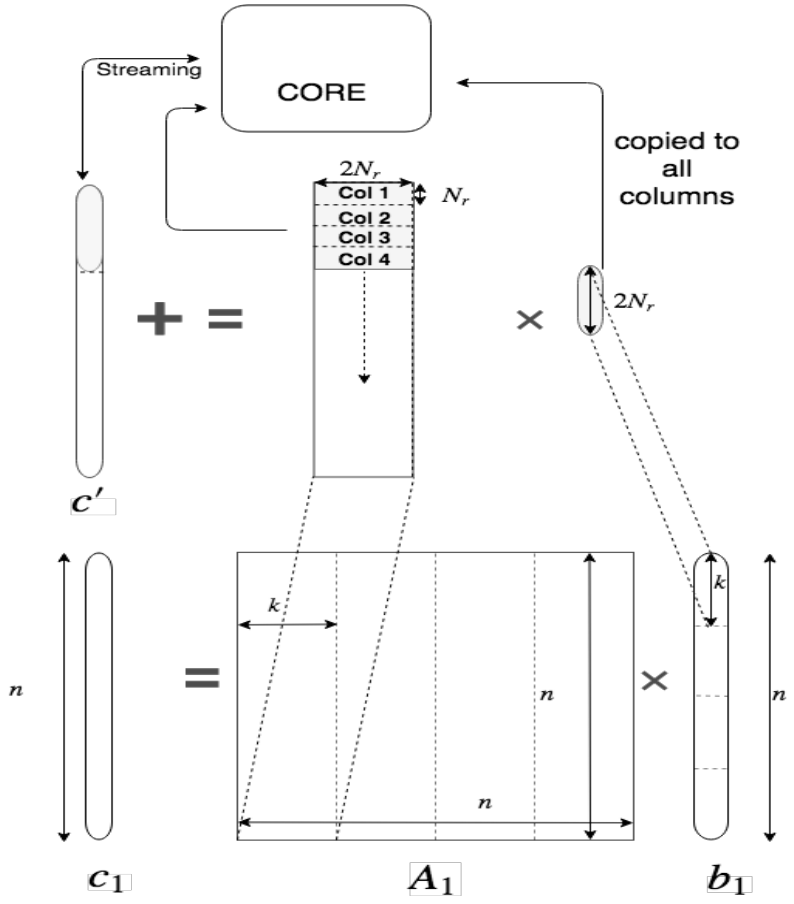

Figure 3.4: GEMV Mapping for Block Updates ($k = 2N_r$)

Block Update: Since our core can only work on small panels of size $N_r \times N_r$, a large problem is broken down as shown in Figure 3.4. To explain the decomposition,

we start with a subproblem $c_1 = A_1 \times b_1$ which has been assigned to a single core. The product vector c_1 is computed by multiple iterations of matrix vector multiplications. In each iteration, k columns of A_1 and k rows of b_1 are processed to produce a partial product vector c'. This partial product is accumulated over multiple iterations to produce the final product vector c_1.

Since elements of vector c' are accessed multiple times, it is important to overlap their data access with computation. Since we have a bandwidth of N_r elements per cycle, it takes N_r cycles to transfer an updated $N_r^2 \times 1$ panel of c' and another N_r cycles to fetch the next panel of c'. Hence, we extend the panel update operations inside a column of PEs from $c_{N_r \times 1} = A_{N_r \times N_r} \times b_{N_r \times 1}$ to $c_{N_r \times 1} = A_{N_r \times k} \times b_{k \times 1}$ to increase the panel update duration in the core from N_r cycles to k cycles, where $k \geq 2N_r$. This allows us to overlap the writing of a previously updated panel of c' to the on-chip memory and prefetching of elements of c' for the next panel update with the panel update computations taking place inside a core.

3.2.2 GEMM

Partitioning: In matrix-matrix multiplication, the dense output matrix C is partitioned into blocks and distributed among the cores. The dense input matrices A and B are partitioned and replicated to multiple cores as shown in Figure 3.5. There is equal load distribution and no synchronization is required between the cores.

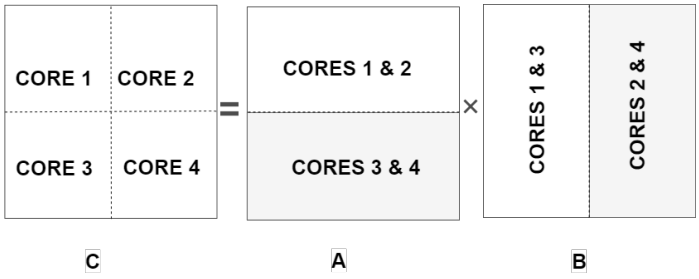

Figure 3.5: Partitioning GEMM input to multiple cores

Panel Update: The $N_r \times N_r$ matrix update for GEMM is explained in Figure 3.6 using the 2-D PE array, where $N_r = 3$. A 3×3 sized matrix A is replicated inside each of the column of PEs such that each PE in row (i) stores a copy of all the elements from row (i) of the matrix A. A 3×3 sized matrix B is mapped to the PE array such that, PE(i, j) stores a single element of B, i.e $B(i, j)$. In the 1^{st} cycle, $B_{1,i}$ is broadcast by the PEs in row 1 to all the PEs in column (i). In the 2^{nd} cycle, all the PEs perform MAC operations using $B(1, i)$, e.g PE$(2, 2)$ updates $C_{2,2} += A_{2,1} \times B_{1,2}$. In the same cycle, $B_{2,i}$ is broadcast by the PEs in row 2 to all the PEs in column i. In subsequent cycles, MAC operations and column broadcasts are overlapped. After the 3^{rd} cycle, all the elements of matrices A and B have been used. Thus, it takes $N_r + 1$ cycles to update the $N_r \times N_r$ matrix C in the core.

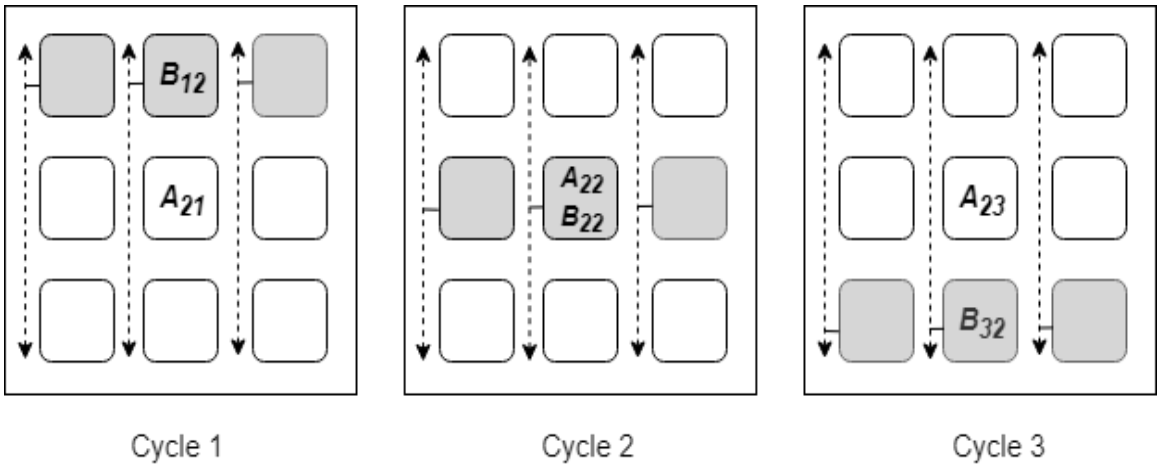

Figure 3.6: GEMM Panel Update in a PE array ($N_r = 3$)

Block Update: A decomposition strategy similar to GEMV was used to break down a large GEMM problem into small panel updates as shown in Figure 3.7. In order to solve the subproblem $C_1 = A_1 \times B_1$ assigned to a core, the output matrix C_1 is computed by multiple iterations of matrix matrix multiplications. In each iteration,

m columns of A_1 and m rows of B_1 are processed to produce a partial product matrix C'. This partial product is accumulated over multiple iterations to produce the final product matrix C_1.

