Universidade Estadual de Campinas
Instituto de Computação

Rafael Cardoso Fernandes Sousa

Convolution Tensor Slicing Optimization for Multicore NPUs

Otimização do Particionamento de Tensores para Convolução em NPUs Multicore

CAMPINAS
2023

Rafael Cardoso Fernandes Sousa

Convolution Tensor Slicing Optimization for Multicore NPUs

Otimização do Particionamento de Tensores para Convolução em NPUs Multicore

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo
Co-supervisor/Coorientador: Dr. Marcio Machado Pereira

Este exemplar corresponde à versão final da Tese defendida por Rafael Cardoso Fernandes Sousa e orientada pelo Prof. Dr. Guido Costa Souza de Araújo.

CAMPINAS
2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Silvania Renata de Jesus Ribeiro - CRB 8/6592

So85c

Sousa, Rafael Cardoso Fernandes, 1988-
Convolution tensor slicing optimization for multicore NPUs / Rafael Cardoso Fernandes Sousa. – Campinas, SP : [s.n.], 2023.

Orientador: Guido Costa Souza de Araújo.
Coorientador: Marcio Machado Pereira.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Compiladores (Computadores). 2. Aprendizado de máquina. 3. Programação paralela (Computação). 4. Redes neurais convolucionais. I. Araújo, Guido Costa Souza de, 1962-. II. Pereira, Marcio Machado, 1959-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações Complementares

Título em outro idioma: Otimização do particionamento de tensores para convolução em NPUs multicore
Palavras-chave em inglês:
Compilers (Eletronic computers)
Machine learning
Parallel programming (Computer science)
Convolutional neural networks
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Márcio Bastos Castro
Alfredo Goldman Vel Lejbman
Hervé Cédric Yviquel
Lucas Francisco Wanner
Data de defesa: 24-11-2023
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-2673-6601
- Currículo Lattes do autor: http://lattes.cnpq.br/1581280350850662

Universidade Estadual de Campinas
Instituto de Computação

Rafael Cardoso Fernandes Sousa

Convolution Tensor Slicing Optimization for Multicore NPUs

Otimização do Particionamento de Tensores para Convolução em NPUs Multicore

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo
  IC/UNICAMP
- Prof. Dr. Márcio Bastos Castro
  INE/UFSC
- Prof. Dr. Alfredo Goldman vel Lejbman
  IME/USP
- Prof. Dr. Hervé Cedric Yviquel
  IC/UNICAMP
- Prof. Dr. Lucas Francisco Wanner
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 24 de novembro de 2023

Acknowledgements

Firstly, I would like to express my deepest gratitude and thanks to my wife, Karina de Oliveira, who supported me throughout my entire Ph.D. process. She was my inspiration and a source of strength; thanks to her, I successfully defended my doctoral thesis. Additionally, a very special thanks to my family, who have also supported me since the beginning.

I would also like to express my sincere gratitude to my advisor, Prof. Guido Araujo, for his constant support, knowledge, and guidance throughout my Ph.D. process. Additionally, I extend my thanks to my co-supervisor, Prof. Marcio Pereira, whom I could consistently rely on for discussions about algorithms and research ideas at various stages.

Furthermore, special thanks to my friends from the Computer Systems Laboratory (LSC), especially to those with whom I had the opportunity to meet and become friends, and to those with whom I had the chance to work and collaborate on their projects. The moments spent in the lab were very important for my Ph.D. process.

I thank the members of my doctoral committee for their significant contributions to my defense, the postgraduate department (sec-pos) for their prompt support, and also the IT support staff of the Institute of Computing (IC).

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and by the Electronics and Telecommunications Research Institute (ETRI/South Korea). Additionally, I would like to thank LG Electronics (San Jose Lab) and SilicoNeuro/AiM Future for their contributions to the development of the project.

Resumo

Embora a geração de código para modelos de Redes Neurais Convolucionais (CNNs) tenha sido extensivamente estudada, a realização de divisão eficiente de dados e paralelização para Unidades de Processamento Neural (NPUs) multicore e com memórias internas restritas ainda é um problema desafiador. Dado o tamanho dos tensores de entrada/saída das convoluções e o pequeno espaço das memórias internas das NPUs, minimizar as transações de memória ao mesmo tempo em que se maximiza o paralelismo e a utilização das unidades de multiplicação e acumulação (MAC) são fundamentais para qualquer solução eficaz. Esta tese propõe um passo de otimização no compilador TensorFlow XLA/LLVM para NPUs multicore, chamado Otimização de Divisão de Tensores (TSO), que: (a) maximiza o paralelismo das convoluções e o reuso de memória entre os núcleos das NPUs, (b) maximiza a reutilização de dados em cada núcleo NPU, selecionando a estratégia de agendamento apropriada que minimiza as transferências de dados entre as memórias no chip e a DRAM, e (c) reduz as transferências de dados entre a memória DRAM do hospedeiro (host) e as memórias internas da NPU ao empregar um modelo baseado em rajadas (bursts) de memória DRAM ao avaliar cada solução no espaço de busca, resultando na seleção daquela que oferece o melhor desempenho. Para avaliar a abordagem proposta, foram realizados experimentos usando o Processador Neuromórfico (NMP), uma NPU multicore contendo 32 núcleos RISC-V estendidos com novas instruções de CNN. Os resultados experimentais mostram que o TSO é capaz de identificar a melhor divisão de tensores que minimiza o tempo de execução para um conjunto de modelos de CNN. Acelerações de até 21,7% são observadas ao comparar a técnica baseada em rajada do TSO com uma abordagem de fatiamento de dados sem rajada. Entre as 236 convoluções avaliadas de 6 modelos CNN, o TSO com o modelo de rajadas supera a técnica sem rajadas em 73% delas, alcançando acelerações de até 3,69x. Grande parte da melhoria provém do fato de que o TSO com a técnica baseada em rajada habilitada otimiza o tempo de carga em até 24% e o tempo de armazenamento em até 79% quando comparado à abordagem sem rajadas na execução completa de todo o modelo CNN. Além disso, o TSO com a técnica baseada em rajada foi também comparado a uma abordagem existente que aplica um modelo de custo baseado em um modelo teórico chamado roofline [57], usado para estimar o desempenho das convoluções. Como resultado, o TSO supera todos os modelos CNN, com acelerações de até 29,2%. Para 84% das convoluções, o TSO alcança acelerações de até 5,19x. O algoritmo também foi adaptado para o framework de Aprendizado de Máquina Glow para validar a generalidade da abordagem TSO. O desempenho dos modelos foi medido tanto nos compiladores Glow quanto TensorFlow XLA/LLVM, revelando resultados similares.

Abstract

Although code generation for Convolution Neural Network (CNN) models has been extensively studied, performing efficient data slicing and parallelization for highly-constrained Multicore Neural Processor Units (NPUs) is still a challenging problem. Given the size of convolutions' input/output tensors and the small footprint of NPU on-chip memories, minimizing memory transactions while maximizing parallelism and MAC utilization are central to any effective solution. This thesis proposes a TensorFlow XLA/LLVM compiler optimization pass for Multicore NPUs, called Tensor Slicing Optimization (TSO), which: (a) maximizes convolution parallelism and memory usage across NPU cores, (b) maximizes data reuse on each NPU core by selecting the appropriate scheduling strategy that minimizes data transfers between the on-chip memories and DRAM, and (c) reduces data transfers between the host DRAM memory and the NPU on-chip memories by employing a DRAM memory burst modeling when evaluating every solution in the search space, which results in the selection of the one that provides the highest performance. To evaluate the proposed approach, experiments were performed using the NeuroMorphic Processor (NMP), a multicore NPU containing 32 RISC-V cores extended with novel CNN instructions. Experimental results show that TSO is capable of identifying the best tensor slicing that minimizes execution time for a set of CNN models. Speedups of up to 21.7% result when comparing the TSO burst-based technique to a no-burst data slicing approach. Among the 236 evaluated convolutions from 6 different CNN models, TSO with the burst-modeling outperforms the no-burst approach in 73% of them, achieving speedups of up to 3.69x. Most of the improvement comes from the fact that TSO with the burst-based technique enabled optimizes the load time by up to 24% and the store time by up to 79% when compared to the no-burst approach in end-to-end execution of a CNN model. Furthermore, TSO with the burst-based technique was compared against an existing approach that applies a cost model based on a theoretical roofline model [57] to estimate the convolution performance, which resulted in TSO outperforming all CNN models with speedups of up to 29.2%. In this comparison, TSO shows improvements in 84% of the convolutions, achieving speedups of up to 5.19x. The TSO algorithm was also ported to the Glow Machine Learning framework to validate the generality of the TSO approach. The performance of the models was measured on both Glow and TensorFlow XLA/LLVM compilers, revealing similar results.

List of Figures

List of Tables

List of Symbols and Acronyms

Tensor terms

`IN`	Input tensor
N	Input/Weight channels
H	Input height
L	Input width
`KS`	Weight tensor
M	Weight/Output channels
K1	Weight height
K2	Weight width
`OUT`	Output tensor
R	Output height
C	Output width

Tile terms

$\texttt{IN}^T$	Input tile
T_N	Input/Weight tile channels
T_H	Input tile height
T_L	Input tile width
$\texttt{KS}^T$	Weight tile
T_M	Weight/Output tile channels
$\texttt{OUT}^T$	Output tile
T_R	Output tile height
T_C	Output tile width

AI terms

AI	Artificial Intelligence

CNN	Convolutional Neural Network
Conv-Layer	Convolutional Layer
FC-Layer	Fully Connected Layer
ML	Machine Learning
ReLU	Rectified Linear Unit

Architectural terms

NPU	Neural Processing Unit
CPU	Central Processing Unit
GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Array
NMP	NeuroMorphic Processor
TLE	NMP Tile
TLT	NMP Tilelet
DME	NMP Data Movement Engine
MB0	NMP MBLOB for input tile
MB1	NMP MBLOB for weight tile
MB2	NMP MBLOB for output tile
MAC	Multiplier Accumulator
DRAM	Dynamic Random-Access Memory
CAS	Column Address Strobe
DDR	Double Data Rate

Compiler terms

PM	Pass Manager
AOT	Ahead-Of-Time compilation
JIT	Just-In-Time compilation
LLVM	Low Level Virtual Machine
LLVM IR	LLVM's Intermediate Representation
HLO	High Level Operations
ONNX	Open Neural Network Exchange

HIR	High-level Intermediate Representation
LIR	Low-level Intermediate Representation
MLIR	Multi-Level Intermediate Representation
DCE	Dead Code Elimination
CSE	Common Subexpression Elimination
TSO terms	
TSO	Tensor Slicing Optimization
IS	Input Stationary
OS	Output Stationary
WS	Weight Stationary
TLE KS	TLE slicing using KS scheme
TLE KS&OUT	TLE slicing using KS&OUT scheme
TLE OUT	TLE slicing using OUT scheme
TLE_R	Output rows in a TLE slice
TLE_W	Filters in a TLE slice
α_{in}	Number of reloads of the input tiles
α_{ks}	Number of reloads of the weight tiles
α_{out}	Number of stores of the output tiles

Contents

Chapter 1

Introduction

Deep Learning using Convolutional Neural Network (CNN) has become a significant Machine Learning (ML) architecture model that considerably increases the accuracy of many modern AI applications. The steady increase in the adoption of CNNs is driven mostly by applications in the Computer Vision domain, where it addresses problems like Object Recognition [29, 75, 90], Object Detection [22, 72], and Video Classification [39, 73]. Other areas, like Speech Recognition and Natural Language Processing (NLP), have also benefited from the application of CNN models [5, 43].

The size and complexity of state-of-the-art CNNs have grown significantly, followed by its accuracy improvements. For instance, LeNet-5 [47], a model that recognizes handwritten digits, has less than 1 million parameters, while more complex models, like InceptionV3 [78], which classifies thousands of different object categories, has more than 23 million parameters. Such an increase in the model complexity and parameters size not only demands more computational power but also produces a significant increase in the data movement between host (off-chip) and the AI accelerator (on-chip) memories, thus considerably impacting energy-consumption and memory traffic [80].

It is well-known that convolution is the most expensive operation of a CNN [53,55,83], accounting for the largest share of a CNN execution. Given the size of its tensor inputs and the wide variety of configuration parameters (e.g., kernel size, stride, etc.), selecting the best data mapping which maximizes convolution parallelism while minimizing memory transactions is a key factor in the performance of any AI accelerator. This is particularly critical for multicore Neural Processing Units (NPUs), which have stringent (on-chip) memory constraints and need to achieve large inference throughput.

Convolution input tensors and weights must be partitioned into small tiles that fit into the NPU on-chip memories. This process is called *tiling*, and once the tiles are defined, these tiles are brought from (slow) external DRAM to (fast) on-chip memories, one pair of input and weight tiles at a time, to compute an output tile of the convolution operation. Depending on how the tile shapes and sizes are selected and the order they are brought to the on-chip memories, the convolution execution time can change drastically.

The tiling process is complex, involving a search space where millions to billions of solutions can be explored. The work presented in this thesis focuses on determining how to efficiently select a solution from this search space. In this sense, three research questions arise: (Q_1) What is the impact of tiling on DRAM burst usage; (Q_2) What is

the impact of tiling on scheduling; and (Q_3) What is the impact of tiling on parallelism. Regarding the first research question (Q_1), this thesis explores techniques to efficiently select tile shapes that maximize memory burst usage. Bringing more data from a single memory access can considerably reduce data transfer time to DRAM. This concept is widely explored in GPUs [20, 85] and can also be extended to work on multicore NPUs. The next research question (Q_2) addresses how to explore different scheduling strategies to maximize data reuse in the on-chip memories and reduce reloads in DRAM. Various scheduling strategies have been explored, especially for FPGAs and CPUs [52, 57, 61]. However, the tile sizes explored by these solutions are not always ideal, as fixed tile sizes are used, and maximizing them up to the capacity of the on-chip memories may lead to better performance. Finally, the last research question (Q_3) explores how to enhance data balance among multiple cores in an NPU to reduce the required memory footprint for each core. Many efforts have been explored in this regard (e.g., [13, 89]), but some may not effectively choose a proper solution due to the use of a unique slicing scheme (e.g., only slicing the filters among the cores), which may consequently lead to more data reload.

This thesis focuses on answering questions Q_1 – Q_3 above and uses these answers to design an LLVM-based optimization for NPU architectures called *Tensor Slicing Optimization* (TSO) that can achieve the following goals:

- To minimize data transfer between the host and NPU's on-chip memory by properly answering Q_1. In Section 6.5 we explain how a specialized tiling selection algorithm is used to leverage memory burst access. The goal is to choose tile sizes and shapes that maximize memory burst reuse;

- To maximize the reuse of on-chip memory tiles by properly answering Q_2. In Section 6.4 we explain how a tiling scheduling algorithm is utilized to reuse as much as possible on-chips memory tiles to reduce the number of times a tile needs to be reloaded/stored from/to DRAM;

- To maximize the parallelization of convolution computations by properly answering Q_3. In Sections 6.2 – 6.3 we explain an approach to evenly distribute the convolution data among the cores of an NPU, ensuring load balancing;

- By applying Q_1 in TSO, a speedup of up to 21.7% is achieved. Regarding Q_2 in TSO, a speedup of up to 39.8% is observed. Finally, concerning Q_3, TSO achieves a speedup of up to 41.0%. Additionally, when compared to an existent solution (roofline-based cost model [57]), TSO shows a speedup of up to 29.2%.

TSO should be capable of modeling, at compile-time, the memory utilization of the various NPU cores in the search for the best input/output tensor slicing that minimizes data transfers between the host and the NPU cores' memories. To evaluate this approach, experiments were performed using the LGE NeuroMorphic Processor (NMP), a multicore NPU containing 32 cores, and the TensorFlow XLA LLVM compiling toolchain.

The work in this thesis resulted in the following scientific publications. The first two publications are focused on NPU architectures and are the contributions of this thesis's

author. The other publications target CPU architectures and are joint work with other authors that reused a variation of the *Tensor Slicing Optimization (TSO)* proposed herein (Section 6.1).

- Sousa, R., Jung, B., Kwak, J., Frank, M. and Araujo, G. (2021, October). Efficient tensor slicing for multicore NPUs using memory burst modeling. In 2021 IEEE 33rd International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD) (pp. 84-93).

- Sousa, R., Pereira, M., Kwon, Y., Kim, T., Jung, N., Kim, C., Frank, M. and Araujo, G. (2023, January). Tensor Slicing and Optimization for Multicore NPUs. Journal of Parallel and Distributed Computing (JPDC), 175, 66-79.

- Ferrari, V., Sousa, R., Pereira, M., de Carvalho, J. P., Amaral, J. N. and Araujo, G. (2022, October). Improving Convolution via Cache Hierarchy Tiling and Reduced Packing. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 538-539.

- Ferrari, V., Sousa, R., Pereira, M., de Carvalho, Moreira, J., J. P., Amaral, J. N. and Araujo, G. (2023, September). Advancing Direct Convolution using Convolution Slicing Optimization and ISA Extensions. ACM Transactions on Architecture and Code Optimization (TACO), 2023.

The TSO algorithm is well-suited for devices with small on-chip memories, as seen in the case of NMP, where careful management of memory access to DRAM is essential for achieving high performance. A variant of TSO can be applied for CPUs, as demonstrated in SConv [21]. However, for GPUs, where large global memory is available, TSO may not be as efficient as in scenarios like NMP. In the case of FPGAs, there is potential for improvement by implementing TSO, although we have not explored its application on those targets.

