Imperial College London

Reconfigurable Computing for Approximate Matrix-Vector Multiplication

A Hardware-Software Co-Design Approach

Author

Justin Huang

CID: 01860968

Supervised by

Professor Christos-Savvas Bouganis

Second Marker

Professor Tim Constandinou

A Thesis submitted in fulfillment of requirements for the degree of
Master of Engineering in Electrical and Electronic Engineering

Department of Electrical and Electronic Engineering
Imperial College London

2024

Abstract

Matrix-vector multiplications are fundamental to a wide range of computational tasks, particularly in machine learning and scientific computing. When deployed on FPGA, these operations can become memory-bound when a large volume of matrices cannot be stored on-chip and has to be retrieved from off-chip. The project aims to address this challenge through a hardware-software co-design framework. Utilising various compression techniques, the framework aims to exploit the inherent redundancies in the given matrices to reduce memory requirements while considering the computational structure of the operations. From a hardware perspective, the framework employs a library of compression-aware and parametrisable compute engines to generate an accelerator that optimises hardware utilisation and computation latency. Given a set of matrix-vector multiplications $\mathcal{M}$ with matrices $\mathbf{W}_j$ and their input vectors $\mathbf{x}_j$, certain application accuracy requirements, and a target FPGA platform, the framework generates matrix decompositions $\mathbf{W}'_j$ and their corresponding hardware accelerators H that minimise the latency of computing these matrix-vector products under hardware resource constraints.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced. I have used ChatGPT v4 as an aid in the preparation of my report. I have used it to improve the quality of my English throughout, however all technical content and references comes from my original text.

Copyright Declaration

Acknowledgments

I would like to express my deepest gratitude to Professor Christos-Savvas Bouganis for his invaluable supervision and guidance throughout this project. His insights and expertise have been instrumental in the successful completion of this work.

I would also like to give special thanks to Mr. Zhewen Yu, PhD student in the Circuit and System Research Group, for his continuous support and assistance with FPGA prototyping. His contributions have been crucial to the development and implementation of this project.

Furthermore, I am grateful to all the teaching staff at Imperial College London who have taught and mentored me during my time here. Their dedication and knowledge have greatly enriched my academic journey and laid the foundation for this research.

Thank you all for your support and encouragement.

Contents

1

Introduction

Contents

This chapter outlines the motivations for the project and highlights the key research problems it addresses. The chapter also provides a high-level specification of the project and briefly introduces the structure of the report.

1.1 Challenges

Matrix-vector multiplications are fundamental to a wide range of computational tasks, particularly in machine learning and scientific computing. In machine learning, for instance, neural networks with fully connected layers rely heavily on matrix-vector multiplications to process and transform input data through successive layers. However, the extensive computational resources required for these operations pose a challenge to their computations in low-latency and power-constrained platforms. Such applications prohibit the use of high-performance, but power-consuming systems like GPUs and multi-core CPUs [1]. In these scenarios, FPGAs offer a compelling solution. FPGAs combine customisation and reconfigurability, allowing for the design of specialised hardware architectures that achieve low latency while maintaining a low energy consumption.

While FPGA offers low latency and superior energy efficiency for these workloads, they are constrained by the limited on-chip memory and the bandwidth available for external memory transaction. In many previous FPGA-based accelerator [2], [3], the entire weight matrices of the neural network is stored on-chip. This approach is not only expensive but also restricts the size of the models that can be implemented. Take Long Short Term Memory (LSTM) as an exmaple. A standard LSTM layer with input and hidden vector dimensions of 512 requires approximately 33.6Mb of storage when using 16-bit precision. This requirement quadruples to 134.3Mb for vector dimensions of 1024. However, the on-chip memory of FPGAs is typically limited. For example, the on-chip memory of Xilinx Zynq 7045 FPGA is only 19.2Mb while the Virtex 7 690T FPGA is equipped a limited on-chip memory of 53Mb, which is insufficient to accommodate a single LSTM layer with a moderate vector size of 1024 [4].

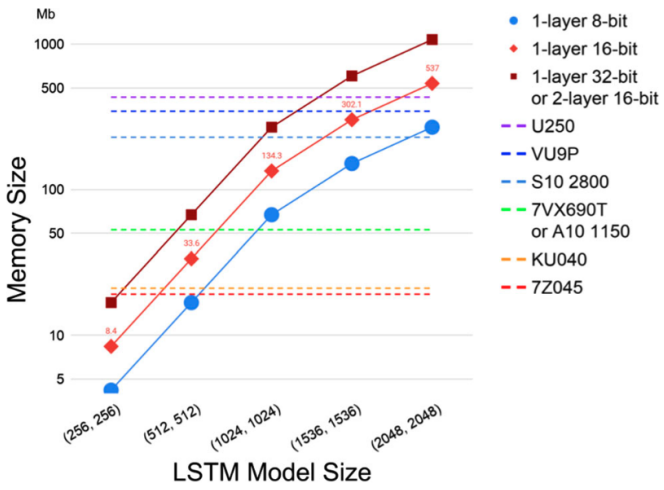


Figure 1.1: Matrix Sizes and On-chip Memory Sizes of Various FPGAs[4].

To support large-scale matrix-vector multiplications, it is necessary to store matrices in off-chip DRAM. This shift effectively transitions the acceleration bottleneck from computation-bounded to memory-bounded. Given the Virtex-7 690T FPGA board's peak DRAM bandwidth of just 13 GB/s, transferring the four matrices with dimensions of 512×1024 takes approximately 2.5 milliseconds to move from DRAM to the on-chip accelerator. This bottleneck is exacerbated by the fact that the retrieved matrices are typically used only once during computation. The frequent loading of matrices results in high power consumption and significant memory transfer overhead. In practical applications, where multiple matrix-vector multiplications run in parallel, this performance limitation becomes even more pronounced. Each additional operation amplifies the demand for data transfer.

1.2 Problem Formulation

The project addresses the generic problem of accelerating matrix-vector multiplications performed in parallel on synchronised inputs for a target FPGA. The matrices are assumed to be stored in off-chip memory, enhancing the approach's applicability to large-scale applications. The research problem is formulated as follows: given a set of matrix-vector multiplications $\mathcal{M}$ with matrices $\mathbf{W}_j$ and their corresponding input vectors $\mathbf{x}_j$, some application accuracy requirement and a target FPGA platform, derive matrix decompositions $\mathbf{W'}_j$ and their corresponding hardware accelerator H that minimise the latency of computing these matrix-vector product under hardware resource constraints. Specifically, the project focuses on lossy compression where numerical error of the approximation is allowed but bounded by specified error threshold.

In real-world workloads, the set of matrix-vector multiplications to be accelerated may have varying dimensionalities, which introduces significant complexity. This diversity can manifest as varying latencies and computational frequencies across matrix-vector multiplications, adding complications to both approximation techniques and hardware architectures. To address these complexities in a structured manner, the project standardises the dimensionality of these matrices. Furthermore, this work assumes that there are no data dependencies during matrix-vector product computation. The entire input vector $\mathbf{x}_j$ is accessible at the time of computation, enabling concurrent scheduling of computations without pipeline stalls.

These assumptions underscore the need for a more in-depth investigation into the diverse nature of matrix-vector multiplications in practical scenarios. Such an investigation would delve into the challenges of managing diverse matrix sizes, synchronising operations with varied latencies, and load balancing to accommodate these variances efficiently. These potential optimisations remain an area for future research and are elaborated in Chapter 7.

1.3 Project Specification

To address the aforementioned challenges, various techniques have been proposed to reduce the computational complexity of matrix-vector multiplications. Popular tensor decomposition methods such as CANDECOMP/PARAFAC and Tucker decomposition involve decomposing a tensor into multiple factor matrices and possibly a core tensor. While these methods are powerful for capturing higher-order interactions in multi-dimensional data, they introduce additional computational overhead due to the more complex multi-dimensional data structures and operations. Furthermore, these tensor decompositions are less straightforward to map onto FPGA architectures because of their irregular computational patterns and higher memory access demands.

In contrast, 2D Singular Value Decomposition (SVD) is more efficient for hardware-accelerated matrix-vector multiplication. Unlike multi-dimensional tensor decompositions, 2D SVD focuses on decomposing a matrix into two simpler vectors, which significantly reduces computational complexity and is easier to implement on FPGA architectures. The regular computational patterns of 2D SVD results in lower memory access overhead, which makes 2D SVD a more practical and scalable approach for approximating MVM on hardware accelerators.

Building on this premise, the project aims to address the research problem via a hardware-software co-design framework. On the software side, the project develops a toolflow that applies various 2D SVD compression methods on a given set of matrices. The assumption is that these matrices exhibit redundancies within their parameters, which SVD can exploit to decrease the data transferred between off-chip memory and the FPGA's compute engines, thereby reduce memory requirement without substantially compromising the accuracy of computation. On the hardware side, the project proposes a library of compression-aware and parametrisable compute engines to generate an accelerator that optimises hardware utilisation and computation latency. Each compute engine is equipped with a corresponding roofline model which is indicative of its performance and resource utilisation. The framework performs design space exploration to determine a set of design points

that achieve a better accuracy-latency trade-off on a target FPGA. Finally, the project uses Long Short-Term Memory (LSTM) networks as a case study, but the methodologies are applicable to a wide range of machine learning architectures, such as transformers.

1.4 Report Structures

The report begins by introducing various SVD-based approximation algorithms to provide a comprehensive background. It then discusses several approximation methods specifically optimised for hardware acceleration. This is followed by an explanation of the top-level framework. Additionally, the development of a roofline model for different approximation methods and their corresponding compute engines is presented. The framework is subsequently evaluated using two LSTM workloads and compared to the state-of-the-art implementation. Finally, the report concludes with a summary and suggestions for future work.

2

Background

Contents

This chapter covers some background and relevant theoretical concepts that underpin the project. Additionally, it provides a review of pertinent research and advancements that have been made in the field.

2.1 Tiled Matrix-vector Multiplication

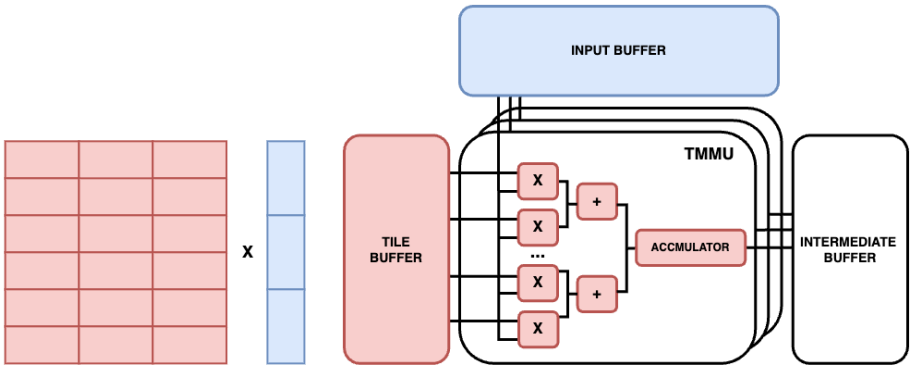


Figure 2.1: Tiled Matrix-Vector Multiplication

Conventionally, large-scale matrix-vector multiplication is done via tiled matrix-vector multiplication. The tiled matrix-vector multiplication unit (TMMU), as discussed in [5], handles large matrix-vector multiplication operations by breaking them down into smaller, more manageable sub-tasks called tiles. In a tiled MVM unit, the input matrix is divided into smaller sub-matrices, or tiles. Similarly, the vector is also divided into segments that match the dimensions of the matrix tiles. Each tile of the matrix is multiplied by the corresponding segment of the vector. As each tile multiplication is completed, the results are accumulated in an intermediate buffer. Once all tiles have been processed, these intermediate results are combined to form the final output vector.

2.2 Iterative Approximation: Single Matrix Approximation

The effectiveness of approximate matrix-vector multiplication has been investigated in many research [6]–[8]. Han et al. [9] presented an FPGA-based speech recognition engine that employs a load-balance-aware compression scheme to reduce network size. Wang et al. [10] proposed a hybrid compression method using circulant matrices, forward nonlinear function approximation, and quantisation. However, these implementations require retraining to restore application-level accuracy. Rizakis et al. [1] presented an approximate computational method for MVMs. This method does not require retraining and compensates for the induced error using iterative refinement. The project will be built around this approximate computational method.

Given a matrix-vector multiplication as shown in 2.1, the matrix-vector multiplication can be approximated using the following method.

$$\mathbf{y}_j = \mathbf{W}_j \cdot \mathbf{x}_j, \quad \mathbf{W}_j \in \mathbb{R}^{M \times N} \tag{2.1}$$

Initially, 2D SVD decomposition is performed on $\mathbf{W}_j$ to obtain $\mathbf{W}_j = \mathbf{U}_j \mathbf{\Sigma}_j \mathbf{V}_j^T$, where $\mathbf{U}_j \in \mathbb{R}^{M \times M}$, $\mathbf{\Sigma}_j \in \mathbb{R}^{M \times N}$ and $\mathbf{V}_j \in \mathbb{R}^{N \times N}$. Then rank-1 approximation is applied by keeping the singular vectors of the largest eigenvalue to obtain $\mathbf{W'}_j = \mathbf{s}_{1,j}\mathbf{u}_{1,j}\mathbf{v}_{1,j}^{\mathrm{T}}$. To further sparsify the matrix, a binary mask vector $\mathrm{mask}_j \in \{0,1\}^{N \times 1}$ is introduced, with each entry indicating whether a parameter in vector $\mathbf{v}_{1,j}$ is removed or not. The sparsified matrix can be represented as $\mathbf{W'}_j = \mathbf{s}_{1,j}\mathbf{u}_{1,j}\left[\,\mathrm{mask}_j \odot \mathbf{v}_{1,j}\,\right]^{\mathrm{T}}$. To minimise the error introduced due to matrix sparsification, the sparsification is transformed into the following optimisation problem.

$$\min_{\mathbf{mask}_j} \left\| \mathbf{W}_j - \mathbf{W}'_j \right\|_2^2, \quad \text{s.t. } \| \,\mathrm{mask}_j \,\|_0 = \mathrm{NZ} \tag{2.2}$$

NZ is the number of non-zero elements in vector mask_j. The optimal solution to the optimisation problem in Equation 2.2 is achieved by retaining NZ elements on each column of vector $\mathbf{v}_{1,j}^{\mathrm{T}}$, with the highest absolute value. Their corresponding indices in mask mask_j are set to 1. By applying both rank-1 approximation and sparsification, the approximated matrix-vector multiplication can be represented as:

$$\mathbf{y'}_j = \mathbf{W'}_j \cdot \mathbf{x}_j \tag{2.3}$$

To refine the error induced by approximation, an iterative refinement algorithm, detailed in Equation 2.4 and Algorithm 1, is used. The matrix is decomposed using rank-1 SVD to obtain a preliminary low-rank representation. This is followed by a sparsification step, ensuring that exactly NZ non-zero elements are retained in each column of vector $\mathbf{v}_1^{\mathrm{T}}$. Subsequent iterations update the approximation by incorporating a rank-1 approximation and sparsification of the residual matrix. The process iterates for a predefined number of steps $\mathrm{N_{steps}}$. This algorithm aims to minimise the Mean Square Error (MSE) between the approximated and original matrix. With a sufficient number of refinement steps, the approximation converges, resulting in a matrix-vector multiplication that closely approximates the original matrix-vector multiplication with reduced computational complexity.

$$\mathbf{y'}_j = \sum_{n=1}^{\mathrm{N_{steps}}} \mathbf{s}_{1,j}^{\mathbf{i(n)}} \mathbf{u}_{1,j}^{\mathbf{i(n)}} \left[(\mathrm{mask}_j^{i(n)} \odot \mathbf{v}_{1,j}^{i(n)})^{\mathrm{T}} \mathbf{x}_j \right] \tag{2.4}$$

Algorithm 1 Iterative Approximation: Single Matrix Approximation

Require:

Matrix $\mathbf{W}_j \in \mathbb{R}^{M \times N}$

Number of non-zero elements **NZ** per refinement step

A MSE threshold $\mathbf{T}_{MSE}$

Iterative Refinement:

Initialise $n \leftarrow 1$

Rank-1 decomposition $\left[\mathbf{u}_{1,j}^{\mathbf{i(1)}}, \mathbf{s}_{1,j}^{\mathbf{i(1)}}, \mathbf{v}_{1,j}^{\mathbf{i(1)}}\right] = \text{SVD}(\mathbf{W}_j)_1$

Generate $\text{mask}_j^{\mathbf{i(1)}} \leftarrow$ solution to Equation 2.2 for vector $\mathbf{v}_{1,j}^{\mathbf{i(1)}}$

$\mathbf{W'}_j^{\mathbf{i(1)}} = \mathbf{s}_{1,j}^{\mathbf{i(1)}} \mathbf{u}_{1,j}^{\mathbf{i(1)}} \left[\text{mask}_j^{\mathbf{i(1)}} \odot \mathbf{v}_{1,j}^{\mathbf{i(1)}}\right]^{\text{T}}$

while $\text{MSE}(\mathbf{W}_j, \mathbf{W'}_j^{\mathbf{i(n)}}) > \mathbf{T}_{MSE}$ **do**

    Increment step counter n $\leftarrow$n + 1

    Compute residual matrix $\mathbf{E}_j = \mathbf{W}_j - \mathbf{W'}_j^{\mathbf{i(n-1)}}$

    Rank-1 decomposition $\left[\mathbf{u}_{1,j}^{\mathbf{i(n)}}, \mathbf{s}_{1,j}^{\mathbf{i(n)}}, \mathbf{v}_{1,j}^{\mathbf{i(n)}}\right] = \text{SVD}(\mathbf{E}_j)_1$

    Generate $\text{mask}_j^{\mathbf{i(n)}} \leftarrow$ solution to Equation 2.2 for vector $\mathbf{v}_{1,j}^{\mathbf{i(n)}}$

    Update matrix approximation $\mathbf{W'}_j^{\mathbf{i(n)}} = \mathbf{W'}_j^{\mathbf{i(n-1)}} + \mathbf{s}_{1,j}^{\mathbf{i(n)}} \mathbf{u}_{1,j}^{\mathbf{i(n)}} \left[\text{mask}_j^{\mathbf{i(n)}} \odot \mathbf{v}_{1,j}^{\mathbf{i(n)}}\right]^{\text{T}}$

end while

Notes:

(i) i(n) stands for refinement step n

(ii) SVD(X)$_1$ returns the rank-1 SVD decomposition of matrix **X**

(iii) MSE(X,Y) returns the mean square error (MSE) between matrix **X** and **Y**

(iv) The final value of n represents the number of refinement steps to reach the MSE threshold, denoted as N_{steps}

2.3 Iterative Approximation: Group Matrix Approximation

The previous iterative approximation method utilises 2D SVD to exploit the redundancy the redundancy within a single matrix. Shashua et al. [11] extended this concept to approximate a group of matrices by leveraging the redundancies across the entire group. They proposed an approach which treats a group of matrices as a tensor and performs approximation based on its tensor rank. Notably, this tensor-rank approximation is designed such that the SVD decomposition emerges in the special case when there is only one matrix in the group.

The proposed approach involves decomposing a given set of matrices into a common set of matrices. The approach produces a set of rank-1 matrices — each can be expressed as the product of two vectors **u** and **v** — whose linear combination approximates the original input matrices. This decomposition also minimises the mean squared error between the approximated and original matrices. The algorithm is elaborated below.