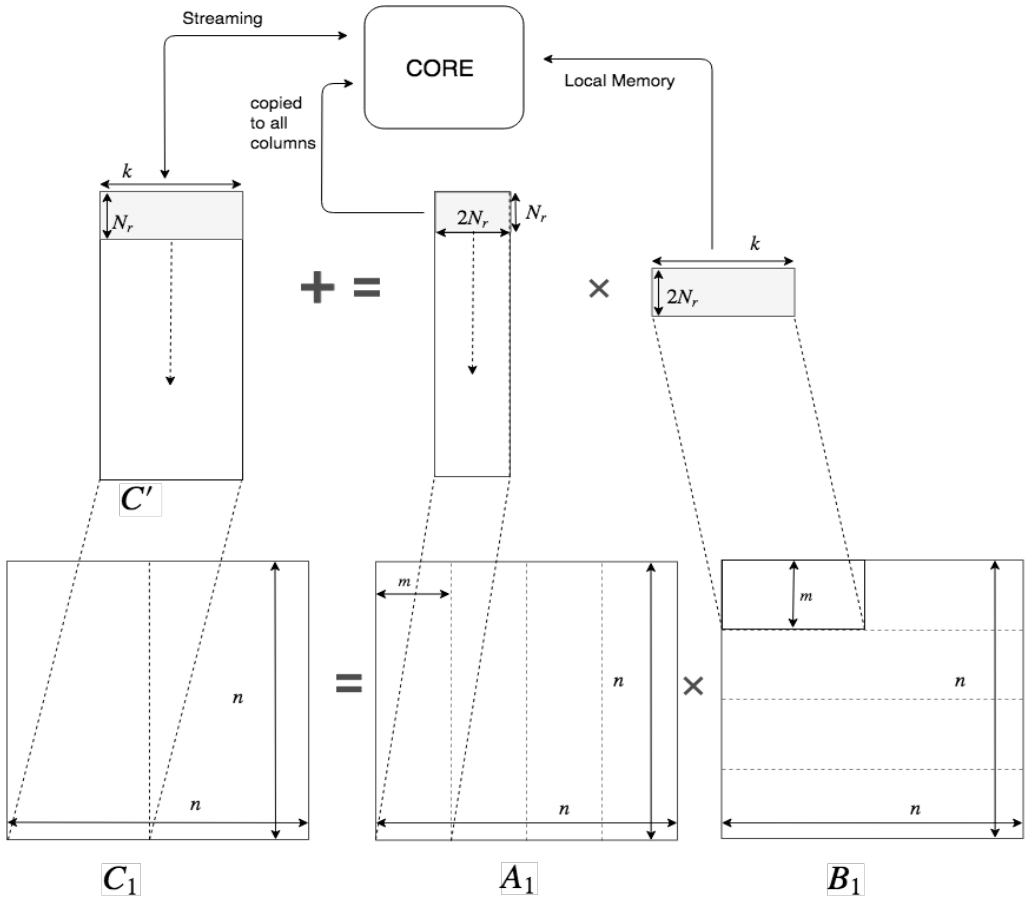

Figure 3.7: GEMM Mapping for Block Updates ($m = 2Nr$)

Similar to the block update in GEMV, the extended panel update operation for GEMM is performed as: $C_{N_r \times N_r} = A_{N_r \times m} \times B_{m \times N_r}$, where $m \geq 2N_r$. This allows us to schedule the writing of a previously updated panel of C' to the on-chip memory and prefetching of elements of C' for the next panel update while the current panel update computations are taking place inside a core.

During a single iteration of the GEMM block update, $n \times m$ elements of matrix A_1

are accessed repeatedly if the entire $m \times n$ block of matrix B_1 is not present inside the core. Hence we map a larger $m \times k$ block of matrix B_1 to the core to reduce repeated access to elements of A_1, where k is dependent on the available local memory size in the PEs in a core.

3.2.3 TRSM

Partitioning: As shown in Figure 2.3, a large TRSM problem ($A \times X = B$) can be partitioned for parallel execution. Column blocks of matrix B are distributed among the cores and matrix A is copied in the on-chip memory of each core. Each of the cores compute an exclusive portion of the result matrix X. There is equal load distribution and no synchronization is required between the cores.

Panel Update: An $N_r \times N_r$ sized panel of a lower triangular matrix A is replicated in all the columns of the PE array such that each PE in row (i) stores a copy of all the elements from row (i) of the matrix A. An $N_r \times N_r$ panel of matrix B is mapped to the core such that, PE_{ij} stores a single element of B, i.e B_{ij}. The initialization step calculates the reciprocal of A_{ii} ($r_{ii} = 1/A_{ii}$) in all diagonal PEs. The reciprocal r_{ii} is then broadcast along the i^{th} row. This is followed by a multiplication in row 1 ($X_{1i} = B_{1i} \times r_{11}$) to obtain the solution in the first row. Next, two steps are repeated for $N_r - 1$ iterations. Let m denote the iteration number:

Step-1: MAC update in all PEs below row(m): $B_{ij} -= A_{im}.X_{mj}$; where X is received through column broadcast from row(m) and $i > m$.

Step-2: Multiplication update in all PEs in row($m+1$) to update

$X_{m+1,j} = B_{m+1,j}.r_{m+1,m+1}$; (in place solution: $b \rightarrow x$)

It takes 3 cycles for initialization and $(N_r - 1) \times 3$ cycles for $(N_r - 1)$ iterations of $broadcast \rightarrow MAC \rightarrow multiplication$. Hence, $3N_r$ cycles are required in total for

a TRSM panel update.

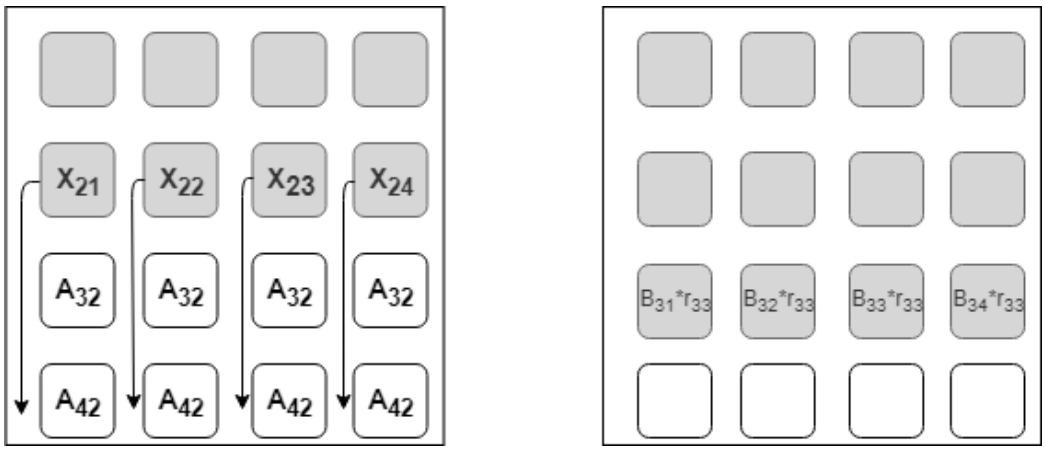

Figure 3.8: TRSM Panel Update in a PE array ($N_r = 4$)

Figure 3.8 shows the 2^{nd} iteration($m = 2$) for a 4×4 panel update. Step-1 updates all elements of matrix B in rows 3 and 4. The elements of matrix A are already present in the PEs and X is received from row 2. Step-2 updates X in row 3.

Block Update: A TRSM problem assigned to a core is solved sequentially, as shown in Figure 3.9, through multiple blocked GEMM and blocked TRSM updates.

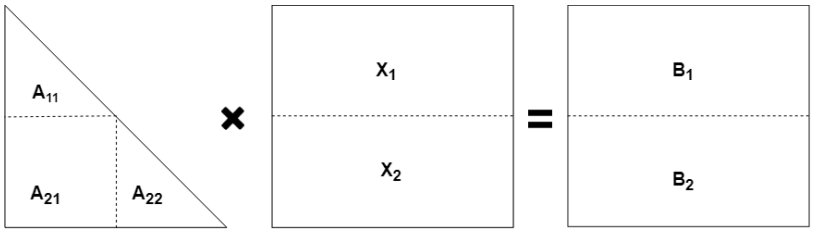

Figure 3.9: Sequential TRSM using GEMM

Step-1:	$A_{11} \times X_1 = B_1$	blocked TRSM
Step-2:	$B_2 - = A_{21} \times X_1$	blocked GEMM
Step-3:	$A_{22} \times X_2 = B_2$	blocked TRSM

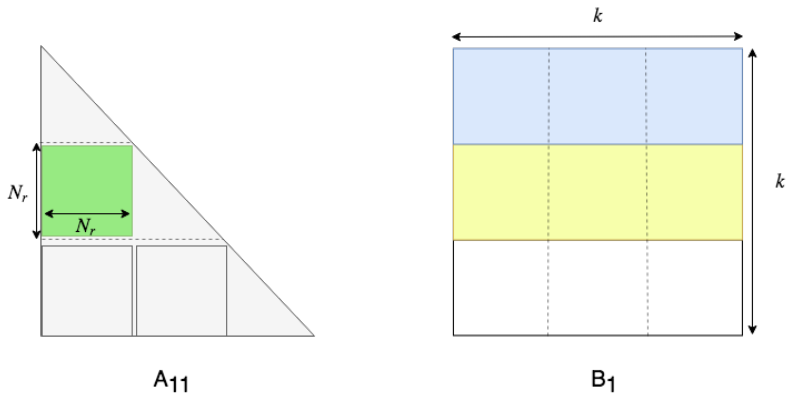

Figure 3.10: TRSM Block Updates ($k = 3N_r$)

For block TRSM update, $k \times k$ sized block of B is mapped to the local memories of the core. $N_r \times N_r$ sized panels of A are streamed in and out of the core to produce a block of X through multiple panel updates.

Figure 3.10 shows an intermediate stage of blocked TRSM where *blue* panels of B_1 have been processed to produce corresponding panels of X_1 by TRSM panel updates. The *yellow* panels of B_1 are being updated using the *blue* panels present on the core and the streaming *green* panel of A_{11} by GEMM panel updates.

3.2.4 LUD

Panel Update: An $N_r \times N_r$ sized panel of matrix A is mapped to the core, such that each $PE_{i,j}$ stores a single element of A, i.e a_{ij}. The initialization step calculates the reciprocal of a_{ii} ($r_{ii} = 1/a_{ii}$) in all diagonal PEs except the last one ($i \neq N_r$).

The reciprocal r_{ii} is then broadcast along the i^{th} column. Next, two steps [Fig. 3.11] are repeated for $N_r - 1$ iterations to obtain L and U in place of A. The elements of L or U generated in a $PE_{i,j}$ are referred to as l_{ij} or u_{ij}. Let m denote the iteration number:

Step-1: Multiplication updates in $PE_{i,m}$: $l_{im} = a_{im} \times r_{mm}$; where $i > m$

Step-2: MAC updates in all PEs $a_{ij} -= l_{im} \times u_{mj}$; where l_{im} is received through the row broadcast from column(m) and u through the column broadcast from row(m); where $i, j > m$.