This thesis is organized as follows. In Chapter 2, the research questions explored in this thesis are detailed. In Chapter 3, a concise review of Convolutional Neural Networks (CNNs) and their main building blocks is provided. Additionally, an in-depth analysis of convolutions is presented, with a focus on tiling techniques for efficient computation. Moving on to Chapter 4, an overview of the NMP architecture is introduced, which serves as the target device for this thesis. Chapter 5 discusses the integration of the TSO into ML compilers, along with other necessary optimizations for adapting computation to the NMP architecture. The subsequent chapter, Chapter 6, introduces the TSO algorithm, the main contribution of this thesis. This includes slicing and partitioning techniques, and the application of a cost model based on memory burst accesses to determine the most suitable solution. In Chapter 7, the results of comparing the TSO algorithm with two data-volume-based solutions are presented, and a thorough analysis of the three research questions ($Q_1 - Q_3$) is provided. A more in-depth analysis to support the TSO effectiveness is presented in Chapter 8. Moving forward to Chapter 9, the main distinctions between TSO and previous research are highlighted. Finally, concluding the thesis in Chapter 10, findings are summarized, and potential future research topics are proposed.

Chapter 2

Thesis' Research Questions

This chapter delves into the research questions initially introduced in Section 1, thoroughly examining the advantages associated with the implementation of each one of them in the context of executing convolutions on multicore NPUs. For a deeper clarification of the research questions, refer to the three sections below, where a detailed discussion on the impact of tiling on DRAM burst usage, scheduling, and parallelism on the convolution execution time is presented. This thesis focuses on using the answers to these questions to design an effective compiler-based convolution-slicing optimization algorithm.

2.1 Q_1: What is the Impact of Tiling on DRAM Burst Usage

Consider Figure 2.1, which shows the time a Convolution takes when using tiles of different shapes. In that example, the input tensor is a single channel with 128×128 16-bit fixed-point elements (row-major) computed over a single kernel of size 1×1. In the figure, tiles are represented as light/dark gray areas, and each red dot represents a (128B) *memory burst* access to the DRAM. Accessing time in a DRAM can be divided into two components: (a) CAS (Column Address Strobe) latency, which is the time taken to read the first byte of a memory burst from the DRAM Row Buffer; and (b) Access latency, which is the time taken to read the following bytes of the burst. For example, reading the first byte from a 128B burst of a typical DDR3 memory takes $\sim$ 14ns, the same time it takes to read all the remaining 127 bytes of that burst. Depending on how data is tiled, memory bursts can enormously impact execution time. For example, in Figure 2.1 the Convolution can be divided into: (a) 8 $128 \times 32B$ tiles resulting in 1024 bursts (red dots) and an execution time of 84us; (b) 4 $128 \times 64B$ tiles corresponding to 512 bursts and a reduced 58us execution time; and (c) 4 $64 \times 128B$ tiles which require 256 bursts and 46us execution time, a 45% reduction in the convolution time when comparing to the tiling in (a). As shown in the graph of Figure 2.1, tiling (c) (4 $64 \times 128B$ tiles) has the shortest memory access time at w = 128B. From that point on, as the width (w) of the tile continues to increase, memory access time worsens and then improves again at the next memory burst alignment (w = 256). In the area of code generation, the problem of ordering memory accesses so as to maximize burst data usage is known as *memory access coalescing* [31, 34, 65, 68]. Although memory access coalescing is a common problem in GPU code generation [20, 85], it has not been extensively explored in multicore NPUs.

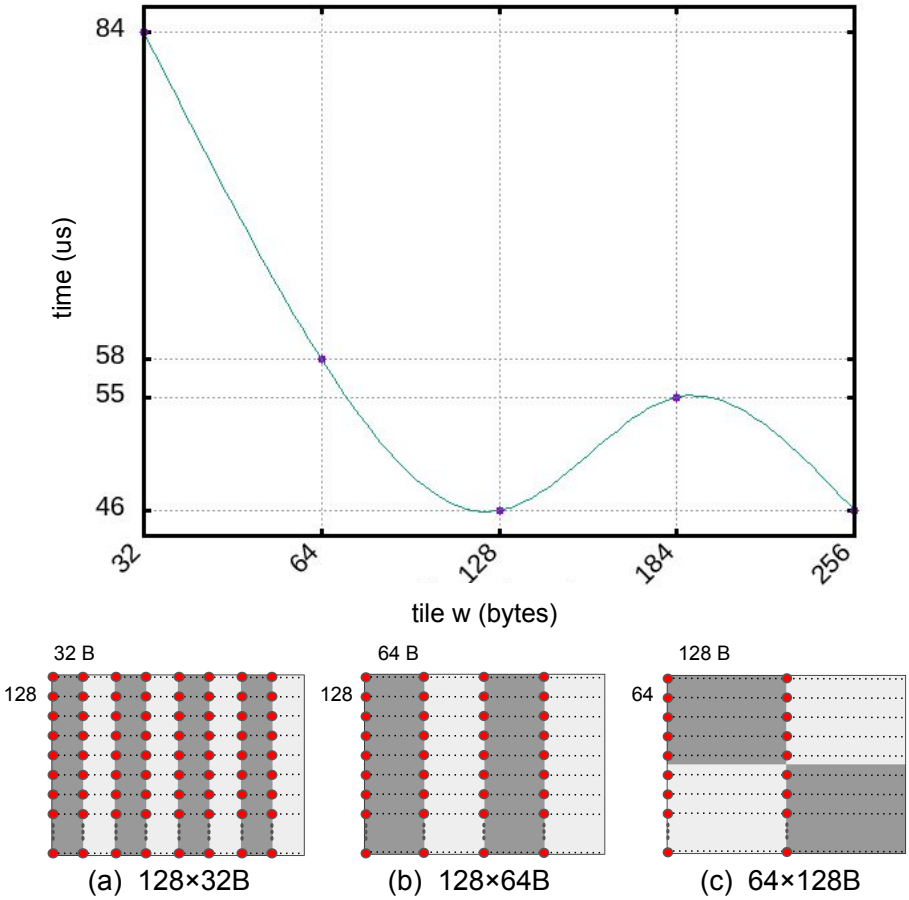

Figure 2.1: Memory access with different tile shapes.

2.2 Q_2: What is the Impact of Tiling on Scheduling

The tile size selection and the scheduling order in which those tiles are executed at runtime are critical factors in optimizing the convolution performance on multicore NPUs. Data movements can be significantly reduced by maximizing the sizes of the input, weight, and output tiles to match (or approximate) the maximum capacity of the NPU's on-chip memories. For instance, consider the execution flow presented in Figure 2.2, where, for simplicity, only a single NPU core is used. Initially, the first input tile ($\texttt{IN}^T$ #1) is loaded from DRAM into the NPU's on-chip memory ① and remains stationary while being computed with a set of weight tiles, one at a time, to generate an output tile. Once the computation of the first weight tile ($\texttt{KS}^T$ #1) ② is finished, an output tile ($\texttt{OUT}^T$ #1) is produced and then stored in the host memory (DRAM) ③. After that, another weight tile ($\texttt{KS}^T$ #2), which needs to be loaded from the DRAM ④, is placed in its corresponding NPU on-chip memory and then computed with the stationary input tile to generate a new output tile ($\texttt{OUT}^T$ #2), which is subsequently stored in the DRAM ⑤. After finishing the computation of the stationary input tile with all weight tiles, another input tile ($\texttt{IN}^T$ #2) ⑥ is loaded and kept stationary, and the same set of weight tiles is loaded again from DRAM, one by one, to generate new output tiles. To minimize the need for multiple reloads, maximizing the sizes of the tiles, particularly the one that remains stationary, is essential. In the context of the execution flow illustrated in Figure 2.2, a reduction

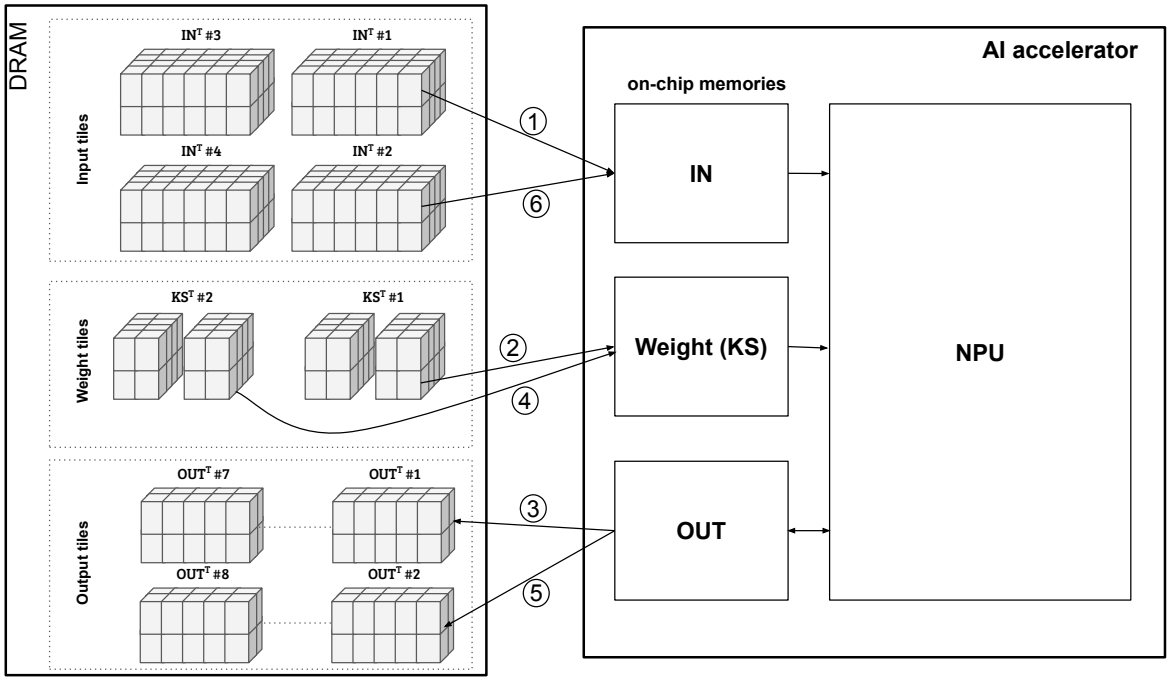

Figure 2.2: Scheduling of multiple tiles with the input tile kept stationary. In this example, 4 input tiles are computed with 2 weight tiles in order to generate 8 output tiles.

in the number of input tiles (e.g., from 4 to 2) would result in a proportional decrease in the number of reloads required for the weight tiles from the DRAM. However, it is important to consider that increasing the size of one tile may impact the sizes of the others, as these are interdependent. Hence, given the range of possible solutions that can be selected from a search space, the task of choosing the one that minimizes data transfer while maintaining high utilization of the NPU's cores becomes critical and challenging. This task is essential to produce optimized code.

2.3 Q_3: What is the Impact of Tiling on Parallelism

To accelerate the convolution computation on a multicore NPU, the partitioning of the convolution data (input, weight, and output tensors) must consider workload balancing for efficient mapping of the operation. Hence, it is essential to distribute the data evenly among the NPU cores to ensure a balanced data workload across all of them. Furthermore, another crucial aspect to consider involves minimizing data transfers between the DRAM and the NPU on-chip memories during this step, which can be achieved by prioritizing the partitioning of the largest tensor (in terms of size). To show how this can be achieved, consider the example illustrated in Figure 2.3, where the input tensor represents an RGB image with 3 channels of 224×224 16-bit fixed-point elements. This input tensor undergoes computation with a weight tensor consisting of 16 filters, each with a size of $3 \times 3 \times 3$ ($C \times H \times W$). This computation results in an output tensor comprising 16 channels, all of the size 224×224. Assume that all tensors have the same data type and that the input tensor is zero-padded in order to generate an output tensor with matching spatial dimensions. In this example, the partitioning of the convolution data distributes a

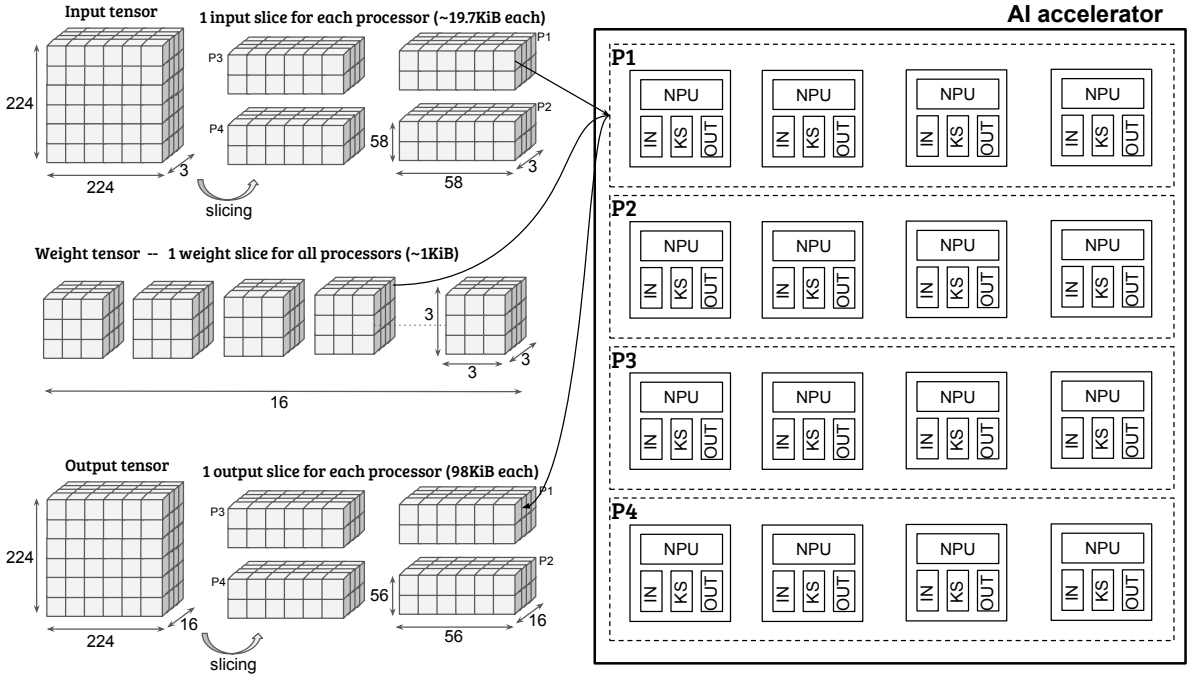

Figure 2.3: The input, weight, and output tensors are partitioned to generate slices for the NPU's processors. Note that distinct slices from the input and output tensors are assigned to the NPU's processors. As for the weight tensor, a single slice is extracted and shared among all NPU's processors for computing.

slice of the input tensor and a slice of the weight tensor to each set of interconnected NPU cores, referred to as processors (e.g., P1) in Figure 2.3. These slices are then distributed internally between the NPU cores and then computed so that each NPU processor produces distinct slices of the output tensor. There are many configurations that can be explored in this partitioning step, for instance, the input tensor could be partitioned into four slices, and the weight tensor partitioned into only a single slice so as to have each slice of the input tensor assigned to a distinct processor and the single slice of the weight tensor assigned to all processors. Contrary to that, one could divide the weight tensor into four slices, and the input tensor into just one slice. Lastly, both the input and weight tensors could be divided, into two slices each. Note that at the end, each processor has to basically generate a distinct slice of the output tensor. Depending on how the input and weight tensors are sliced, either by prioritizing the division of the input or weight tensor into slices (or in some cases dividing both in the same proportion), the number of transactions from/to the DRAM can considerably decrease. For the example presented in Figure 2.3, the chosen configuration is the one where the input tensor is divided into one slice for each processor and only a single slice is extracted from the weight tensor and shared among all processors. As a result, a workload of approximately 118KiB is assigned to each processor. Alternatively, if the decision is made to divide the weight tensor instead, the workload would be around 686KiB (5.8x bigger). Although this particular case benefits from slicing the input tensor, it might not be the appropriate solution for other scenarios where the sizes of the convolution tensors may vary. As a result, choosing the correct slicing scheme becomes a crucial factor in minimizing data transfers.

Chapter 3

Background

This chapter provides an overview of the main layers that constitute a Convolutional Neural Network (CNN). Furthermore, it introduces the concept of tiling and explains its application in enabling the execution of convolutions with large memory requirements on devices with limited memory, such as NPUs.

3.1 Convolutional Neural Network

A Convolutional Neural Network (CNN) model can be visualized as a directed acyclic dataflow graph, where nodes represent specific CNN operations and edges represent n-dimensional arrays called tensors. This graph follows a sequential flow, with each node taking one or more input tensors and producing an output tensor. This output tensor becomes the input tensor for subsequent operations in the graph (as shown in Figure 3.1). The CNN model takes an initial input tensor, such as a 3-dimensional image, and applies a series of operations to generate a final output tensor representing the inference result. Typically, this output tensor contains a probability distribution that classifies the input image into predefined classes, such as "dog" or "cat".

The most common operations found in CNN models are illustrated in Figure 3.1. These operations include Convolution, AddBias, ReLU, MaxPooling, Flatten, Linear, and Softmax. Each of these operations plays a crucial role in a CNN model's overall functioning and structure.