Algorithm 2 Iterative Approximation: Group Matrix Approximation

Require:

A set of $M \times N$ matrices $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_k$
A MSE threshold $\mathbf{T}_{MSE}$
A user-defined threshold $\mathbf{T}_{user}$

Iterative Refinement:

Initialise $n \leftarrow 0$
Initialise $\mathbf{W}_j^{\mathbf{i(1)}} = \mathbf{W}_j,\ \forall j \in [1, k]$
while $\left(\frac{1}{k}\sum_{j=1}^{k} \mathrm{MSE}(\mathbf{W}_j^{\mathbf{i(n)}}, 0) > T_{\mathrm{MSE}}\right)$ **do**
    Increment step counter n ←n + 1
    Compute the eigenvector $\mathbf{u}^{\mathbf{i(n)}}$ of $\sum_{j=1}^{k} \mathbf{W}_j^{\mathbf{i(n)}} \mathbf{W}_j^{\mathbf{i(n)T}}$ with the largest eigenvalue
    Form the matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}} = \left[\, \mathbf{W}_1^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}}, \mathbf{W}_2^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}}, \ldots, \mathbf{W}_k^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}} \,\right]$
    Compute the eigenvector $\mathbf{v}^{\mathbf{i(n)}}$ of matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}}\widehat{\mathbf{W}}^{\mathbf{i(n)T}}$ with the largest eigenvalue
    repeat
        Form the matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}} = \left[\, \mathbf{W}_1^{\mathbf{i(n)}}\mathbf{v}^{\mathbf{i(n)}}, \mathbf{W}_2^{\mathbf{i(n)}}\mathbf{v}^{\mathbf{i(n)}}, \ldots, \mathbf{W}_k^{\mathbf{i(n)}}\mathbf{v}^{\mathbf{i(n)}} \,\right]^{\mathrm{T}}$
        Compute the eigenvector $\mathbf{u}^{\mathbf{i(n)}}$ of matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)T}}\widehat{\mathbf{W}}^{\mathbf{i(n)}}$ with the largest eigenvalue
        Form the matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}} = \left[\, \mathbf{W}_1^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}}, \mathbf{W}_2^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}}, \ldots, \mathbf{W}_k^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}} \,\right]$
        Compute the eigenvector $\mathbf{v}^{\mathbf{i(n)}}$ of matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}}\widehat{\mathbf{W}}^{\mathbf{i(n)T}}$ with the largest eigenvalue
    until $\mathbf{u}^{\mathbf{i(n)}}$ and $\mathbf{v}^{\mathbf{i(n)}}$ vectors change less than $\mathbf{T}_{user}$
    $\mathbf{A}^{\mathbf{i(n)}} = \mathbf{u}^{\mathbf{i(n)}}\mathbf{v}^{\mathbf{i(n)T}}$
    $\mathbf{s}_j^{\mathbf{i(n)}} = \mathbf{v}^{\mathbf{i(n)T}}\mathbf{W}_j^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}},\ \forall j \in [1, k]$
    $\mathbf{W}_j^{\mathbf{i(n+1)}} = \mathbf{W}_j^{\mathbf{i(n)}} - \mathbf{s}_j^{\mathbf{i(n)}}\mathbf{A}^{\mathbf{i(n)}},\ \forall j \in [1, k]$
end while

Notes:

(i) i(n) stands for refinement step n
(ii) MSE(X,Y) returns the mean square error (MSE) between matrix **X** and **Y**
(iii) The final value of n represents the number of refinement steps to reach the MSE threshold, denoted as $\mathrm{N_{steps}}$

The group of matrices can then be approximated as the a linear combination of matrix $\mathbf{A}^{\mathbf{i(n)}}$:

$$\mathbf{W'}_j = \sum_{n=1}^{\mathrm{N_{steps}}} \mathbf{s}_j^{\mathbf{i(n)}}\mathbf{A}^{\mathbf{i(n)}}, \quad \forall j \in [1, k] \tag{2.5}$$

Consequently, the set of matrix-vector multiplications using these group of matrices can be approximated as:

$$\mathbf{W}_j\mathbf{x}_j \approx \sum_{n=1}^{\mathrm{N_{steps}}} \mathbf{s}_j^{\mathbf{i(n)}}\mathbf{A}^{\mathbf{i(n)}}\mathbf{x}_j, \quad \forall j \in [1, k] \tag{2.6}$$

2.4 Mean Squared Error (MSE)

The error injected due to iterative approximation is distributed across all elements of the approximated weight matrix. The objective of the approximation algorithm is to identify matrix decomposition and fine-tune the sparsity of singular vector at each refinement iteration such that the Mean Square Error (MSE) between the approximated weight matrix $\mathbf{W}'$ the original matrix $\mathbf{W}$ is minimised. The following definition of Mean Square Error is used throughout the work. All subtraction and square operations are performed element-wise.

$$\text{MSE} = \frac{1}{\text{M} \cdot \text{N}} \sum_{i=1}^{\text{M}} \sum_{j=1}^{\text{N}} (W_{ij} - W'_{ij})^2 \tag{2.7}$$

An effective approximation algorithm should converge within a finite number of refinement steps. By design, the approximation algorithm will terminate refinement once the MSE between the approximated and original weight matrices falls below the MSE threshold. Ideally, as the MSE approaches zero, the approximated matrix-vector product should increasingly resemble the exact solution, which in turn should lead the system's output to converge to the output of the unapproximated system. Therefore, mean squared error serves as proxy for quantifying application accuracy drop. The validity of this accuracy model is evaluated in Chapter 6.

3

Iterative Approximation

Contents

This chapter explores the iterative approximation methods used to accelerate matrix-vector multiplications. These methods are specifically optimised for hardware acceleration on FPGA.

3.1 Iterative Approximation Optimisation

3.1.1 Sparsification Optimisation

As elaborated in Section 2.2, iterative approximation introduces sparsity at the granularity of single elements. This unstructured sparsity, where non-zero elements are distributed randomly, requires complex indexing and data handling during computation, which is less efficient for hardware acceleration. In addition, the sparsity only applies along the column dimension of the decomposition. There is further opportunity to apply sparsity along the row dimension.

The vectors $\mathbf{u^{i(n)}}$ and $\mathbf{v^{i(n)}}$ are divided by tile sizes Tr and Tc, each representing the tile size in the row and column dimensions. To facilitate hardware acceleration, the tile size is coupled with the parallelism of the compute engine. Sparsity is applied tile-wise and in both row and column dimensions. The optimal solution to the optimisation problem in Equation 2.2 is reformulated as retaining NZr non-zero tiles from the vector $\mathbf{u^{i(n)}}$ and the NZc non-zero tiles from vector $\mathbf{v^{i(n)}}$ with the highest absolute magnitude. Please note that $\mathbf{s_j^{i(n)}}$ are scalars and therefore remains untouched.

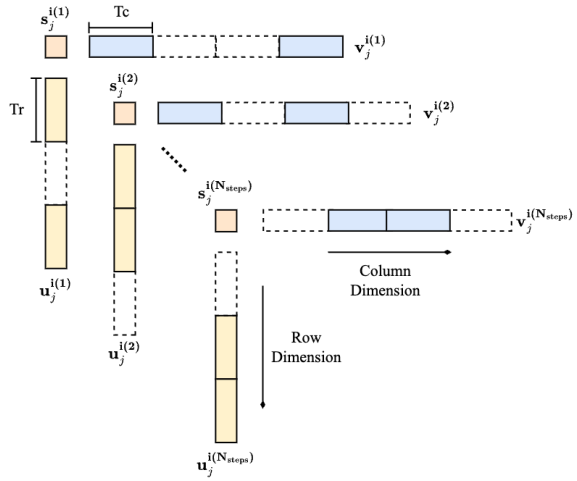


Figure 3.1: Sparisfication with Tr = 4, Tc = 4, NZr = 2 and NZc = 2.

By organising zero and non-zero elements into tiles, this structured sparsity enables hardware accelerators to exploit this regularity to streamline memory access and data movement. In addition, structured sparsity reduces the complexity of indexing and data handling, enabling accelerators to skip over zero elements with minimal computational overhead. However, it is important to note that there is a trade-off with the convergence of approximation, which will be discussed later.

3.1.2 Mask Optimisation

To reduce memory demand and computational latency, the accelerator streams only the non-zero tiles of the decompositions. Two masks are introduced to indicate the positions of these non-zero elements. Given a weight matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$, two binary mask vectors, $\text{masku}^{\mathbf{i(n)}} \in \mathbb{R}^{M \times 1}$ and $\text{maskv}^{\mathbf{i(n)}} \in \mathbb{R}^{N \times 1}$, are created at each refinement step. Each bit in these masks indicates whether an element has been zeroed out.

Furthermore, to efficiently manage and transmit the index of the non-zero tiles, two compressed binary masks denoted in bold as $\mathbf{masku}^{\mathbf{i(n)}}$ and $\mathbf{maskv}^{\mathbf{i(n)}}$ are used. Each of these compressed masks has a length of $\frac{\text{M}}{\text{Tr}}$ and $\frac{\text{N}}{\text{Tc}}$ respectively, where Tr and Tc are the tile sizes in the row and column dimensions. In these compressed masks, each bit represents whether the corresponding tile is retained (1) or removed (0). Specifically, $\mathbf{masku}^{\mathbf{i(n)}}$ contains NZr ones and $\mathbf{maskv}^{\mathbf{i(n)}}$ contains NZc ones.

These compressed masks are streamed into the accelerator for computation. To illustrate the mask generation process, an example is provided below:

v	0.7824	-0.7624	0.2511	-0.2168	0.2731	0.8217	0.0213	-0.8237
Absolute Magnitude	1.5448		0.4679		1.0948		0.845	
maskv	1	1	0	0	1	1	0	0
Compressed **maskv**	1		0		1		0	

Figure 3.2: Mask Generation Example with Tc = 2 and NZc = 2

3.1.3 Single Matrix Approximation Optimisation

The iterative approximation algorithm for single matrix approximation as mentioned in Section 2.2 is optimised for hardware acceleration. Sparsity is applied in both the row and column dimensions, and on a tile-wise basis. Specifically, the algorithm retains the NZr non-zero tiles from the vector $\mathbf{u^{i(n)}}$ and the NZc non-zero tiles from $\mathbf{v^{i(n)}}$ that have the highest absolute magnitudes.

Algorithm 3 Optimised Iterative Approximation: Single Matrix Approximation

Require:

Matrix $\mathbf{W}_j \in \mathbb{R}^{M \times N}$

Number of non-zero tiles **NZr** and **NZc** per refinement step

A MSE threshold $\mathbf{T}_{MSE}$

Iterative Refinement:

Initialise $n \leftarrow 1$

Rank-1 decomposition $\left[\mathbf{u}_{1,j}^{\mathbf{i(1)}}, \mathbf{s}_{1,j}^{\mathbf{i(1)}}, \mathbf{v}_{1,j}^{\mathbf{i(1)}}\right] = \text{SVD}(\mathbf{W}_j)_1$

Set $\text{maskv}_j^{\mathbf{i(1)}}$ to 1 at indices of the **NZc** highest abs. magnitude tiles in $\mathbf{v}_{1,j}^{\mathbf{i(1)}}$

Set $\text{masku}_j^{\mathbf{i(1)}}$ to 1 at indices of the **NZr** highest abs. magnitude tiles in $\mathbf{u}_{1,j}^{\mathbf{i(1)}}$

$\mathbf{W'}_j^{\mathbf{i(1)}} = \mathbf{s}_{1,j}^{\mathbf{i(1)}} \left[\text{masku}_j^{\mathbf{i(1)}} \odot \mathbf{u}_{1,j}^{\mathbf{i(1)}}\right] \left[\text{maskv}_j^{\mathbf{i(1)}} \odot \mathbf{v}_{1,j}^{\mathbf{i(1)}}\right]^{\text{T}}$

while $\text{MSE}(\mathbf{W}_j, \mathbf{W'}_j^{\mathbf{i(n)}}) > \mathbf{T}_{MSE}$ **do**

  Increment step counter n $\leftarrow$n + 1

  Compute residual matrix $\mathbf{E}_j = \mathbf{W}_j - \mathbf{W'}_j^{\mathbf{i(n-1)}}$

  Rank-1 decomposition $\left[\mathbf{u}_{1,j}^{\mathbf{i(n)}}, \mathbf{s}_{1,j}^{\mathbf{i(n)}}, \mathbf{v}_{1,j}^{\mathbf{i(n)}}\right] = \text{SVD}(\mathbf{E}_j)_1$

  Set $\text{maskv}_j^{\mathbf{i(n)}}$ to 1 at indices of the **NZc** highest abs. magnitude tiles in $\mathbf{v}_{1,j}^{\mathbf{i(n)}}$

  Set $\text{masku}_j^{\mathbf{i(n)}}$ to 1 at indices of the **NZr** highest abs. magnitude tiles in $\mathbf{u}_{1,j}^{\mathbf{i(n)}}$

  $\mathbf{W'}_j^{\mathbf{i(n)}} = \mathbf{W'}_j^{\mathbf{i(n-1)}} + \mathbf{s}_{1,j}^{\mathbf{i(n)}} \left[\text{masku}_j^{\mathbf{i(n)}} \odot \mathbf{u}_{1,j}^{\mathbf{i(n)}}\right] \left[\text{maskv}_j^{\mathbf{i(n)}} \odot \mathbf{v}_{1,j}^{\mathbf{i(n)}}\right]^{\text{T}}$

end while

Notes:

(i) $\mathbf{SVD(X)}_1$ returns the rank-1 SVD decomposition of matrix **X**

(ii) MSE(X,Y) returns the mean square error (MSE) between matrix **X** and **Y**

(iii) The final value of n represents the number of refinement steps to reach the MSE threshold, denoted as N_{steps}

Furthermore, at compile-time, the product of the singular value $\mathbf{s}_{1,j}^{\mathbf{i(n)}}$ and the singular vector $\mathbf{v}_{1,j}^{i(n)}$, denoted as $\mathbf{v}_j^{\mathbf{i(n)}}$, is precomputed. This precomputation not only eliminates one multiplication operation in each compute engine but also reduces the number of operands that need to be transmitted from off-chip memory. During the approximate MVM operation, the non-zero tiles of vectors $\mathbf{u}_j^{i(n)}$ and $\mathbf{v}_j^{\mathbf{i(n)}}$, along with the compressed masks $\mathbf{masku}_j^{\text{i(n)}}$ and $\mathbf{maskv}_j^{\text{i(n)}}$, are streamed to the compute engine from off-chip memory. The compressed masks are used to indicate the positions of the non-zero tiles, allowing them to be aligned with the corresponding tiles of the input $\mathbf{x}_j$ during computation.

After the optimisation, the approximate MVM using single matrix approximation can be expressed as the following:

$$\mathbf{W}_j\mathbf{x}_j \approx \sum_{n=1}^{\mathrm{N_{steps}}} (\mathrm{masku}_j^{\mathrm{i(n)}} \odot \mathbf{u}_j^{\mathbf{i(n)}})(\mathrm{maskv}_j^{\mathrm{i(n)}} \odot \mathbf{v}_j^{\mathbf{i(n)}})^{\mathrm{T}}\mathbf{x}_j \tag{3.1}$$

3.1.4 Group Matrix Approximation Optimisation

The iterative approximation algorithm for group matrix approximation as mentioned in Algorithm 2 is optimised for hardware acceleration. Similar as before, sparsity is applied in both the row and column dimensions, and on a tile-wise basis.

The group of matrices can then be approximated as a linear combination of a shared pair of vectors $\mathbf{u}^{i(n)}$ and $\mathbf{v}^{i(n)}$, as expressed in the equation below:

$$\mathbf{W'}_j = \sum_{n=1}^{\mathrm{N_{steps}}} \mathbf{s}_j^{\mathbf{i(n)}} \left(\mathrm{masku}^{\mathbf{i(n)}} \odot \mathbf{u}^{\mathbf{i(n)}}\right) \left(\mathrm{maskv}^{\mathbf{i(n)}} \odot \mathbf{v}^{\mathbf{i(n)}}\right)^{\mathrm{T}}, \quad \forall j \in [1, k] \tag{3.2}$$

Consequently, the set of matrix-vector multiplications using these group of matrices can be approximated as:

$$\mathbf{W}_j\mathbf{x}_j \approx \sum_{n=1}^{\mathrm{N_{steps}}} \mathbf{s}_j^{\mathbf{i(n)}} \left(\mathrm{masku}^{\mathbf{i(n)}} \odot \mathbf{u}^{\mathbf{i(n)}}\right) \left(\mathrm{maskv}^{\mathbf{i(n)}} \odot \mathbf{v}^{\mathbf{i(n)}}\right)^{\mathrm{T}} \mathbf{x}_j, \quad \forall j \in [1, k] \tag{3.3}$$

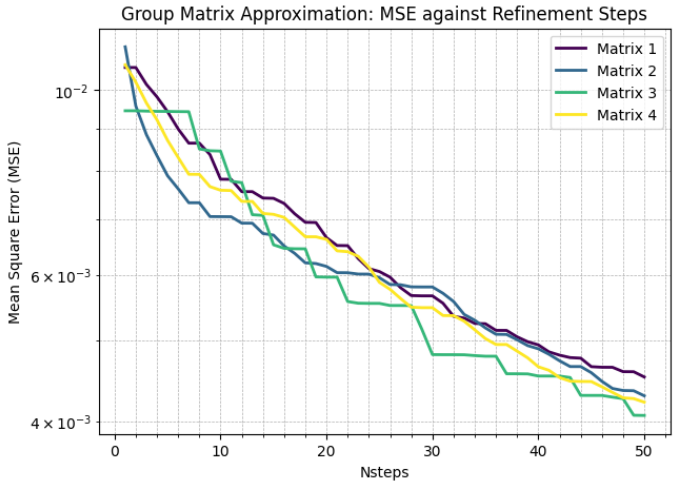


Figure 3.3: Convergence of Group Matrix Approximation with 4 Matrices

Algorithm 4 Optimised Iterative Approximation: Group Matrix Approximation

Require:

A set of $M \times N$ matrices $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_k$
A MSE threshold $\mathbf{T}_{MSE}$
Number of non-zero tiles **NZr** and **NZc** per refinement step
A user-defined threshold $\mathbf{T}_{user}$

Iterative Refinement:

Initialise $n \leftarrow 0$
Initialise $\mathbf{W}_j^{\mathbf{i(1)}} = \mathbf{W}_j,\ \forall j \in [1, k]$
while $\left(\frac{1}{k}\sum_{j=1}^{k} \mathrm{MSE}(\mathbf{W}_j^{\mathbf{i(n)}}, 0) > T_{\mathrm{MSE}}\right)$ **do**
    Increment step counter n $\leftarrow$n + 1
    Compute the eigenvector $\mathbf{u}^{\mathbf{i(n)}}$ of $\sum_{j=1}^{k} \mathbf{W}_j^{\mathbf{i(n)}} \mathbf{W}_j^{\mathbf{i(n)T}}$ with the largest eigenvalue
    Form the matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}} = \left[\, \mathbf{W}_1^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}}, \mathbf{W}_2^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}}, \ldots, \mathbf{W}_k^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}} \,\right]$
    Compute the eigenvector $\mathbf{v}^{\mathbf{i(n)}}$ of matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}}\widehat{\mathbf{W}}^{\mathbf{i(n)T}}$ with the largest eigenvalue
    repeat
        Form the matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}} = \left[\, \mathbf{W}_1^{\mathbf{i(n)}}\mathbf{v}^{\mathbf{i(n)}}, \mathbf{W}_2^{\mathbf{i(n)}}\mathbf{v}^{\mathbf{i(n)}}, \ldots, \mathbf{W}_k^{\mathbf{i(n)}}\mathbf{v}^{\mathbf{i(n)}} \,\right]^{\mathrm{T}}$
        Compute the eigenvector $\mathbf{u}^{\mathbf{i(n)}}$ of matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)T}}\widehat{\mathbf{W}}^{\mathbf{i(n)}}$ with the largest eigenvalue
        Form the matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}} = \left[\, \mathbf{W}_1^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}}, \mathbf{W}_2^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}}, \ldots, \mathbf{W}_k^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}} \,\right]$
        Compute the eigenvector $\mathbf{v}^{\mathbf{i(n)}}$ of matrix $\widehat{\mathbf{W}}^{\mathbf{i(n)}}\widehat{\mathbf{W}}^{\mathbf{i(n)T}}$ with the largest eigenvalue
    until $\mathbf{u}^{\mathbf{i(n)}}$ and $\mathbf{v}^{\mathbf{i(n)}}$ vectors change less than $\mathbf{T}_{user}$
    $\mathbf{A}^{\mathbf{i(n)}} = \mathbf{u}^{\mathbf{i(n)}}\mathbf{v}^{\mathbf{i(n)T}}$
    $\mathbf{s}_j^{\mathbf{i(n)}} = \mathbf{v}^{\mathbf{i(n)T}}\mathbf{W}_j^{\mathbf{i(n)T}}\mathbf{u}^{\mathbf{i(n)}},\ \forall j \in [1, k]$
    Set $\mathrm{maskv}^{\mathbf{i(n)}}$ to 1 at indices of the **NZc** highest abs. magnitude tiles in $\mathbf{v}^{\mathbf{i(n)}}$
    Set $\mathrm{masku}^{\mathbf{i(n)}}$ to 1 at indices of the **NZr** highest abs. magnitude tiles in $\mathbf{u}^{\mathbf{i(n)}}$
    $\mathbf{W}_j^{\mathbf{i(n+1)}} = \mathbf{W}_j^{\mathbf{i(n)}} - \mathbf{s}_j^{\mathbf{i(n)}} \left[\mathrm{masku}^{\mathbf{i(n)}} \odot \mathbf{u}^{\mathbf{i(n)}}\right] \left[\mathrm{maskv}^{\mathbf{i(n)}} \odot \mathbf{v}^{\mathbf{i(n)}}\right]^{\mathrm{T}},\ \forall j \in [1, k]$
end while

Notes:

(i) MSE(X,Y) returns the mean square error (MSE) between matrix **X** and **Y**
(ii) The final value of n represents the number of refinement steps to reach the MSE threshold, denoted as $\mathrm{N}_{\mathrm{steps}}$

During the approximate MVM operation, the non-zero tiles of vectors $\mathbf{u}_j^{i(n)}$ and $\mathbf{v}_j^{\mathbf{i(n)}}$, along with the scalars $\mathbf{s}_j^{\mathbf{i(n)}}$, compressed masks $\mathbf{masku}_j^{\mathrm{i(n)}}$ and $\mathbf{maskv}_j^{\mathrm{i(n)}}$, are streamed to the compute engine from off-chip memory.