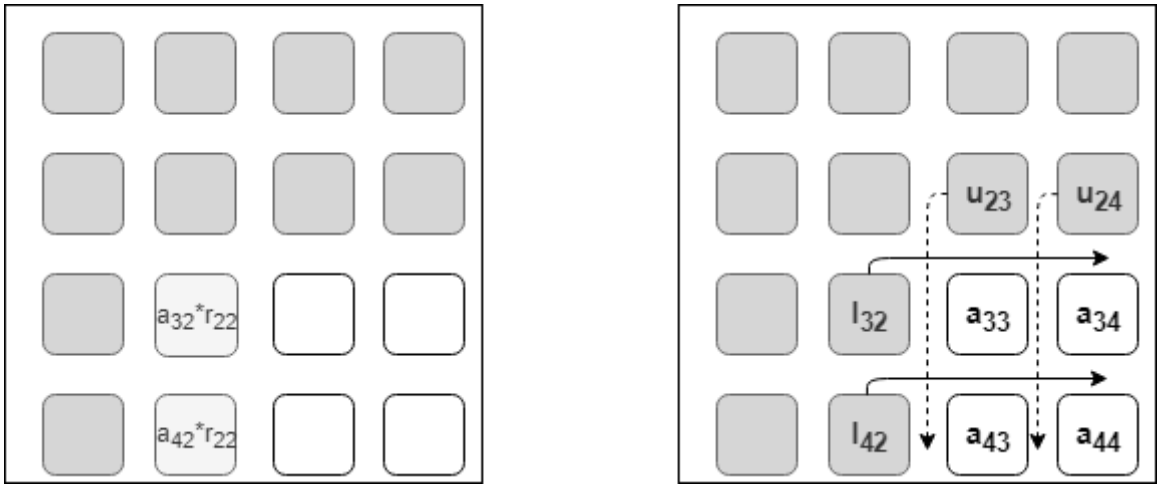

Figure 3.11: LUD Panel Update in a PE array $N_r = 4$)

It takes 2 cycles for initialization and $(N_r - 1) \times 3$ cycles for $(N_r - 1)$ iterations of *multiplication* $\rightarrow$ *broadcast* $\rightarrow$ *MAC*. Hence, $3N_r - 1$ cycles are required in total for an LUD panel update.

Figure 3.11 shows the 2^{nd} iteration($m = 2$) for a 4×4 panel update. Step-1 updates elements of the lower triangular matrix L below the diagonal $PE_{2,2}$, ie. l_{32} and l_{42}. Step-2 updates elements of matrix A: $a_{ij} -= l_{i2} \times u_{2j}$; where l_{i2} values are received through the row broadcast from column 2, $j_{2,j}$ values a re received through the column broadcast from row 2 and $i, j > 2$.

Block Update: An LUD problem assigned to a core is solved sequentially, as shown in Figure 3.12, through multiple blocked LUD, blocked TRSM and blocked GEMM updates.

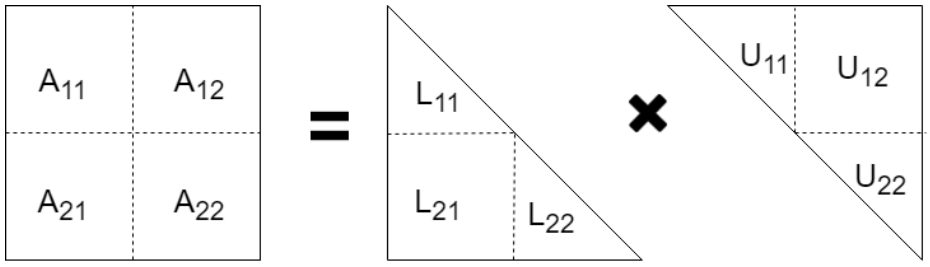

Figure 3.12: Sequential LUD using GEMM and TRSM

Step-1:	$A_{11} = L_{11} \times U_{11}$	blocked LUD
Step-2:	$L_{11} \times U_{12} = A_{12}; \quad L_{21} \times U_{11} = A_{21}$	blocked TRSM
Step-3:	$A_{22} -= L_{21} \times U_{12}$	blocked GEMM
Step-4:	$A_{22} = L_{22} \times U_{22}$	blocked LUD

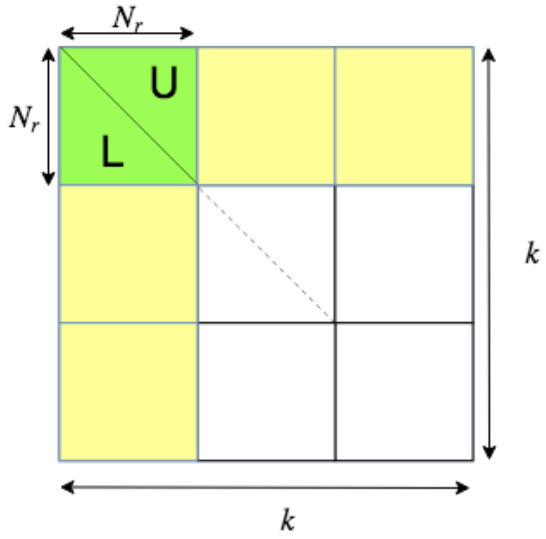

Figure 3.13: LUD Block Updates ($k = 3N_r$)

For a block LUD update, $k \times k$ sized block of A is mapped to the local memories of the core. Through a sequence of GEMM, TRSM and LUD panel updates, an in-place solution is obtained in the core. Figure 3.13 shows an intermediate step of blocked LUD where the *green* panel has been processed by LUD panel update. The *yellow* panels are being updated using the *green* panel by TRSM panel updates. This would be followed by GEMM update of *white* panels using the *yellow* and *green* panels.

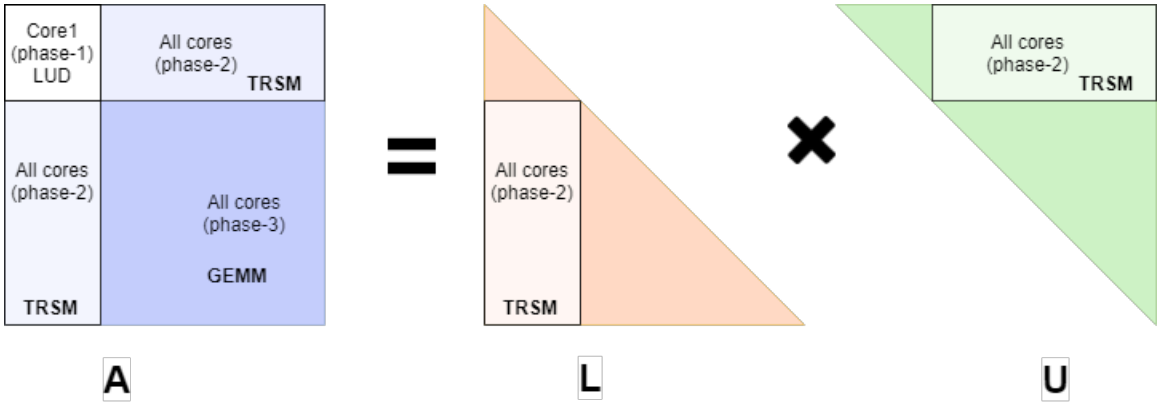

Figure 3.14: Partitioning for a large LU Decomposition problem

Partitioning: The LU Decomposition solution is inherently sequential. As such it cannot be partitioned for parallel processing using multiple cores. However, a large matrix A can be decomposed by recursively applying the sequential $LUD \rightarrow TRSM \rightarrow GEMM \rightarrow LUD$ method discussed earlier. This presents opportunity for utilizing the multi-core architecture during the TRSM and GEMM phase [Fig. 3.14]. This partitioning scheme has an imbalance during the solution of the first LUD subproblem when only one core will be used. The subsequent TRSM, GEMM and LUD sub-problems can be solved by using all the cores.

3.2.5 Matrix Inverse

A large matrix inversion problem can be solved in 3 phases: (*i*) LU Decomposition of the matrix, (*ii*) two TRSM calls for finding inverse of the lower and upper triangular matrices and (*iii*) GEMM to obtain the inverse of the original matrix.

$$A = L \times U; \qquad \text{(LUD)}$$

$$L \times X = I; \qquad U \times X = I; \qquad \text{(TRSM)}$$

$$A^{-1} = U^{-1} \times L^{-1} \qquad \text{(GEMM)}$$

This approach allows us to utilize all the cores during each of the LUD, TRSM and GEMM phases. When more than one cores are being used, the output data after each phase has to be brought back to the external memory to begin with the next phase. The time complexity of this routine is the sum of individual time complexities of LUD, TRSM and GEMM.