The main objective of a convolution operation is to extract features from the input tensor by applying a set of pre-trained filters, also known as kernels. Convolution involves computing a dot product between the filters and local patches of the input tensor. Additionally, a bias term, represented as a 1-dimensional array, is added to the resulting tensor. The bias helps shift the output tensor's elements towards positive or negative values, ensuring that the network does not produce tensors with only zero elements. An activation function is normally applied to the tensor after the convolution and bias addition. The Rectified Linear Unit (ReLU) is commonly used as the activation function in CNN models, which sets all negative elements to zero, effectively deeming them dead or inactive for subsequent operations. Other activation functions, such as ReLU6, which sets a threshold between 0 and 6, can also be employed. By combining these three operations

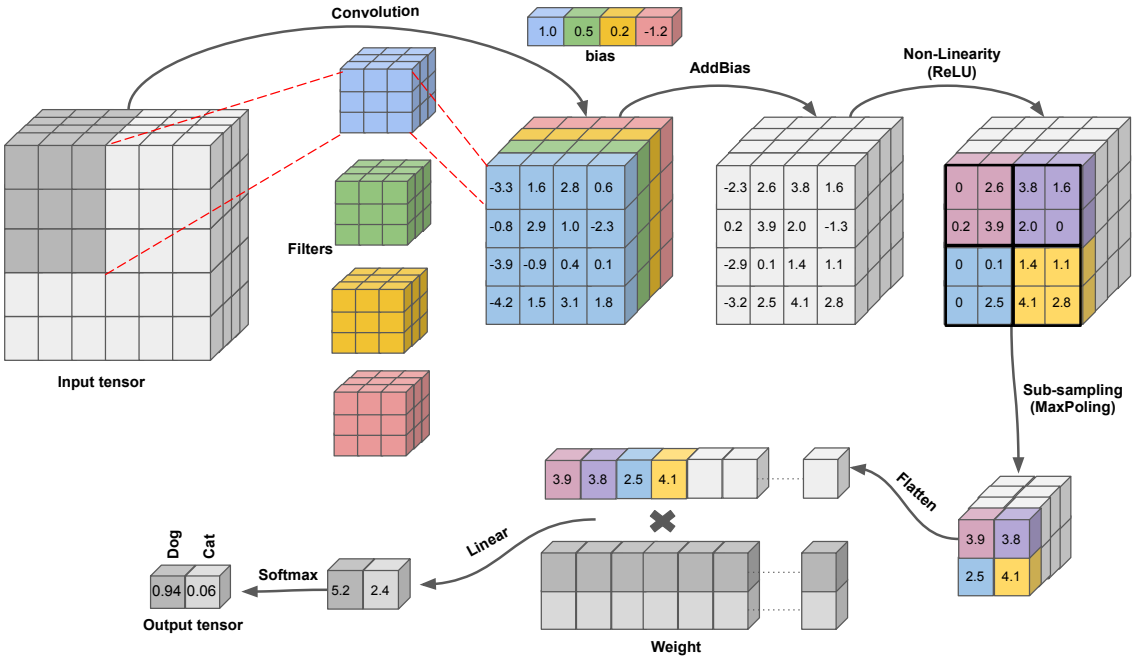

Figure 3.1: Illustration of a Convolutional Neural Network (CNN) model comprising Convolution, AddBias, ReLU, MaxPooling, Flatten, Linear, and Softmax operations. The model processes an input image through a sequence of operations to classify it into a predefined set of classes.

- convolution, bias addition, and ReLU activation - a Convolutional Layer (referred to as Conv-Layer throughout this thesis) is formed, serving as a fundamental building block in a CNN model.

The Pooling Layer, as depicted by the MaxPool function in Figure 3.1, is responsible for reducing the spatial dimensions (height and width) of the input tensor through a process called sub-sampling. In this layer, a sliding window of a specified size (e.g., 2×2) moves over the input tensor, and the maximum element within each window is selected and written to the output tensor. This operation effectively reduces the size of the tensor while preserving the most significant features. A stride value determines the step size for shifting the window over the input tensor. In the case of the example in Figure 3.1, the stride is also 2×2, indicating that each window does not overlap with neighboring windows and skips a certain number of elements in between. By applying non-overlapping windows, the pooling layer achieves dimension reduction without redundancy. In addition to MaxPooling, another commonly used pooling operation in CNN models is AVGPooling. Unlike MaxPooling, AVGPooling calculates the average of the elements within each window, providing a different way to downsample the input tensor. Both MaxPooling and AVGPooling contribute to the overall feature extraction and spatial dimension reduction process within CNN models.

Before performing the computation of a Fully Connected Layer, represented by the Linear operation in Figure 3.1, the input tensor needs to be flattened from a 3-dimensional array to a 1-dimensional array. This flattening operation, also known as the Flatten Layer, reshapes the tensor without modifying the data layout in memory, where only

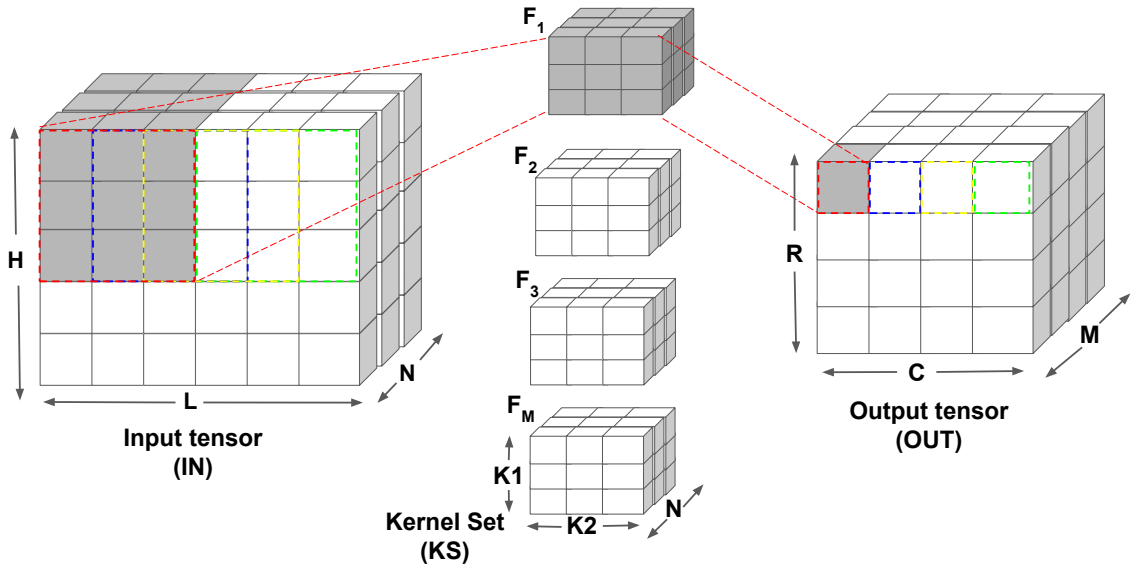

Figure 3.2: A Convolution operation where the gray area in the input tensor is computed with a filter to generate the element highlighted in gray in the output tensor. The dashed boxes represent the computation performed by the same filter afterward.

the view of the data is changed, and this is done at zero cost. Once the input tensor is flattened, the Linear operation multiplies the resulting 1-dimensional array by a pre-trained 2-dimensional weight array. This multiplication process, along with an optional bias addition and activation function (e.g., ReLU), forms the fully connected layer (FC-Layer).

Finally, the Softmax operator is employed for image classification models to compute the probabilistic distribution for each neuron (element) in the output tensor. This distribution assigns a probability to each class label that the network is trained to classify. The neuron with the highest probability is typically considered the Top-1 classification. However, other Top-k values, such as Top-5, can also be extracted to better understand the network's predictions. For instance, in the InceptionV3 model, the resulting output tensor contains 1000 elements, with each element corresponding to the classification probability of a distinct class label.

3.2 Convolutional Layer (Conv-Layer)

Among all possible layers composing a CNN, the Conv-Layer usually accounts for more than 90% of the execution time [17] of a model and generates a large amount of data movements. This is especially critical on architectures with small on-chip memories, such as NPUs. Due to the burden imposed by this layer, this work aims to design specialized data tiling/scheduling algorithms for Conv-Layers, focusing on reducing memory transaction overhead. Moreover, since NPUs are used to perform *Inference* on mobile and embedded devices, this work is restricted to optimizing inference on pre-trained models.

As shown in Figure 3.2, a Conv-Layer is composed of an input (`IN`) tensor with N input

feature maps of size H (height) $\times$ L (width) each and a set of pre-trained weights. The weight set (KS) is a set of M multidimensional 3-D array kernels/filters of size $N \times K1 \times K2$. Each filter slides over the IN tensor performing a 3-D convolution with a stride factor of S. Note in the IN tensor of Figure 3.2 the presence of red, blue, yellow, and green dashed blocks. They represent the order in which the filter is passed over the IN tensor during the computation. Additionally, note that the dashed blocks are shifted only by a factor of 1, which indicates the stride of the convolution. After sliding a single filter over the entire input image (IN tensor), an $R \times C$ output feature map is generated in the output (OUT) tensor. In the OUT tensor, it is possible to see the corresponding output to each dashed block of the IN tensor, represented with the same color. A set of M output feature maps results after applying all M filters in KS to the input (IN) tensor. The sequence of operations performed by the Conv-Layer, including the bias addition and activation function (ReLU), is shown in Equation 3.1.

$$\texttt{OUT}[m][r][c] = \texttt{ReLU}(\texttt{Bias}[m] + \sum_{k=0}^{N-1} \sum_{i=0}^{K1-1} \sum_{j=0}^{K2-1} \texttt{IN}[k][S \times r + i][S \times c + j] \times \texttt{KS}[m][k][i][j]),$$

$$where, \begin{cases} 0 \leq m < M, \\ 0 \leq r < R, \\ 0 \leq c < C, \\ R = (H - K1 + S)/S, \\ C = (L - K2 + S)/S \end{cases} \tag{3.1}$$

The number of operations required to compute a single element in the OUT tensor is $2 \times N \times K1 \times K2$, where 2 indicates both multiplication and accumulation. Note that the number of operations is proportional to the size of the filters in KS, which can grow considerably depending on the kernel size, impacting as well as the number of memory transactions required to read both the IN and KS tensors from DRAM. In order to compute the entire convolution, including all elements in the OUT tensor, a total of $2 \times M \times R \times C \times N \times K1 \times K2$ operations are required. The computation of each element in the OUT tensor can be performed independently of the others, i.e., in parallel. Thus, in principle, one can schedule the computation of each filter in KS to be processed in parallel with respect to the others. The schedule can be done in various ways, but selecting the appropriate one can lead to better data reuse and better use of the computational resources available on the target device. The computation of the OUT tensor can be formulated using the Equation 3.1. The bias addition (Bias) and activation (ReLU) operations shown in Equation 3.1 are not as computationally expensive as the computation of the convolution itself. Both the bias addition and the activation function (ReLU) perform $M \times R \times C$ operations. As for the bias addition, it involves element-wise additions, whereas the ReLU entails computing the maximum between zero and the resulting output tensor from the bias addition.

Two other concepts commonly used to define a convolution are *batch* and *padding*.

The batch determines the number of input images that are fed to a CNN model to be executed at once. A batch B results in an input tensor of size $B \times N \times H \times L$, which in turn produces an output tensor of size $B \times M \times R \times C$. In this work, the batch is defined as 1 for all tested CNN models. The other concept is padding, which is necessary when the spatial dimensions (R and C) of the `OUT` tensor are intended to have the same sizes as the spatial dimensions of the `IN` tensor (H and L). Padding is used in those cases to increase the H and L dimensions of each input feature map so that new rows and columns filled with zeros are appended to their borders. A special case where the filters have size $N \times 1 \times 1$ and stride equal to 1 does not require padding, as the `OUT` tensor is already produced with the same spatial dimensions as the `IN` tensor. Padding is applicable for the other cases when either K1 or K2 values are bigger than 1. Consider, for example, a filter with a size of $N \times 3 \times 3$. In this case, each input feature map (H and L) in the `IN` tensor is expanded by adding 2 additional rows (one at the top and one at the bottom) and 2 additional columns (one at the left and one at the right). By employing this expansion, sliding the filters over the padded `IN` tensor results in an `OUT` tensor with the same feature map dimensions as the original `IN` tensor. Although zero-padding is the most common case when padding the input (`IN`) tensor, padding with other constants may also be applied.

3.3 Tiled Convolution

Convolutions in a CNN model typically have different kernel sizes and different numbers of input (M) and output (N) feature maps with distinct sizes and variable strides. For instance, the first convolution from the Inception-V3 model [78] has $3 \times 299 \times 299$ elements in its `IN` tensor. Considering that this tensor's data type is a 16-bit fixed-point, the total size required to allocate it in memory becomes 524KiB, which is impractical when working with NPUs with constrained on-chip memories. Note that this 524KiB includes only the `IN` tensor. Hence, if we consider the `OUT` tensor and the filters in `KS`, this size would increase considerably more. To circumvent this problem, data tiling becomes a mandatory task when computing convolution.

Data tiling of a convolution operation consists in partitioning its `IN` and `OUT` tensors into small tiles and dividing the filters in `KS` so that each $\texttt{IN}^T$, $\texttt{OUT}^T$ and $\texttt{KS}^T$ tiles fit together at the same time in the NPU on-chip memories. Figure 3.3 shows how the `IN` and `OUT` tensors data are respectively divided into $\texttt{IN}^T = (T_N, T_H, T_L)$ tiles, and $\texttt{OUT}^T = (T_M, T_R, T_C)$ tiles. Notice that the dimensions T_H and T_L of the $\texttt{IN}^T$ can be computed using the dimensions T_R and T_C of the $\texttt{OUT}^T$ tensor through Equation 3.2.

$$\begin{aligned} T_H &= (T_R - 1) \times S + K1 \\ T_L &= (T_C - 1) \times S + K2 \end{aligned} \tag{3.2}$$

, where $K1$, $K2$, and S are the kernel sizes and stride, respectively. Moreover, if the filters/kernels in `KS` do not fit in their respective NPU on-chip memories, `KS` is also partitioned into $\texttt{KS}^T = (T_M, T_N, K1, K2)$ tiles, where T_M is the number of filters and T_N the number of channels to be loaded from each filter. In this work, the full values of K1

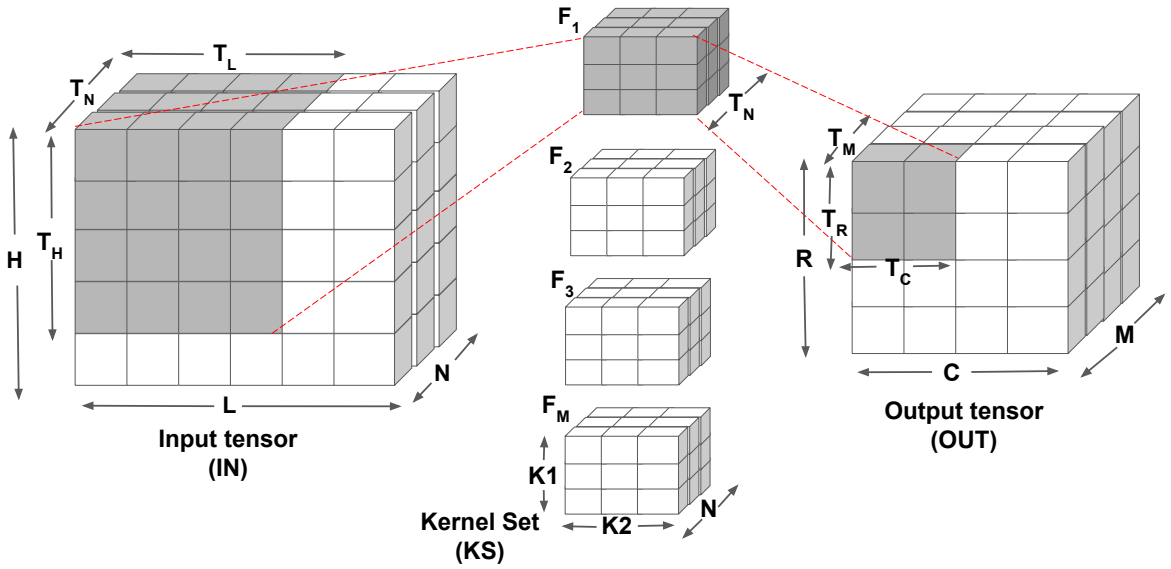

Figure 3.3: A Convolution operation with tiling applied to it to allow its data to fit in memory-constrained devices. The tiles are formed as follows: (a) $\texttt{IN}^T = (T_N, T_H, T_L)$, (b) $\texttt{KS}^T = (T_M, T_N, \text{K1}, \text{K2})$, and (c) $\texttt{OUT}^T = (T_M, T_R, T_C)$.

and K2 are used when defining the $\texttt{KS}^T$ tile, but tiling could also be applied to these two terms.

As shown in Figure 3.3, data tiling extends beyond a single convolution component, such as the KS tensor, and encompasses all involved components, including the IN, KS, and OUT tensors. Hence, when applying tiling to one component (e.g., the KS tensor), it becomes crucial to consider the memory requirements of the other two components (IN and OUT tensors) as well. This ensures that all three ($\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$) tiles can fit within their respective on-chip memories on the accelerator at the same time when computing an $\texttt{IN}^T$ tile with a $\texttt{KS}^T$ tile to generate an $\texttt{OUT}^T$. Therefore, a coordinated approach is essential for data tiling, considering the memory constraints of all convolution components and the NPU's memory limitations, in order to optimize their storage and computation on the device (more details in Section 6.3).