3.2 Hybrid Matrix Approximation

Building on the two matrix approximation methods, hybrid matrix approximation is proposed to leverage the benefits of both single and group matrix approximation methods. The primary idea is to decompose a given set of matrices into two sets of components: one set that is common across all matrices and another set that is specific to each individual matrix. This idea can be further extended by considering dynamic groupings of matrices.

In this extended hybrid approach, matrices are grouped based on their ability to collectively minimize MSE, with each group sharing the same decompositions $\mathbf{u^{i(n)}}$ and $\mathbf{v^{i(n)}}$ but different scalars $\mathbf{s_j^{i(n)}}$. The grouping can be adjusted dynamically at each refinement step to optimise the approximation quality. This method spans two extremes: when the grouping encapsulates all matrices, the approximation is equivalent to group matrix approximation; when each group contains only a single matrix, the approximation is equivalent to single matrix approximation. This hybrid decomposition approach captures both the shared structure among matrices and the unique characteristics of each matrix. By balancing these aspects, the approach may improve the convergence properties of the approximation. The approach is visualised in Figure 3.4. The number of groups per refinement step is denoted as $\mathrm{N}_{\mathrm{groups}}^{\mathbf{i(n)}}$.

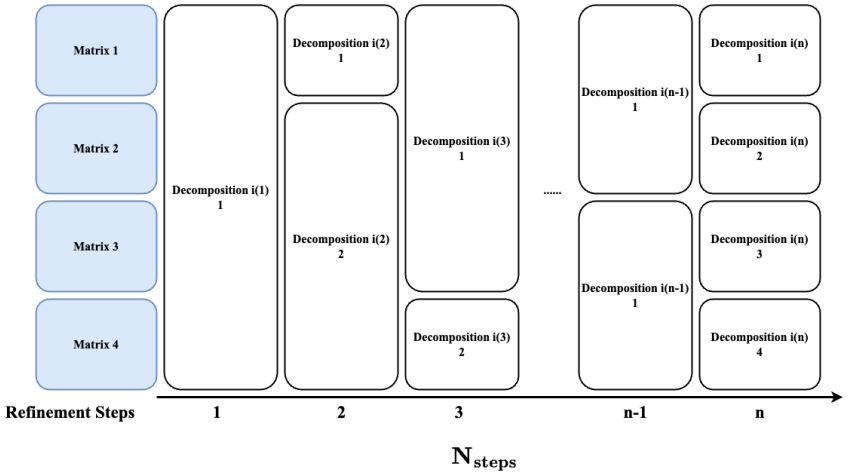


Figure 3.4: Hybrid Matrix Approximation Visualisation: The set contains four matrices. In refinement step 1, there is 1 group of matrices ($\mathrm{N}_{\mathrm{groups}}^{(1)} = 1$), where all 4 matrices are approximated together using group matrix approximation. In refinement step 2, there are 2 groups of matrices ($\mathrm{N}_{\mathrm{groups}}^{(2)} = 2$); one group contains 1 matrix, and the other group contains 3 matrices. Each group is separately approximated using the group matrix approximation method. For refinement step n, where there is only one matrix in each group, the approximation is equivalent to single matrix approximation.

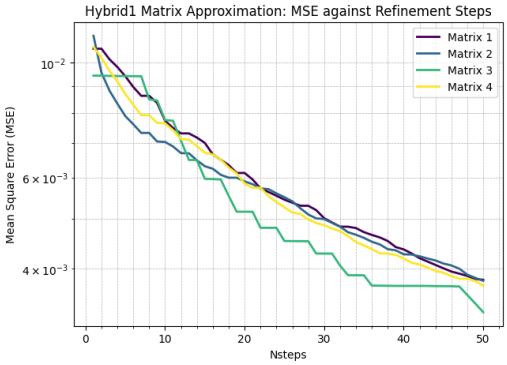
While increasing the number of groupings yields yields better convergence for individual matrices, it also increases memory overhead by requiring more distinct decompositions per refinement step. To navigate around this trade-off, several metrics are defined to assess the grouping's ability to collectively minimise MSE:

- **Metric 1: Average MSE decrement of the entire set of matrices per memory access.** This metric measures the overall efficiency of the approximation process by calculating the average reduction in MSE for all matrices in the set, normalised by the number of memory accesses.

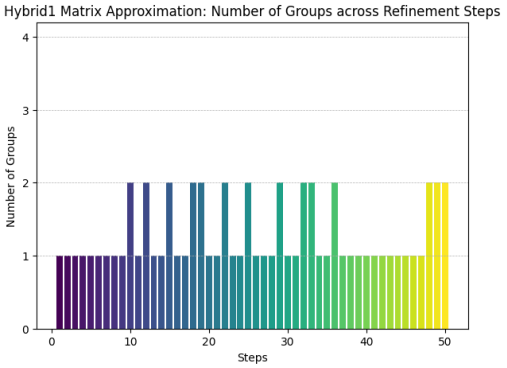
- **Metric 2: Prioritise the largest MSE decrement of a single matrix per memory access.** This metric focuses on the matrix with the highest individual error and aims to maximise the reduction in its MSE for each memory access. By targeting the matrix with the most significant error, this metric ensures that the most critical approximations are optimised first, potentially leading to more noticeable improvements in overall system performance.

- **Metric 3: Cumulative average MSE decrement of the entire set of matrices per cumulative memory access.** This metric considers the cumulative improvement in MSE from the start of the optimisation process, divided by the total number of memory accesses up to the current point. It provides a comprehensive view of the efficiency of the optimisation process over time, highlighting how well the approximation strategy performs in the long run by balancing short-term gains and overall progress.

A set of four matrices is tested using the hybrid matrix approximation with different metrics. The MSE of each individual matrix is plotted against refinement steps. Additionally, the number of groups $\mathrm{N}_{\mathrm{groups}}^{(n)}$ is plotted against refinement steps to highlight the differences between metrics. Approximation using different metrics shows convergence with increasing number of refinement steps. As shown in Figure 3.5, Metric 1 favours large groupings of matrices per step, as this balances the average MSE decrement per memory access. In Figure 3.6, Metric 2 initially decomposes large groups of matrices to exploit shared information and later focuses on single matrices to more efficiently represent the information unique to each matrix. Figure 3.7 illustrates Metric 3, the approximation uses 4 groups for each step which is equivalent to single matrix approximation. These phenomenon is matrix-dependent, and the effectiveness of the hybrid approach with different metrics is explored in Chapter 6.

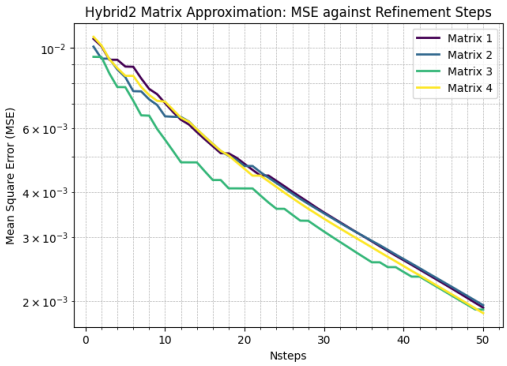


(a) MSE Against Refinement Steps

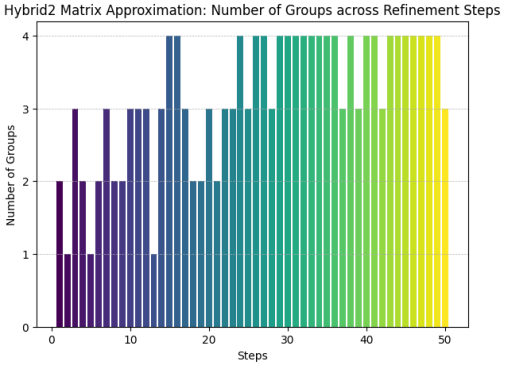


(b) Number of Groups Across Refinement Steps

Figure 3.5: Hybrid Matrix Approximation: Metric 1

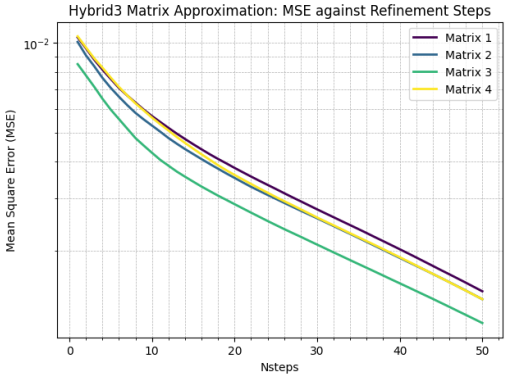


(a) MSE Against Refinement Steps

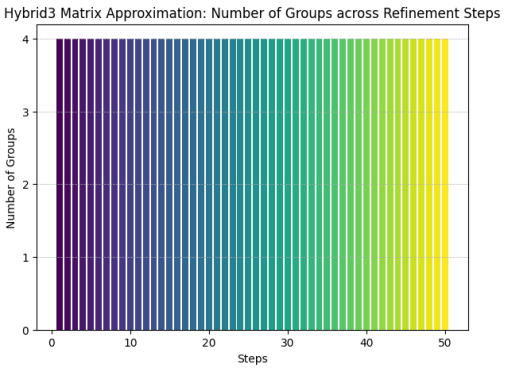


(b) Number of Groups Across Refinement Steps

Figure 3.6: Hybrid Matrix Approximation: Metric 2



(a) MSE Against Refinement Steps



(b) Number of Groups Across Refinement Steps

Figure 3.7: Hybrid Matrix Approximation: Metric 3

3.3 Stack Matrix Approximation

Some real-world workloads involve a set of MVMs with different matrices but the same input. For this special case, matrices with the same dimensionality can be stacked and approximated together using single matrix approximation, as shown in Algorithm 3. Consider a set of $\mathrm{N_{mvm}}$ dependent MVMs, these MVMs can be combined into a single stacked MVM operation as:

$$\mathbf{y}_j = \mathbf{W}_j \cdot \mathbf{x}, \quad \mathbf{W}_j \in \mathbb{R}^{M \times N}, \quad \forall j \in [0, \mathrm{N_{mvm}} - 1] \tag{3.4}$$

$$\mathbf{y} = \begin{bmatrix} \mathbf{W}_0 \\ \mathbf{W}_1 \\ \cdots \\ \mathbf{W}_j \end{bmatrix} \cdot \mathbf{x}, \quad \forall j \in [0, \mathrm{N_{mvm}} - 1] \tag{3.5}$$

Here, $\mathbf{y}$ is the concatenated result vector, and the stacked matrix consists of all individual $\mathbf{W}_j$ matrices. Using the equation for single matrix approximation, the approximated MVM result $\mathbf{y}'$ of the stacked matrix can be expressed as shown in Equation 3.5. Similar to single matrix approximation, the product of the singular value $\mathbf{s}_{1,j}^{\mathbf{i(n)}}$ and the singular vector $\mathbf{v}_{1,j}^{\mathbf{i(n)}}$ is precomputed and denoted as $\mathbf{v}^{\mathbf{i(n)}}$.

$$\mathbf{y}' \approx \sum_{n=1}^{N_{\mathrm{steps}}} (\mathrm{masku}^{\mathbf{i(n)}} \odot \mathbf{u}^{\mathbf{i(n)}}) \cdot (\mathrm{maskv}^{\mathbf{i(n)}} \odot \mathbf{v}^{\mathbf{i(n)}})^{\mathrm{T}} \cdot (\mathrm{maskv}^{\mathbf{i(n)}} \odot \mathbf{x}) \tag{3.6}$$

The approximated result vector $\mathbf{y}'_j$ for each individual MVM can be extracted from the overall approximation:

$$\mathbf{y}'_j \approx \begin{bmatrix} y'_{jM} \\ y'_{jM+1} \\ \cdots \\ y'_{(j+1)M} \end{bmatrix}, \quad \forall j \in [0, \mathrm{N_{mvm}} - 1] \tag{3.7}$$

It is important to note that the compressed mask $\mathbf{masku}^{\mathbf{i(n)}}$ has a length of $\frac{\mathrm{N_{mvm}} \cdot \mathrm{M}}{\mathrm{Tr}}$, while the length of $\mathbf{maskv}^{\mathbf{i(n)}}$ remains the same.

3.4 Normalisation

In approximation methods involving the collective approximation of a set of matrices (such as stack, group, and hybrid matrix approximation), normalisation is a crucial step to ensure consistent scaling and numerical stability. The normalisation process involves the following steps. First, each matrix in the set is normalised by dividing it by the user-specified norm. The framework supports two types of norms:

- **Frobenius Norm:** The Frobenius norm of a matrix $\mathbf{W}$, denoted as $\|\mathbf{W}\|_F$, is defined as:

$$\|\mathbf{W}\|_F = \sqrt{\sum_{i=1}^{M}\sum_{j=1}^{N}|w_{ij}|^2} \tag{3.8}$$

  where w_{ij} represents the elements of $\mathbf{W}$. This norm is useful for capturing the overall energy of the matrix and is less sensitive to individual element magnitudes.

- **Spectral Norm:** The spectral norm of a matrix $\mathbf{W}$, denoted as $\|\mathbf{W}\|_2$, is defined as the largest singular value of $\mathbf{W}$:

$$\|\mathbf{W}\|_2 = \sigma_{\max}(\mathbf{W}) \tag{3.9}$$

  where $\sigma_{\max}(\mathbf{W})$ is the maximum singular value of $\mathbf{W}$. This norm corresponds to the maximum stretching factor the matrix applies to any vector and is more sensitive to extreme values, providing insight into the matrix's most significant transformation properties.

Mathematically, the normalisation step involves creating a normalised matrix $\widehat{\mathbf{W}}_j$ as follows:

$$\widehat{\mathbf{W}}_j = \frac{\mathbf{W}_j}{\|\mathbf{W}_j\|}, \quad \forall j \tag{3.10}$$

Next, the approximation is performed on these normalised matrices as before. If $\widehat{\mathbf{W}}'_j$ is the approximated normalised matrix, the final approximated matrix $\mathbf{W'}_j$ is obtained by multiplying back by the original norm:

$$\mathbf{W'}_j = \widehat{\mathbf{W}}'_j \times \|\mathbf{W}_j\|, \forall j \tag{3.11}$$

Normalisation ensures that the approximation respects the relative importance and magnitude of the original matrices, making the iterative approximation more robust to initial scale differences among the matrices. The effect of normalisation at the application level is discussed in Chapter 6.

3.5 Iterative Approximation Analysis

Several trade-offs exist in iterative approximation. These trade-offs are inherent in all approximation strategies.

3.5.1 Trade-off: Number of Non-Zero Tiles and Rate of Convergence

In this analysis, a matrix is approximated using single matrix approximation without introducing sparsity in the column dimension. The tile sizes Tc and Tr are set to 2. The approximation using different NZc values and the resulting MSE of the approximation are plotted against the number of refinement steps N_{steps}.

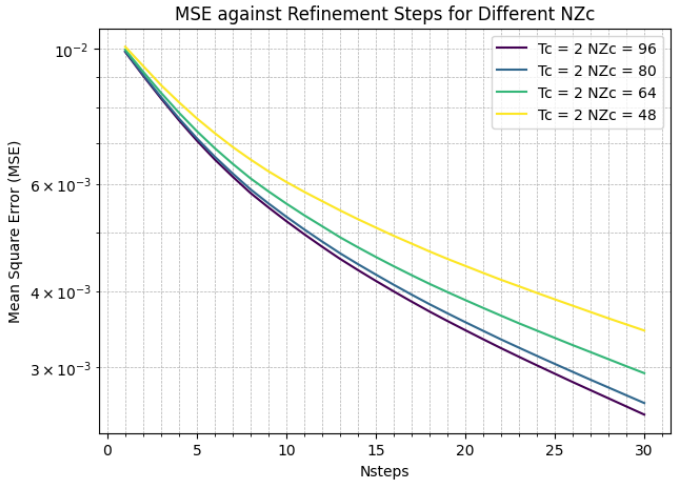


Figure 3.8: MSE vs. Number of Refinement Steps for Different NZc Values

As the number of refinement iterations increases, the MSE gradually decreases, and the approximated matrix increasingly resembles the original matrix. Additionally, as the number of non-zero tiles in column dimension NZc increases, the approximation exhibits a faster rate of convergence, requiring fewer refinements to achieve the same level of MSE. However, computing more non-zero tiles per refinement step requires more cycles and memory access, potentially leading to longer latency in hardware acceleration. Thus, there is a trade-off between the number of non-zero tiles per step and the rate of convergence of the approximation.

3.5.2 Trade-off: Parallelism and Rate of Convergence

In the same experimental setup, a matrix is approximated using single matrix approximation without introducing sparsity in the column dimension. The total number of non-zero elements in the column dimension is kept constant, i.e., the product of Tc and NZc remains the same. The tile size Tc is varied, and the MSE is plotted against the number of refinement steps for different Tc and NZc pairs.

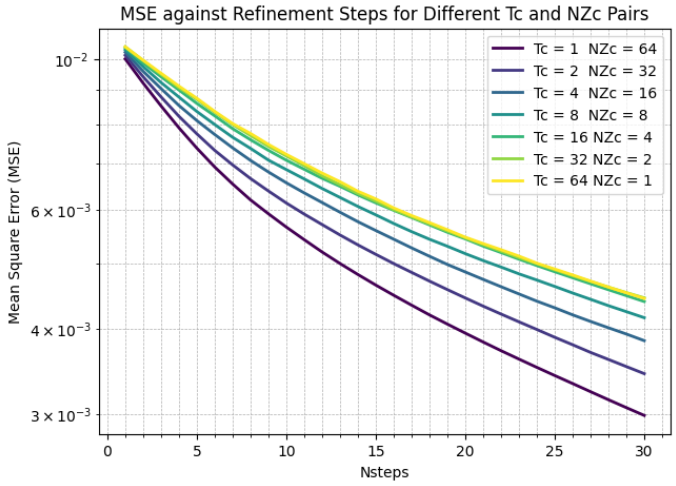


Figure 3.9: MSE vs. Number of Refinement Steps for Different Tc and NZc Pairs

As shown in Figure 3.9, smaller tile sizes result in a faster rate of convergence of the approximation. A smaller tile size Tc achieves the same MSE with fewer refinement steps. This is because smaller tiles provide finer granularity in sparsity, leading to better approximation in the least square sense at each refinement step. However, in hardware acceleration, tile size is coupled with the parallelism of the compute engine. Smaller tiles result in fewer computations running in parallel and longer latency of computation. This analysis highlights the trade-off between parallelism and the rate of convergence of approximation.

These analyses demonstrate that iterative approximation involves balancing multiple parameters to optimise performance. The top-level framework developed later in this work leverages these trade-offs to achieve efficient hardware acceleration.

3.6 Iterative Approximation Parameters

The table below summarises all the parameters used in iterative approximation. These parameters influence how matrices are decomposed and approximated during the iterative refinement. By adjusting these parameters, the top-level framework can exploit various trade-offs and identify promising design points with improved performance.

Table 3.1: List of Compile Time Parameters of Iterative Approximation

Symbol	Description
Strategy	The approximation strategy used in approximation
Norm	The normalisation method used in approximation
$\mathbf{N_{steps}}$	Number of refinement steps
Tr	Tile size of the $\mathbf{u}^{(i)}$ vectors
Tc	Tile size of the $\mathbf{v}^{(i)}$ vectors
NZr	Number of non-zero tiles of vector $\mathbf{u}^{(i)}$ in each refinement step
NZc	Number of non-zero tiles of vector $\mathbf{v}^{(i)}$ in each refinement step

4

Hardware Architecture

Contents

This chapter discusses the hardware architecture of the project. It begins with an overview of the top-level accelerator architecture, followed by several implementations of SVD kernels, each tailored to different approximation strategies.

4.1 Accelerator Architecture

The accelerator architecture is designed to accelerate the computation of multiple matrix-vector multiplications (MVMs). The accelerator consists of multiple SVD kernels, Direct Memory Access (DMA) units, and pipeline control logic. The overall architecture is depicted in Figure 4.1.