Sparse Matrix Compression: It is well known that sparse matrices arising from higher-order discretizations and those encountered in Finite Element Analysis exhibit a dense block substructure in which the blocks are constant-sized and aligned[12]. Such block sparse patterns are also common in small world network connectivity graphs which are globally sparse but locally dense. This suggests that preprocessing the sparse matrices into a block compressed form can be done while maintaining a high fill ratio [Eqn. 3.1] for many important problem domains.

$$Fill\ Ratio = \frac{Number\ of\ nonzero\ elements}{Number\ of\ Stored\ Values} \tag{3.1}$$

$$(Stored\ Values = nonzero\ elements + fill\ in)$$

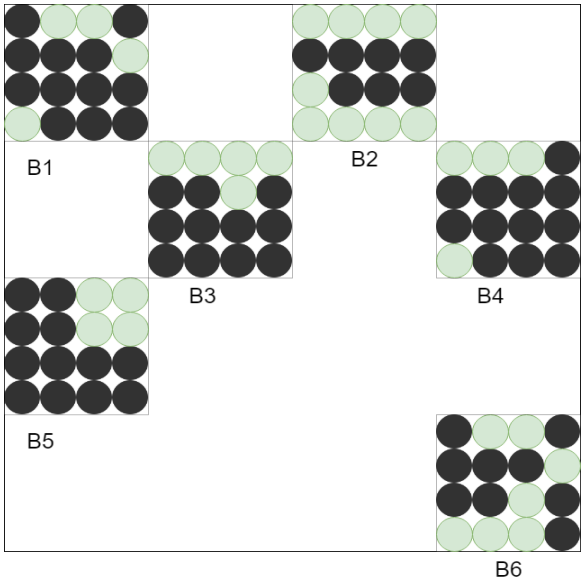

Figure 3.15: Dense 4×4 sized block creation using fill-ins

Figure 3.15 shows a sparse matrix where the *black* dots are non-zero elements. *grey* dots are fill-ins which are treated as non-zero elements in spite of being zeroes to create dense blocks. It can be seen that larger block size would lead to lower fill ratio

in general and thus poor performance. There are partitioned data structures that support distribution of a large size problem into multiple cores. Two partitioned-block compressed formats, namely, Partitioned Block Compressed Row(PBCSR) and Partitioned Block Compressed Column(PBCSC) were used for sparse matrix representation in our mapping schemes. Sparse matrices are preprocessed to create $row \times col$ sized dense blocks before partitioning; these dense blocks are referred to as *dblks* in the following sections.

PBCSR representation:	PBCSC representation:
Val : [B1, B2, B3, B4, B5, B6]	Val : [B1, B5, B3, B2, B4, B6]
Part_ptr: [0, 4]	Part_ptr: [0, 3]
BlkRow_ptr: [0, 2, 4, 5]	BlkCol_ptr: [0, 2, 3, 4]
BlkCol_id: [0, 2, 1, 3, 0, 3]	BlkRow_id: [0, 2, 1, 0, 1, 3]

3.3.1 SpMV

Partitioning: The partitioned data structure allows distribution of a large problem among multiple cores as shown in Figure 3.16. Dense vector b is shared among the cores. Equal load distribution is not guaranteed in this method but no synchronization is required between the cores. The central controller is responsible for load balancing that might arise because of unequal partitions.

Panel Update: With $row = col = N_r$ for the *dblks* in the compressed format, the $N_r^2 \times 1$ vector update for SpMV can be achieved by N_r columns of PEs inside the core as in the GEMV panel update [Fig. 3.3]. It takes $N_r + 1$ cycles for the SpMV panel update. However since we are working with a compressed data structure of A, vector c would be corrupted if multiple columns of the PE array try to update the same elements of c. The local controller would need to stall the PEs in such cases

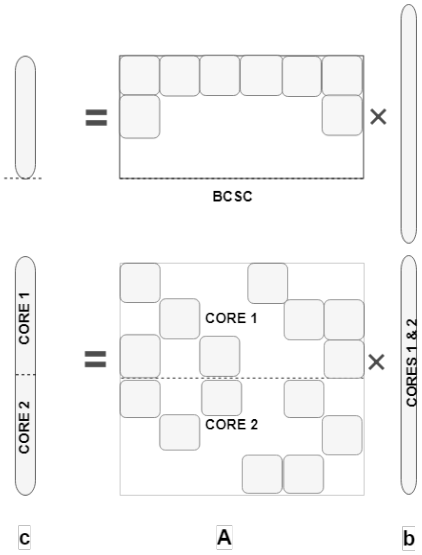

Figure 3.16: Partitioning SpMV input to multiple cores

and this could lead to $2N_r$ cycles for SpMV panel update in the worst case.

Block Update: Since our core can only work on small panels, a large problem is broken down as shown in Figure 3.17. To explain the decomposition, we start with a subproblem $c_1 = A_1 \times b_1$ which has been assigned to a single core; where A_1 is stored in BCSC format. The local controller stores the BlkCol_ptr and BlkRow_id which is used for index matching when loading elements of c' to the core. The product dense vector c_1 is computed by multiple iterations of sparse matrix vector multiplication. In each iteration, multiple $row \times col$ sized $dblks$ of A_1 and col elements of b_1 are processed to produce a partial product vector c'. The partial product is accumulated over multiple iterations to produce the final product vector c_1.

Similar to GEMV block update, the extended panel update operation for SpMV inside a column of PEs: $c_{N_r \times 1} = A_{N_r \times k} \times b_{k \times 1}$; where $k \geq 2N_r$ allows to hide the time taken to transfer elements of c'. However here we strictly limit k to $2N_r$ because a larger size of $dblks$ would lead to smaller fill-ratio which is not favourable.

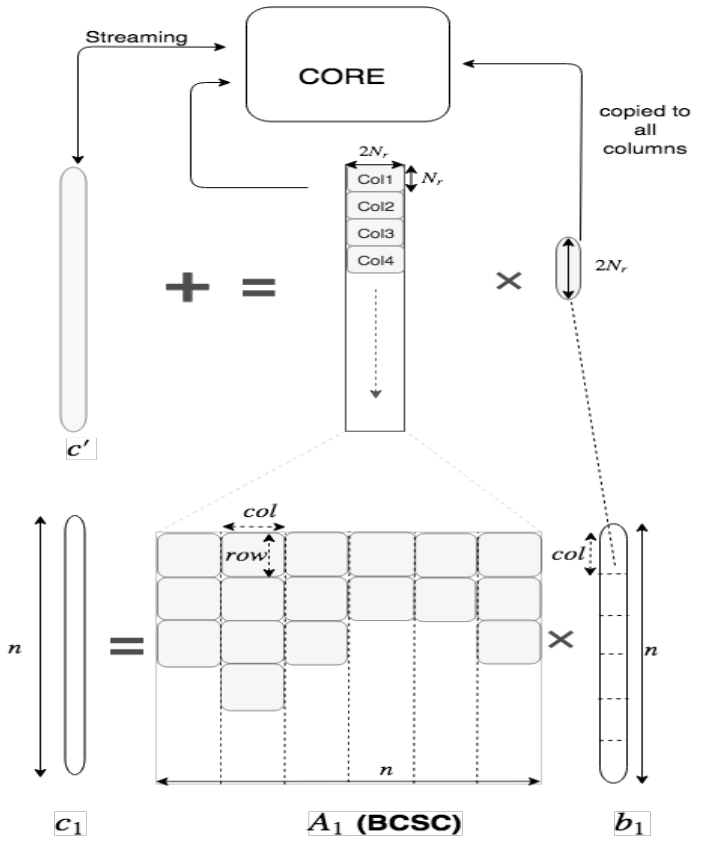

Figure 3.17: SpMV Mapping for Block Updates ($row = N_r$ and $col = 2N_r$)

3.3.2 SpMM

Partitioning: The partitioned data structure allows distribution of a large problem among multiple cores as shown in Figure 3.18. The dense output matrix C is distributed among the cores. Similar to SpMV, the central controller is responsible for load balancing in case of unequal partitions.

Panel Update: With $row = col = N_r$ for *dblks* in the compressed format, the $N_r \times N_r$ matrix update for SpMM is performed in the core as in GEMM panel update [Fig. 3.6]. It takes $N_r + 1$ cycles for the SpMM panel update.

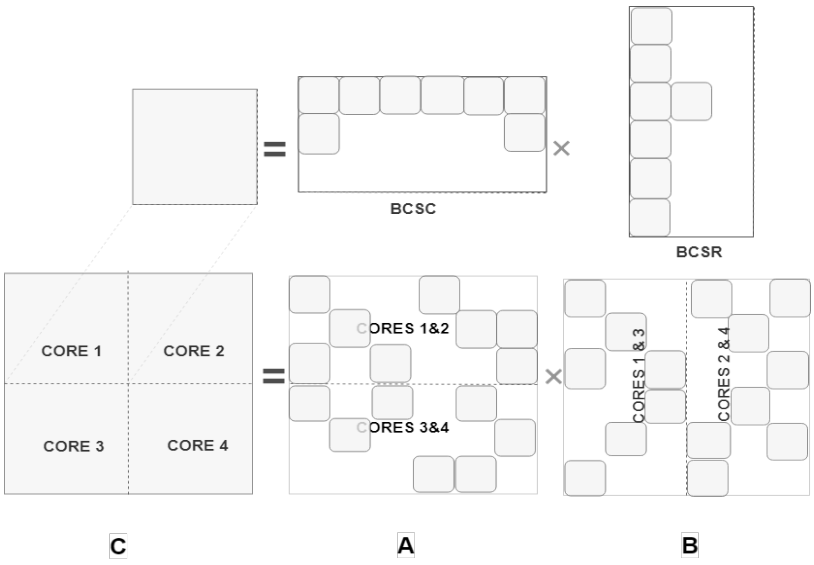

Figure 3.18: Partitioning SpMM input to multiple cores

Block Update: Since our core is designed to work on small panels, a large problem is decomposed as shown in Figure 3.19. We start with a subproblem $C_1 = A_1 \times B_1$ which has been assigned to a core; where A_1 is stored in BCSC format and B_1 in BCSR format. The local controller stores the BlkCol_ptr and BlkRow_id for matrix A and the BlkRow_ptr and BlkCol_id for matrix B, which is used for index matching when loading elements of C' to the core. The product matrix C is computed in multiple iteration of sparse matrix matrix multiplication. In each iteration, multiple $row_A \times col_A$ sized $dblks$ of A_1 and multiple $row_B \times col_B$ sized $dblks$ of B_1 are processed to produce a partial product matrix C'. This partial product is accumulated over multiple iterations to produce the final matrix C_1.