When partitioning a convolution for execution on an NPU, it is possible to explore different tile shapes. However, it is essential to consider certain constraints. Firstly, increasing the number of filters (T_M) in $\texttt{KS}^T$ will also increase the number of output feature maps in T_M, by the same amount, as T_M is also defined in $\texttt{OUT}^T$ tile. Secondly, increasing the number of filter channels (T_N) in $\texttt{KS}^T$ will result in a corresponding increase in the number of channels in the $\texttt{IN}^T$ tile, as both the $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles are determined based on T_N. Finally, increasing the spatial dimensions (T_R and T_C) of the output feature maps in the $\texttt{OUT}^T$ tile will lead to an increase in T_H and T_L in the $\texttt{IN}^T$ tile, as they are derived through a linear function of T_R and T_C (as depicted in Equation 3.2).

Each tile shape leads to different memory accesses and usage of the resources available on the NPU. Besides that, different scheduling strategies can be explored, each one with a specific memory access pattern that leads to different data reuse. The way computation

is mapped can considerably affect the data movement between host and NPU memories leading to poor data re-use. Moreover, if not properly done, tiling can also result in poor utilization of the NPU's Multiplier Accumulator (MAC) units, which can become idle during the computation (more details in Section 6.4).

Chapter 4

Hardware Architecture

Although CPUs have been proposed to accelerate CNNs by relying on multicore parallelism and SIMD instructions [48, 82], the number and complexity of the layers in modern CNN models make it very difficult to run the entire network on CPUs. To improve inference throughput, (fast) GPU solutions have been proposed to process a large amount of data [16, 76]. Field Programmable Gate Arrays (FPGAs), on the other hand, have been extensively used as an alternative to this problem as they offer good performance and reconfigurability [10, 14, 23, 64, 71]. Nevertheless, these architectures are not efficient power-performance solutions for critical edge applications, like surveillance cameras and cellphone face recognition, etc., which have stringent execution and power consumption constraints. Several types of accelerators have been proposed to accelerate CNNs in a power-efficient way. Specialized ASICs [9], Neural Processing Units (NPUs) [41, 54], and Tensor Processing Units (TPUs) [36] are some examples.

This thesis uses the NeuroMorphic Processor (NMP) by LG Electronics (LGE) as a compiling target. The NMP is an AI accelerator designed primarily for integration into embedded systems, such as televisions and refrigerators. Its architecture was specifically tailored for the execution of CNN models, including tasks like image classification and object detection, thanks to its well-suited computing units and on-chip memories. The key idea behind the NMP architecture is to use RISC-V ISA Extensions to design relevant CNN instructions like Conv-Layers, fully connected layers, pooling layers, element-wise operations, etc.

4.1 The NMP Architecture

The NMP architecture (Figure 4.1) is a multicore NPU that contains an ARM57 processor that works as a host for a set of multiple Tile (TLE) processors, containing each a set of Tilelet (TLT) cores. Each TLT has one RISC-V core and three on-chip (scratchpad) memories, namely MB0, MB1, and MB2, which respectively store the $\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$ tiles from the `IN`, `KS`, and `OUT` tensors. Besides that, each TLT is also equipped with a MAC acceleration unit to execute CNN operations. The MAC unit execution is triggered by the RISC-V core and is capable of executing 8- and 16-bit fixed-point operations with the memory layout organized in NCHW format. The datapath between the MAC unit

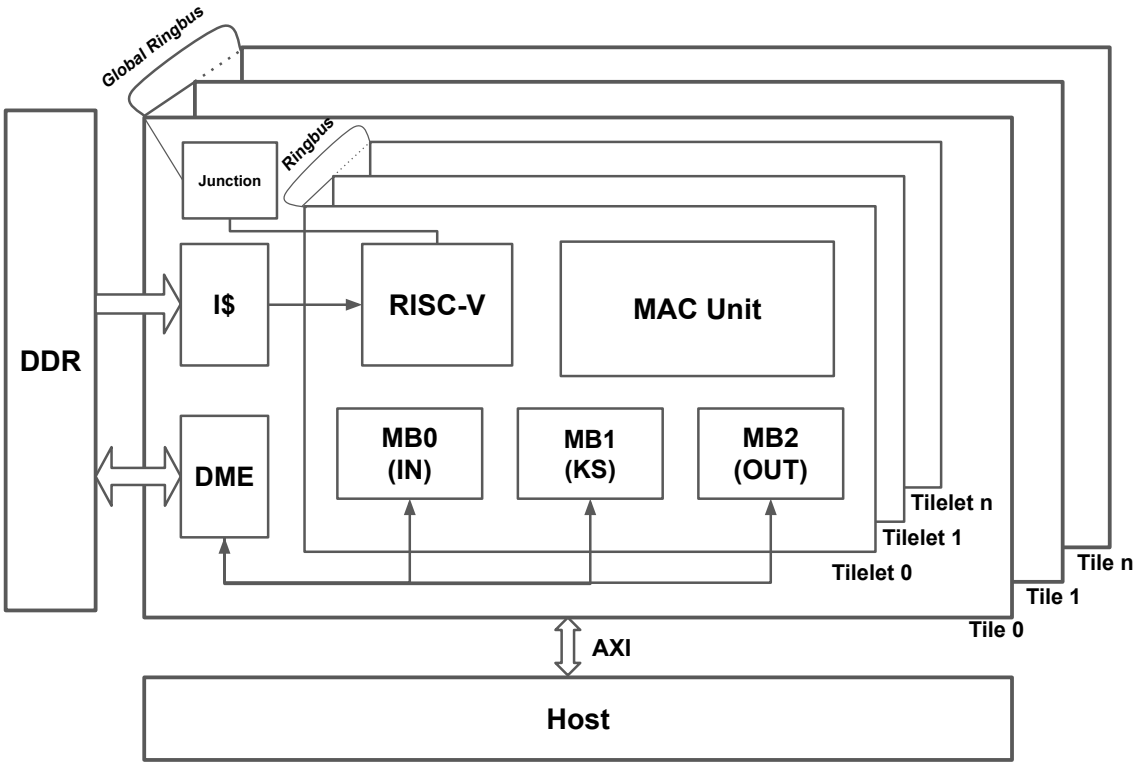

Figure 4.1: NMP Architecture.

and TLT on-chip memories (MBLOBs - MB0, MB1, and MB2) is 128-bit wide, which means that for 8-bit fixed-point, up to 16 MAC operations are executed per cycle, while for 16-bit fixed-point, up to 8 MAC operations per cycle may be executed.

The data transfers between NMP and host happen through a Data Movement Engine (DME) module, as shown in Figure 4.1, with one DME module for each TLE. The DME module slices the memory requests into transactions of up to 128 bytes. The host communicates with the NMP through an AXI interface, and data can be shared between TLTs of different TLEs by using a Global Ringbus. The TLTs of a TLE also have their own Ringbus to communicate data between them. The instructions executed by the RISC-V cores are fetched from the host memory and stored into a cache instruction shared between the TLTs of the same TLE.

To execute computations on the NeuroMorphic Processor (NMP), a CNN model needs to be compiled using an ML compiler like TF-XLA [3]. The compilation process generates two binary files: (1) a binary file containing RISC-V instructions and (2) a binary file containing the weights for both convolutional and fully connected layers (more details in Chapter 5). These files need to be loaded into specific memory addresses using the NMP device descriptor. Once the loading is complete, a system call is invoked to initiate the computation on NMP. The execution of the compiled model starts on the RISC-V cores of each TLT, with each TLT operating independently. Similar to GPUs, each TLT is identified by a unique ID, allowing them to determine the portion of data they are responsible for computing.

TLTs primarily use primitive RISC-V instructions for control flow, such as looping over tiled data. In addition to these instructions, the NMP architecture includes an *extended*

Type of Instruction	Instructions
Data Movement	nmp_load[3d]
	nmp_store[3d]
Layers	nmp_activation
	nmp_conv2d
	nmp_percept
	nmp_pool
	nmp_veop
Synchronization	nmp_wait
	nmp_signal

Table 4.1: Extended instructions supported by NMP.

set of RISC-V instructions for (refer to Table 4.1): (a) Data movement: these instructions, such as *nmp_load* and *nmp_load3d*, are executed by the RISC-V core to load data from the DRAM to the TLT's on-chip memory. The Data Movement Engine (DME) handles the data transfer. To bring data back from each TLT's on-chip memory to the host DRAM, the RISC-V core executes the extended *nmp_store* and *nmp_store3d* instructions; (b) Layer computation: to perform computations on the Multiply-Accumulate (MAC) units, the RISC-V core invokes one of the following extended instructions: *nmp_activation*, *nmp_conv2d*, *nmp_percept*, *nmp_pool*, and *nmp_veop*; and (c) Synchronization: two instructions, *nmp_wait* and *nmp_signal*, are used for synchronization. *nmp_wait* represents a barrier that waits until a specific task is completed, such as a data load from the host memory. When the barrier is resolved, a signal is emitted from one RISC-V core to inform other participating RISC-V cores that the task has been completed. Overall, the extended RISC-V instructions in the NMP architecture enable data movement, layer computation, and synchronization to facilitate the execution of CNN computations efficiently.

Despite the independent execution of computations by each TLT within a single TLE, NMP introduces a special instruction called the *multicast* load. This instruction operates on a single TLT but enables all TLTs within the same TLE to load the same data from the DRAM simultaneously. To illustrate this, consider a scenario where all TLTs within a particular TLE process the same $\texttt{IN}^T$ tile but compute with different $\texttt{KS}^T$ tiles. Instead of loading the identical $\texttt{IN}^T$ tile multiple times, one for each TLT in the TLE, a multicast load can be employed. This capability allows for concurrent processing of the $\texttt{IN}^T$ tile by different filters in parallel, maximizing performance and throughput. It is important to note that TLTs from distinct TLEs cannot utilize the same multicast load to load the same data. In such cases, multiple multicast loads are required, with each TLE executing its own multicast load to ensure that the same data is loaded into all of its TLTs.

NMP provides three levels of hardware-based semaphores to facilitate synchronization among TLEs and TLTs. These semaphores support various synchronization operations, including: (a) Synchronizing computation within a TLT – after invoking the MAC unit to perform an operation (e.g., *nmp_conv2d* instruction), the RISC-V execution can be blocked until the MAC unit completes its task. This enables precise coordination between the RISC-V core and the MAC unit; (b) Synchronizing computation between TLTs of the

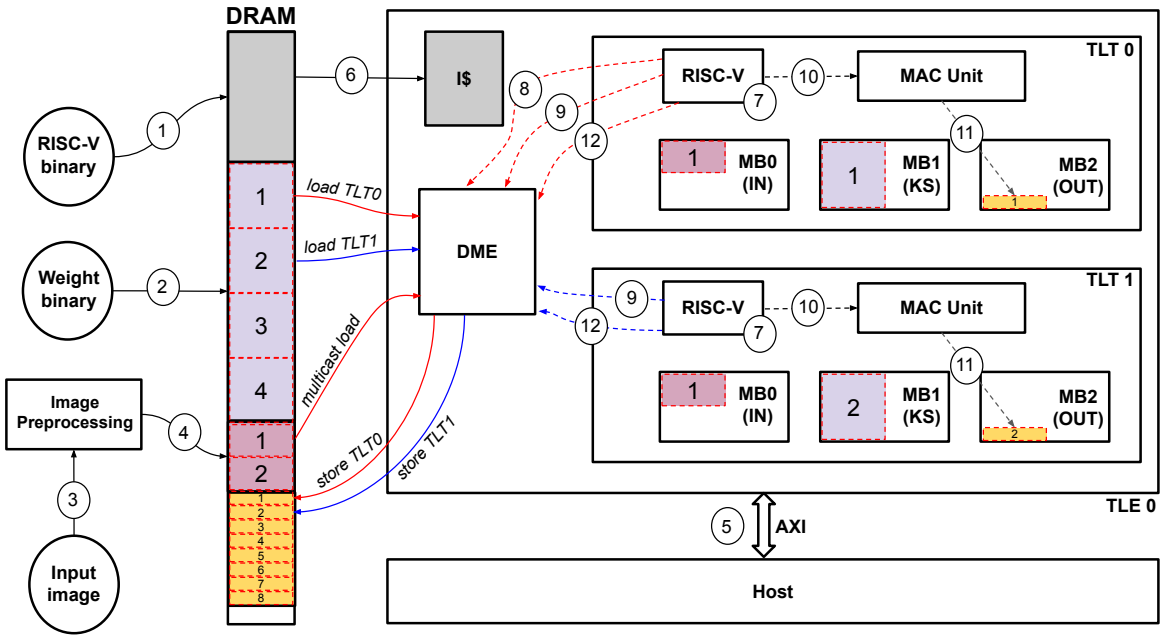

Figure 4.2: The NMP workflow where a step-by-step execution is shown for a CNN operation.

same TLE – during a multicast load, where all TLTs within a specific TLE load the same data from the DRAM, synchronization plays a crucial role. The semaphores ensure that all TLTs are blocked until their respective on-chip memories receive the data. This guarantees synchronized access to shared data within the TLE; and (c) Synchronizing computation between TLTs of different TLEs – in a multi-layer CNN model, performing proper synchronization among TLTs (of different TLEs) is required before initiating the computation of the next layer. This is because a TLT within a TLE may depend on the data generated by a TLT from a different TLE. To ensure correct sequencing, if a TLT completes its computation before other TLTs working on the same layer, it must be blocked from proceeding to the subsequent layer. This blocking mechanism ensures that the TLT waits until all relevant computations across TLEs are finished, maintaining the integrity of the computation flow within the model. Therefore, the availability of hardware-based semaphores in the NMP architecture enables efficient and reliable synchronization at different levels, ensuring correct execution order and data consistency within and across TLEs and TLTs.

As stated before, the NMP workflow is similar to that of a GPU. To illustrate the NMP workflow, see Figure 4.2, which shows each step in a simplified NMP architecture with 1 TLE containing 2 TLTs (for simplicity). Initially, the host application loads both the RISC-V binary ① and the Weight binary ② files into the host memory (DRAM). The data in the Weight binary file constitutes multiple `KS` tensors of different operations besides the *bias* vector that are consumed by the Conv- and FC-Layers. In the sequence, the host application pre-processes the input image according to the range expected by the CNN model ③. As an example of pre-processing, there are models that expect the RGB channels to be within the range of [-1,1], while others expect them to be within the

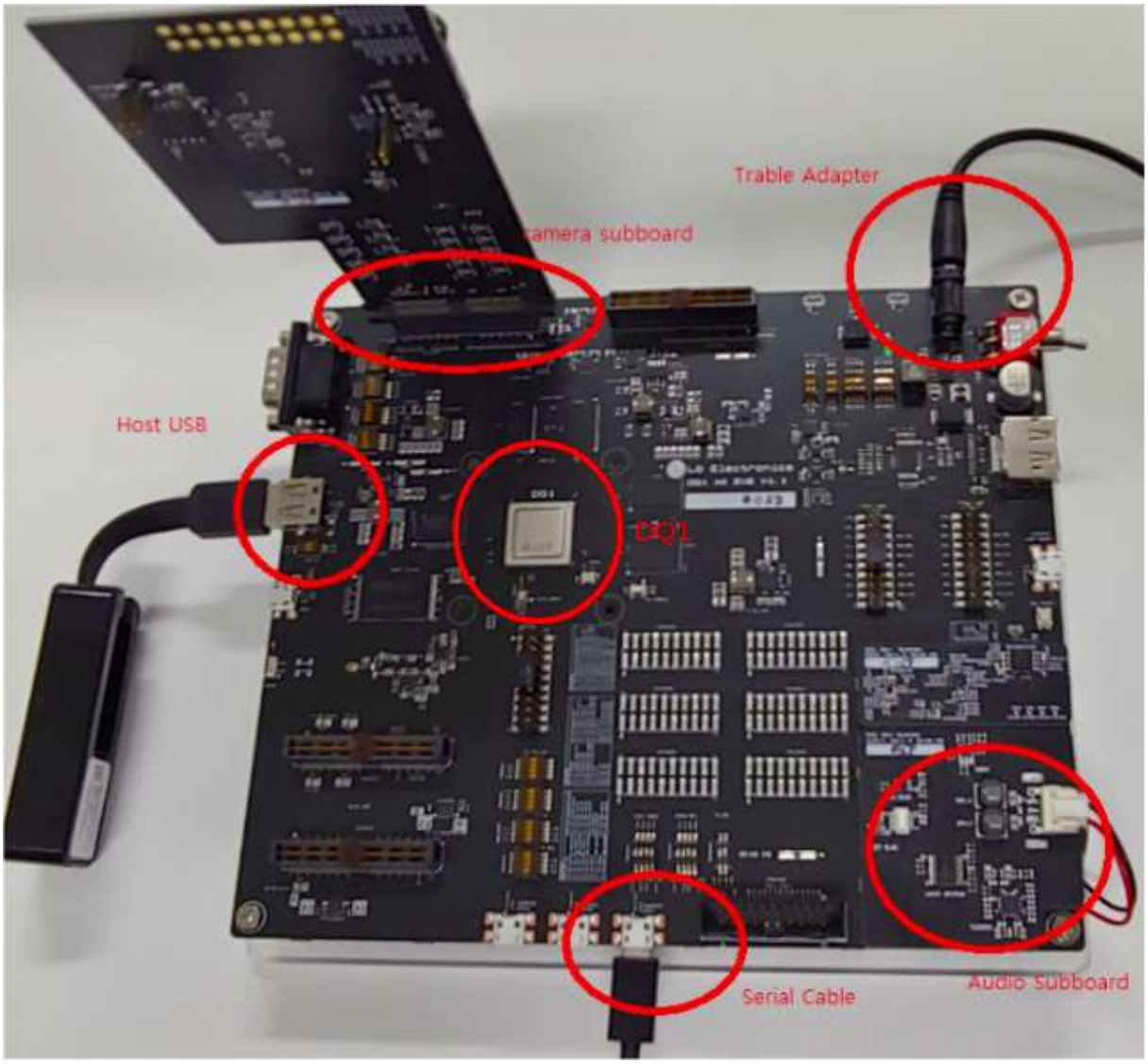

Figure 4.3: Development board with an NMP DQ1-A0 chip integrated on it.