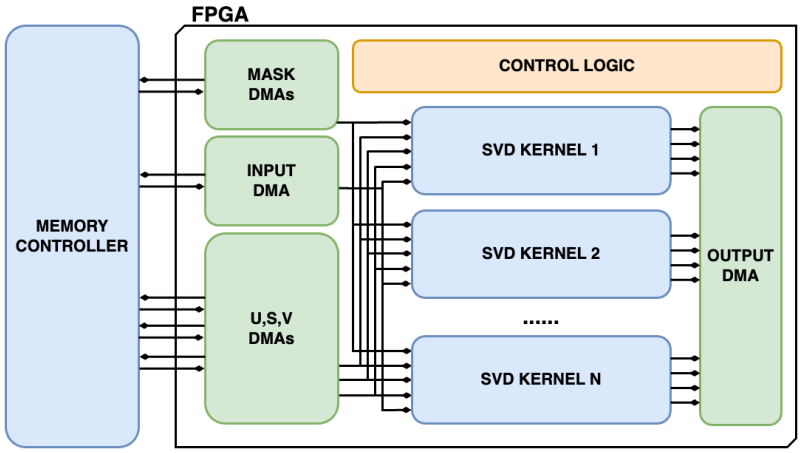


Figure 4.1: Accelerator Architecture

4.1.1 DMA Units and Memory Interface

The DMA units serve as the interface with the DDR3 memory controller on the FPGA, orchestrating data transfers between the on-chip accelerator and off-chip memory. DMAs units are shared across all SVD kernels, which unifies off-chip memory access and reduces wiring complexity. Data are prefetched and buffered to mask memory access latency and prevent pipeline stalls, ensuring a continuous data stream to the respective SVD kernel.

- **Input DMA:** Input vectors are prefetched ahead of computations and stored in designated input buffers during processing to maximize data reuse.
- **U, S, V DMAs:** These DMAs fetch the non-zero elements of the vectors $\mathbf{u^{i(n)}}$ and $\mathbf{v^{i(n)}}$, and possibly the scalars $\mathbf{s^{i(n)}}$, tile-by-tile, delivering them to the corresponding SVD kernels.
- **Mask DMAs:** The compressed mask **maskv** and **masku** for each refinement iteration are streamed into the accelerator via mask DMAs and buffered for the duration of one refinement step. These are used to indicate the position of non-zero tiles in vectors $\mathbf{u^{i(n)}}$ and $\mathbf{v^{i(n)}}$.
- **Output DMA:** Once the MVMs are finished, the output DMA unit streams the approximated matrix-vector product back to off-chip memory for subsequent processing.

4.1.2 SVD Kernels

The accelerator supports a library of compression-aware SVD kernels tailored to different approximation strategies as discussed in Chapter 3. The architectural parameter $N_{\text{SVD kernel}}$ indicates the number of SVD kernels in the accelerator. Each kernel is designed to receive a continuous stream of decomposed matrix tiles from the DMAs and perform approximate MVMs. The accelerator supports both homogeneous and heterogeneous configurations:

- **Homogeneous Configuration:** All SVD kernels utilise the same approximation strategy and have the same level of parallelism, providing a uniform approach to all MVMs.
- **Heterogeneous Configuration:** Each SVD kernel is tailored to different approximation strategies with varying levels of parallelism. This flexibility allows the accelerator to efficiently handle MVM workload across a range of matrix sizes.

4.1.3 Pipeline Control Logic

The pipeline control logic coordinates the overall operation of the accelerator. It ensures synchronisation between data prefetching, processing, and storing operations. By managing data flow and controlling the execution of SVD kernels, the pipeline control logic minimises idle times, enhancing overall computational efficiency.

4.2 SVD Kernel Library

4.2.1 SVD Kernel for Single Matrix Approximation

Architecture

The architecture of the SVD kernel optimised for single matrix approximation, adapted from the design proposed by Rizakis et al. [1], is illustrated in Figure 4.2. The SVD kernel consists of multiple datapaths, each including one input buffer and dispatcher, one v kernel, and one u kernel. Each datapath can support one independent MVM with independent input vector. The number of independent MVMs the SVD kernel can support is denoted as N_{mvm}.

$$\mathbf{y}_j = \mathbf{W}_j \cdot \mathbf{x}_j, \quad \mathbf{W}_j \in \mathbb{R}^{M \times N}, \quad \forall j \in [0, N_{\text{mvm}} - 1] \tag{4.1}$$

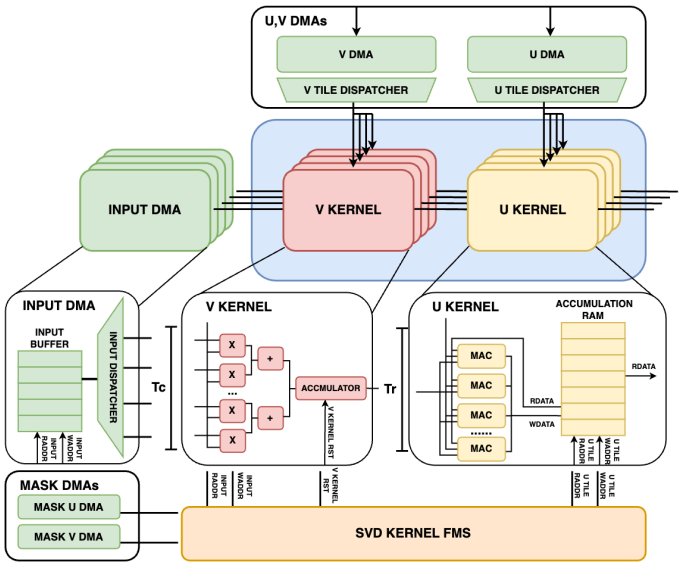


Figure 4.2: SVD kernel for single matrix approximation: the kernel shown in the figure has four datapaths and can support four independent MVMs operating in parallel

To reduce memory demand and improve hardware utilisation, the accelerator streams only non-zero tiles of the vectors $\mathbf{u^{i(n)}}$ and $\mathbf{v^{i(n)}}$. The SVD kernel then performs approximate MVM using theses non-zero tiles and the buffered input vector $\mathbf{x}_j$. As shown in Figure 4.3, non-zero tiles of vector $\mathbf{v}_j^{\mathbf{i(n)}}$ must be aligned with their corresponding input tiles $\mathbf{x}_j$ for computation. In j^{th} datapath, the input buffer addresses corresponding to the non-zero tiles of $\mathbf{v}_j^{\mathbf{i(n)}}$ are decoded from the compressed mask $\mathbf{maskv}_j^{\mathbf{i(n)}}$. The input dispatcher reads the input tiles from the buffer sequentially and dispatches them to the v kernel. The input vector buffer is partitioned into tiles of size Tc to supply the entire input tile in parallel. Concurrently, the v tile dispatcher forwards the non-zero tiles of $\mathbf{v}_j^{\mathbf{i(n)}}$ to v kernel. This mechanism effectively aligns the non-zero $\mathbf{v}_j^{\mathbf{i(n)}}$ tiles with their corresponding input tiles.

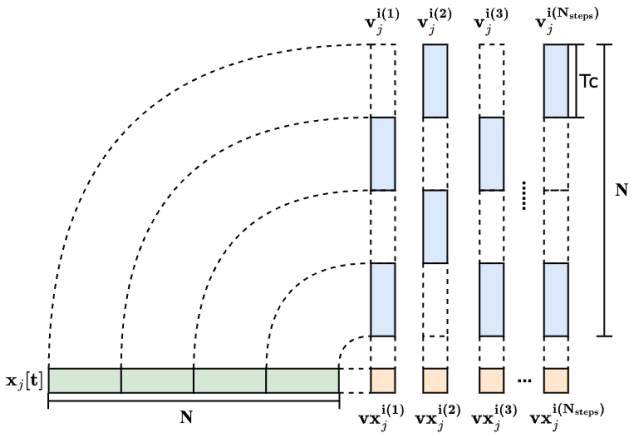


Figure 4.3: Visualisation of dot product $\mathbf{x}_j \cdot \mathbf{v}_j^{\mathbf{i(n)}}$ with N=16, Tc =4 and NZc = 2

The v kernel then performs dot product between the non-zero tiles of $\mathbf{v}_j^{\mathbf{i(n)}}$ and their corresponding input tiles as shown in Equation 4.2. The v kernel is implemented using an multiplier-adder tree. The multiplier-adder tree is constructed with a width of Tc which aligns with the tiling dimension of $\mathbf{v}_j^{\mathbf{i(n)}}$. The partial dot product is accumulated in the accumulator until the entire dot product between the non-zero tiles of $\mathbf{v}_j^{\mathbf{i(n)}}$ and the their corresponding input tiles are added. The final dot product $\mathbf{vx}_j^{\mathbf{i(n)}}$ is then passed as a constant operand to the subsequent u kernel, and the accumulator is cleared for the next refinement step.

$$\mathbf{vx}_j^{\mathbf{i(n)}} = (\mathrm{maskv}_j^{\mathbf{i(n)}} \odot \mathbf{v}_j^{\mathbf{i(n)}})^{\mathrm{T}} \cdot (\mathrm{maskv}_j^{\mathbf{i(n)}} \odot \mathbf{x}_j), \quad \forall n \in [1, \mathrm{N_{steps}}], \quad \forall j \in [0, \mathrm{N_{mvm}} - 1] \tag{4.2}$$

Likewise, the u tile dispatcher supplies the non-zero tiles of $\mathbf{u}_j^{\mathbf{i(n)}}$ to u kernel. The u kernel performs the approximate MVM as shown in Equation 4.3. The computation is carried out via a multiply–accumulate (MAC) array. The MAC array is constructed with a width of Tr which aligns with the tiling dimension of $\mathbf{u}_j^{\mathbf{i(n)}}$. The u kernel multiplies the constant $\mathbf{vx}_j^{\mathbf{i(n)}}$ with non-zero tiles of $\mathbf{u}_j^{\mathbf{i(n)}}$, and the result is accumulated to corresponding entry in the accumulation ram. The ram address to be accumulated is decoded from compressed mask $\mathbf{masku}_j^{\mathbf{i(n)}}$. The accumulation RAM has and is partitioned into tiles (tile size Tr) to enable parallel access by the MAC array.

$$\mathbf{W}_j \cdot \mathbf{x}_j \approx \sum_{n=1}^{\mathrm{N_{steps}}} (\mathrm{masku}_j^{\mathbf{i(n)}} \odot \mathbf{u}_j^{\mathbf{i(n)}}) \cdot \mathbf{vx}_j^{\mathbf{i(n)}}, \quad \forall j \in [0, \mathrm{N_{mvm}} - 1] \tag{4.3}$$

As shown in the scheduling diagram (Figure 4.4), the datapath is pipelined to boost throughput. V kernel starts computing the dot product $\mathbf{vx}_j^{\mathbf{i(n)}}$ for new $\mathbf{v}_j^{\mathbf{i(n)}}$ while u kernel multiplies and accumulates the current $\mathbf{u}_j^{\mathbf{i(n)}}$. This process repeats until all $\mathrm{N_{steps}}$ refinement steps are completed. Once all the refinement steps are finished, the u kernel stream out the approximated matrix-vector product from its accumulation RAM.

Note that this kernel can also support $\mathbf{N_{mvm}}$ dependent MVMs with the same input vector. While the datapath construct remains unchanged, the memory access pattern is slightly different. The difference is highlighted in the next section.

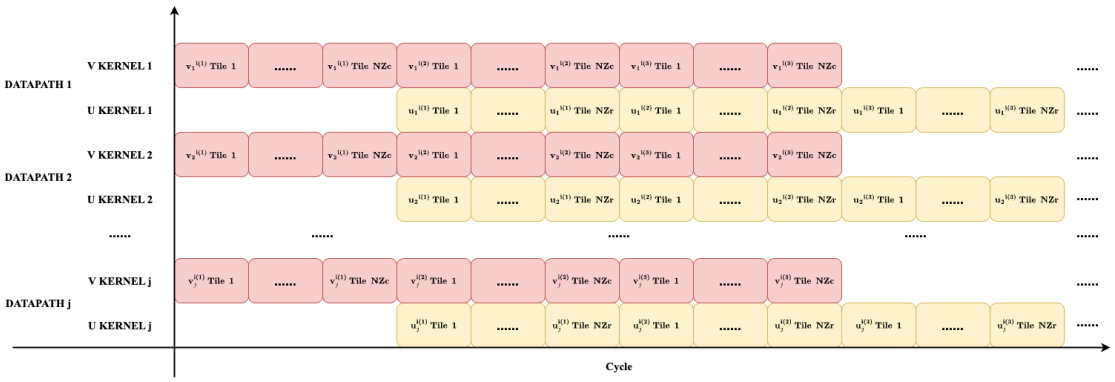


Figure 4.4: Scheduling diagram: SVD kernel for singlex matrix approximation

Memory Access

The total memory access of this SVD kernel type comprises the following components. The memory access due to the input and output vectors of N_{mvm} independent MVMs is shown below. The byte size of the quantised input and matrix decompositions are defined as Precision.

$$\textbf{input} + \textbf{output} = N_{mvm} \cdot (N + M) \cdot \text{Precision}$$

For N_{mvm} dependent MVMs, the input vector is shared and is accessed only once. The memory access due to the input and output vectors is expressed as:

$$\textbf{input} + \textbf{output} = (\ N + N_{mvm} \cdot M\ ) \cdot \text{Precision}$$

The u and v DMA units only stream in the non-zero tiles of vector $\mathbf{v_j^{i(n)}}$ and $\mathbf{u_j^{i(n)}}$. The total number of non-zero tiles per step are NZc and NZr, respectively. Additionally, each refinement step requires $\textbf{masku}_j$ and $\textbf{maskv}_j$. For each matrix $\mathbf{W}_j$, the byte size of matrix decomposition $\mathbf{W}'_j$ **size** includes $\mathbf{v}_j$ **size**, $\mathbf{u}_j$ **size**, $\textbf{maskv}_j$ **size**, and $\textbf{masku}_j$ **size**.

$$\mathbf{u}_j \textbf{ size} = N_{steps} \cdot \text{Precision} \cdot NZr \cdot Tr \qquad \textbf{masku}_j \textbf{ size} = N_{steps} \cdot \frac{M}{Tr}$$

$$\mathbf{v}_j \textbf{ size} = N_{steps} \cdot \text{Precision} \cdot NZc \cdot Tc \qquad \textbf{maskv}_j \textbf{ size} = N_{steps} \cdot \frac{N}{Tc}$$

$$\mathbf{W}'_j \textbf{ size} = v_j \text{ size} + u_j \text{ size} + \text{maskv}_j \text{ size} + \text{masku}_j \text{ size}$$

The total byte size of matrix decomposition of N_{mvm} MVMs is expressed as:

$$\mathbf{W'\ size} = N_{mvm} \cdot N_{steps} \cdot \left[\text{Precision} \cdot (NZc \cdot Tc + NZr \cdot Tr) + \left(\frac{N}{Tc} + \frac{M}{Tr} \right) \right]$$

The total memory access of this SVD kernel type is expressed as:

$$\mathbf{mem\ access\ (byte)} = \text{input} + \text{output} + W' \text{ size}$$

4.2.2 SVD Kernel for Group Matrix Approximation

Architecture for Independent MVMs

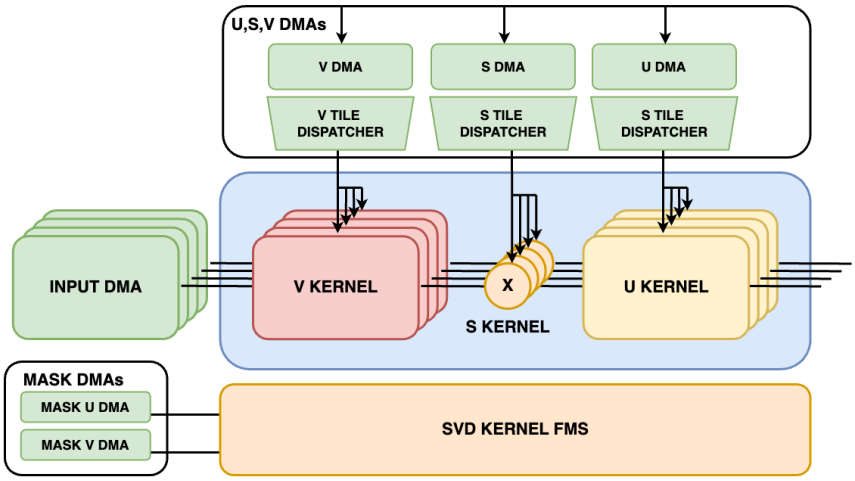


Figure 4.5: SVD kernel optimised for group matrix approximation: the kernel shown in the figure has four datapath and supports four independent MVMs operating in parallel

The architecture of the SVD kernel optimised for group matrix approximation, adapted from the design proposed by Ribes et al. [12], is illustrated in Figure 4.5. This SVD kernel can perform N_{mvm} independent MVMs using a group matrix approximation strategy.

All MVMs share the same vector $\mathbf{v^{i(n)}}$ and $\mathbf{u^{i(n)}}$, but have different input vectors $\mathbf{x}_j$, and scalars $\mathbf{s_j^{i(n)}}$. During computation, the shared v DMA sequentially broadcast the non-zero tiles of vector $\mathbf{v^{i(n)}}$ to separate v kernel. Concurrently, separate input dispatcher delivers the corresponding input tiles $\mathbf{x}_j$ to respective v kernel. The v kernel computes the following dot product similar to Equation 4.2.

The dot product $\mathbf{vx}_j^{i(n)}$ is then multiplied by the scalars $\mathbf{s}_0^{i(n)}$, $\mathbf{s}_1^{i(n)}$ ... $\mathbf{s}_j^{i(n)}$ separately to obtain $\mathbf{svx}_j^{i(n)}$ and forwarded to separate u kernels to maintain throughput. Concurrently, the u DMA sequentially broadcasts the non-zero tiles of vector $\mathbf{u}^{i(n)}$ to $\mathrm{N_{mvm}}$ parallel u kernels in the datapath. Each u kernel performs the computation over $\mathrm{N_{steps}}$ refinement steps as shown in Equation 4.5.

$$\mathbf{svx}_j^{\mathbf{i(n)}} = \mathbf{vx}^{\mathbf{i(n)}} \cdot \mathbf{s}_j^{\mathbf{i(n)}}, \quad \forall j \in [0, \mathrm{N_{mvm}} - 1], \quad \forall n \in [1, \mathrm{N_{steps}}] \tag{4.4}$$

$$\mathbf{W}_j \cdot \mathbf{x}_j \approx \sum_{n=1}^{\mathrm{N_{steps}}} (\mathrm{masku}^{\mathrm{i(n)}} \odot \mathbf{u}^{\mathbf{i(n)}}) \cdot \mathbf{svx}_j^{\mathbf{i(n)}}, \quad \forall j \in [0, \mathrm{N_{mvm}} - 1] \tag{4.5}$$

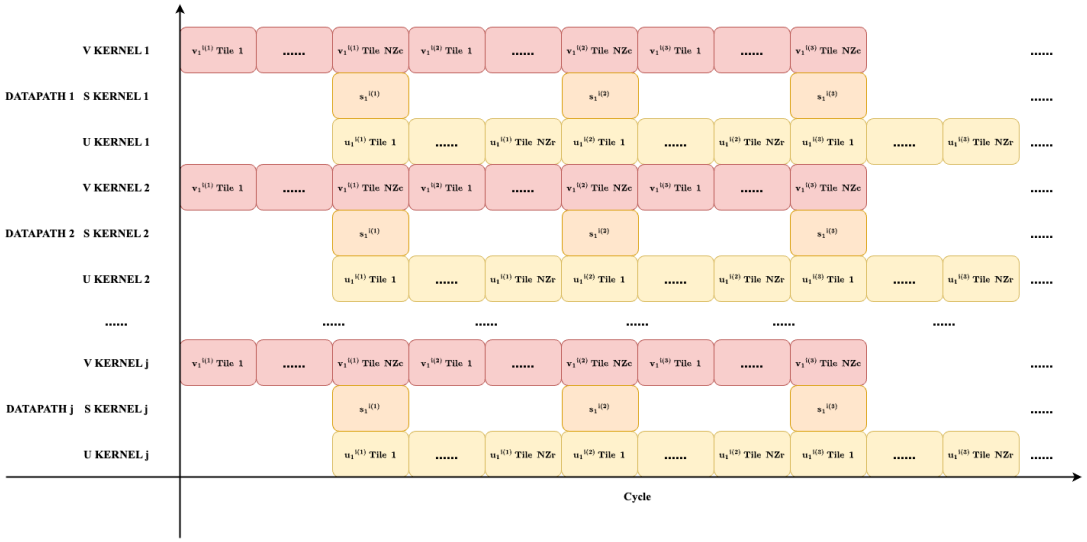


Figure 4.6: Scheduling diagram: SVD kernel for group matrix approximation

Architecture for Dependent MVMs

In the case of dependent MVMs, all v kernels in the datapath perform the same dot product operation, leading to inefficient hardware resource utilisation. To enhance hardware efficiency, the previous architecture is optimised specifically for computing dependent MVMs. The optimised SVD kernel features a single input dispatcher and a shared v kernel, which is utilised across all MVMs. To maintain the same throughput, the shared v kernel computes the dot product and then distributes the result to separate s kernels and u kernels for the next stage of the approximate MVM. This restructured SVD kernel architecture allows for the concurrent execution of $\mathrm{N_{mvm}}$ dependent MVMs using a group matrix approximation strategy, with improved hardware utilisation.