Similar to GEMM block update, the extended panel update operation for SpMM inside the core is given by: $C_{N_r \times N_r} = A_{N_r \times m} \times B_{m \times N_r}$, where $m \geq 2N_r$ allows to hide the time taken to transfer elements of C'. Again we strictly limit m to $2N_r$ because a larger size of $dblks$ would lead to smaller fill-ratio.

During a single iteration of the SpMM block update, $dblks$ in a compressed column

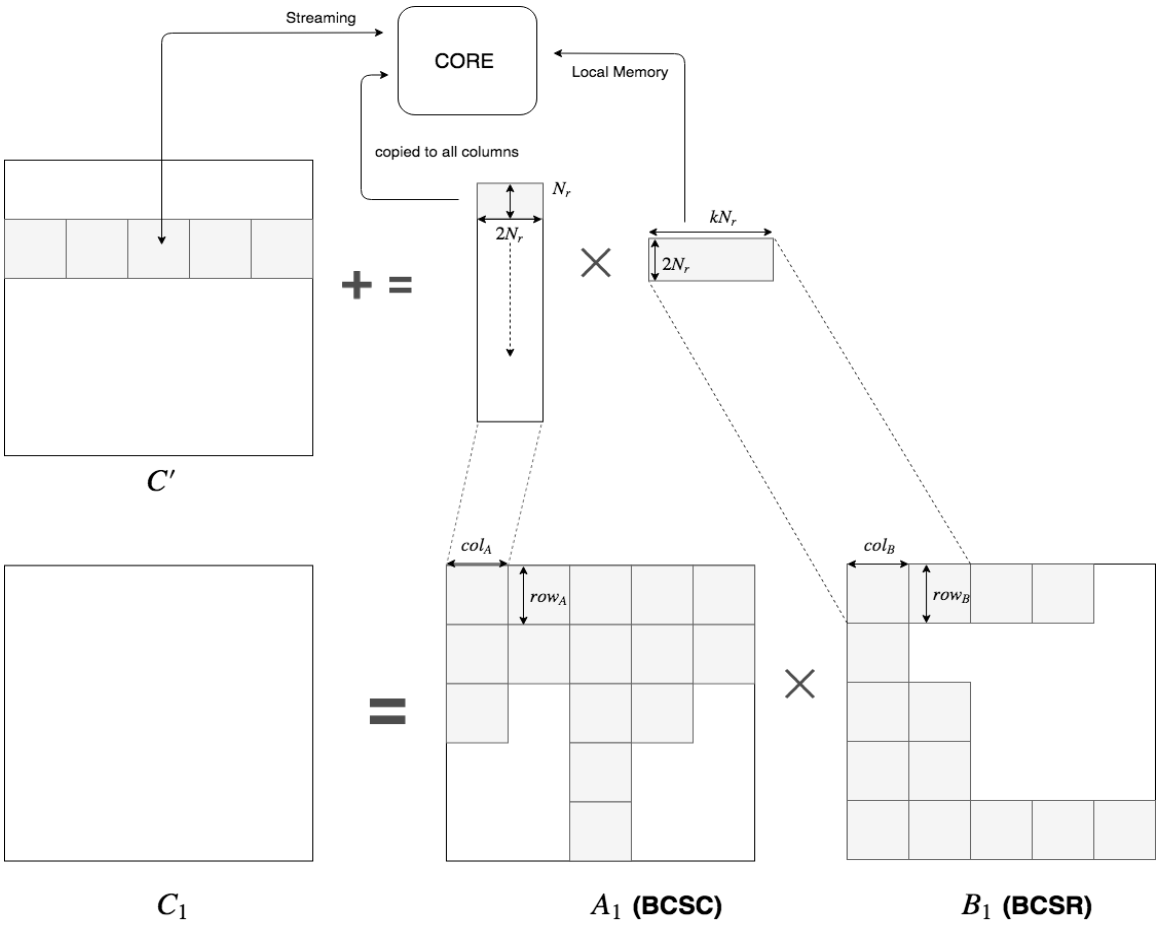

Figure 3.19: SpMM Mapping for Block Updates ($row_A = col_B = N_r$ and $col_A = row_B = 2N_r$)

of matrix A_1 are accessed repeatedly if all the *dblks* of the corresponding compressed row of matrix B_1 are not present inside the core. Hence we map a large number (k) of *dblks* to the core in order to reduce repeated access to *dblks* of A_1, where k is dependent on the available local memory in the PEs in a core.

Chapter 4

DESIGN TRADE-OFFS

In this chapter we investigate the trade-offs associated with the architecture in terms of number of PEs in the core, local memory size and the bandwidth between the on-chip memory and the core. Hardware implementation results for a single core have also been presented later in the chapter which show the feasibility of our design.

4.1 Core Configuration (N_r)

A higher count of PEs inside a core provides more parallelism by running $N_r \times N_r$ computations in parallel. However all the PEs may not be active in every cycle. This means that all the execution units available inside a core would not be utilized in every cycle. To ensure that PEs are not under-utilized for the selected core configuration, we studied the PE Utilization Ratio for each of the panel updates in the PE array.

$$PE_{Util} = \frac{\sum_{k=1}^{p} no.\ of\ active\ PEs\ in\ the\ k^{th}\ cycle}{p \times N_r^2} \tag{4.1}$$

where p is the no. of cycles required for an $N_r \times N_r$ panel update. ($N_r \geq 2$)

The PE_{Util} is 1 for all the panel updates with a single PE inside a core ($N_r = 1$). We evaluated the PE_{Util} based on Equation 4.1 for each of the kernels [Appendix]. The results have been shown in Table 4.1. Based on these results we obtained a plot that shows the variation of PE_{Util} with the number of processing elements.

Kernel	p (cycles for panel update)	PE_{Util} (utilization ratio)
GEMV	$N_r + 1$	1
GEMM	$N_r + 1$	1
TRSM	$3N_r$	$\frac{2+N_r}{3N_r}$
LUD	$3N_r - 1$	$\approx \frac{3+2N_r}{3(3N_r-1)}$
INV	$10N_r$	$\frac{18+11N_r}{30N_r}$
SpMV	$N_r + 1$	1
SpMM	$N_r + 1$	1

Table 4.1: PE Utilization Ratio

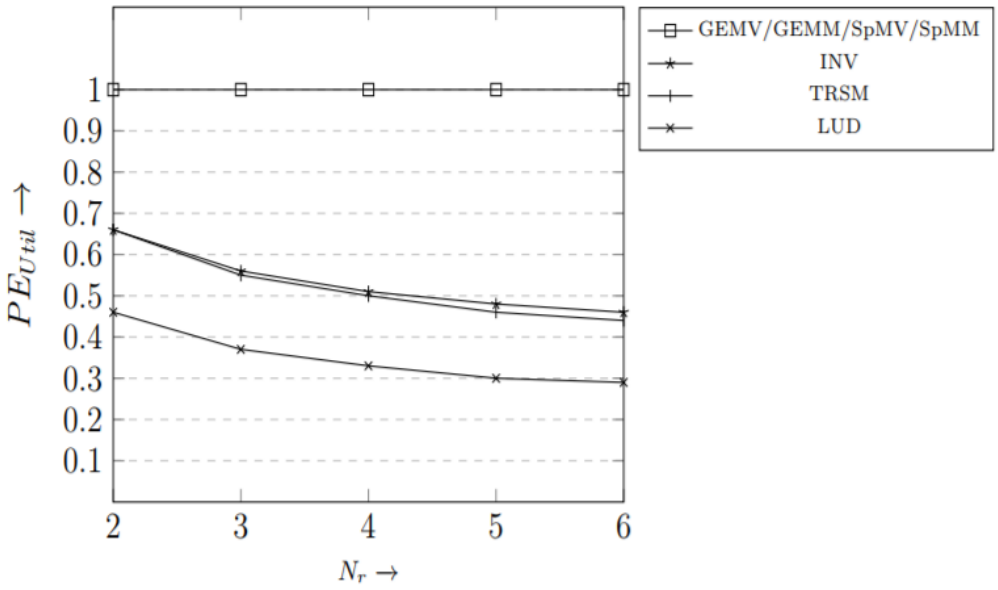

Figure 4.1: Effect of Core Configuration on PE Utilization

As the PE count increases, we observe a deterioration in the PE_{util} for TRSM, LUD, and INV kernels [Fig. 4.1]. Thus, a lower value of N_r such as 2, 3 or 4 ensures a higher PE utilization. We selected $N_r = 4$ to maintain a minimum PE_{util} of 0.33 for LUD, and ≈ 0.5 for TRSM and INV. This configuration allows 16 parallel MAC updates for GEMV, GEMM, SpMV and SpMM.

Another argument in support of a lower value of N_r arises from the block compressed data structure where the dense block ($dblks$) size was fixed as ($N_r \times 2N_r$). Thus, keeping N_r low ensures higher fill ratio [Equation 3.1] for any arbitrary sparse matrix where dense block sub-structures do not exist naturally.