range of [0,1]. The pre-processed image is copied to the DRAM to be later consumed by NMP ④ as an `IN` tensor. After that, the host application issues a signal ⑤ to first copy the RISC-V instructions (which includes the extended instructions) from DRAM into the TLE's instruction cache via a dedicated AXI ⑥ and then wake up the RISC-V cores ⑦ to start computing those instructions. Before starting the computation of the layer itself on NMP, it is necessary to transfer the input data ($\texttt{IN}^T$ and $\texttt{KS}^T$ tiles) from the host memory (DRAM) to the accelerator on-chip memory. The TLT0's RISC-V core begins by issuing a multicast load instruction to the DME unit ⑧ to bring the first $\texttt{IN}^T$ tile (referred to as 1) from DRAM into MB0 of each TLT. After this instruction is issued, TLT0 enters a barrier and waits until the DME finishes the load. Meanwhile, TLT1 also runs into the same barrier. Once both TLTs are notified by the DME on the completion of the multicast load instruction, the RISC-V core of each TLT proceeds to the next RISC-V instructions. The next load instruction is requested, this time, each TLT requests to the DME unit ⑨ a distinct $\texttt{KS}^T$ tile to be loaded from DRAM (refereed as 1 and 2, respectively). Each load instruction is then attended at a different moment, and a semaphore is employed

individually on each TLT to ensure the data arrives at the destination (MB1). With the $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles available at both TLTs in their MB0 and MB1 on-chip memories, respectively, the RISC-V core of each TLT emits a special instruction that sends a signal to its MAC Unit (10) to run the CNN layer instruction (e.g., *nmp_ conv2d*) over the loaded data. The TLTs wait for the completion of the computation ((11)). Once finished, they receive notifications from their respective MAC Units indicating the completion of the computation and the availability of the resulting $\texttt{OUT}^T$ in their corresponding MB3 on-chip memories. Finally, each TLT's RISC-V core issues a signal to the DME unit to write their $\texttt{OUT}^T$ tile into DRAM (12). The same process explained herein is repeated as many times as the number of remaining $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles that are left to be computed on the DRAM. After the completion of the entire CNN layer, control is passed back to the CPU to consume the output data or, in the presence of a subsequent layer in the CNN model, the same steps are repeated.

The NMP architecture used in this work, namely DQ1-A0 (see Figure 4.3), is composed of 4 TLEs, each containing 8 TLTs (RISC-V + MAC unit). Each TLT has three on-chip memories of size 8KiB each. With an operating frequency of @1GHz, the NMP (i.e., all the 32 TLTs together) has a theoretical peak performance of either 512- or 256-GMACs/sec when executing 8- or 16-bit fixed-point data, respectively. The DRAM memory is a DDR3 that operates at a 1066MHz clock rate (DDR3-2133 – 17GiB/s). For this edition of the architecture, NMP does not enable MAC/LOAD overlap as it is single-ported. For future NMP architectures, we anticipate the addition of extra on-chip (dual-port) memories that will allow the compiler to software-pipeline MAC/LOADs.

Although the NMP architecture shows to be very efficient at tasks like image classification and object detection, a limitation imposed by its architecture, in addition to the constraint posed by the size of its on-chip memories, is the limited mapped DRAM memory space, restricted to 256MB. This limitation presents challenges for the execution of numerous models due to their extensive memory requirements for weight allocation.

Chapter 5

The Software Framework

This chapter discusses essential compiler techniques necessary for code transformation and optimization when targeting NPUs in general, and the NeuroMorphic Processor (NMP) in particular. These techniques are applicable to all compilers that have implemented Tensor Slicing Optimization (TSO). Furthermore, a concise overview of each compiler utilized in this study is presented, along with an explanation of how NMP integrates within their frameworks.

5.1 Code Generation for NMP

When targeting compilation for NMP, the initial input for the compiler is a CNN model, as illustrated in Figure 5.1. The compilation process consists of several steps aimed at generating the required files for executing computations on NMP. These files include: (1) a binary file containing RISC-V instructions; and (2) a binary file containing the pre-trained weights/filters and bias. The compiler performs a series of analyses and optimization passes using a Pass Manager (PM) to generate the RISC-V instruction file. The PM plays an important role in producing highly optimized code that is specifically tailored for efficient execution on NMP.

After reading the **CNN model**, the compiler proceeds to convert it into an intermediate representation (IR), referred to as **Compiler IR** in Figure 5.1, utilizing a compiler pass called **Model converter**. The Compiler IR then undergoes a series of graph-level transformations, analyses, and optimizations. These operations are performed through a sequence of **OPT passes** defined within the Pass Manager (PM). Each optimization pass generates a new Compiler IR, although it is not explicitly shown in Figure 5.1 for simplicity. The series of OPT passes work iteratively to refine and improve the Compiler IR, leading to a highly optimized representation of the CNN model for efficient execution on NMP.

5.1.1 Fusion

The OPT passes primarily focus on *loop fusion* [51] as the main optimization technique, which involves combining multiple CNN operations into a single operation. Depending on the ML compiler, operations like convolution, bias summation, and activation functions

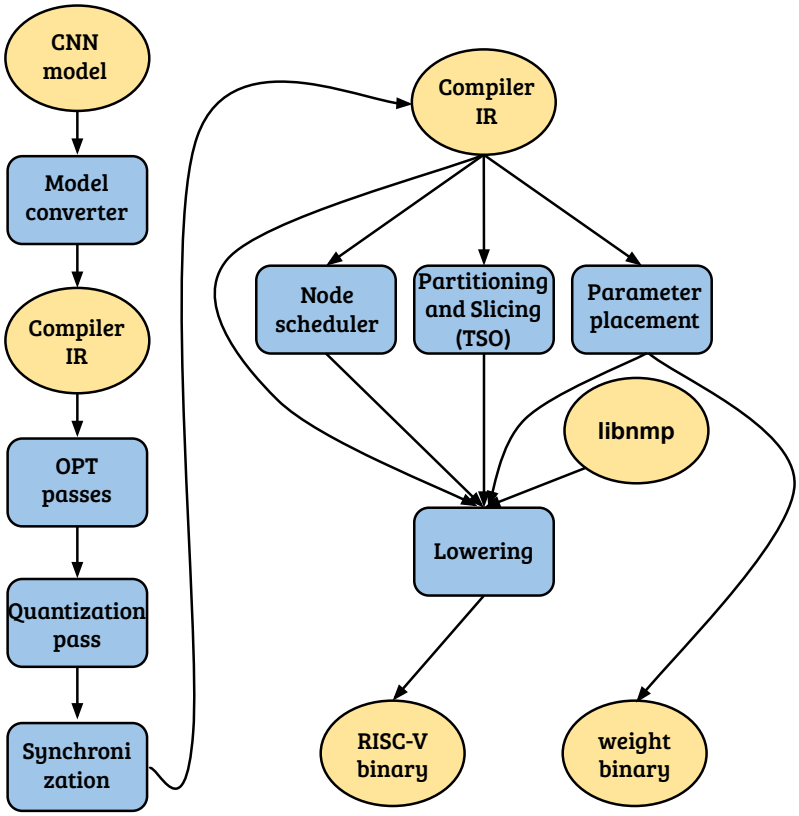

Figure 5.1: Common compilation flow required for any ML compiler when targeting code to NMP.

(e.g., ReLU) may be treated as separate operations. Fusing these operations together can significantly reduce the amount of data movement required between DRAM and the NMP's on-chip memories at runtime. As previously mentioned, when NMP computes an $\texttt{OUT}^T$ tile, it is written back to the host memory (DRAM) due to the limited capacity of the on-chip memory (MB3) to store all the assigned $\texttt{OUT}^T$ tiles for a TLT at once. Consequently, when computing the next layer, the $\texttt{OUT}^T$ tile of the current layer needs to be loaded back from DRAM as the $\texttt{IN}^T$ tile for the next layer's computation. This process of storing data from the NPU's on-chip memory to DRAM and then retrieving it incurs significant overhead. To address this issue, loop fusion is applied. By sequentially computing the fused operations, the data is stored back to DRAM only after all the fused operations have been computed. This eliminates unnecessary data transfers between DRAM and on-chip memory, improving efficiency and reducing overhead. By minimizing the frequency of data movement, loop fusion enhances the overall performance when running CNN models on the NMP architecture. For a detailed discussion of the impact of loop fusion on the execution time of NMP, refer to Section 8.1.

The main fusion patterns used in this work are (1) $convolution + bias + ReLU$, and (2) $linear + bias + ReLU$. Merging the bias summation and activation function is straightforward since these operations have a one-to-one relationship, meaning that each output

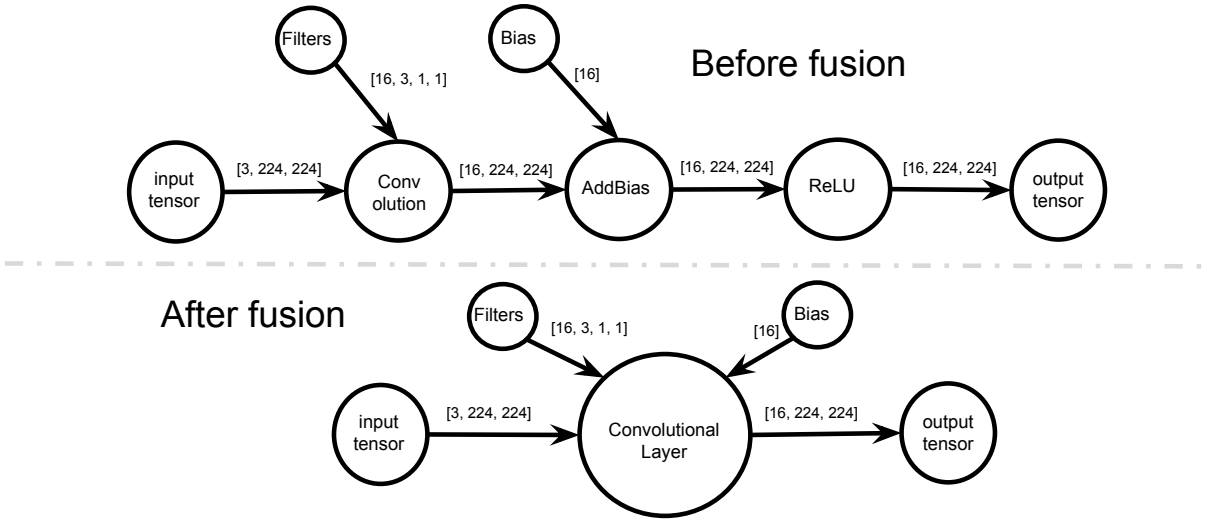

Figure 5.2: Before and after applying the compiler pass – Loop/Operation Fusion.

element depends on a single input element. While fusing other operations like MaxPool with its predecessor is possible, it introduces complexity. MaxPool operates on windows of elements to produce a single output element. However, not all input elements required for this operation to compute a window may be available during computation. This can happen either because these elements are computed by another TLT or because they are part of the next $\texttt{IN}^T$ tile. In such cases, it is necessary to load the required data from DRAM or from the on-chip memories of other TLTs through inter-core communication. Due to the complexity involved and the limited gain achieved (due to the trade-off of inter-core communications), fusion with MaxPool and other operations has not been explored in this work.

To demonstrate the impact of fusion on NPU multicore systems such as NMP, refer to Figure 5.2. Before the operations are fused, it is important to understand the sequence: the output tensor produced from the convolution operation serves as the input tensor for the AddBias operation, which generates another output tensor. This resulting output tensor, in turn, becomes the input for the activation function (ReLU), producing a subsequent output tensor. The challenge with this approach, as previously noted, arises from the limited size of the NMP's on-chip memories. This limitation requires storing the $\texttt{OUT}^T$ tiles in DRAM between every two operations as they are computed on NMP. This is required so as to release space for the computation of the next $\texttt{OUT}^T$ tile of the same operation. As a result, this requires on NMP to place the output produced by the first operation into the DRAM memory and subsequently load the same data again (but as $\texttt{IN}^T$ tiles) from DRAM back into the NMP's on-chip memory to compute the next operation. Thus, in the example depicted in Figure 5.2, after the output tensor produced by the convolution operation is written into DRAM, that tensor has to be loaded again from DRAM to compute the AddBias and later again to compute the ReLU operations (the same number of stores from the on-chip memories to DRAM applies). For the illustrated example, this effectively requires additional load/store operations of 6.125MiB between the DRAM and the NMP's on-chip memories. In contrast, for the fused version, where the AddBias and ReLU are executed on top of the convolution's output before storing it

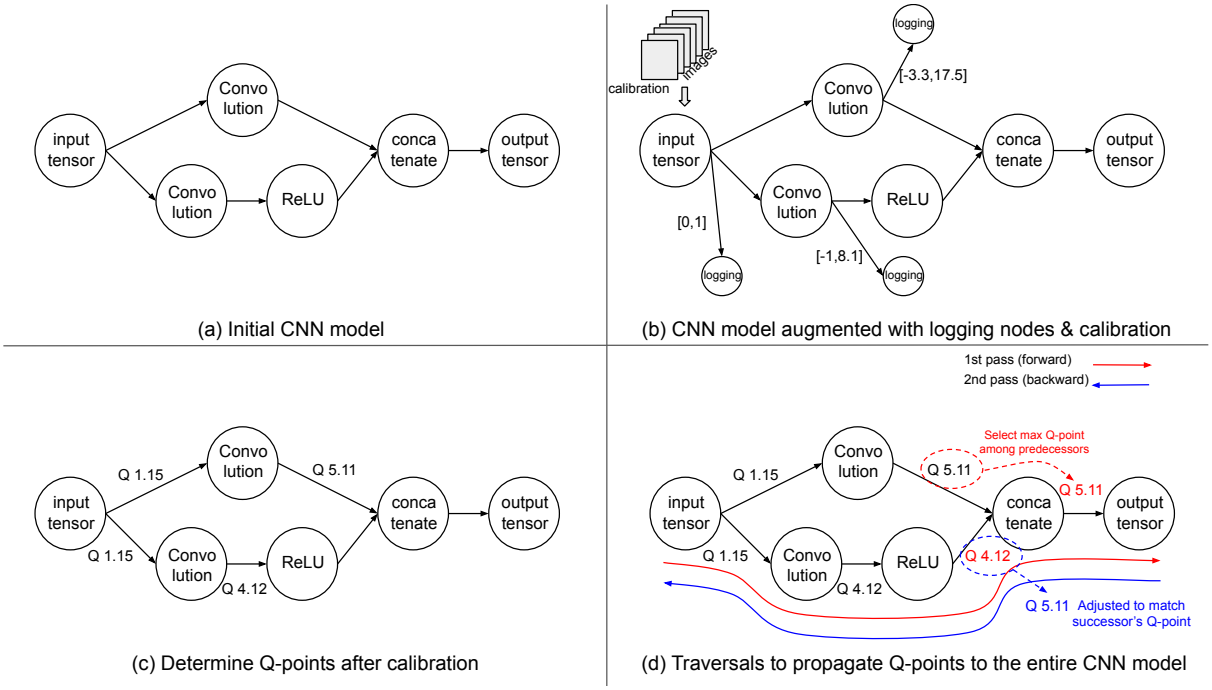

Figure 5.3: Quantization pass applied to a CNN model with steps (a)-(d) explaining from the initial model, through the insertion of logging nodes and calibration until the application of the traversal passes (forward and backward) to propagate the Q-points.

in DRAM, these additional memory transactions can be completely avoided.

In addition to loop fusion, the PM applies two other important optimizations within the OPT passes: (1) fused batch normalization into a convolution, and (2) concatenate resolver. The first optimization involves removing the batch normalization operation entirely if it follows a convolution operation. Fusing these operations together eliminates the need for separate batch normalization computations. This optimization reduces the computational overhead and memory requirements associated with batch normalization, resulting in improved performance. The second optimization, known as the concatenate resolver, aims to eliminate the concatenate operation by reorganizing the addresses where each input tensor to the concatenate operation is written. This optimization reduces the need for explicit concatenation operations and minimizes data movement between the host memory (DRAM) and the NPU's on-chip memories.

5.1.2 Quantization

The **Quantization pass** in Figure 5.1 converts the weights from 32-bit floating-point to either 8-bit or 16-bit fixed-point representation and quantizes the CNN operations to the same precision. It uses the *symmetric with power of 2 scale* scheme [58], where the zero-point is always set to 0. For each input, output, and weight tensor, this pass determines a *Q-point*, which represents the number of bits assigned to the integer and fractional parts of the tensor data through a radix point. The goal is to ensure that the integer part has enough bits to represent the minimum and maximum output values produced by the operation or constant. The Q-point is represented as $Q\ i.f$, where i represents the number

of bits for the integer part and f for the fractional part. To show the quantization flow, refer to Figure 5.3. Starting from an initial CNN model (a), to capture the ranges of the tensors, a calibration step is required (b). This step involves augmenting the CNN model by adding logging nodes after some selected CNN operations. The augmented graph is then executed over hundreds or thousands of images on a CPU, allowing the capture of the minimum and maximum values produced by each logged operation. Finally, these values are used to determine the Q-point for each CNN operation (c).