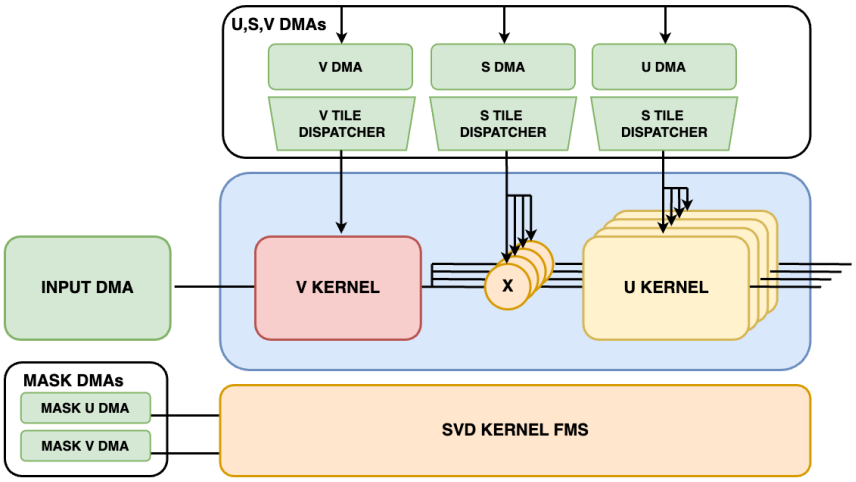


Figure 4.7: SVD kernel optimised for group matrix approximation: the kernel shown in the figure has one datapath and supports four dependent MVMs operating in parallel

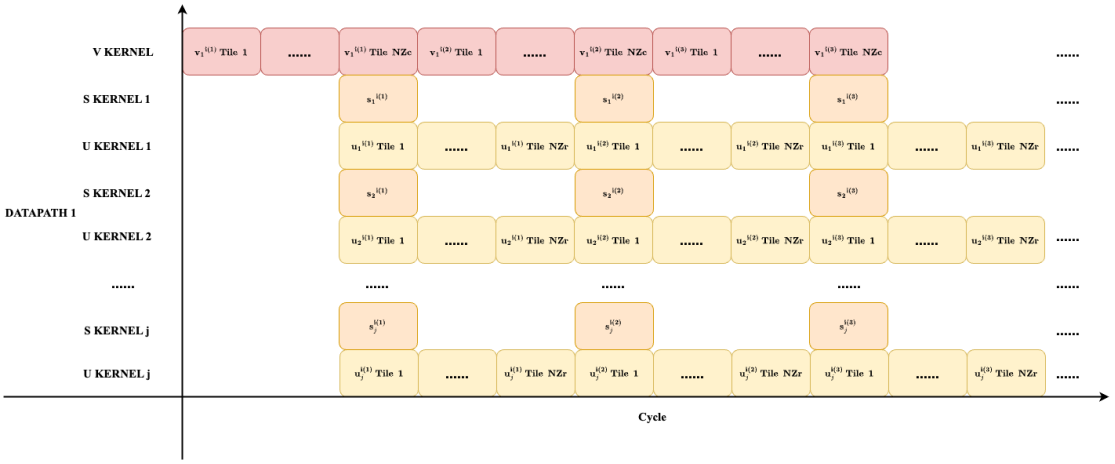


Figure 4.8: Scheduling diagram: SVD kernel for group matrix approximation

Memory Access

The total memory access of this SVD kernel comprises the following components. The memory access due to the input and output vectors of N_{mvm} independent or dependent MVMs is the same as before.

Similarly, the u and v DMA units only stream in the non-zero tiles of vectors $\mathbf{v^{i(n)}}$ and $\mathbf{u^{i(n)}}$. The total number of non-zero tiles per step are NZc and NZr, respectively. The retrieved data are supplies to the v kernel(s) and all u kernels in the datapath. In addition, group matrix approximation requires scalars $\mathbf{s}_j^{i(n)}$, with each s scalar being dispatched to respective s kernel for

computation at each refinement step.

$$\textbf{u size} = \mathrm{N_{steps}} \cdot \mathrm{Precision} \cdot \mathrm{NZr} \cdot \mathrm{Tr} \qquad \textbf{masku size} = \mathrm{N_{steps}} \cdot \frac{\mathrm{M}}{\mathrm{Tr}}$$

$$\textbf{v size} = \mathrm{N_{steps}} \cdot \mathrm{Precision} \cdot \mathrm{NZc} \cdot \mathrm{Tc} \qquad \textbf{maskv size} = \mathrm{N_{steps}} \cdot \frac{\mathrm{N}}{\mathrm{Tc}}$$

$$\textbf{s size} = \mathrm{N_{steps}} \cdot \mathrm{N_{mvm}} \cdot \mathrm{Precision}$$

The total byte size of matrix decomposition $\mathbf{W'}$ **size** includes **v size**, **s size**, **u size**, **maskv size**, and **masku size**.

$$\begin{aligned}\mathbf{W'\ size} &= \mathrm{v\ size} + \mathrm{u\ size} + \mathrm{s\ size} + \mathrm{maskv\ size} + \mathrm{masku\ size} \\ &= \mathrm{N_{steps}} \cdot \left[\mathrm{Precision} \cdot (\mathrm{NZc} \cdot \mathrm{Tc} + \mathrm{N_{mvm}} + \mathrm{NZr} \cdot \mathrm{Tr}) + \left(\frac{\mathrm{N}}{\mathrm{Tc}} + \frac{\mathrm{M}}{\mathrm{Tr}}\right)\right]\end{aligned}$$

Finally, the total memory access of this SVD kernel is expressed as:

$$\textbf{mem access (byte)} = \mathrm{input} + \mathrm{output} + \mathrm{W'\ size}$$

4.2.3 SVD Kernel for Hybrid Matrix Approximation

Architecture

The datapath design of the SVD kernel optimised for hybrid matrix approximation is identical to that of the SVD kernel used for group matrix approximation with independent MVMs. The primary difference lies in the dispatch logic. In hybrid matrix approximation, the decomposition is shared dynamically due to different groupings. When streamed into the SVD kernel, each vector $\mathbf{u^{i(n)}}$, $\mathbf{v^{i(n)}}$ and scalars $\mathbf{s^{i(n)}}$ are encoded with the kernel index to which they belong. The dispatcher sequentially broadcasts the tiles corresponding to the kernel, as indicated by the kernel index. The dynamic nature of the tile dispatch logic allows for efficient handling of different matrix groupings across refinement steps.

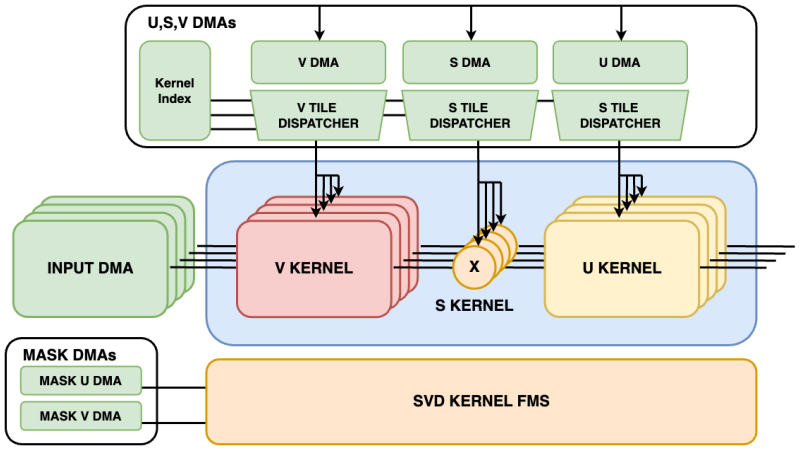


Figure 4.9: SVD kernel for stack matrix approximation: the kernel has only one datapath and supports N_{mvm} dependent MVMs operating in parallel

Memory Access

The total memory access of this SVD kernel comprises the following components. The memory access due to the input and output vectors of N_{mvm} independent or dependent MVMs is the same as before. In hybrid approximation, the total byte size of matrix decomposition $\mathbf{W'}$ **size** depends on the number of groups $N_{group}^{\mathbf{i(n)}}$ at each refinement step.

$$\begin{aligned}
\textbf{u size per step} &= N_{group}^{\mathbf{i(n)}} \cdot \text{Precision} \cdot \text{NZr} \cdot \text{Tr} \qquad \textbf{masku size per step} &= N_{group}^{\mathbf{i(n)}} \cdot \frac{\text{M}}{\text{Tr}} \\
\textbf{v size per step} &= N_{group}^{\mathbf{i(n)}} \cdot \text{Precision} \cdot \text{NZc} \cdot \text{Tc} \qquad \textbf{maskv size per step} &= N_{group}^{\mathbf{i(n)}} \cdot \frac{\text{N}}{\text{Tc}}
\end{aligned}$$

$$\textbf{s size per step} = N_{group}^{\mathbf{i(n)}} \cdot \text{Precision}$$

$$\begin{aligned}
\mathbf{W'\ size} &= \text{v size} + \text{u size} + \text{s size} + \text{maskv size} + \text{masku size} \\
&= \sum_{n=1}^{N_{steps}} N_{group}^{\mathbf{i(n)}} \cdot \left[\text{Precision} \cdot (\text{NZc} \cdot \text{Tc} + 1 + \text{NZr} \cdot \text{Tr}) + \left(\frac{\text{N}}{\text{Tc}} + \frac{\text{M}}{\text{Tr}} \right) \right]
\end{aligned}$$

Finally, the total memory access of this SVD kernel is expressed as:

$$\textbf{mem access (byte)} = \text{input} + \text{output} + \text{W}' \text{ size}$$

4.2.4 SVD Kernel for Stack Matrix Approximation

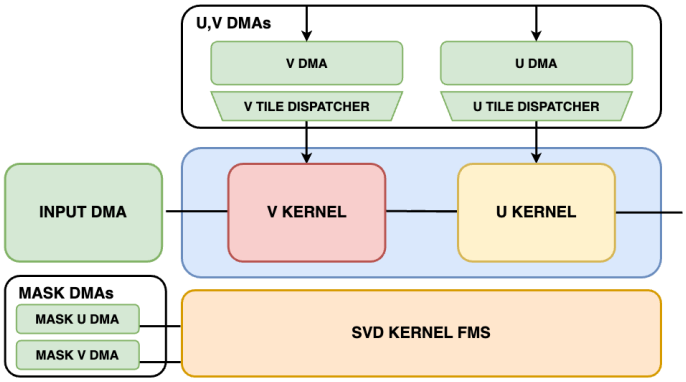


Figure 4.10: SVD kernel for stack matrix approximation: the kernel has only one datapath and supports N_{mvm} dependent MVMs operating in parallel

The architecture of the SVD kernel optimised for stack matrix approximation is illustrated in Figure 4.10. This SVD kernel can perform N_{mvm} dependent MVMs using the stack matrix approximation strategy, as elaborated in Section 3.3. These dependent MVMs use different matrices but share the same input vector.

$$\mathbf{y} = \begin{bmatrix} \mathbf{W}_0 \\ \mathbf{W}_1 \\ \cdots \\ \mathbf{W}_j \end{bmatrix} \cdot \mathbf{x}, \quad \forall j \in [0, N_{mvm} - 1] \tag{4.6}$$

The SVD kernel computes approximate MVM for the stacked matrix as expressed in Equation 4.7. The kernel consists of a single datapath, following a similar construct as discussed in Section 4.2.1. Both the v kernel and the u kernel are parameterisable with respect to tile sizes Tc and Tr. The shared input vector is stored in a single input buffer of the input DMA. Notably, the depth of the accumulation RAM in the u kernel is scaled in proportion to N_{mvm} because the accumulation RAM need to store the matrix-vector product of the stacked matrix. Once all refinement steps are complete, the u kernel streams out an approximated matrix-vector product that comprises N_{mvm} approximated matrix-vector products as expressed in Equation 4.8.

$$\mathbf{y}' \approx \sum_{n=1}^{N_{steps}} [(\text{maskv}^{\mathbf{i(n)}} \odot \mathbf{v}^{\mathbf{i(n)}})^{\mathrm{T}} \cdot (\text{maskv}^{\mathbf{i(n)}} \odot \mathbf{x})] \odot (\text{masku}^{\mathbf{i(n)}} \odot \mathbf{u}^{\mathbf{i(n)}}) \tag{4.7}$$

$$\mathbf{y}_j \approx \begin{bmatrix} \mathrm{y}'_{jM} \\ \mathrm{y}'_{jM+1} \\ \cdots \\ \mathrm{y}'_{(j+1)M} \end{bmatrix}, \quad \forall j \in [0, \mathrm{N_{mvm}} - 1] \tag{4.8}$$

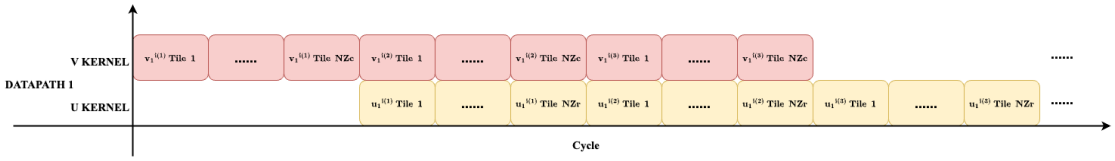


Figure 4.11: Scheduling diagram: SVD kernel for stack matrix approximation

Memory Access

The total memory access of this SVD kernel type comprises the following components. As before, the input vector is shared by $\mathrm{N_{mvm}}$ dependent MVMs and is accessed only once. Similar to single matrix approximation, the u and v DMA units only stream in the non-zero tiles of vectors $\mathbf{v}^{i(n)}$ and $\mathbf{u}^{i(n)}$. The total number of non-zero tiles per step are NZc and NZr, respectively. Additionally, each refinement step requires compressed mask **masku** and **maskv**.

The total byte size of matrix decomposition $\mathbf{W}'$ **size** includes **v size**, **u size**, **maskv size**, and **masku size**. Note that for stack matrix approximation, each **masku** has size $\frac{\mathrm{N_{mvm}} \cdot \mathrm{M}}{\mathrm{Tr}}$.

$$\textbf{u size} = \mathrm{N_{steps}} \cdot \mathrm{Precision} \cdot \mathrm{NZr} \cdot \mathrm{Tr} \qquad \textbf{masku size} = \mathrm{N_{steps}} \cdot \frac{\mathrm{N_{mvm}} \cdot \mathrm{M}}{\mathrm{Tr}}$$

$$\textbf{v size} = \mathrm{N_{steps}} \cdot \mathrm{Precision} \cdot \mathrm{NZc} \cdot \mathrm{Tc} \qquad \textbf{maskv size} = \mathrm{N_{steps}} \cdot \frac{\mathrm{N}}{\mathrm{Tc}}$$

The total byte size of the matrix decomposition for $\mathrm{N_{mvm}}$ dependent MVMs is expressed as:

$$\begin{aligned} \mathbf{W}' \textbf{ size} &= \textbf{v size} + \textbf{u size} + \textbf{maskv size} + \textbf{masku size} \\ &= \mathrm{N_{steps}} \cdot \left[\mathrm{Precision} \cdot (\mathrm{NZc} \cdot \mathrm{Tc} + \mathrm{NZr} \cdot \mathrm{Tr}) + \left(\frac{\mathrm{N}}{\mathrm{Tc}} + \frac{\mathrm{N_{mvm}} \cdot \mathrm{M}}{\mathrm{Tr}} \right) \right] \end{aligned}$$

The total memory access of this SVD kernel type is expressed as:

$$\textbf{mem access (byte)} = \mathrm{input} + \mathrm{output} + \mathrm{W}' \ \mathrm{size}$$

4.3 Run-time Parameters

When deployed on the FPGA, the pipeline control logic of the accelerator leverages these run-time parameters to manage data flow and control the execution of SVD kernels.

Table 4.1: List of Run Time Parameters of the Accelerator

Symbol	Description
$\mathbf{N_{steps}}$	Number of refinement steps.
M	The number of rows of matrix $\mathbf{W}_j$
N	The number of columns of matrix $\mathbf{W}_j$
NZr	Number of non-zero tiles of vector $\mathbf{u}^{(i)}$ in each refinement step
NZc	Number of non-zero tiles of vector $\mathbf{v}^{(i)}$ in each refinement step

4.4 Architectural Parameters

The table below summarises all the architectural parameters of the accelerator. During FPGA configuration, these parameters define the memory access pattern of the DMA, the size of DMA buffers, the accumulation RAM, the width of MAC arrays, the multiplier-adder tree, and other component characteristics. These parameters directly impact the resource usage and performance of the accelerator. The top-level framework leverages these architectural parameters to identify high-performance configurations under hardware resource constraints, as elaborated in the next chapter.

Table 4.2: List of Architectural Parameters of the Accelerator

Symbol	Description
$\mathbf{N_{SVD\ kernel}}$	Number of SVD kernels in the accelerator
Kernel Type	The approximation strategy of each SVD kernel in the accelerator
$\mathbf{N_{mvm}}$	The number of MVMs each SVD kernel support
M	The number of rows of matrix $\mathbf{W}_j$
N	The number of columns of matrix $\mathbf{W}_j$
Tr	The parallelism of u kernel in each SVD kernel
Tc	The parallelism of v kernel in each SVD kernel
Precision	Byte size of the quantised input and matrix values

5

Framework

Contents

This chapter explains the framework developed to identify design points with optimal performance. Additionally, the chapter demonstrates the accelerator model employed by the framework to estimate theoretical performance and resource utilisation.

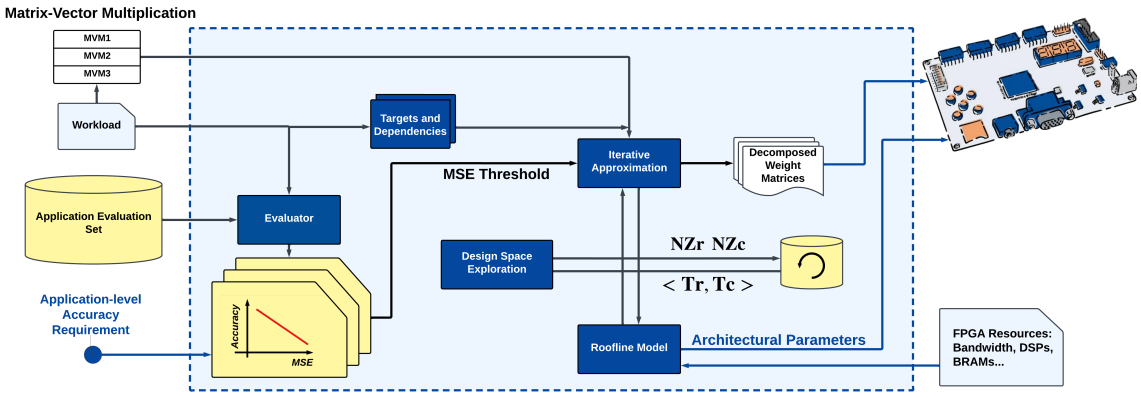


Figure 5.1: Framework Overview

5.1 Design Parameters

As shown in Table 3.1, 4.1, and 4.2 a variety of design parameters influence the efficiency of matrix-vector product computations using approximation.

Each design parameter can take on a wide range of possible values, making the design space huge and impractical to search exhaustively. Besides, if there are accuracy requirements at the application level, the framework must run an application evaluation set for all possible design points, which introduces significant overhead. To address these challenges in a structured manner, the following methodologies has been developed to identify a set of design points that achieve better accuracy-latency trade-offs while meeting hardware resource constraints.