The updates inside a core assumed that the on-chip memory could transfer data to/from the core through row broadcast busses at N_r elements/cycle [Section 3.2]. Hence the bandwidth between on-chip memory and the core for $N_r = 4$ is $BW_{in} = 4$ elements/cycle.

4.2 Memory Size

The Local Memory (LM) acts as a buffer and supports data reuse. A larger LM_{size} reduces the time spent in accessing data and thus increases a core's utilization.

$$CORE_{Util} = \frac{Comp_{cycles}}{Comp_{cycles} + Comm_{cycles}} \tag{4.2}$$

The block update techniques described in Chapter-3, indicate that the performance for most of the kernels is largely dictated by that of GEMM. Thus, for selecting the optimal LM_{size} we maximize $CORE_{Util}$ for GEMM. Figure 3.7 showed that $n \times k$ MAC updates for matrix C requires a transfer of $2N_r \times k$ elements of B and $n \times 2N_r$ elements of A.

$$
\begin{aligned}
Comm_{cycles} &= \frac{(n \times 2N_r + 2N_r \times k)elements}{BW_{in}} \\
&= \frac{(2n + 2k)(N_r)elements}{N_r \frac{elements}{cycle}} \\
&= (2n + 2k)\ cycles \\
Comp_{cycles} &= \frac{2nk}{N_r} \\
CORE_{Util} &= \frac{\frac{nk}{N_r}}{n + k + \frac{nk}{N_r}}
\end{aligned}
$$

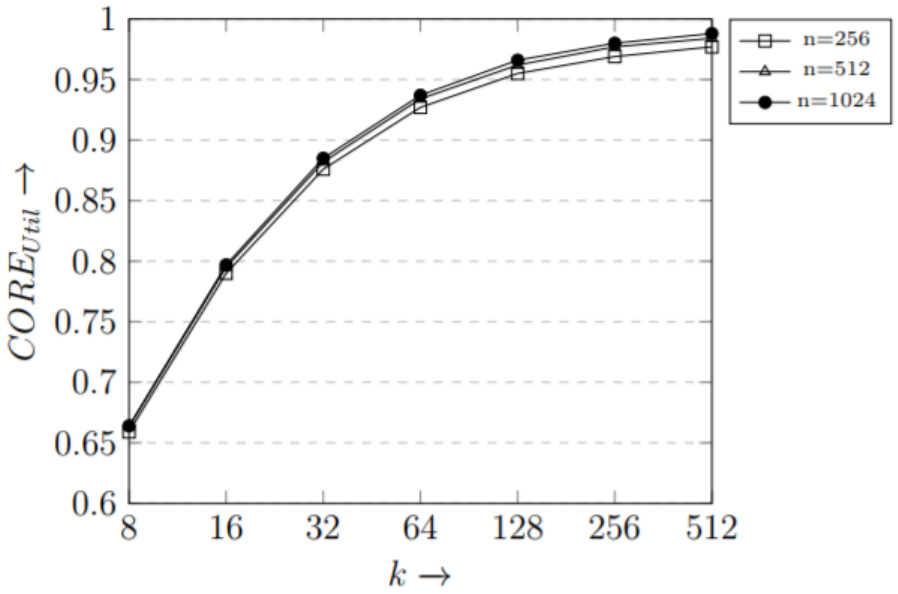

Figure 4.2: Effect of Local Memory size on Core Utilization

For GEMM updates, $LM_{size} = \frac{2k}{N_r}$ elements per PE. We plotted the effect of LM_{size} on $CORE_{Util}$ for $n = 256$, 512 and 1024 with $N_r = 4$. We choose $k = 512$ since it provides a very high $CORE_{Util}$ [Fig. 4.2].

4.3 Bandwidth vs Local Memory Size

The previous section assumed a fixed bandwidth $BW_{in} = N_r$ elements per cycle. If the bandwidth is kept variable, Equation 4.2 reduces to the following:

$$CORE_{Util} = \frac{\frac{nk}{N_r}}{\frac{(n+k)N_r}{BW_{in}} + \frac{nk}{N_r}}$$

We fix $N_r = 4$ and a $CORE_{Util} = 0.99$ in the above equation and plot the effect of $LM_{size}(= \frac{2k}{N_r} = \frac{k}{2})$ on bandwidth BW_{in} for multiple values of n considering a GEMM update as earlier.

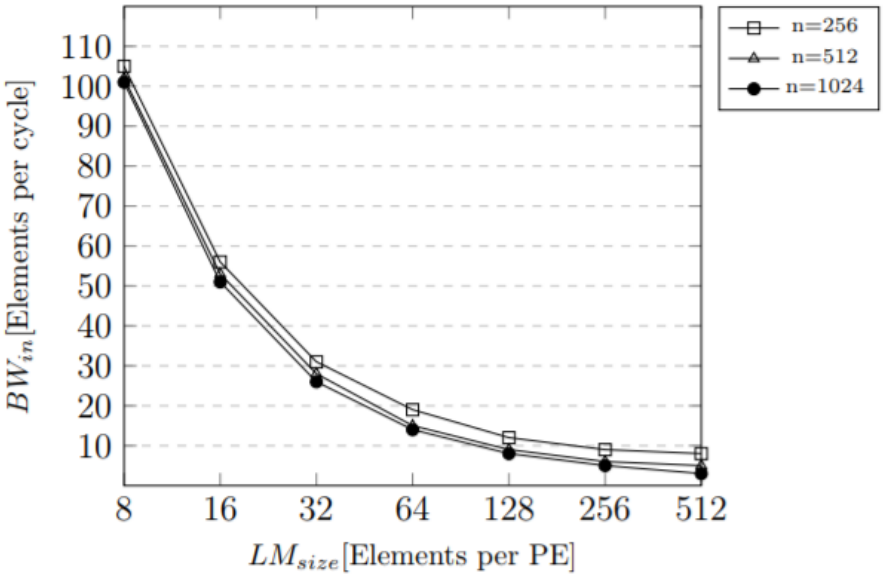

Figure 4.3: Local memory size vs Bandwidth

Thus, it can be seen from Figure 4.3 that a very high core utilization can be maintained with a smaller local memory size, provided there is sufficient bandwidth between the core and its on-chip memory.

Element Size	4 Bytes
PE Count	4×4
Register File Size	64 Bytes
Local Memory Size	1 KB
On-chip Memory Size	1 MB
Memory Bandwidth	16 Bytes/cycle

Table 4.2: Core Configurations

We fix $LM_{size} = 256$ elements per PE or $k = 512$. Also, based on the mappings explained in Chapter-3, the maximum number of elements stored in the register file is always below $3N_r = 12$, so we fix the Register File as 16 elements wide (a power of 2). We also fix the on-chip memory size as 1 MB, which is sufficient to store square matrices A, B and C of sizes as large as 256×256. Table 4.2 summarizes an optimal core configuration for our accelerator.

4.4 Hardware Implementation

A 4×4 PE array capable of performing panel updates for all the 7 kernels was implemented in SystemVerilog. Local memory was not required for implementing the panel updates. The implementation was a flat design. The PEs contain an ALU, 8 registers (sufficient to support all the panel updates) and an FSM controller. The reciprocal operation is considered as a single cycle read from an LUT stored inside the PE, however the LUT was not used in the RTL design. The FSM controller inside each PE produces a unique control word every cycle. The control word depends on the particular kernel update in progress and a PE's location in the 2D array. Thus, the datapath is reconfigured for running different panel updates inside the core.

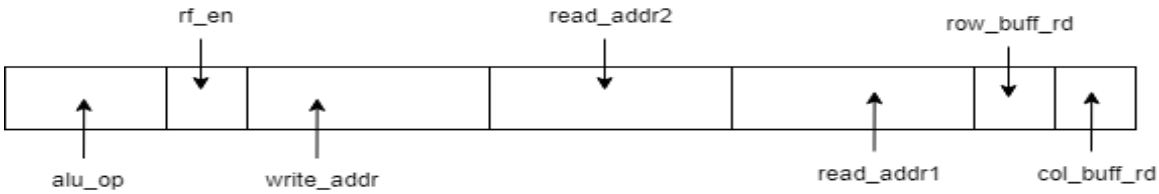

Figure 4.4: Control Word

A control word is 14 bits long [Fig. 4.4]. It has 2 control bits for the column buffer and the row buffer to read from or write to the broadcast busses. There are 6 bits for two read addresses and 3 bits for one write address of the register file which also has 1 bit for the enable signal. There are 2 bits for controlling the ALU operation.

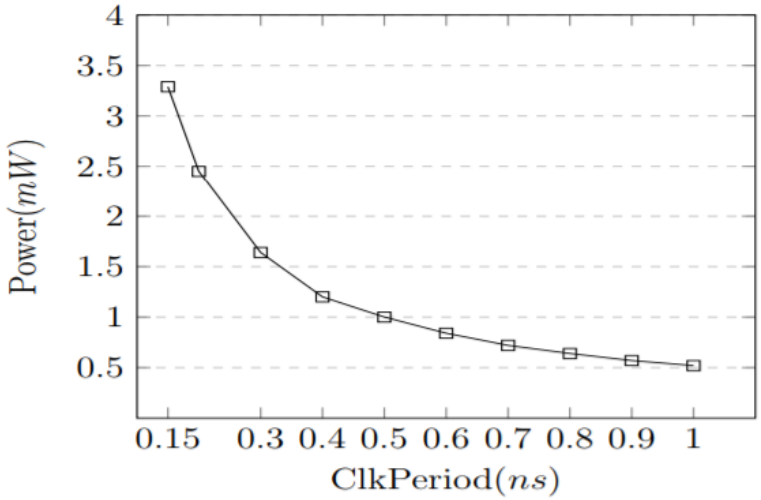

Figure 4.5: Power vs Delay

We verified the functionality of the core based on 32bit integer values using simulation results. The core was also synthesized using a $7nm$ PDK and a clock period of $500ps$ was chosen based on the power-delay curve [Fig. 4.5]. The power consumption was $1mW$ and area for the core (excluding Local Memory and LUT) was $0.076mm^2$. Thus, with 16 parallel MAC units a single core clocked at $500ps$ can achieve a maximum throughput of 32 GOPS.