Since the calibration step is computationally expensive, optimizing the process involves restricting the logging nodes to specific layers (e.g., Convolution and Linear layers). Thus, in order to calculate the Q-points for the other layers (e.g., Pooling layer), it is used a dataflow analysis that consists of two traversals over the dataflow graph (refer to step (c) in Figure 5.3). The first traversal propagates the Q-points from the input tensor to the output tensor of the CNN model for the non-augmented nodes. The second traversal is performed in reverse order, from the output tensor to the input tensor of the CNN model, and works by adjusting the Q-points to ensure that the input tensors to the Concatenate and Add operations have the same Q-points. Quantizing the weights/filters and bias is straightforward since they are constant, making it easy to determine the Q-point for them, which can then be used to convert the data from 32-bit floating-point to either an 8-bit or 16-bit fixed-point representation.

5.1.3 Synchronization

The next compiler pass in Figure 5.1 is called **Synchronization** and is implemented using the insertion of a *global barriers*. This pass inserts global barrier nodes after each CNN operation. The purpose of this pass is to ensure that the TLTs do not start computing the next layer while the current layer is still being computed by other TLTs. This is necessary since the data produced by one TLT may be used by other TLTs during the computation of the next layer. By inserting global barrier nodes, synchronization is enforced among the TLTs, allowing data dependencies to be respected. After the Synchronization pass, a slightly optimized **Compiler IR** is generated. This optimized IR represents the modified dataflow graph with the inserted barrier nodes. It serves as an intermediate representation for further compilation steps and code generation.

5.1.4 Tiling and Scheduling

The following three passes run on top of the optimized Compiler IR (refer to Figure 5.1). They are: (1) **Node scheduler** pass is executed to extract a topological order of the CNN operations in the dataflow graph. It determines the order in which those operations run at runtime. The (2) **Partitioning and Slicing (TSO)** pass initially performs the slicing of the convolution data among the TLEs (Section 6.2). It then runs the following sequence of tasks: (a) it divides the convolution data at each TLE among their corresponding TLTs; (b) computes how the workload of an individual TLT is tiled into multiple $\texttt{IN}^T$, $\texttt{KS}^T$ and $\texttt{OUT}^T$ tiles (Section 6.3); and (c) determines how these tiles are scheduled to be executed at runtime (Section 6.4). TSO tasks (a)-(c) use a cost model that selects the

appropriate solution for each convolution from a complex tiling/scheduling solution space (Section 6.5).

Although the other CNN operations are also sliced and then partitioned among the TLTs of each TLE, they are not discussed in this thesis either due to their simplicity or to the fact that their tiling does not significantly impact the model's final performance.

5.1.5 RISC-V and Weight Binary Files Generation

The (3) **Parameter placement** pass generates the **weight binary** file, which contains the weight/filters and bias utilized at runtime by the Conv-Layers and FC-Layers. Additionally, this compiler pass determines the base address for each filter and bias within this file. These addresses are later used by the code generation pass to instruct the TLTs on from where to load the required constants (e.g., filters). Besides determining the base address within the weight file, this compiler pass also determines the addresses from which to load input tensors and where to write the output tensors for each CNN operation. Finally, the **Lowering** pass lowers the Compiler IR using the analysis data produced by the last three steps (refer to the other 2 steps in Subsection 5.1.4 for details) along with an optimized library, the libnmp, that implement the layers' computation to generate the **RISC-V binary** file.

5.2 Compilers for Machine Learning

In recent years, the increased attention to machine learning models has led to the development of several compilers to address the growing demand. Notable among these compilers are Tensorflow XLA [4], ONNX-MLIR [35], and Glow [69]. In the subsequent subsections, we will briefly describe how each of these compilers can be extended to work with NMP.

5.2.1 TF-XLA

TensorFlow [4] is a widely-used open-source machine learning library that supports both research and production applications. It provides APIs for developing machine learning models across various platforms, including desktop, mobile, web, and cloud. When working with TensorFlow, one of the key challenges is achieving high inference throughput on mobile and embedded devices, considering constraints such as performance, low latency, small model size, and portability. To address these constraints, Google has developed TensorFlow XLA [3], or simply TF-XLA, a specialized compiler.

TF-XLA is designed to optimize linear algebra operations in the TensorFlow Computational Graphs (TCGs), which represent computations as a dataflow graph with nodes as operations and edges as tensors. Instead of focusing solely on individual operations, TF-XLA takes a group of operations into consideration when optimizing the TCG. These optimizations include operation fusion, algebraic simplification, dead code elimination (DCE), and common subexpression elimination (CSE). By applying these optimizations, TF-XLA aims to maximize the efficiency and performance of the compiled executable.

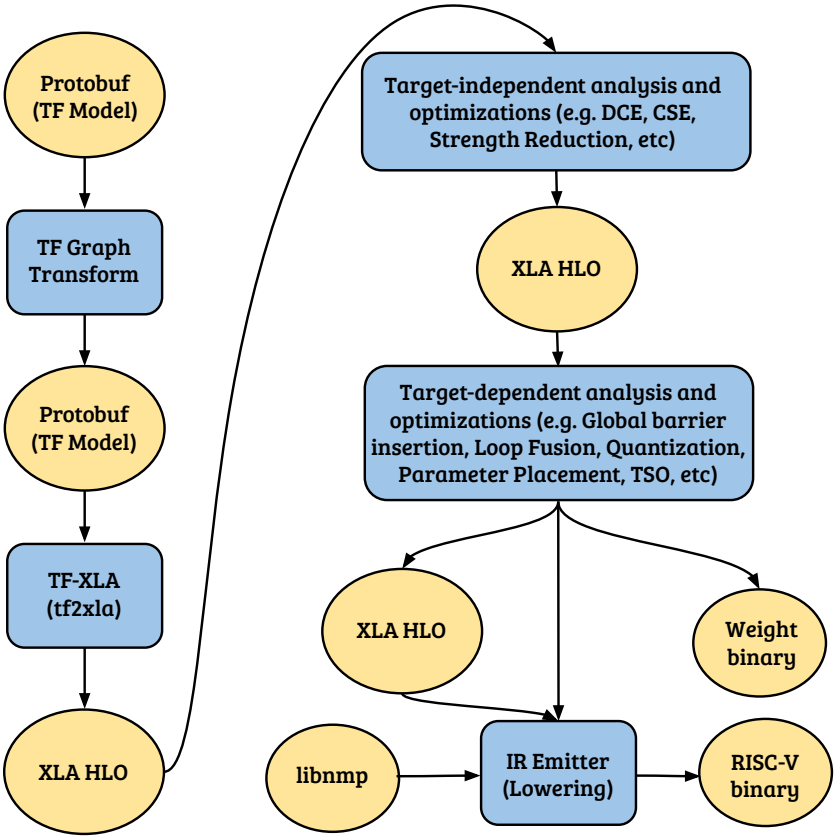

Figure 5.4: Tensorflow XLA flow for NMP.

TF-XLA offers two types of flows to the user: (a) just-in-time (JIT) compilation and (b) ahead-of-time (AOT) compilation. The JIT compilation is the simpler option, allowing the user to select a group of operations in the Python code to be compiled at runtime. However, it relies on the Tensorflow runtime, which can be a bottleneck for execution. On the other hand, AOT compilation requires some additional effort from the user but has the advantage of not depending on the Tensorflow runtime. For NMP, the AOT compilation flow was chosen due to the characteristics of NMP, as supporting the Tensorflow runtime would have been impractical due to its hardware requirements.

The TF-XLA compiler takes a *protobuf* file as input (refer to Figure 5.4), which is a serialized data structure file containing the network's definition, where the operations and their connections are listed, as well as the associated weights. Prior to lowering the protobuf file into an intermediate representation, the TF Graph Transform tool is utilized to perform various transformations, such as fold batch normalization into convolutions, hence simplifying the network for subsequent compilation steps. This process generates a new protobuf file, which serves as input for creating the XLA HLO (High Level Operations) intermediate representation within the compiler using the *tf2xla* tool. Beginning with the initial XLA HLO representation, target-independent optimizations, such as Dead Code Elimination (DCE) and Common Subexpression Elimination (CSE), are applied to produce an optimized HLO representation. Following this, target-dependent optimiza-

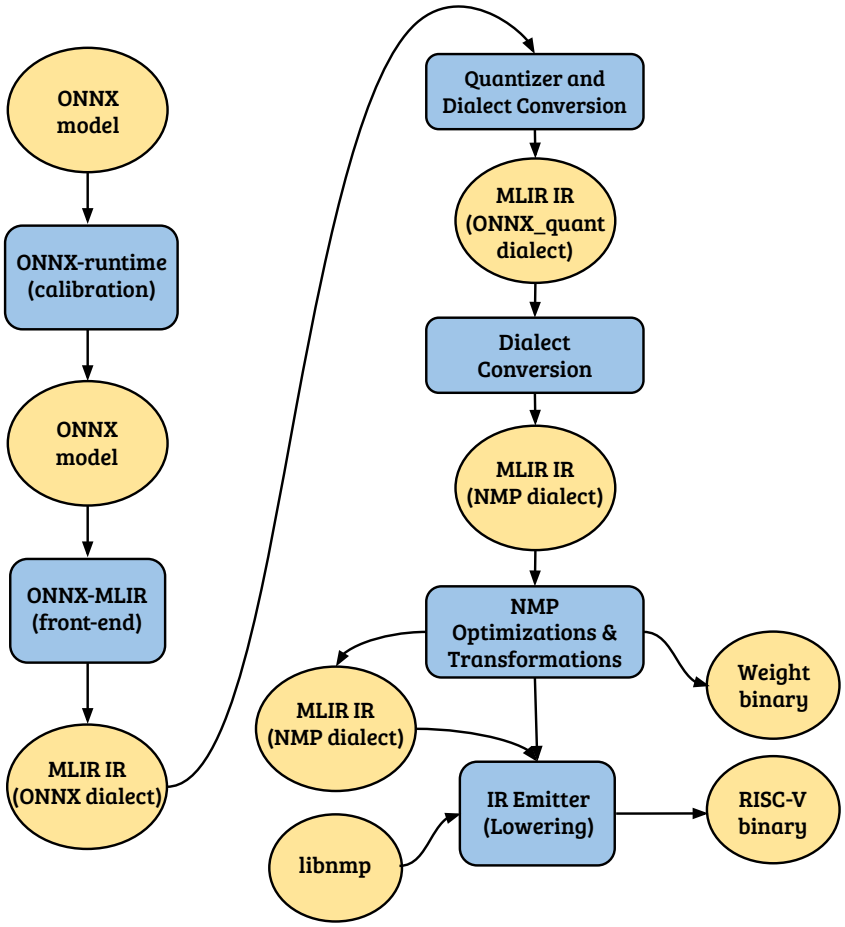

Figure 5.5: ONNX-MLIR flow for NMP.

tions, including quantization, are executed, along with other optimizations defined in the PM (e.g., global barrier insertion, parameter placement, etc). The TSO pass is also included in this step, being executed just before the lowering from the XLA HLO to the LLVM IR. During this lowering pass, the HLO instructions map to intrinsics within an optimized NMP library (libnmp), which encompasses a range of typical CNN operations. As part of code generation, the compiler produces a RISC-V executable (RISCV.bin) that is utilized by the TLTs, along with the quantized weight file (Weight.bin).

5.2.2 ONNX-MLIR

ONNX-MLIR [35] is an open-source MLIR-based compiler that focuses on accelerating the execution of machine learning models represented in the ONNX format [8]. Built on top of the MLIR framework [46], it leverages the benefits of a variety of dialects that aim to enhance performance in many ways. ONNX, like the protobuf format, utilizes a serialized object to represent the operations and the tensors within the CNN model. Its wide adoption among frameworks like PyTorch [62] and MXNet [12], as well as runtimes like ONNX-runtime [19], demonstrates its versatility and popularity within the machine learning community.

The MLIR framework offers a varied range of dialects, each serving specific purposes to fulfill various optimization and transformation requirements. For instance, the *affine*

dialect allows users to generate and manipulate loops using a powerful dependence analysis that enables a wide range of affine transformations. The *arith* dialect provides support for integer and floating-point mathematical operations. The *linalg* dialect, on the other hand, offers high-level optimizations for linear algebra operations. Within the context of ONNX-MLIR, established dialects like affine and arith are employed alongside two internally defined dialects called *ONNX* and *Krnl*. Tailored specifically for the project, the ONNX dialect aims to represent each ONNX operation internally as a map of one-to-one, while the Krnl dialect facilitates functionalities such as loop creation, loop permutation, and code generation for highly optimized matrix multiplication.

To fully exploit the advantages offered by specific dialects, the MLIR framework introduces an essential concept: dialect conversion. This process is employed in ONNX-MLIR as follows: (1) The initial dialect created in ONNX-MLIR is ONNX, which faithfully represents each operation within the ONNX model. Several high-level optimizations are applied to this dialect. (2) The ONNX dialect is subsequently transformed into the Krnl dialect, along with other dialects like arith. (3) Further optimizations are applied to the Krnl dialect before lowering it to the affine dialect. This conversion to the affine dialect enables a new set of optimizations. (4) The affine dialect is lowered to the LLVM dialect, which serves as the last step before generating the LLVM IR. (5) Finally, the LLVM IR is generated and further optimized to later generate a shared library that contains the optimized code generated for the entire machine-learning model.

For NMP, the ONNX-MLIR compiler works as follows (refer to Figure 5.5): Run calibration, where the CNN model represented in ONNX format is augmented with ReduceMin and ReduceMax nodes after each operation to capture the minimum and maximum values produced by each layer. The calibrated model is then executed over a set of pre-selected images using the ONNX runtime; Generate a new ONNX file from this step, preserving the original network structure but integrating the logs (minimum and maximum values) produced in the previous step into the header of the ONNX file to later use them in the compiler to quantize the network; Read the new ONNX file using the compiler's front-end, which is responsible for generating the first intermediate representation. In this first representation, all operations in the ONNX model are represented in the ONNX dialect; Perform a quantization pass using the minimum and maximum values obtained during the calibration step; Convert each operation to a new dialect called ONNX_quant; Run TSO (Tiling and Scheduling Optimization) to determine the tiling and scheduling configurations; Lower the ONNX_quant dialect to the NMP dialect, which includes operations supported by NMP. This step encompasses all the necessary optimizations and transformations to adapt the code for NMP, including the extraction of the Weight binary; Finally, lower the NMP dialect to the LLVM dialect, generate the LLVM IR, create a shared library that links with the NMP library, and extract the RISC-V binary containing the model code for execution.

5.2.3 Glow

The Glow compiler [69] is a machine learning compiler designed to generate highly optimized code for heterogeneous hardware architectures. It accomplishes this by trans-

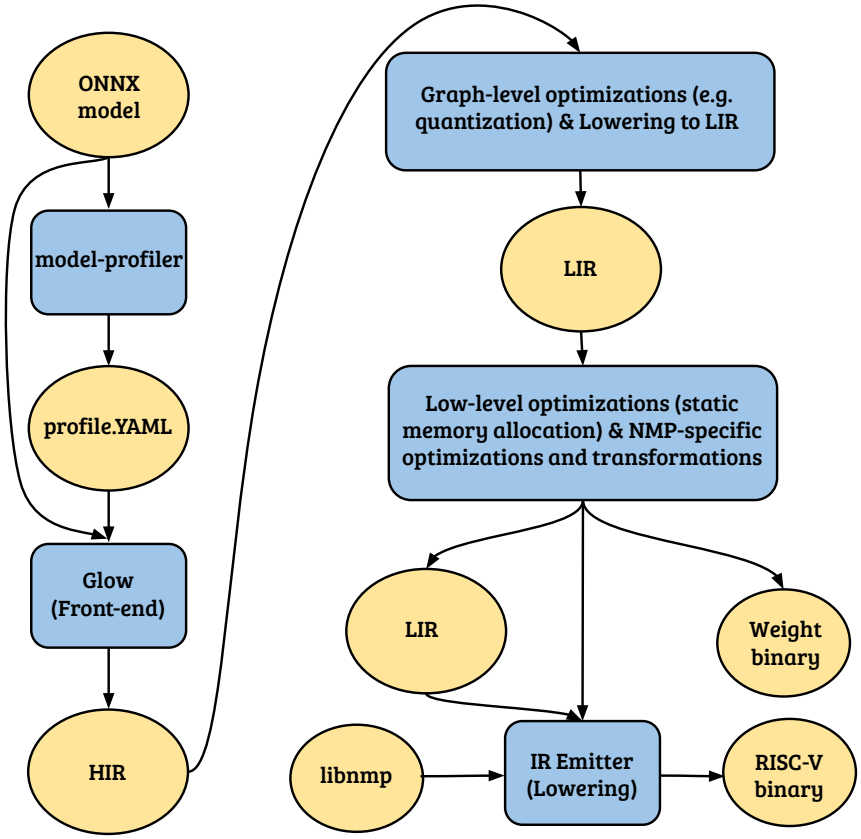

Figure 5.6: GLOW flow for NMP.

forming traditional ONNX models into a two-phase intermediate representation. Firstly, the High-level Intermediate Representation (HIR) applies graph-level optimizations such as loop fusion to reduce memory transactions. Subsequently, the Lower-level Intermediate Representation (LIR) employs memory-related optimizations like copy elimination, further enhancing performance. Once this set of optimizations is complete, the LIR representation is translated to LLVM IR, where additional optimizations are applied. Finally, the compiler generates an object containing the optimized model code, ready for execution on the target hardware.