5.2 Framework Methodology

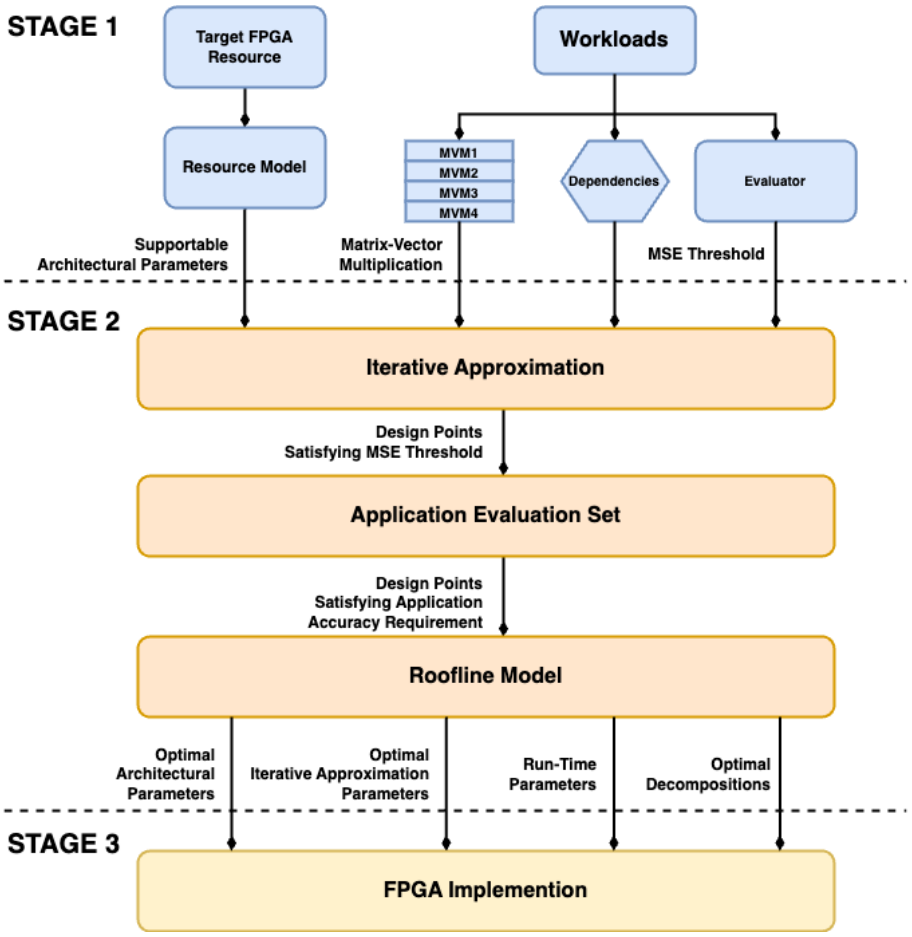


Figure 5.2: Framework Flowchart

Figure 5.2 illustrates the framework's flowchart. The framework is divided into three distinct stages, as described below.

In stage 1, the framework decomposes the workload into a sequence of matrix-vector operations and identifying critical targets and dependencies that govern the sequence. These dependencies and targets pose constraints on how the matrices can be approximated. Concurrently, the Evaluator leverages the application's evaluation dataset to determine a MSE threshold for iterative approximation that aligns with the application-level accuracy requirements. Given the FPGA hardware resources, such as the number of DSPs and BRAMs, the framework narrows down the design space by identifying the feasible number of SVD kernels ($N_{\text{SVD kernel}}$) and the extent of parallelism (Tr and Tc) achievable within each engine. To estimate resource utilisation without actual hardware implementation, an analytical resource model is developed. Design points that consume excessive resources are eliminated. The remaining design points which satisfies the resource constraint proceed to the next step.

In stage 2, the framework applies a collection of iterative approximation strategies and determines a set of design points that satisfy the specified MSE threshold. These design points are evaluated using the application evaluation dataset. Design points that do not meet the application-level accuracy targets are discarded. The design points that satisfies the application-level accuracy requirements are selected for design space exploration. For these selected design points, roofline models are instantiated, and their theoretical performance is estimated. The framework then performs design space exploration to identify the design point that achieves the lowest latency while meeting both application-level accuracy requirements and hardware resource constraints.

In stage 3, the FPGA architectural parameters of the selected design point are used to configure the hardware accelerator. The matrix decompositions are then loaded into the off-chip memory of the target FPGA for acceleration.

5.3 Accelerator Models

In the following section, the SVD kernels are assumed to have a homogeneous configuration as discussed in Section 4.1.2, though a similar accelerator model can be created for heterogeneous configurations. In a homogeneous configuration, all the SVD kernels in the accelerator have the same level of parallelism and use the same approximation strategy. Additionally, each SVD kernel approximates the same number of MVMs and uses the same number of refinement steps Nsteps.

5.3.1 Hardware Resource Model

FPGA prototyping reveals that DSPs and BRAMs are the most critical resources. To estimate resource utilisation and eliminate design points that exceed available resources, a resource model targeting these two components has been developed. The target device for this resource model is Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA.

DSP Model

Inside the SVD kernel, DSP usage come from the following sources: v kernel, u kernel and possibly s kernel. The v kernel is a multiplier-adder tree with width Tc. The u kernel is a MAC array with width Tr. The s kernel is a single multiplier. All fixed-point arithmetic unit are mapped to DSPs.

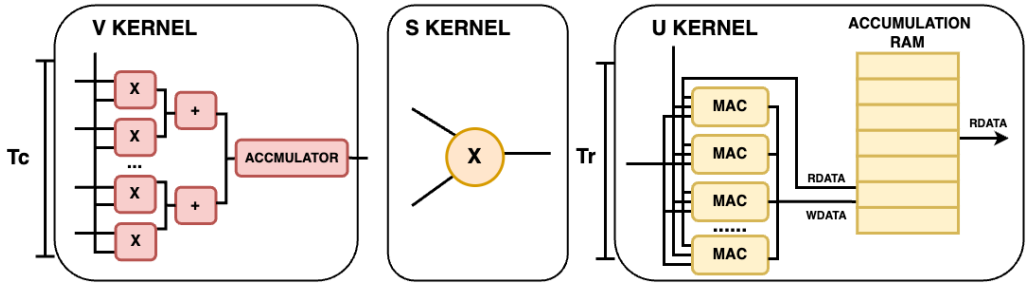


Figure 5.3: Structure of V Kernel, S Kernel and U Kernel

The DSP utilisation of a single adder, single multiplier and single accumulator are defined as DSP_{adder}, $DSP_{multiplier}$, and $DSP_{accumulator}$, respectively. The DSP utilisation of each component is listed in Table **??**.

Table 5.1: DSP Usage of Each Component

Symbol	**Number of DSPs**
$\mathbf{DSP_{v\ kernel}}$	$Tc \cdot DSP_{multiplier} + (Tc\ -1) \cdot DSP_{adder} + DSP_{accumulator}$
$\mathbf{DSP_{s\ kernel}}$	$DSP_{multiplier}$
$\mathbf{DSP_{u\ kernel}}$	$Tr \cdot DSP_{multiplier} + Tr \cdot DSP_{adder}$

The accelerator has $N_{SVD\ kernel}$ homogeneous SVD kernels and each kernel approximates N_{mvm} MVMs. Therefore, the total DSP usage of each SVD kernel type can be estimated as:

Table 5.2: DSP Usage for Each SVD Kernel Type

Symbol	**Number of DSPs**
$\mathbf{DSP_{Single}}$	$N_{mvm} \cdot [\, DSP_{v\ kernel} + DSP_{u\ kernel} \,]$
$\mathbf{DSP_{Stack}}$	$DSP_{v\ kernel} + DSP_{u\ kernel}$
$\mathbf{DSP_{Group}}$	$DSP_{v\ kernel} + N_{mvm} \cdot [\, DSP_{s\ kernel} + DSP_{u\ kernel} \,]$
$\mathbf{DSP_{Hybrid}}$	$N_{mvm} \cdot [\, DSP_{v\ kernel} + DSP_{s\ kernel} + DSP_{u\ kernel} \,]$

BRAM Model

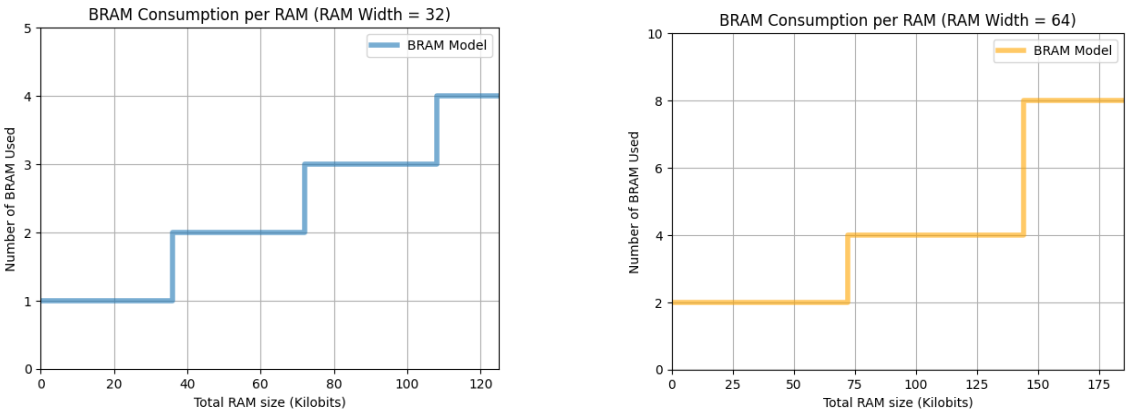
Within the accelerator, BRAM usage mainly comes from the following sources: the input vector buffer and the accumulation RAM in the u kernel. The BRAM usage due to the DMA buffer is considered minimal, as the buffer only needs to store the matrix decomposition of one refinement step, and is thus neglected in the model.

Each input vector buffer has a data width of Tc $\times$ Precision to supply the entire input tile in parallel. Similarly, each accumulation RAM has a data width of Tr $\times$ Precision to allow parallel access by the MAC array. The size of both the input vector buffer and the accumulation RAM are contingent on the matrix size of the MVMs. Specifically, each input vector buffer has size N$\times$Precision, and each accumulation RAM has size N$\times$Precision. It is important to highlight that for stack matrix approximation, each accumulation RAM has size $N_{mvm}\times$M$\times$Precision.

For the target device, BRAMs come in fixed sizes of 36 Kb and support various data widths, such as 8-bit, 16-bit, and 32-bit, with a maximum data width of 32-bit. The total depth of the BRAM changes depending on the data width. For example, a 36Kb BRAM can store:

- 4K 8-bit entries
- 2K 16-bit entries
- 1K 32-bit entries

If the required RAM size does not exceed the capacity of a single BRAM, a single BRAM is used. For larger RAM sizes that exceed the capacity of a single BRAM, multiple BRAMs are employed. Similarly, for RAM that exceeds the maximum data width, multiple BRAMs are used. Therefore, the BRAM can be modeled as a step function, as shown in Figure 6.18.



(a) (b)

Figure 5.4: BRAM Consumption per RAM with data width 32-bit and 64-bit

The accuracy of the resource model is evaluated the Chapter 6.

5.3.2 Roofline Models

Methods as discussed in works [1], [13], are used to create a roofline model for the proposed architecture. This model establishes a relationship between the maximum possible throughput for each configuration and its operational intensity, which is the ratio of computational workload to off-chip memory traffic. This model is crucial for predicting performance across different design points.

As shown in SVD scheduling diagram of the SVD kernels, the number of cycles to compute one refinement step can be estimated as the critical path between the v kernel and u kernel. The v kernel and u kernel take NZc and NZr cycles respectively to compute one refinement step. In addition, the cycles required to read out the final matrix-vector product, denoted as $\mathrm{OUT_{cycles}}$, are taken into account. Therefore, the total number of cycles to perform one approximate MVM can be expressed as:

$$\mathbf{Ncycles} = \max\left(\mathrm{NZc} \cdot \mathrm{N_{steps}}\,,\, \mathrm{NZr} \cdot \mathrm{N_{steps}}\right) + \mathrm{OUT_{cycles}} \tag{5.1}$$

Table 5.3: Number of Cycles Required to Read Out the Final Result

Approximation	$\mathbf{OUT_{cycles}}$
Single, Group, Hybrid	$\frac{\mathrm{M}}{\mathrm{Tr}}$
Stack	$\frac{\mathrm{N_{mvm}} \cdot \mathrm{M}}{\mathrm{Tr}}$

The total number of fix-point operations to approximate all MVMs is defined as Nops. The value of Nops depends on number of SVD kernels $\mathrm{N_{SVD\ kernel}}$, $\mathrm{N_{mvm}}$ per SVD kernel, and the approximation strategy used. The Nops for different approximation strategies are listed in Table 5.4.

Table 5.4: Total Number of Fix-point Operations for Different Approximation Strategies

Approximation	**Number of Fix-Point Operations (Nops)**
Single	$\mathrm{N_{SVD\ kernel}} \cdot \mathrm{N_{mvm}} \cdot \mathrm{N_{steps}}\,[2\mathrm{NZc} \cdot \mathrm{Tc} + 2\mathrm{NZr} \cdot \mathrm{Tr}]$
Stack	$\mathrm{N_{SVD\ kernel}} \cdot \mathrm{N_{steps}}\,[2\mathrm{NZc} \cdot \mathrm{Tc} + 2\mathrm{NZr} \cdot \mathrm{Tr}]$
Group	$\mathrm{N_{SVD\ kernel}} \cdot \mathrm{N_{steps}}\,[2\mathrm{NZc} \cdot \mathrm{Tc} + \mathrm{N_{mvm}}\,(1 + 2\mathrm{NZr} \cdot \mathrm{Tr})]$
Hybrid	$\mathrm{N_{SVD\ kernel}} \cdot \mathrm{N_{steps}} \cdot \mathrm{N_{mvm}}\,[2\mathrm{NZc} \cdot \mathrm{Tc} + 1 + 2\mathrm{NZr} \cdot \mathrm{Tr}]$

For each refinement step, each v kernel performs $2\mathrm{NZc} \times \mathrm{Tc}$ fix-point operations to compute the dot product, which includes Tc multiplication operations and Tc addition operations per cycle. Similarly, each u kernel performs $2\mathrm{NZc} \times \mathrm{Tc}$ fix-point operations to multiply the u tile with the dot product (Tr multiply operations per cycle) and accumulate them in the accumulation ram (Tr add operations per cycle). Note that the group approximation strategy also requires $\mathrm{N_{mvm}}$ multiply operations to multiply the scalars $\mathbf{s_j^{i(n)}}$.

The Computational Performance (CP) is defined as the total number of fix-point operations per second the accelerator can perform. CP can be estimated using total number of cycles Ncycles and the operating frequency of FPGA as:

$$\mathbf{CP} = \frac{\text{Nops} \cdot \text{frequency}}{\text{Ncycles}} \left[\frac{\text{ops}}{\text{s}}\right] \tag{5.2}$$

The Computation-to-Communication (CTC) ratio is defined as the total number of fix-point operations per memory access in bytes. Memory access for different types of SVD kernels are detailed in Chapter 4.

$$\mathbf{CTC} = \frac{\text{Nops}}{\sum \text{mem access (byte)}} \left[\frac{\text{ops}}{\text{byte}}\right] \tag{5.3}$$

In a digital system, the maximum achievable performance is constrained by either the platform's peak throughput or highest data throughput that the platform's memory system can sustain. Therefore, the attainable performance of the accelerator is the minimum value between the Computational Performance (CP) and the product of the maximum available memory bandwidth of the FPGA and the Communication-to-Computation (CTC) ratio.

$$\mathbf{Att_{perf}} = \min(\text{ CP }, \text{ CTC} \cdot \text{Bandwdith }) \left[\frac{\text{ops}}{\text{s}}\right] \tag{5.4}$$

The execution time for approximation can be estimated by dividing the total number of fixed-point operations by the attainable performance of the accelerator.

$$\mathbf{t_{exe}} = \frac{\text{Nops}}{\text{Att}_{\text{perf}}} \left[\frac{\text{ops}}{\text{ops/s}}\right] \tag{5.5}$$

It is also noteworthy that if the accelerator is not memory-bound, i.e., the compute performance (CP) is smaller than the product of bandwidth and the compute-to-communication (CTC) ratio, the execution time is simply:

$$\mathbf{t_{exe}} = \frac{\text{Nops}}{\text{CP}} = \frac{\text{Ncycles}}{\text{Frequency}} \left[\frac{\text{ops}}{\text{ops/s}}\right] \tag{5.6}$$

In the roofline model, the design points with low execution time are promising for achieving better performance in actual FPGA implementation. This is explored in the following chapter.

6 Evaluation

Contents

In this chapter, the proposed framework framework is evaluated using Long Short-Term Memory (LSTM) workload. The performance and resource utilisation of actual FPGA implementations are tested and compared with model estimations. Additionally, the framework's performance are compared against baseline and state-of-the-art methods.

6.1 Experimental Setup

Two Long Short Term Memory (LSTM) workloads are used to demonstrate the applicability of the proposed framework.

6.1.1 Long Short-Term Memory (LSTM) Networks

Long Short-Term Memory network (LSTM) is a type of neural network that incorporate feedback connections and internal memory cells to record past information and process sequences over arbitrary time intervals. These networks excel in a variety of sequence processing tasks, such as translation, speech processing, and video classification due to their ability to accurately capture long-term dependencies.

The LSTM model employed in this work is depicted schematically in Figure 6.1. The hidden state $\mathbf{h}[t]$ at any time step is computed using the following set of equations:

$$\begin{aligned}
\mathbf{i}[t] &= \sigma\left(\mathbf{W}_{ix}\mathbf{x}[t] + \mathbf{W}_{ih}\mathbf{h}[t-1]\right) \\
\mathbf{f}[t] &= \sigma\left(\mathbf{W}_{fx}\mathbf{x}[t] + \mathbf{W}_{fh}\mathbf{h}[t-1]\right) \\
\mathbf{g}[t] &= \tanh\left(\mathbf{W}_{gx}\mathbf{x}[t] + \mathbf{W}_{gh}\mathbf{h}[t-1]\right) \\
\mathbf{o}[t] &= \sigma\left(\mathbf{W}_{ox}\mathbf{x}[t] + \mathbf{W}_{oh}\mathbf{h}[t-1]\right) \\
\mathbf{c}[t] &= \mathbf{f}[t] \odot \mathbf{c}[t-1] + \mathbf{i}[t] \odot \mathbf{g}[t] \\
\mathbf{h}[t] &= \mathbf{o}[t] \odot \tanh\left(\mathbf{c}[t]\right)
\end{aligned} \tag{6.1}$$

In these equations, σ denotes the sigmoid activation function, tanh() is the hyperbolic tangent function, and $\odot$ represents element-wise multiplication. The variables i, f, g, and o represent the input, forget, input modulation, and output gates, respectively. The symbols $\mathbf{W}_x$ and $\mathbf{W}_h$ represent the weight matrices associated with the input and the hidden states, respectively. The internal memory state is denoted by $\mathbf{c}[t]$, and $\mathbf{h}[t]$ is defined as the output of the LSTM cell, which is transferred to the next time step or an external layer. The LSTM gates regulate the flow of information within the unit: the input gate selects new information to be incorporated into the memory cell; the forget gate identifies what prior information should be discarded; the input modulation gate modulates the contribution of new input to the cell state; and the output gate determines the next hidden state of the LSTM [14].

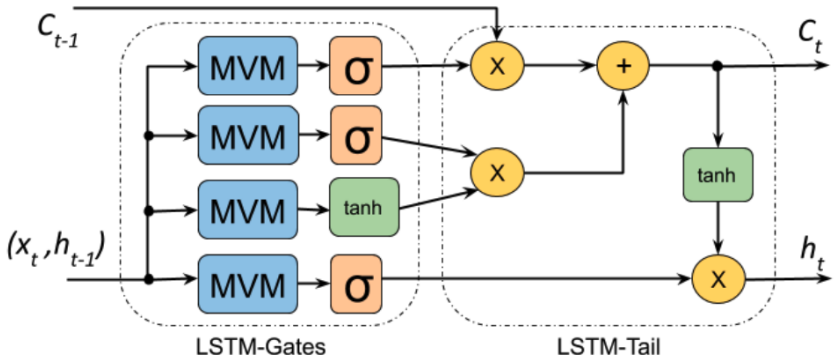


Figure 6.1: LSTM Cell Structure [14].

If there is no temporal dependencies, the input and hidden state matrices can be concatenated together. Similarly, the weight matrices of the input vector and hidden state can be augmented together. Therefore, the set of LSTM equations can be generalised by Equation 6.2.

$$\begin{aligned} \mathbf{x}[t] &= \begin{bmatrix} \mathbf{x}[t] \\ \mathbf{h}[t-1] \end{bmatrix} \\ \mathbf{W}_j &= [\, \mathbf{W}_{jx},\, \mathbf{W}_{jh} \,], \quad \forall j \in [1,4] \\ \mathbf{gate}_j &= \text{nonlin}\,(\mathbf{W}_j \mathbf{x}[t]), \quad \forall j \in [1,4] \end{aligned} \tag{6.2}$$

The computation of a LSTM cell can be transformed into a set of four dependent MVMs. These MVMs involve four distinct augmented weight matrices $\mathbf{W}_j$ and a shared concatenated input $\mathbf{x}[t]$. After performing the MVMs, the LSTM back-end adds biases to these matrix-vector products, applies non-linear activation functions, and carries out a sequence of element-wise operations to complete the LSTM computation.

The proposed framework framework generates a hardware accelerator to accelerate the computation of this set of MVMs. The matrix-vector products calculated by the hardware accelerator are then plugged into the LSTM back-end model implemented using Keras 2.4.3 and TensorFlow 2.4.1 to verify their accuracy at application-level.