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

A multicore architecture capable of executing multiple dense and sparse linear algebra computations, namely GEMV (Dense General Matrix Vector Multiplication), GEMM (Dense General Matrix Matrix Multiplication), TRSM (Triangular Matrix Solver), LU Decomposition, Matrix Inverse, SpMV (Sparse Matrix Vector Multiplication), SpMM (Sparse Matrix Matrix Multiplication) was presented. The selected linear algebra kernels represent both Level-2 and Level-3 dense and sparse BLAS. Table 5.1 summarizes the scope of this work and compares it with competing implementations such as [32], [25], [5], and [39].

The unified architecture consists of multiple cores which are used to perform 4×4 sized panel updates for the selected kernels. Multiple panel updates are executed sequentially by the core to solve a larger problem size. Mapping and scheduling schemes for each of the kernels was presented which ensured that the architecture can be used to work with any given input size.

The detailed analysis of the selected kernels showed that they all relied on the same execution units, i.e a MAC unit and a reciprocal unit. Optimization techniques such as data blocking and prefetching were used.

Design trade-offs related to the number of PEs inside a core, the local memory size inside each PE and the bandwidth between the on-chip memory and the core were analyzed. Based on the trade-offs, PE array utlization and core utilizations were maximized and an optimal core configuration was derived. An RTL level im-

plementation for a single core with 4×4 processing elements was done and results for the panel updates were verified. Synthesis results using a $7nm$ PDK were used to demonstrate that a single core could achieve a performance of upto 32GOPS.

Kernels	Linear Algebra Processor [32]	REDEFINE CGRA [25]	Dense-Sparse Matrix Vector Unit [5]	Layers CGRA [39]	Our Design
GEMV		✓	✓	✓	✓
GEMM	✓	✓		✓	✓
TRSM	✓				✓
SYRK	✓				
LU Decomposition	✓			✓	✓
QR Decomposition	✓	✓			
Cholesky Decomposition	✓				
Matrix Inverse					✓
SpMV			✓		✓
SpMM					✓

Table 5.1: Existing Linear Algebra Accelerators

5.2 Future Work

The work presented in this thesis is an initial exploration of a unified architecture design for multiple matrix computation kernels.

- The design presented in this work can be extended to target a specific domain of application by selecting the specific algorithms used in that domain and systematically mapping them to this architecture.
- Using the already mapped kernels more complex algorithms can be implemented with minimum software overhead, e.g Conjugate Gradient algorithm can be performed on this architecture using the SpMV mapping.
- Mapping more bandwidth limited kernels from Level-1 BLAS could transform the architecture to a unified BLAS accelerator with the addition of Floating point arithmetic units.
- The processing elements can be optimized for performance by pipelining the execution units and increasing the bandwidth can allow for more computation communication overlap.
- The partitioning schemes for LUD, and Matrix Inverse can be improved to reduce the transfer of intermediate results from the on-chip memory to the external memory.
- The design of a local controller was not explored in this work and it must be studied in future extensions to present a realistic power and performance analysis of the entire core.

REFERENCES

[1] Alle, M., K. Varadarajan, A. Fell, R. R. C., N. Joseph, S. Das, P. Biswas, J. Chetia, A. Rao, S. K. Nandy and R. Narayan, "Redefine: Runtime reconfigurable polymorphic asic", ACM Trans. Embed. Comput. Syst. **9**, 2, 11:1–11:48 (2009).

[2] Anderson, E., Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof and D. Sorensen, "Lapack: A portable linear algebra library for high-performance computers", in "Proceedings of the 1990 ACM/IEEE Conference on Supercomputing", Supercomputing '90, pp. 2–11 (1990).

[3] Blackford, L. S., J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry and D. Walker, "Scalapack: A portable linear algebra library for distributed memory computers - design issues and performance", in "Proceedings of the 1996 ACM/IEEE Conference on Supercomputing", Supercomputing '96 (1996).

[4] Buttari, A., J. Langou, J. Kurzak and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures", Parallel Comput. **35**, 1, 38–53, URL `http://dx.doi.org/10.1016/j.parco.2008.10.002` (2009).

[5] Calderon, H. and S. Vassiliadis, "Reconfigurable fixed point dense and sparse matrix-vector multiply/add unit", in "IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)", pp. 311–316 (2006).

[6] Cheema, U. I., G. Nash, R. Ansari and A. A. Khokhar, "Invarch: A hardware eficient architecture for matrix inversion", in "2015 33rd IEEE International Conference on Computer Design (ICCD)", pp. 180–187 (2015).

[7] Dongarra, J., M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with gpus", Numerical Computations with GPUs pp. 1–26 (2014).

[8] Dongarra, J. and D. Walker, "Software libraries for linear algebra computations on high performance computers", SIAM Review **37**, 2, 151–180, URL `https://doi.org/10.1137/1037042` (1995).

[9] Dongarra, J. J., J. Du Croz, S. Hammarling and I. S. Duff, "A set of level 3 basic linear algebra subprograms", ACM Trans. Math. Softw. **16**, 1, 1–17, URL `http://doi.acm.org/10.1145/77626.79170` (1990).

[10] Dongarra, J. J., J. Du Croz, S. Hammarling and R. J. Hanson, "An extended set of fortran basic linear algebra subprograms", ACM Trans. Math. Softw. **14**, 1, 1–17, URL `http://doi.acm.org/10.1145/42288.42291` (1988).

[11] Duff, I. S., M. A. Heroux and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum", ACM Trans. Math. Softw. **28**, 2, 239–267, URL http://doi.acm.org/10.1145/567806.567810 (2002).

[12] Eberhardt, R. and M. Hoemmen, "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures", in "2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)", pp. 663–672 (2016).

[13] Elmroth, E., F. Gustavson, I. Jonsson and B. Kgstrm, "Recursive blocked algorithms and hybrid data structures for dense matrix library software", SIAM Review **46**, 1, 3–45, URL https://doi.org/10.1137/S0036144503428693 (2004).

[14] Gonzalez, J. and R. C. Nez, "Lapackrc: Fast linear algebra kernels/solvers for fpga accelerators", Journal of Physics: Conference Series **180**, 1, 012042 (????).

[15] Goto, K. and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication", ACM Trans. Math. Softw. **34**, 3, 12:1–12:25 (2008).

[16] Greathouse, J. L., K. Knox, J. Po, K. Varaganti and M. Daga, "clsparse: A vendor-optimized open-source sparse blas library", in "Proceedings of the 4th International Workshop on OpenCL", IWOCL '16, pp. 7:1–7:4 (ACM, New York, NY, USA, 2016), URL http://doi.acm.org/10.1145/2909437.2909442.

[17] Guo, S., Y. Dou, Y. Lei and G. Wu, "A deeply-pipelined fpga-based spmv accelerator with a hardware-friendly storage scheme", IEICE Electronics Express **12**, 11, 20150161–20150161 (2015).

[18] Intel, "Intel® MKL, Developer Reference", URL https://software.intel.com/en-us/articles/mkl-reference-manual (2009).

[19] Jouppi, N. P., C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit", in "Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on", pp. 1–12 (IEEE, 2017).

[20] Kågström, B., P. Ling and C. van Loan, "Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark", ACM Trans. Math. Softw. **24**, 3, 268–302, URL http://doi.acm.org/10.1145/292395.292412 (1998).

[21] Khan, F. A., R. A. Ashraf, Q. H. Abbasi and A. A. Nasir, "Resource efficient parallel architectures for linear matrix algebra in real time adaptive control algorithms on reconfigurable logic", in "2008 Second International Conference on Electrical Engineering", pp. 1–9 (2008).

[22] Lawson, C. L., R. J. Hanson, D. R. Kincaid and F. T. Krogh, "Basic linear algebra subprograms for fortran usage", ACM Trans. Math. Softw. **5**, 3, 308–323, URL http://doi.acm.org/10.1145/355841.355847 (1979).

[23] Mahfoudhi, R., S. Achour and Z. Mahjoub, "Parallel triangular matrix system solving on cpu-gpu system", in "2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)", pp. 1–6 (2016).

[24] Merchant, F., A. Maity, M. Mahadurkar, K. Vatwani, I. Munje, M. Krishna, S. Nalesh, N. Gopalan, S. Raha, S. K. Nandy and R. Narayan, "Micro-architectural enhancements in distributed memory cgras for lu and qr factorizations", in "2015 28th International Conference on VLSI Design", pp. 153–158 (2015).

[25] Merchant, F., T. Vatwani, A. Chattopadhyay, S. Raha, S. K. Nandy and R. Narayan, "Accelerating BLAS on custom architecture through algorithm-architecture co-design", CoRR **abs/1610.06385**, URL `http://arxiv.org/abs/1610.06385` (2016).

[26] Nvidia, "CUBLAS Library User Guide", URL `http://docs.nvidia.com/cublas` (2016).

[27] Nvidia, "CUSPARSE Library User Guide", URL `https://docs.nvidia.com/cuda/cusparse/index.html` (2018).