The Glow compiler provides a flexible interface for retargeting to various architectures by facilitating the integration of new backends. This enables users to define custom operators and specify their lowering to LLVM, while also implementing architecture-specific transformations and optimizations. Through this process, users can guide high-level optimizations, including operator fusion as well as the supported operations. As part of this versatile approach, the NMP architecture has been successfully incorporated into the Glow compiler as a new backend.

The Glow compiler takes an ONNX model as input (refer to Figure 5.6), which undergoes a profile-guided quantization process. This process, along with a set of images, assesses the numeric ranges (minimum and maximum) for each layer through a tool called *model-profiler* using the *symmetric with power of 2 scale* scheme. The quantization tool

generates an output file in YAML format, which the compiler utilizes for network quantization. The quantization and other optimizations like layer fusion, occur at the HIR level. The HIR is subsequently lowered to the LIR format, where additional transformations and optimizations are applied, including operation scheduling, tensor memory allocations, and other NMP-related optimizations. Finally, the LIR is lowered to LLVM IR, which generates the RISC-V code for NMP.

5.2.4 Extending AI Compilers for NMP

The TSO algorithm has been deployed on the three presented compilers (TF-XLA, Glow, and ONNX-MLIR). To achieve this, we implemented an NMP backend on all three compilers, integrating the lowering part where the IR is lowered to RISC-V assembly and incorporating the optimizations explained in this chapter. One of these optimizations is the fusion of operations, which involves matching sequences of operations and merging them into a single operation. For each merged operation, a corresponding implementation in libnmp is required. Regarding the quantization pass, we have implemented it from scratch on TF-XLA and ONNX-MLIR, while on Glow, we have used the one already available in the compiler. All the other optimizations and required passes, such as weight binary file generation, TSO algorithm, global synchronization, etc., have been developed on all compilers. Regarding the libnmp library, we added operations to support certain layers and used others that had been added by LG before we started working on NMP. Concerning the runtime libraries – those used to load the model, preprocess the input, communicate with the board, and validate the outputs, among other tasks – we implemented support for all CNN models. Finally, with respect to debugging, since NMP does not provide users with any tools for debugging, we achieved this by dumping the memory, as we know where data should be written to and read from. By doing that, we could analyze the output produced by each layer for correctness, etc.

Chapter 6

NMP Mapping Strategies

During the execution of a Conv-Layer, various mapping strategies exist for transferring data from the host memory (DRAM) to the NPU on-chip memory. However, improper execution of these data movements can result in significant increases in performance degradation and energy consumption [80]. To tackle this problem specifically for multicore NPUs, this thesis introduces an optimization algorithm called *Tensor Slicing Optimization* (TSO), which explores a search space to identify the most efficient TLE/TLT data partitioning/scheduling strategy, primarily aiming to minimize the number of memory transfers needed during the execution of Conv-Layers. Additionally, TSO aims to maximize parallelism across the multiple cores that comprise the NPU.

6.1 TSO Algorithm

When mapping a convolution to NMP, a vast range of potential mapping solutions exists within the tiling/scheduling search space. Each mapping solution leads to distinct memory access patterns, parallelism across the TLEs (and consequently across the TLTs), and varying memory footprints. The choice of a mapping solution significantly impacts the performance, efficiency, and resource utilization to compute a convolution on NMP. Therefore, careful consideration and analysis are crucial to select an efficient mapping solution that aligns with the specific requirements and constraints of the target device (NMP). For this purpose, the TSO algorithm plays an important role in determining the mapping solution for each convolution, and this is done at compile-time as a compiler pass in the ML compiler, thus ensuring efficient decision-making for the mapping process.

TSO works by exhaustively exploring the solution space in the search for the best convolution tiling/scheduling strategy that minimizes execution time. It first slices the input tensor of the convolution (`IN`) and its corresponding filters (`KS`) among the TLE processors of the NMP so that each TLE computes a different slice of the convolution's output (`OUT`). After that, each TLE slice is further partitioned into multiple tiles so as to distribute the computation among the TLT cores of the corresponding TLE processor. These two steps of the TSO algorithm are detailed below. Before moving further, please consider that every mention to *slice* refers to a TLE data partitioning and every mention to *tile* refers to a TLT partitioning.

Algorithm 1 Select best TLE/TLT mapping

```
1:  function TSO(CONVS, #TLE, #TLT)
2:      #pragma omp parallel
3:      #pragma omp single
4:      for each conv ∈ CONVS do
5:          #pragma omp task
6:              ▷ Let conv = (IN, KS, OUT)
7:              map[conv].bestTile.time ← ∞
8:              ▷ Let PART_TLE = {KS, KS&OUT, OUT}
9:              for each p ∈ PART_TLE do
10:                 ▷ Let slice = (TLE_R, TLE_W)
11:                 slice ← TLESlicing(p, conv, #TLE)
12:                 ▷ Let PART_TLT = {IS, OS, WS}
13:                 for each q ∈ PART_TLT do
14:                     ▷ Let tile = (IN^T, KS^T, OUT^T, time, schedule)
15:                     tile ← TLTTiling(q, conv, slice, #TLT)
16:                     if tile.time < map[conv].bestTile.time then
17:                         map[conv].bestSlice ← slice
18:                         map[conv].bestTile ← tile
19:     return map
```

Initially (refer to Algorithm 1), TSO takes as input the set of convolutions of the model ($CONVS$) and the number of TLEs ($\#TLE$) and TLTs ($\#TLT$) of the architecture (line 1). It then iterates over all convolutions (line 4) and initializes a *map* which stores the best TLE slice and TLT tile for that specific convolution (lines 7). Then for all possible TLE slicing schemes p available in $PART_{TLE}$ (line 9, see Section 6.2 for details), the algorithm uses a call to function $TLESlicing$ to divide the convolution `IN` and `KS` tensors data across the TLEs. $TLESlicing$ returns tuple $slice = (TLE_R, TLE_W)$, where TLE_R refers to the part of the `OUT` (rows) that is generated from the slice of the `IN` designated to the TLE processor, and TLE_W a subset of the `KS` filters that will run on that TLE processor.

Remember that each TLE processor in NMP has a set of TLT cores, and thus for each TLE slice produced in line 11, the slice data needs to be divided among its corresponding TLT cores. Hence, for each TLT scheduling strategy q (line 13, see Section 6.3 for details), TSO computes the best TLT tile for the current TLE data slice using a call to $TLTTiling$ (line 15). This function takes as input the TLT scheduling strategy q, the convolution data (*conv*), the current TLE *slice*, and the number of TLTs ($\#TLT$). It then determines the best tiling of the TLE data among the TLT cores. The $TLTTiling$ function returns tuple $tile = (IN^T, KS^T, OUT^T, time, schedule)$, where $\texttt{IN}^T$, $\texttt{OUT}^T$ are the tiles of the `IN` and `OUT` tensors assigned to the TLTs of that TLE, and $\texttt{KS}^T$ is a tile that contains a subset

of the filter in KS.

The tuple also returns an estimate of the time taken to compute the convolution using that specific combination of TLE slice and TLT tile for the best possible scheduling (*schedule*) strategy (see Section 6.5 for details). To achieve that, it takes into consideration the cost to load the $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles from DRAM into the (on-chip) TLT memories MB0 (IN) and MB1 (KS), respectively, and the time to store the $\texttt{OUT}^T$ from the MB2 TLT (OUT) memory back to the host DRAM. Moreover, *time* also includes the time each evaluated partitioning takes to run on the MAC Unit using the various scheduling alternatives (see Section 6.4 for details).

After returning from $TLTTiling$, TSO compares (line 16) the estimated *time* for the evaluated partitioning with the best time ($map[conv].bestTile.time$) found so far for that specific convolution. It then stores it into the appropriate *map* entry (i.e., $map[conv]$), the corresponding TLE slice (line 17), and TLT tile (line 18). Finally, the *map* containing the best slices/tiles for each convolution is then returned (line 19), so it can be used later by the code generator to synthesize and schedule the code for the TLT cores.

TSO is an algorithm that exhaustively explores the solution space of all possible tensor-slicing solutions for each Convolution of a model. As such, it may take a long time to execute, particularly when the CNN model has many Convolutions. Given that estimating the execution time of a Convolution is independent of the others, the process of exploring the solution space is highly parallel. In this work, we use OpenMP task parallelism to accelerate this exploration by running the simulation of the execution time of all Convolutions in parallel. The parallel execution starts at line 2 with the creation of a thread pool. At this point of the execution, a unique thread is selected from the thread pool (line 3) to create a task for each Convolution (line 5). The tasks are then distributed across the threads within the thread pool to compute the Convolutions' TLE/TLT data partitioning and scheduling in parallel.

6.2 TLE Slicing

The first step in the TSO optimization involves dividing the filters in the KS tensor among the TLEs and determining the corresponding portion of the OUT tensor (rows) that the selected filters will produce at runtime. It is also determined which part of the IN tensor is required to generate this output. Since the slicing process is the first division performed over the IN, KS, and OUT tensors, improper division can significantly affect performance. For instance, assigning more data to the slices may increase communication, as more LOAD/STORE operations from/to DRAM, respectively, may be required. Hence, carefully optimizing the selection of proper slices for each TLE is essential to achieve load balance and minimize the data transfer time. In this work, we explore three TLE slicing schemes, as defined in Algorithm 1: $PART_{TLE} = KS, KS\&OUT, OUT$ (line 8). Even though the TLE slicing schemes used in this work may assign the same slices either from the IN or KS tensors to two or more TLE processors, the OUT slices produced by them do not overlap in any way. The Algorithm 2 shows how the division is performed in each of these TLE slicing schemes, and further details are provided below.

Algorithm 2 TLE Slicing

```
1: function TLESLICING(p, conv, #TLE)
2:     ▷ Get the number of rows of the OUT tensor
3:     rows ← conv.R
4:     ▷ Get the number of filters of the KS tensor
5:     filters ← conv.M
6:     if p = KS then
7:         TLE_W = ⌈filters/#TLE⌉
8:         TLE_R = rows
9:     if p = KS&OUT then
10:        TLE_W = ⌈filters/(#TLE/2)⌉
11:        TLE_R = ⌈rows/(#TLE/2)⌉
12:    if p = OUT then
13:        TLE_W = filters
14:        TLE_R = ⌈rows/#TLE⌉
15:    return (TLE_R, TLE_W)
```

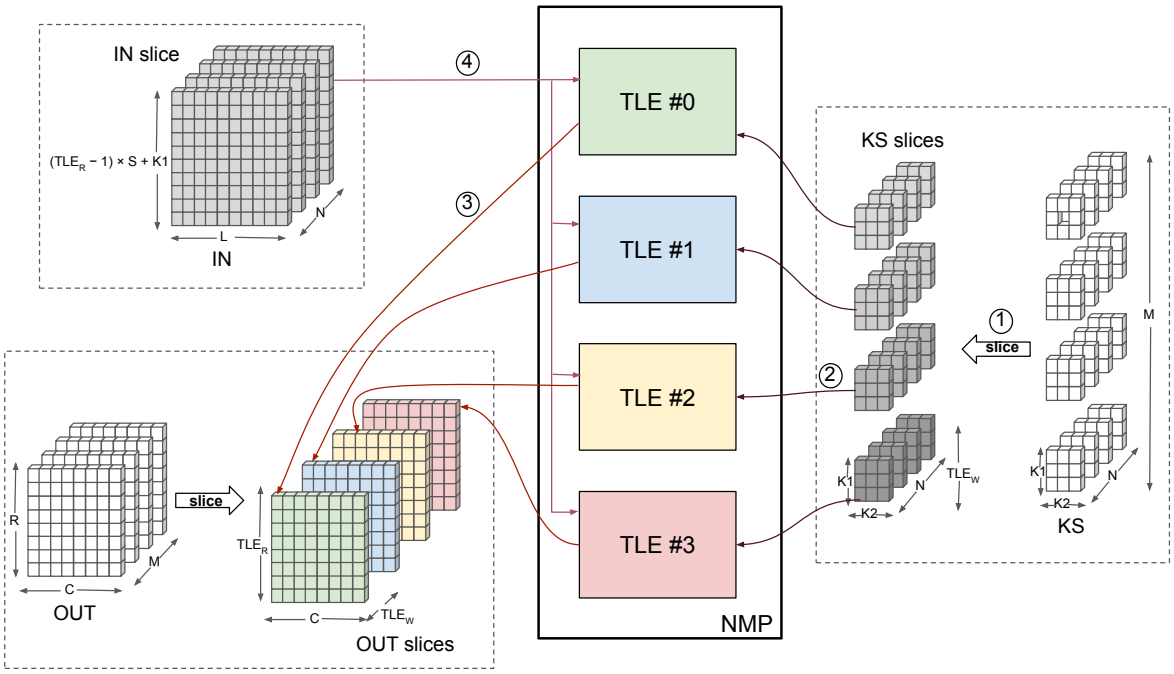

Figure 6.1: The TLE KS slicing scheme divides the filters in the KS tensor into slices, one for each TLE processor. A single IN slice is created and used by all TLE processors.

KS slicing scheme – In the first slicing scheme (line 6), only the convolution filters in KS are divided into slices among the TLEs (line 7), as illustrated in Figure 6.1. The number of slices generated from the KS tensor corresponds to the number of TLEs in the NMP accelerator. Figure 6.1 illustrates the division of the KS tensor into slices ①

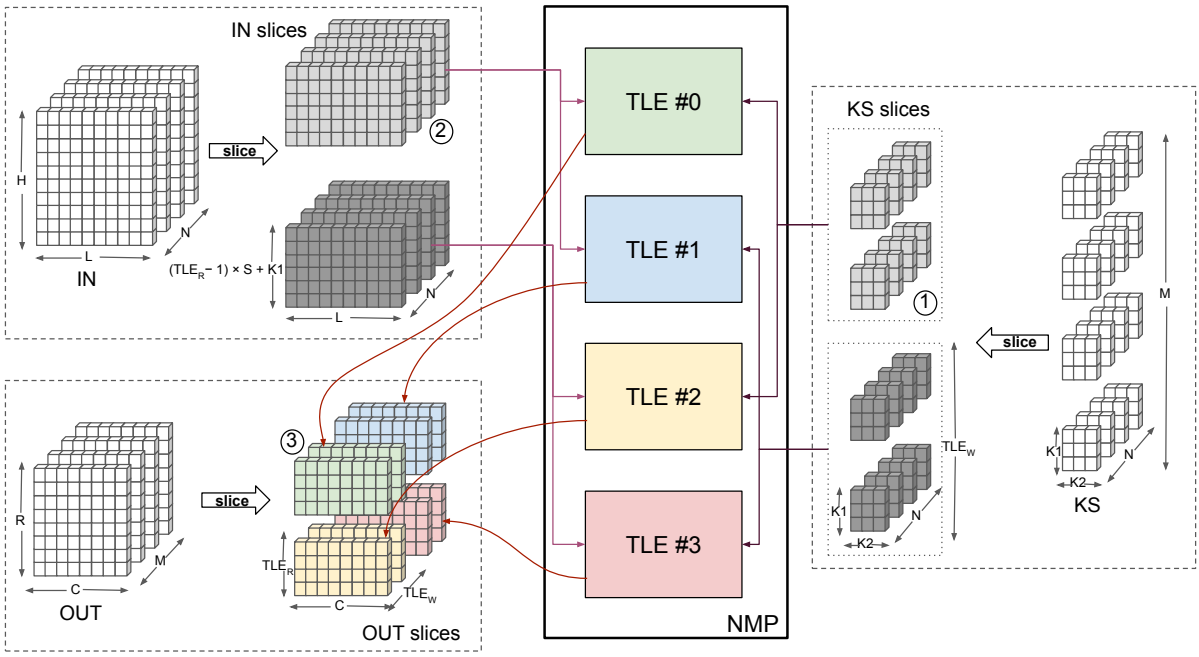

Figure 6.2: The TLE KS&OUT slicing scheme divides both the `IN` tensor and the filters in the `KS` tensor into slices. These slices are then combined to generate one `OUT` slice for each TLE processor.

and their corresponding TLE assignments (see in ② the assigned `KS` slice to TLE #2). Additionally, it also indicates the portion of the `OUT` tensor that each TLE is responsible for generating. For instance, in ③, it is shown the `OUT` slice that is produced by TLE #0. Regarding the `IN` tensor, only a single slice is produced, which is also the one consumed by all TLE processors ④. In terms of data replication, the entire output feature map, which includes the $R \times C$ elements of the `OUT` tensor, needs to be computed by the TLE, requiring the complete `IN` tensor to be loaded at runtime on each TLE processor. This slicing scheme usually works well in the last Conv-Layers of a CNN model due to the increased number of filters, their channel's depth, and the reduced spatial dimension of the `IN` tensors, which become less representative in terms of size. Thus, dividing the filters may reduce data transfers between the host memory (DRAM) and NMP on-chip memories.