6.1.2 FashionLSTM

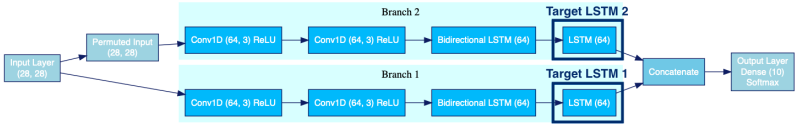


Figure 6.2: FashionLSTM Structure

The first workload, named FashionLSTM, is a multi-LSTM network trained on the Fashion MNIST dataset for image classification. This network consists of two branches: one branch operates on the original input, while the other branch processes a permuted version of the input. Each branch independently extracts temporal features from the sequential input data before combining the outputs for final classification. In the LSTM layer of this model, the dimension of the input vector $\mathbf{x}$[t] is 128 and the dimension of hidden state $\mathbf{h}$[t-1] is 64. The augmented weight matrices $\mathbf{W}_j$ have dimension 64×192. The target LSTM layers are highlighted in Figure 6.2.

The two target LSTM layers involve two sets of four dependent MVMs, each using different input vectors. These input vectors are extracted from the preceding bidirectional LSTM layers. The two set of MVMs are approximated separately by the framework. Iterative refinement stops when the average MSE of both sets of approximations reaches the sepcified MSE threshold. The two set of MVMs are allocated to two separate SVD kernels for computation. The accelerator is configured homogeneously, meaning that both SVD kernels employ the same approximation strategy, exhibit the same level of parallelism, and require the same number of refinement steps to approximate all MVMs.

6.1.3 HarLSTM

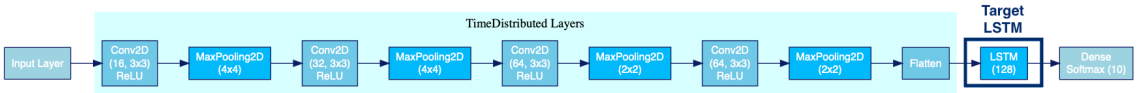


Figure 6.3: HarLSTM Structure

The second workload, named HarLSTM, is a single LSTM network trained on the UCF50 dataset for human activity recognition. This network processes sequential frames from video data to learn temporal dependencies and patterns associated with different human activities. In this LSTM model, the dimension of the input vector $\mathbf{x}$[t] is 64 and the dimension of hidden state $\mathbf{h}$[t-1] is 128. The augmented weight matrices $\mathbf{W}_j$ have dimension 128×192. The target LSTM layer is highlighted in Figure 6.3.

The target LSTM layer involves a single set of four dependent MVMs. These matrices are approximated by the proposed framework. Iterative refinement stops when the average MSE of the approximations reaches the specified MSE threshold. The input vector is extracted from the preceding TimeDistributed Layers. The set of MVMs is allocated to a single SVD kernel for computation.

6.1.4 Setup

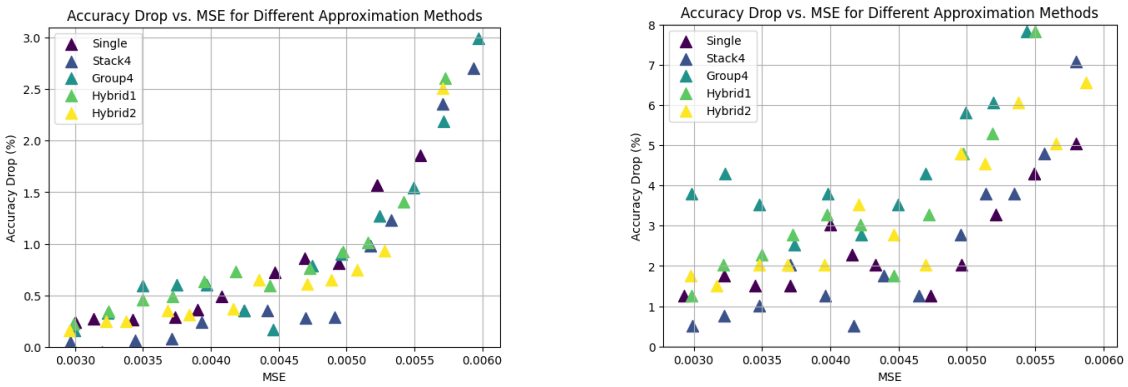
The target FPGA device for the evaluation is the Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA. The bandwidth is **10 GB/s**. The target frequency is **200 MHz**. The arithmetic units use **INT32** precision with 28 fractional bits. The proposed framework is compared with two hardware implementations:

- **Baseline:** The tiled matrix multiplication unit (TMMU) described in [5] serves as the baseline implementation. The baseline implementation does not involve approximation. For consistency with the architectural parameters defined in this work, the width of the TMMU is denoted as Tc, and the number of TMMUs is represented as Tr. Together, they constitute a single MVM engine. Each MVM operation of the LSTM is allocated to one MVM engine.
- **SVD HW1:** A hardware implementation following the design methodology described in [12] is considered the state-of-the-art. This approach focuses on an approximation strategy referred to as merging.

6.2 Accuracy Model

6.2.1 Accuracy Model Validation

The proposed framework assumes that as the MSE of an approximation approaches zero, the approximated weight matrix will more closely resemble the original, leading to the system's output converging with that of the unapproximated system. Therefore, MSE can serve as a proxy for application-level accuracy drop. To validate this assumption, design points with progressively decreasing MSE values are evaluated in terms of their application-level accuracies. As illustrated in Figure 6.4, a lower MSE generally corresponds to a smaller accuracy drop. However, there are instances where a low MSE results in a significant accuracy drop. Thus, selecting a design point with a low MSE is necessary but not sufficient for achieving a low accuracy drop at the application level. Consequently, design points meeting the MSE threshold are further evaluated at the application level for validation.



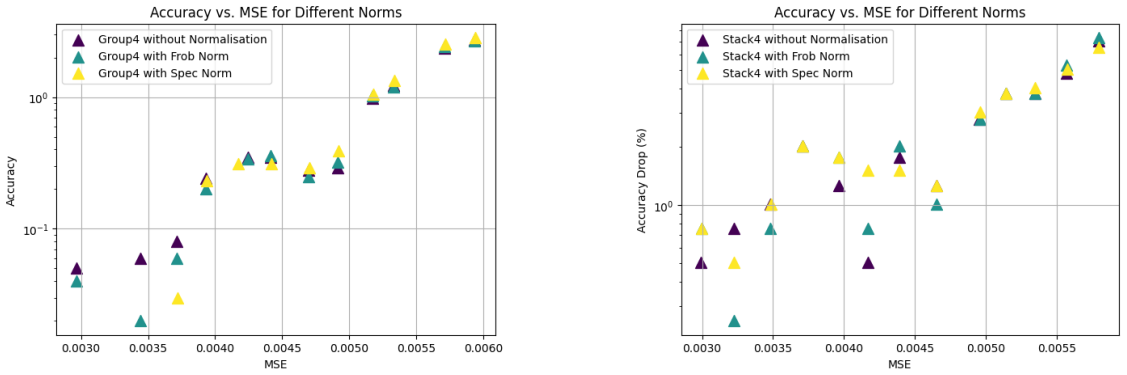
(a) FashinLSTM (b) HarLSTM

Figure 6.4: Correlation between accuracy drop and MSE of the approximation

6.2.2 Effect of Normalisation

The same setup is repeated using HarLSTM to evaluate the effect of normalisation on application-level accuracy, as discussed in Section 3.4. The accuracy drop for each design point is plotted against MSE for different normalisation methods.

As shown in Figure 6.5, the impact of normalisation on application-level accuracy varies across different MSE regions. In certain MSE ranges, specific normalisation methods yield better accuracy. Additionally, the effect of normalisation is influenced by factors such as dimensionality, magnitude and underlying distribution of the set of matrices. The outcome is not deterministic. Therefore, the framework exposes normalisation method as an additional parameter, allowing users to select the most appropriate normalisation method tailored to their specific application needs.



(a) Group Matrix Approximation

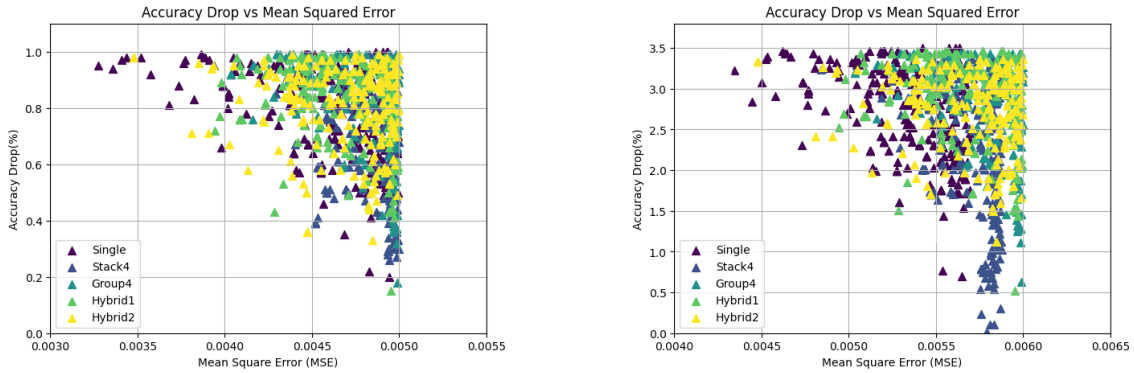
(b) Stack Matrix Approximation

Figure 6.5: Accuracy Drop vs. MSE for Different Normalisation Methods

6.3 Roofline Model

6.3.1 Roofline Model Analysis

The proposed roofline model is applied to the two LSTM workloads. For FashionLSTM, an MSE threshold of 0.005 is used, while for HarLSTM, the MSE threshold is set to 0.006. The application-level accuracy is then evaluated. The application-level accuracy tolerances are set to 1% and 3.5%, respectively. The design points, which have varying design parameters as listed in Section 5.1, are evaluated against these tolerances. Design points that exceed the accuracy tolerances are discarded. The accuracy and MSE of the selected design points are plotted in Figure 6.6. As shown in the figure, all selected design points fall within the specified MSE and accuracy thresholds. These design points are chosen for further design space exploration.



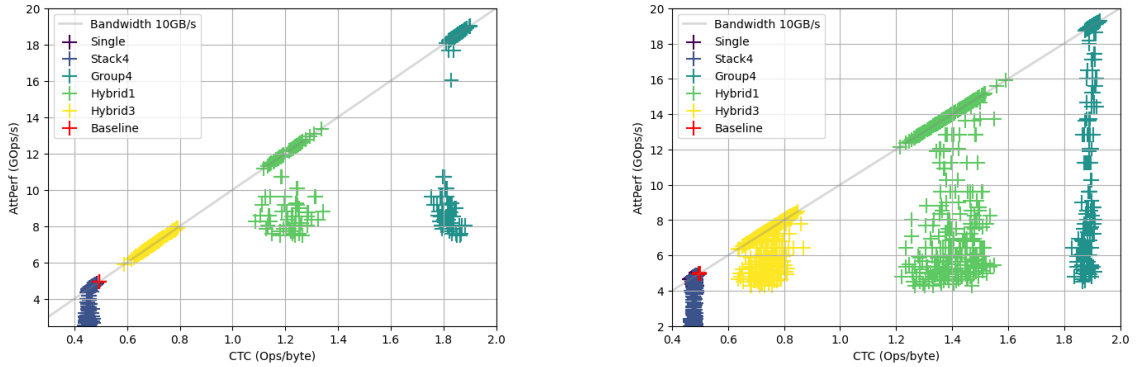
(a) FashionLSTM (b) HarLSTM

Figure 6.6: Design Points Meeting Application-Level Accuracy Requirements

The roofline models of accelerators processing the two LSTM workloads are plotted separately in Figure 6.7. For the target FPGA device, the memory bandwidth is constrained to 10GB/s. Under this memory constraint, a significant portion of the design points lie on the memory roof, indicating that their computations are memory-bound. FashionLSTM is particularly affected by this constraint as it consists of two LSTM units operating in parallel, which requires increased memory transfer for approximate computations. Notably, the baseline implementation is memory-bound under the 10GB/s bandwidth constraint.

The design points using different approximation methods are distributed across the entire roofline spectrum. For both workloads, the stack and single approximation methods exhibit the lowest CTC ratios and attainable performances. In these methods, each retrieved decomposition is utilised to

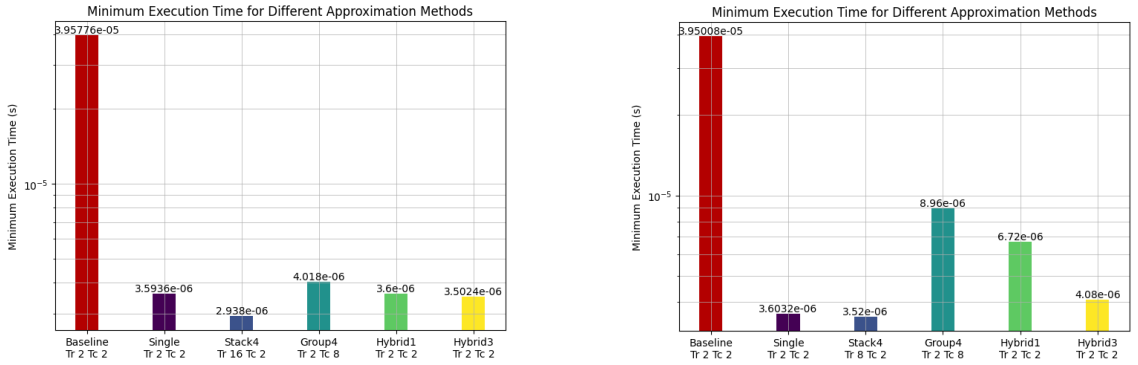
approximate one MVM, resulting in low data reuse and fewer operations performed on the data retrieved from off-chip memory. Conversely, the hybrid and group approximation methods achieve better CTC ratios and higher attainable performance. Each retrieved decomposition is shared by the computation of several MVMs, leading to a higher number of operations performed per memory access, which indicates greater on-chip data reuse. However, this higher attainable performance does not necessarily translate to reduced execution time.



(a) FashionLSTM (b) HarLSTM

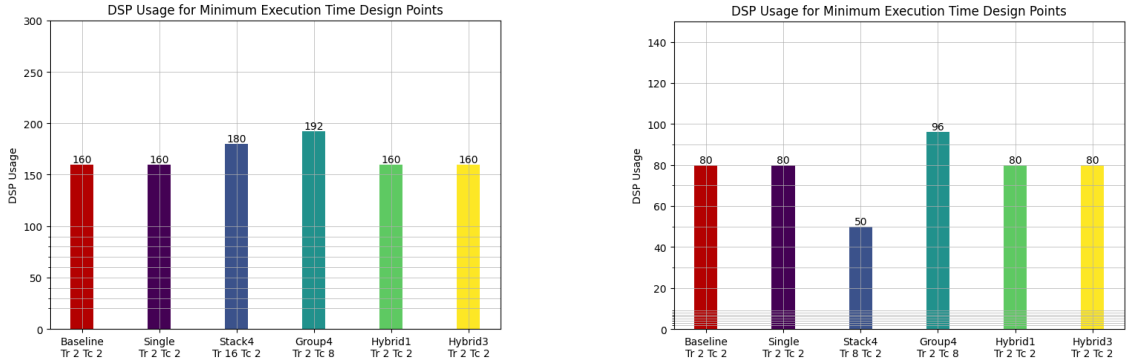
Figure 6.7: Roofline Model for Different LSTM Workloads

Design space exploration is performed on selected design points. Under the given accuracy, bandwidth, and hardware resource constraints, the optimal design points for each approximation strategy are presented in Figure 6.9. For both workloads, stack approximation is the best performing strategy, resulting in 13.5× and 11.2× speed up for the two workloads compared to baseline implementation. It is important to note that the optimal design is application-specific. Different accuracy targets, dimensions of matrices, workload characteristics, and hardware constraints can significantly reshape the roofline model landscape and, in turn, alter the approximation strategy and parameters of the optimal design point.



(a) FashionLSTM (b) HarLSTM

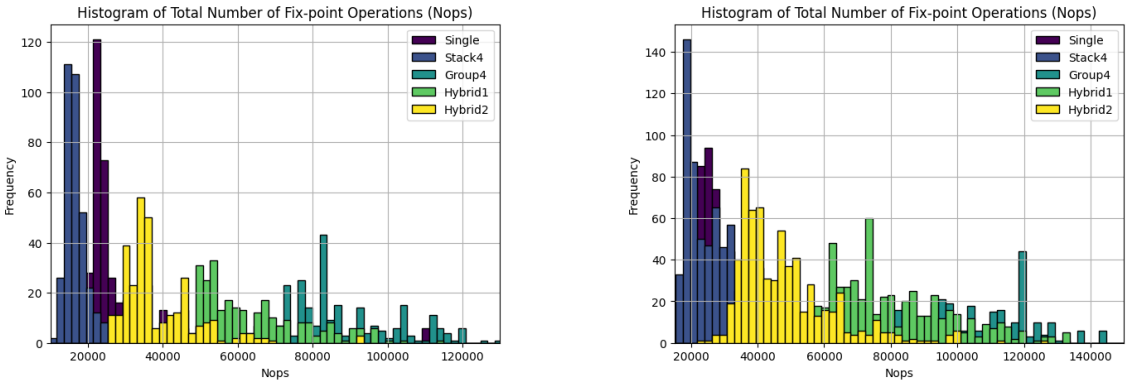
Figure 6.8: Execution Time of Optimal Design Points for Different Approximation Methods



(a) FashionLSTM (b) HarLSTM

Figure 6.9: DSP Usage of Optimal Design Points for Different Approximation Methods

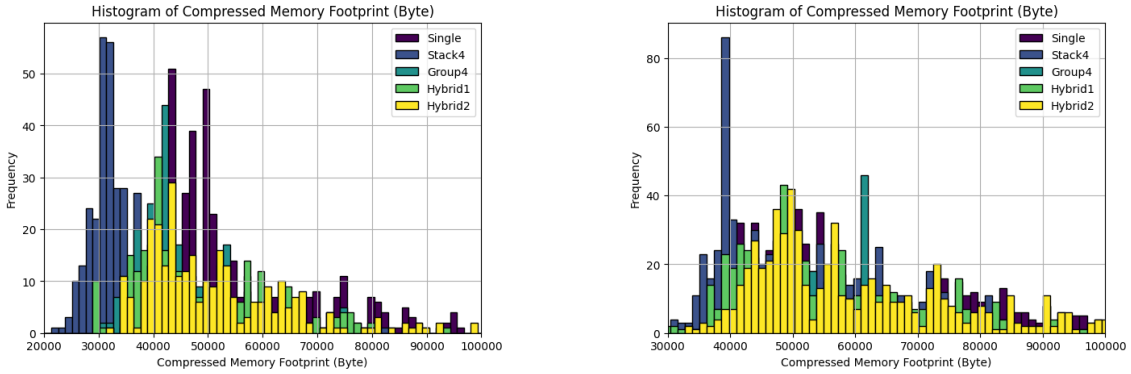
In this experimental setup, stack approximation consistently yields better performance. Despite having low data reuse and performing more operations per memory access, stack approximation achieves the required MSE target in fewer refinement steps, resulting in fewer net operations. This is supported by the histogram of the number of fix-point operations, as shown in Figure 6.10. The significant reduction in number of operations outweighs the fact that stack approximation has low attainable performance, making it the optimal design point for the given experimental setup.



(a) FashionLSTM (b) HarLSTM

Figure 6.10: Number of Fix-Point Operations of Selected Design Points

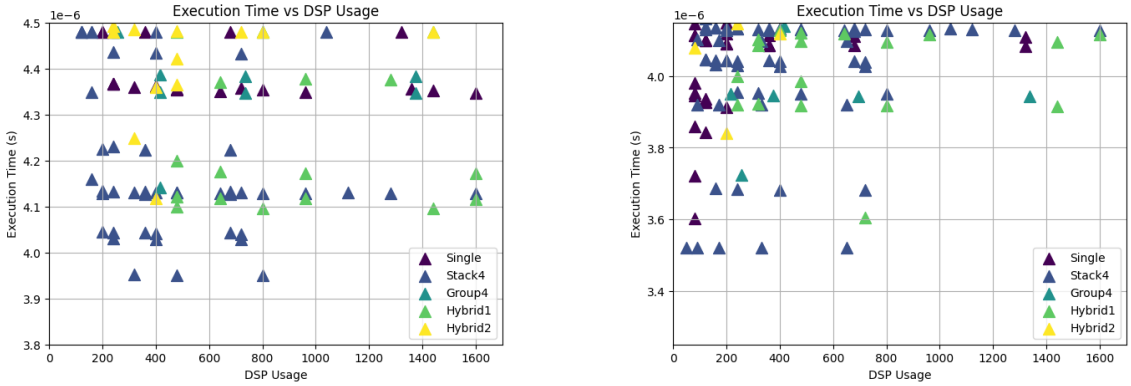
Furthermore, histograms of memory footprint of selected design points are plotted for two LSTM workloads. As shown in Figure 6.12, stack approximation also results in more efficient memory footprint when stored in off-chip memory.