[28] Pal, S., J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator", in "2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)", pp. 724–736 (2018).

[29] Papa, G. and J. Šilc, "Linear algebra in one-dimensional systolic arrays", Informatica **24**, 2, 249–257 (2000).

[30] Peddawad, S. C. and A. Goel, "Matrix-matrix multiplication using systolic array architecture in bluespec team", (2015).

[31] Pedram, A., "Algorithm/architecture codesign of low power and high performance linear algebra compute fabrics", in "2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum", pp. 2214–2217 (2013).

[32] Pedram, A., A. Gerstlauer and R. A. v. d. Geijn, "A high-performance, low-power linear algebra core", in "ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors", pp. 35–42 (2011).

[33] Pedram, A., A. Gerstlauer and R. A. van de Geijn, "Floating point architecture extensions for optimized matrix factorization", in "2013 IEEE 21st Symposium on Computer Arithmetic", pp. 49–58 (2013).

[34] Pedram, A., R. A. van de Geijn and A. Gerstlauer, "Codesign tradeoffs for high-performance, low-power linear algebra architectures", IEEE Transactions on Computers **61**, 12, 1724–1736 (2012).

[35] Prasanna, V. K. and G. R. Morris, "Sparse matrix computations on reconfigurable hardware", Computer **40**, 3, 58–64 (2007).

[36] Rákossy, Z. E., D. Stengele, A. Acosta-Aponte, S. Chafekar, P. Bientinesi and A. Chattopadhyay, "Scalable and efficient linear algebra kernel mapping for low energy consumption on the layers cgra", in "International Symposium on Applied Reconfigurable Computing", pp. 301–310 (Springer, 2015).

[37] Rosado, A., T. Iakymchuk, M. Bataller and M. Wegrzyn, "Hardware-efficient matrix inversion algorithm for complex adaptive systems", in "2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)", pp. 41–44 (2012).

[38] Rkossy, Z. E., T. Naphade and A. Chattopadhyay, "Design and analysis of layered coarse-grained reconfigurable architecture", in "2012 International Conference on Reconfigurable Computing and FPGAs", pp. 1–6 (2012).

[39] Rkossy, Z. E., D. Stengele, G. Ascheid, R. Leupers and A. Chattopadhyay, "Exploiting scalable cgra mapping of lu for energy efficiency using the layers architecture", in "2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)", pp. 337–342 (2015).

[40] Schwarzenberg-Czerny, A., "On matrix factorization and efficient least squares solution.", **110**, 405 (1995).

[41] Strassen, V., "Gaussian elimination is not optimal", Numer. Math. **13**, 4, 354–356, URL `http://dx.doi.org/10.1007/BF02165411` (1969).

[42] Volkov, V. and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra", in "SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing", pp. 1–11 (2008).

[43] W. M. Gentleman, H. T. K., "Matrix triangularization by systolic arrays", URL `https://doi.org/10.1117/12.932507` (1982).

[44] Wang, L., W. Wu, Z. Xu, J. Xiao and Y. Yang, "Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing", in "Proceedings of the 2016 International Conference on Supercomputing", ICS '16, pp. 20:1–20:11 (2016).

[45] Whaley, R. C. and J. J. Dongarra, "Automatically tuned linear algebra software", in "Proceedings of the 1998 ACM/IEEE Conference on Supercomputing", SC '98, pp. 1–27 (IEEE Computer Society, Washington, DC, USA, 1998), URL `http://dl.acm.org/citation.cfm?id=509058.509096`.

[46] Zhu, Q., T. Graf, H. E. Sumbul, L. Pileggi and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware", in "2013 IEEE High Performance Extreme Computing Conference (HPEC)", pp. 1–6 (2013).

APPENDIX

PE UTILIZATION RATIO FOR PANEL UPDATES

In this appendix, we provide the mathematical derivations of the PE utilization ratios for panel updates of GEMV, GEMM, SpMV, SpMM, TRSM, LUD, and INV.

$$PE_{Util} = \frac{\sum_{k=1}^{p} no.\ of\ active\ PEs\ in\ the\ k^{th}\ cycle}{p \times N_r^2}$$

where p is the no. of cycles required for an $N_r \times N_r$ panel update. ($N_r \geq 2$)

GEMV/GEMM/SpMV/SpMM: For these kernels, $p = N_r + 1$ and all the PEs are utilized throughout the entire panel update operation.

$$\begin{aligned} PE_{Util} &= \frac{\sum_{k=1}^{N_r+1}(N_r \times N_r)}{(N_r+1) \times N_r^2} \\ &= 1 \qquad \text{(for GEMV/GEMM/SpMV/SpMM)} \end{aligned}$$

TRSM: For a panel update of this kernel, $p = 3N_r$. It takes 3 cycles for initialization. In the 1^{st} cycle N_r diagonal PEs are active, in the 2^{nd} cycle all the PEs are active and in the 3^{rd} cycle only the first row of PEs is active. After the initialization, two steps repeated for $N_r - 1$ iteration. In an iteration i, broadcasts are required for the first step during which all PEs of row r ($r \geq i$) are active, all PEs below row (i) are active in the first step and all PEs of row ($i + 1$) are active in the second step.

$$
\begin{aligned}
PE_{Util} &= \frac{N_r + N_r^2 + N_r + \sum_{i=1}^{N_r-1}(N_r(N_r+1-i) + N_r(N_r-i) + N_r)}{(3N_r) \times N_r^2} \\
&= \frac{2N_r + N_r^2 + N_r(\sum_{i=1}^{N_r-1}((N_r+1-i) + (N_r-i) + 1))}{(3N_r) \times N_r^2} \\
&= \frac{2N_r + N_r^2 + 2N_r(\sum_{i=1}^{N_r-1}(N_r+1-i))}{(3N_r) \times N_r^2} \\
&= \frac{2N_r + N_r^2 + 2N_r(2+3+4+..+N_r)}{(3N_r) \times N_r^2} \\
&= \frac{2N_r + N_r^2 + N_r(N_r^2+N_r-2)}{(3N_r) \times N_r^2} \\
&= \frac{2N_r + N_r^2 + N_r^3 + N_r^2 - 2N_r}{(3N_r) \times N_r^2} \\
&= \frac{2+N_r}{3N_r} \quad \text{(for TRSM)}
\end{aligned}
$$

LUD: For a panel update of this kernel, $p = 3N_r - 1$. It takes 2 cycles for initialization. In the 1^{st} cycle $N_r - 1$ diagonal PEs are active and in the 2^{nd} cycle all the PEs below these $N_r - 1$ diagonal PEs are active. After the initialization, two steps repeated for $N_r - 1$ iteration. In an iteration i all PEs below the diagonal PE(i, i) are active in the first step, broadcasts are required for the second step during which the sub-array of PEs bounded by row$(i-1)$ and column$(i-1)$ is active except the diagonal PE(i, i). In the second step, the sub-array of PEs bounded by row(i) and column(i) is active.

$$
\begin{aligned}
PE_{Util} &= \frac{N_r - 1 + (2 + 3 + 4 + .. + N_r) + \sum_{i=1}^{N_r-1}((N_r - i) + (N_r + 1 - i)^2 - 1 + i^2)}{(3N_r - 1) \times N_r^2} \\
&= \frac{N_r - 1 - 1 + \sum_{i=1}^{N_r}(i) + \sum_{i=1}^{N_r-1}(i) + \sum_{i=1}^{N_r-1}((N_r + 1 - i)^2 + i^2) - N_r + 1}{(3N_r - 1) \times N_r^2} \\
&= \frac{\sum_{i=1}^{N_r}(i) + \sum_{i=1}^{N_r-1}(i) + \sum_{i=1}^{N_r-1}((N_r + 1 - i)^2 + i^2) - 1}{(3N_r - 1) \times N_r^2} \\
&= \frac{N_r^2 + \sum_{i=1}^{N_r-1}((N_r + 1 - i)^2 + i^2) - 1}{(3N_r - 1) \times N_r^2} \\
&= \frac{N_r^2 + \sum_{i=2}^{N_r}(i^2) + \sum_{i=1}^{N_r-1}(i^2) - 1}{(3N_r - 1) \times N_r^2} \\
&= \frac{N_r^2 - 1 + \frac{2N_r^3 + N_r}{3}}{(3N_r - 1) \times N_r^2} \\
&= \frac{(2N_r^3 + 3N_r^2) + (N_r - 3)}{3(3N_r - 1) \times N_r^2} \\
&\approx \frac{3 + 2N_r}{3(3N_r - 1)} \quad \text{(for LUD)}
\end{aligned}
$$

(Sum of squares of natural numbers: $\sum_1^n(i^2) = \frac{n(n+1)(2n+1)}{6}$)

INV: For a panel update of this kernel, $LUD \rightarrow TRSM \rightarrow TRSM \rightarrow GEMM$ panel updates are performed sequentially. We use the PE_{Util} ratios for GEMM, TRSM and LUD to find the PE_{Util} for Inverse.

$$
\begin{aligned}
PE_{Util} &= \frac{(N_r + 1) \times 1 + 2 \times (3N_r) \times \frac{2+N_r}{3N_r} + (3N_r - 1) \times \frac{3+2N_r}{3(3N_r-1)}}{10N_r \times 1} \\
&= \frac{N_r + 1 + 2(2 + N_r) + \frac{3+2N_r}{3}}{10N_r} \\
&= \frac{18 + 11N_r}{30N_r} \quad \text{(for INV)}
\end{aligned}
$$