(a) FashionLSTM (b) HarLSTM

Figure 6.11: Memory Footprint of Selected Design Points

As illustrated in the zoomed execution time vs. DSP usage plot on the right, it is notable that the optimal design points for all approximation methods are characterised by low DSP usage. The accelerator achieves low latency for the computation despite exhibiting low parallelism. Additionally, increasing DSP usage does not reduce the latency of computation. This observation may initially appear counter-intuitive. To delve deeper into this phenomenon, a lower target frequency of 150 MHz was tested.



(a) 150MHz (b) 200MHz

Figure 6.12: Execution Time against DSP Usage for HarLSTM (Zoomed)

As demonstrated in the figure on the left, for a target frequency of 150 MHz, increasing DSP usage initially reduces latency, but this improvement quickly saturates, resulting in the execution time vs. DSP usage becoming a straight line. For a target frequency of 200 MHz, the design point with the lowest DSP usage already hits the memory bound. Design points with higher DSP usage are more affected by memory bandwidth constraints, as they request larger tiles of decomposition for approximate MVM computation during each refinement step. Additionally, as discussed in Section 3.5.2, increasing tile size increases the granularity of sparsity, making the decomposition suboptimal in the least square sense. The rate of convergence of the approximation deteriorates with larger tile sizes, requiring more refinement steps to reach the specified MSE target. This increase in the total number of operations outweighs the gains in computational performance. These analyses explain why the best-performing design points are those with low parallelism.

Figure 6.13 presents a set of stack matrix approximation design points without pruning. In this scenario, an increase in DSP usage results in higher performance and lower execution time.

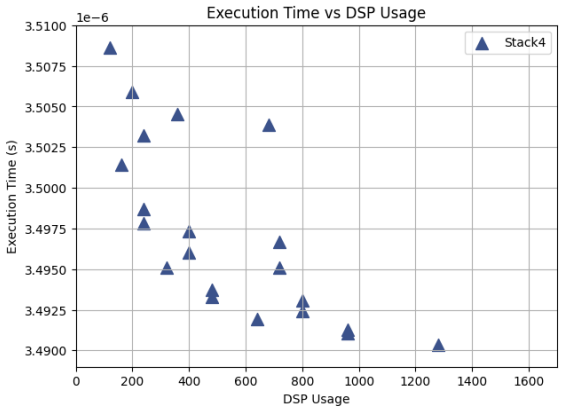


Figure 6.13: Stack Matrix Approximation: Execution Time against DSP Usage without Pruning

6.3.2 Advantage of Different Approximation Methods

In the previous experimental setup, the stack approximation strategy consistently outperformed other approximation methods. However, to demonstrate that there are scenarios where other methods may exhibit better performance, several matrices were constructed for evaluation. Given the lack of an actual workload, the performance is evaluated in terms of execution time required to reach a specified MSE threshold. All other aspects of the experimental setup remains unchanged.

Advantage of Single Matrix Approximation

A set of four matrices is constructed using LSTM input gate matrices, each trained on different subsets of the UCF50 dataset with various human activities. This setup effectively minimises the correlation within this set of matrices, thereby reducing the performance of stack and group matrix approximations, as these methods rely on exploring inter-matrix correlations. A set of design points is selected, and their MSE is plotted against the execution time estimated by the roofline model. As shown in Figure 6.15, the Pareto front for single matrix approximation is superior.

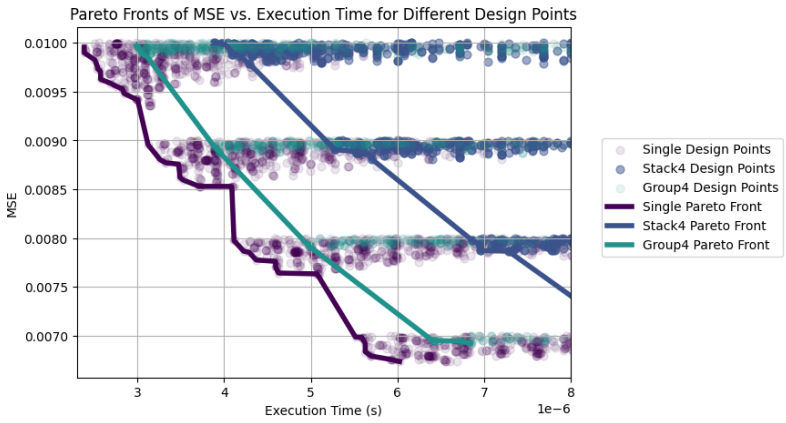


Figure 6.14: Pareto Fronts of MSE vs. Execution Time for Different Design Points

The figure illustrates that for this set of matrices, the single matrix approximation consistently achieves a lower MSE for a given execution time compared to stack and group matrix approximations. This results in a better trade-off between execution time and MSE. In contrast, the stack and group matrix approximations struggle to achieve similar performance levels due to the reduced inter-matrix correlation.

Therefore, for applications involving sets of matrices with low inter-correlation, single matrix approximation proves to be a more effective strategy.

Note that hybrid approximation results in the grouping of matrices into sets of 1 across all refinement steps, which is equivalent to the single matrix approximation method. Therefore, their results are not presented.

Advantage of Group Matrix Approximation

Another set of four matrices is constructed. This set is composed of scalar multiples of an LSTM input gate matrix with low-level noise added. These matrices exhibit a high degree of structural similarity. A set of design points is selected, and their MSE is plotted against the execution time estimated by the roofline model. As shown in Figure 6.15, the Pareto front for group matrix approximation is superior.

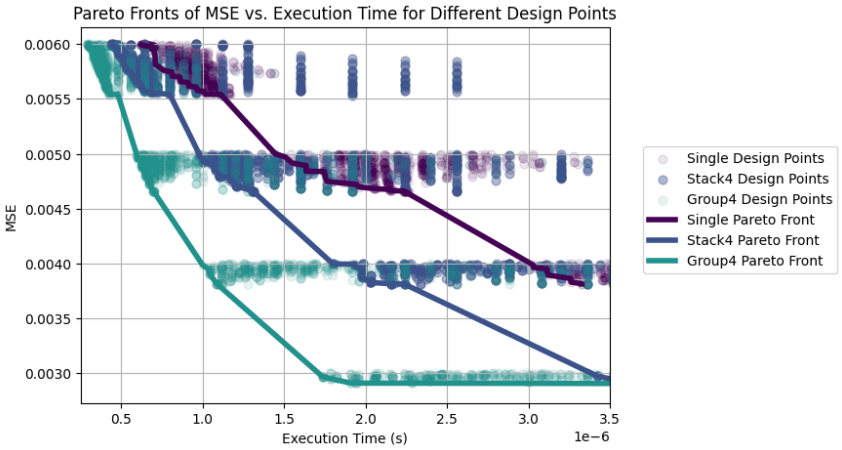


Figure 6.15: Pareto Fronts of MSE vs. Execution Time

The figure illustrates that for this set of matrices, the group matrix approximation consistently achieves a lower MSE for a given execution time compared to single and stack matrix approximations. This result is expected because the group matrix approach leverages the structural similarity among matrices by relying on linear combinations of a shared set of decompositions. This method effectively exploits the high inter-matrix correlation, leading to significant computational efficiency and better performance. The single and stack matrix approximations, on the other hand, do not capitalise on the structural similarity to the same extent. While the single matrix approximation treats each matrix independently, the stack matrix approximation, although it attempts to explore inter-matrix correlations, does not achieve the same level of performance improvement as the group matrix approximation.

Therefore, for applications involving sets of matrices with high inter-correlation and structural similarity, group matrix approximation proves to be the most effective strategy.

Note that hybrid approximation results in the grouping of matrices into sets of 4 across all refinement steps, which is equivalent to the group matrix approximation method. Therefore, their results are not presented.

Advantages of Hybrid Matrix Approximation

In the given experimental setup, and among all the matrix-vector multiplications tested, the hybrid approximation method did not emerge as the best-performing strategy. Despite these initial findings, hybrid matrix approximation holds significant potential. The hybrid method combines elements of both single and group approximations, aiming to dynamically harness the strengths of each while mitigating their respective weaknesses. This flexibility may achieve better trade-offs between accuracy and computational latency under certain conditions, making hybrid approximation a versatile tool in the approximation toolkit. Further research is necessary to identify the specific conditions under which hybrid approximation methods excel.

6.3.3 Roofline Model Validation

A subset of design points that satisfy the bandwidth constraints are selected for FPGA implementation to validate the performance model. The accelerator was simulated using Vivado 2019.1, and the simulated cycles are divided by the operating frequency of the FPGA to calculate the execution time. As illustrated in Figure 6.16, the simulated execution times of the selected design points are plotted against the execution times estimated by the roofline model.

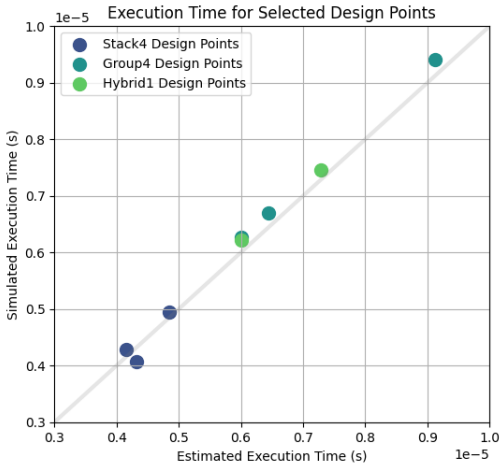


Figure 6.16: Comparison of Simulated and Estimated Execution Times

A high correlation between the estimated and simulated execution times can be observed. The slight deviations are primarily attributed to the pipeline warm-up and cool-down phases which are not captured by the model. Therefore, it is reliable to use the roofline model to predict the best-performing design points.

6.4 Resource Model

To validate the hardware resource model proposed in Section 5.3.1, another subset of design points with varying levels of parallelism were selected for FPGA implementation.

6.4.1 DSP Model Validation

The DSP usage estimated by the hardware resource model aligns closely with the synthesis reports. As demonstrated in Figure 6.17, there is a one-to-one correspondence between the DSP usage reported by the synthesis tool and the DSP usage estimated by the model.

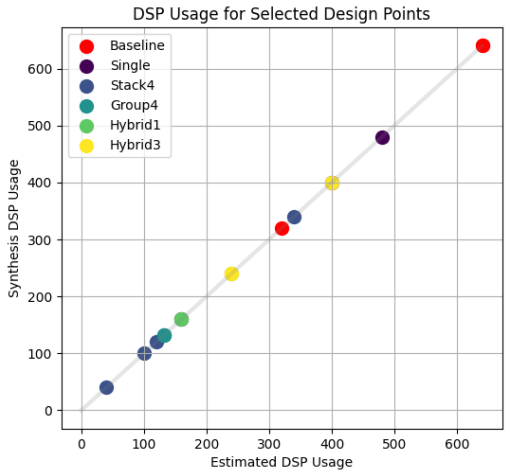


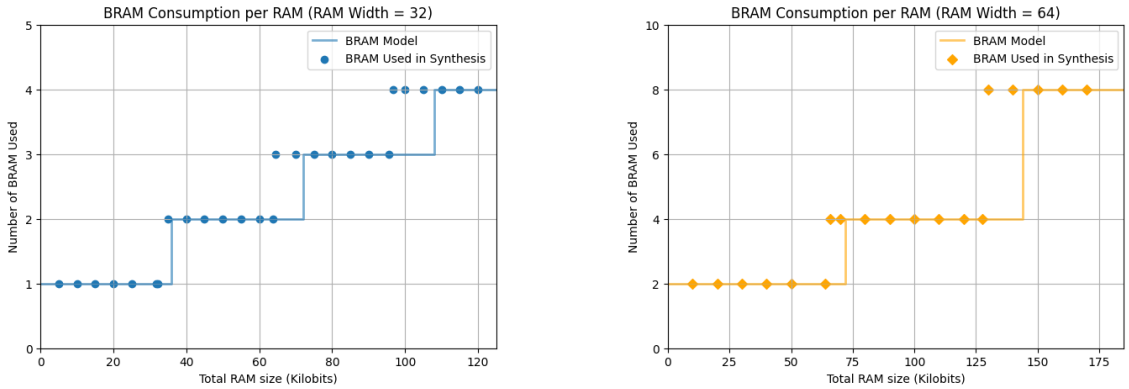
Figure 6.17: Comparison of Synthesised DSP Usage and Model-Estimated DSP Usage

6.4.2 BRAM Model Validation

The BRAM model is first validated by synthesising a dual-port RAM with various sizes and data widths. The estimated BRAM utilisation from the model is compared against the actual utilisation obtained from synthesis reports. As shown in Figure 6.18, the BRAM model demonstrates high reliability for relatively small RAM sizes, with notable deviations for larger RAM sizes. Additionally, deviations are evident at the boundaries where the RAM size approaches the capacity limits of a single BRAM block.

These discrepancies are primarily due to the additional routing and control logic required during synthesis, which can result in more BRAMs being utilised than predicted by the model. For example, when the RAM size slightly exceeds the capacity of a single BRAM block, the synthesis

tool may allocate additional BRAMs to accommodate the overflow, leading to a step increase in BRAM utilisation. This effect is more pronounced for larger RAM sizes and near the boundary conditions, where the synthesis tool introduces optimisation to maintain timing closure and minimise routing congestion.



(a) RAM Data Width 32 (b) RAM Data Width 64

Figure 6.18: BRAM Consumption per RAM with different data widths

As shown in Figure 6.19, the estimated BRAM usage closely matches the synthesised BRAM usage. No discrepancies are observed because the LSTM model used are of moderate size. The size of input vector buffer and accumulation RAM is small and not close to the capacity limits of a single BRAM block.

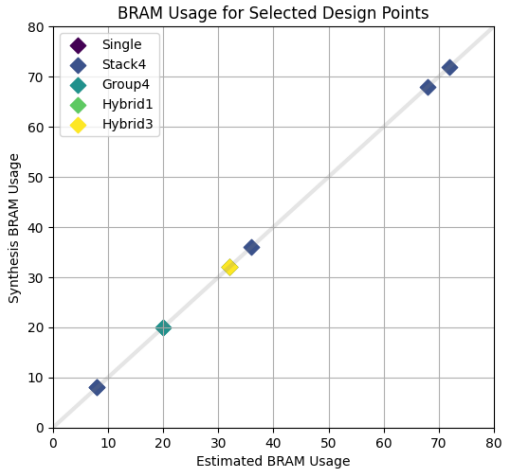


Figure 6.19: BRAM Usage of Selected Design Points

Overall, while the BRAM model provides a good approximation for small to moderate RAM sizes, careful consideration is needed for larger sizes and boundary conditions to account for the additional BRAM usage induced by synthesis-related factors.

6.5 Comparison with State-of-the-Art Method

A hardware implementation following the design methodology described in [12] is considered state-of-the-art. Due to the lack of data from their implementation, we perform a qualitative comparison with our work. Their method, referred to as merging, approximates weight matrices of the same type (input and hidden state) and gate matrices of two LSTM cells together using a group approximation method. In contrast, our approach augments the input and hidden state matrices and concatenates the input vector and hidden state, as we empirically found these to have similar structures. Additionally, our approach considers stacking the matrices of dependent matrix-vector multiplications to leverage the correlation among matrices. Our work also explores other approximation methods, such as single and hybrid approximation, and designs corresponding SVD kernels for hardware acceleration. Furthermore, we proposed optimised architecture for dependent MVMs with improved hardware utilisation. These enhancements potentially offer more flexibility and better performance across a diverse set of real-world applications compared to the state-of-the-art implementation.

7

Conclusions and Future Work

7.1 Conclusion

This project presented a comprehensive framework for approximate matrix-vector multiplication on FPGA. By altering the structure of the computations, the proposed method allows for the co-optimisation of both the approximate computations and the underlying hardware accelerator, while taking into account the resource constraints of the targeted device. The framework incorporates a library of approximation methods, each tailored to different levels of inter-matrix correlation: single matrix approximation excels in cases of low inter-matrix correlation, stack matrix approximation is effective for medium inter-matrix correlation, and group matrix approximation performs best in cases of high inter-matrix correlation. Additionally, the project introduced a novel hybrid approximation method. The framework is evaluated using two Long Short Term Memory network. Under the same hardware resource and bandwidth constraints, the proposed framework achieves up to a 13.5× speedup compared to the baseline implementation. With its versatile library of approximation techniques, the framework is able to achieve a better accuracy-latency trade-off and is applicable to a wide range of machine learning and scientific computing workloads.

7.2 Further Work

The hybrid matrix approximation approach has demonstrated significant potential due to its flexibility. Future work can explore scenarios where this hybrid approach achieves a better accuracy-latency trade-off.

One area for improvement is hardware utilisation. Currently, when the non-zero tiles in the row and column dimensions do not match, either the v kernel or the u kernel remains idle for several cycles. To address this inefficiency, further work can focus on enforcing the reuse of these kernels, thereby boosting overall hardware utilisation.

In hybrid approximation, matrices can be dynamically grouped across refinement steps. Similarly, experimentation can be conducted by stacking different numbers of matrices at each refinement step. This may necessitate the development of a new SVD kernel tailored to this approach. This kernel would need to accommodate the varied structure and dimensionality of the matrices involved.

All current accelerator implementations employ a homogeneous configuration. Future research could investigate the feasibility of supporting different types of SVD kernels with varying levels of parallelism. This improved versatility can efficiently handle MVM in a wide range of real-world applications. Such advancements would require a redesign of the current approximation method to consider matrices of different sizes, along with novel scheduling and load-balancing techniques.

Additionally, the present work only demonstrates its applicability on LSTM networks. Future investigations could extend to evaluate the framework's performance with Transformer models, thereby broadening the applicability and robustness of the proposed approach.

Bibliography

[1] M. Rizakis, S. I. Venieris, A. Kouris, and C.-S. Bouganis, *Approximate fpga-based lstms under computation time constraints*, 2018. arXiv: 1801.02190 [cs.CV].

[2] J. Fowers, K. Ovtcharov, M. Papamichael, *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14. DOI: 10.1109/ISCA.2018.00012.

[3] E. Nurvitadhi, D. Kwon, A. Jafari, *et al.*, "Why compete when you can work together: Fpga-asic integration for persistent rnns," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 199–207. DOI: 10.1109/FCCM.2019.00035.

[4] Z. Que, Y. Zhu, H. Fan, J. Meng, X. Niu, and W. Luk, "Mapping large lstms to fpgas with weight reuse," *Journal of Signal Processing Systems*, vol. 92, Sep. 2020. DOI: 10.1007/s11265-020-01549-8.

[5] F. Shehzad, M. Rashid, M. Sinky, S. Alotaibi, and M. Y. I. Zia, "A scalable system-on-chip acceleration for deep neural networks," *IEEE Access*, vol. PP, pp. 1–1, Jul. 2021. DOI: 10.1109/ACCESS.2021.3094675.

[6] S. Chen, J. Zhou, W. Sun, and L. Huang, "Joint matrix decomposition for deep convolutional neural networks compression," *Neurocomput.*, vol. 516, no. C, pp. 11–26, Jan. 2023, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2022.10.021. [Online]. Available: https://doi.org/10.1016/j.neucom.2022.10.021.

[7] S. Cao, C. Zhang, Z. Yao, *et al.*, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 63–72, ISBN: 9781450361378. DOI: 10.1145/3289602.3293898. [Online]. Available: https://doi.org/10.1145/3289602.3293898.

[8] M. Tukan, A. Maalouf, M. Weksler, and D. Feldman, *Compressed deep networks: Goodbye svd, hello robust low-rank approximation*, 2020. arXiv: 2009.05647 [cs.LG].

[9] S. Han, J. Kang, H. Mao, *et al.*, *Ese: Efficient speech recognition engine with sparse lstm on fpga*, 2017. arXiv: 1612.00694 [cs.CL].

[10] Z. Wang, J. Lin, and Z. Wang, "Accelerating recurrent neural networks: A memory-efficient approach," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2763–2775, 2017. DOI: 10.1109/TVLSI.2017.2717950.

[11] A. Shashua and A. Levin, "Linear image coding for regression and classification using the tensor-rank principle," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1, 2001, pp. I–I. DOI: 10.1109/CVPR.2001.990454.

[12] S. Ribes, P. Trancoso, I. Sourdis, and C.-S. Bouganis, "Mapping multiple lstm models on fpgas," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 1–9. DOI: 10.1109/ICFPT51103.2020.00010.

[13] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. [Online]. Available: https://doi.org/10.1145/1498765.1498785.

[14] Z. Que, H. Nakahara, E. Nurvitadhi, *et al.*, "Optimizing reconfigurable recurrent neural networks," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 10–18. DOI: 10.1109/FCCM48280.2020.00011.