KS and OUT (KS&OUT) slicing scheme – The second slicing scheme (line 9) involves dividing both the filters in `KS` and the rows of the `OUT` tensor among the TLEs. In our study, which utilizes an NMP with 4 TLEs, both the `IN` and `KS` tensors are divided into two sets. These divided portions (`IN` and `KS` slices) are then combined to generate 4 `OUT` slices, one for each TLE processor. In cases where an even distribution is not possible, the TLEs with lower IDs may handle a slightly larger workload. Figure 6.2 illustrates this process, featuring the created slices from the `KS` tensor (as illustrated in ① with the first #0 `KS` slice) and the `IN` tensor (highlighted in ② with the first #0 `IN` slice). It also depicts the portion of the `OUT` tensor (labeled as ③) that is generated by TLE processor #0 when computing the combined `IN` and `KS` slices #0. The two created slices from the `IN` tensor may have overlapping elements because their corresponding `OUT`

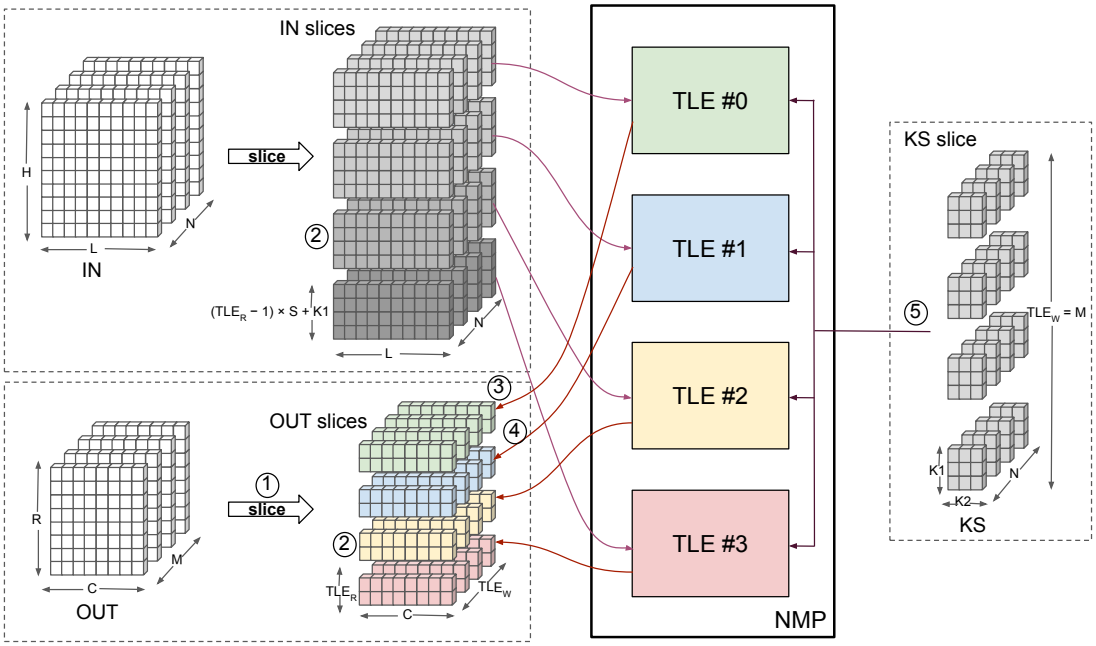

Figure 6.3: The TLE OUT slicing scheme divides the IN tensor into slices, one for each TLE processor. A single KS slice with all filters is created and used by all TLE processors.

slices may require the same input elements from the IN tensor to compute certain OUT elements. The overlap only occurs at the borders and is limited to $L \times (K1 - 1)$ elements. This overlap specifically arises when $K1 > 1$. This TLE slicing scheme reduces the data transfer over the IN tensor compared to the KS slicing scheme. However, it increases the load on the filters in KS since more filters are assigned to the slices. This scheme typically performs well when the KS and IN tensors have similar sizes, which is often the case for Conv-Layers positioned in the middle of a CNN model.

OUT slicing scheme – Finally, in the third slicing scheme (line 12), only the rows of the OUT tensor are divided among the TLEs (line 14). Figure 6.3 illustrates the slicing process of the OUT tensor ① into four distinct slices, each containing a different collection of elements from the OUT tensor. For each OUT slice, a corresponding IN slice is required (refer to the IN and OUT slices annotated with ② for an example). These IN slices may have overlapping elements depending on the filter size. For instance, in scenarios where $K1 > 1$, the first element of the last row in the first OUT slice (produced by TLE #0 – see ③) requires input elements that are also needed to compute the first element of the first row in the second OUT slice (produced by TLE #1 – see ④). This results in overlapping elements in the IN slices of both TLEs, where certain elements from the IN tensor are demanded by both TLEs at runtime. The number of overlapping elements in the first and last IN slices is $L \times (K1 - 1)$. Moreover, the second and third IN slices have an overlap of $2 \times L \times (K1 - 1)$ input elements with their neighboring slices. On the other hand, for $K1 = 1$, no overlapping elements are applied. For this scheme, all the filters in the KS tensor are incorporated into a single KS slice, which is required by each TLE ⑤. This leads to a scenario where the filters are loaded multiple times from the DRAM, as many times as the number of TLE processors. As a result, this slicing scheme is typically more

Algorithm 3 TLT Tiling

```
1: function TLTTILING(q,conv,slice,#TLT)
2:     ▷ Let tile = (IN^T, KS^T, OUT^T, time, schedule)
3:     bestTile.time ← ∞
4:     for T_R ← 1 to slice.TLE_R do
5:         for T_C ← 1 to conv.C do
6:             for T_N ← 1 to conv.N do
7:                 T_M ← GETFILTERS(T_R, T_C, q, slice.TLE_W, #TLT)
8:                 KS^T ← GENTILE_KS(T_M, T_N, conv, q)
9:                 OUT^T ← GENTILE_OUT(T_M, T_R, T_C, q)
10:                IN^T ← GENTILE_IN(T_N, T_R, T_C, q)
11:                (time, schedule) ← CALCTIME(IN^T, KS^T, OUT^T, q)
12:                if time < bestTile.time then
13:                    bestTile ← (IN^T, KS^T, OUT^T, time, schedule)
14:    return bestTile
```

effective for the first Conv-Layers, where the IN tensors are significantly larger compared to the filters in the KS tensors.

Since the NMP board used to collect the experiments for this thesis does not have a global shared buffer (shared among the TLEs), we have not considered this feature in designing TSO. However, TSO can be easily extended to consider a global shared buffer since different slices of the IN/KS from different TLEs may be the same.

6.3 TLT Partitioning

After choosing a TLE slicing scheme, the workload of each TLE is divided among their corresponding TLTs by means of a call to function *TLTTiling* in line 15 of Algorithm 1. *TLTTiling* takes as input the TLT scheduling strategy (q), the convolution data (*conv*), the TLE slice (*slice*) resulting in line 11 of Algorithm 1 and the number of TLTs at each TLE ($\#TLT$). It then produces as output the tuple ($IN^T, KS^T, OUT^T, time, schedule$), which will be used to generate code for the TLTs.

Initially (refer to Algorithm 3), *TLTTiling* initializes variable *bestTile.time* with infinity as it will store the shortest (estimated) execution time of all possible tiles visited by the function. To achieve that, a sequence of three nested loops (lines 4-6) generates the values T_R, T_C, and T_N that are used to explore all possible $\texttt{IN}^T$, $\texttt{KS}^T$ and $\texttt{OUT}^T$ tiles shapes that can be formed from a TLE slice. But before computing the $\texttt{IN}^T$ and $\texttt{OUT}^T$ tiles for that TLE slice, the convolution filters in TLE_W need to be divided among the various TLTs. This is done in line 7, which also determines the maximum number of filters (T_M) that can fit into the MB1 (KS) memory of a TLT, and in line 8, which generates the corresponding $\texttt{KS}^T$ tile. In the case of an unbalanced filter partitioning, the remaining filters are spread among the TLTs, which have the lowest IDs. This is followed by calling

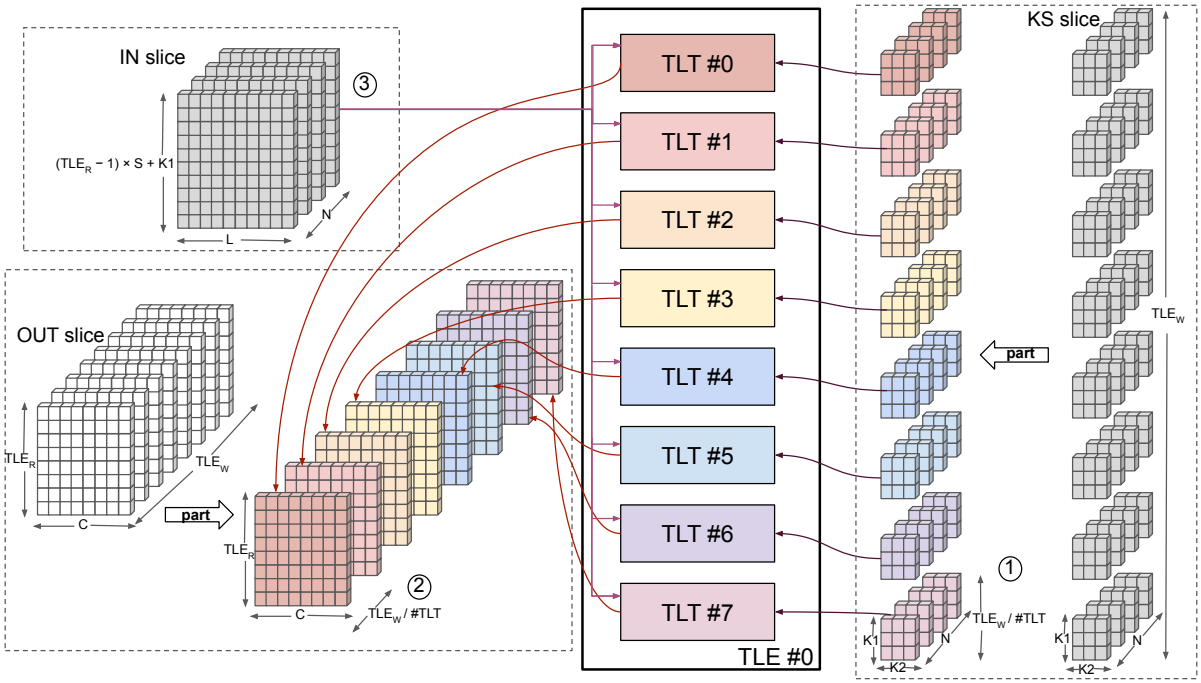

Figure 6.4: The TLT Partitioning involves distributing the filters in a TLE slice across the TLTs of the TLE. Additionally, a single input slice is formed and is necessary for all TLTs. To reduce load time, multicast loads are employed when loading the input.

functions to generate the $\texttt{OUT}^T$ tile ($GenTILE_{OUT}$ in line 9) and $\texttt{IN}^T$ tile ($GenTILE_{IN}$ in line 10). These two functions also check if the tiles $\texttt{OUT}^T$ and $\texttt{IN}^T$ respectively fit into memories MB2 (OUT) and MB0 (IN) of a TLT, as shown in Equation 6.1, where type stands for either 8- or 16-bit fixed-point. The functions between lines 8-10 ($GenTILE_{KS}$, $GenTILE_{OUT}$ and $GenTILE_{IN}$) also calculate the number of times the $\texttt{IN}^T$, $\texttt{OUT}^T$ and $\texttt{KS}^T$ tiles have to be loaded/stored from/to the DRAM to cover all the workload of a TLE slice, and this is done based on the selected scheduling technique, denoted as q in Algorithm 3 (more details in Section 6.4).

$$\begin{cases} |IN^T| = T_N \times T_H \times T_L \times type \leq |MB0| \\ |KS^T| = T_M \times T_N \times K1 \times K2 \times type \leq |MB1| \\ |OUT^T| = T_M \times T_R \times T_C \times type \leq |MB2| \end{cases} \tag{6.1}$$

To illustrate how the IN, KS, and OUT slices of a given TLE are divided across its TLTs, please refer to Figure 6.4. For simplicity, assume that the number of filters in the KS tensor is a multiple of $\#TLT$. During the partitioning process, the filters in the KS slice of a TLE are evenly distributed across its TLT cores, resulting in a total of $TLE_W/\#TLT$ filters and $(TLE_W/\#TLT) \times N \times K1 \times K2$ elements assigned to each TLT core individually ①. Consequently, the OUT slice is also partitioned across the TLT cores, resulting in a total of $(TLE_W/\#TLT) \times TLE_R \times C$ elements for each TLT core ②. On the other hand, the IN slice is not partitioned. Instead, each TLT loads the entire

$N \times ((TLE_R - 1) \times S + K1) \times L$ elements from the IN slice ③. However, multicast loads are used during runtime to optimize the loading of the $\texttt{IN}^T$ tiles from DRAM, where only a single TLT loads the data and distributes it to the other TLTs within the same TLE. Although the TLTs work independently of each other, in order to use multicast loads, they need to be synchronized during the load operations to ensure that they all receive the same $\texttt{IN}^T$ tile data. After slicing the convolution data into the TLEs and partitioning it across the TLTs, it is not always guaranteed that the workload for each TLT will fit within their on-chip memories (MBs). To address this, tiling is employed to ensure that a single $\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$ tiles can fit simultaneously in their respective MBLOBs. By tiling the workload assigned to a TLT, multiple $\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$ tiles may be created, posing a challenge in scheduling these tiles for execution at runtime (refer to Section 6.4 for more details). Additionally, the selection of tile sizes plays a crucial role in optimizing performance, given that choosing larger tiles can significantly increase reuse and reduce data transfers.

The $\texttt{IN}^T$, $\texttt{KS}^T$, $\texttt{OUT}^T$ tiles and the tiling strategy q, are then passed to function *CalcTime* (line 11), so it can estimate the best *schedule* and *time* to compute the TLE slices using the generated tiles (more details in Section 6.5). Finally, the algorithm tests if the *time* computed for the current tiling is smaller than the $bestTile.time$ seen so far, and if so, it updates the $bestTile$.

The NMP architecture enables other partitioning strategies beyond the ones used in this thesis, which leverages a MULTICAST instruction to load IN slices into 8 TLTs/TLE in parallel. For example, one could consider an approach that divides the IN tensor (channels) among the TLEs so that they compute partial sums of the same $\texttt{OUT}^T$ tile, which are later reduced to the final $\texttt{OUT}^T$ tile result. Unfortunately, in NMP, the process of reducing partial sums would require many ring-network messages between the TLEs, thus impacting performance and making some TLTs idle while others use a tree-reduction algorithm to accumulate the $\texttt{OUT}^T$ tile result. Moreover, besides leveraging MULTICAST loads to reduce data transfers, the partitioning strategies proposed herein also guarantee load-balancing between the TLEs/TLT.

6.4 Scheduling

With the data divided among the TLEs/TLTs, the next step of the TSO algorithm is to find the best order in which input/filter tiles are read from DRAM onto on-chip memory to perform tiled convolution. Depending on the memory access pattern that tiles are read from memory (*schedule*), more or less DRAM reload operations can occur, thus impacting the final convolution performance. There are basically three different tile scheduling strategies that TLT cores can use, which depend on the shape and size of the convolution input data – Input Stationary (IS), Output Stationary (OS), and Weight Stationary (WS). Given that the data transfers of the scheduling strategies presented herein can be determined at compile-time, TSO computes the number of accesses to the DRAM required by each one of them according to their data-flow patterns so as to determine the one that can result in the best data reuse. The scheduling strategies are described in detail below.

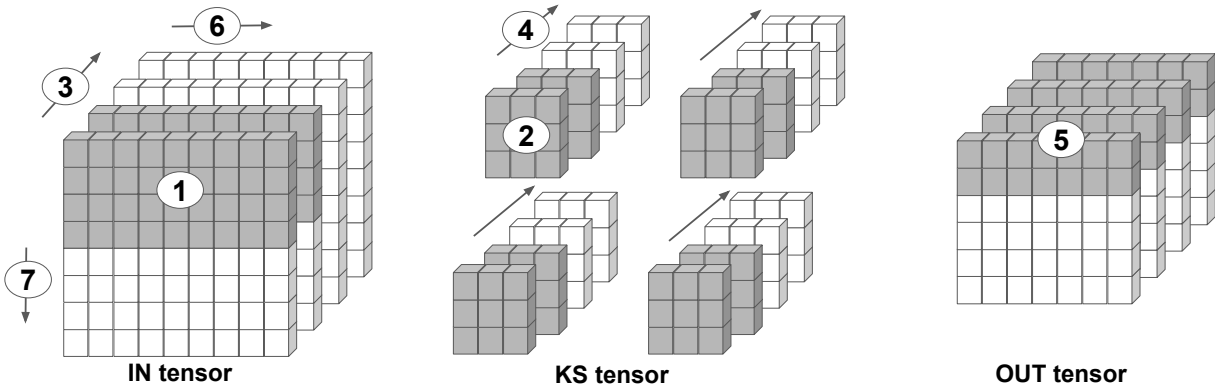

Figure 6.5: Input Stationary.

Input Stationary (IS) – is a scheduling strategy that focuses on reusing the $\texttt{IN}^T$ tiles. Figure 6.5 shows the execution flow of IS. The first step (① in Figure 6.5) is to load the $\texttt{IN}^T$ tile from the DRAM into the NMP MB0 on-chip memory; then, the $\texttt{KS}^T$ tile is also loaded ② from the DRAM into MB1. To make full reuse of the $\texttt{IN}^T$ tile, the $\texttt{KS}^T$ tile has to include all the filters designated to the TLT – even if just a small part of each one of them. With the $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles already loaded, the MAC Unit executes the convolution on them. The result is stored into the MB2 (OUT) memory, which, at this point, only contains a partial sum of the Convolution – the final result of the $\texttt{OUT}^T$ tile is only generated after computing all elements through the depth of the IN tensor. To do that, multiple $\texttt{IN}^T$ ③ and $\texttt{KS}^T$ tiles ④ may be required to be loaded while going through the channel (depth) direction. After computing and accumulating the results, the $\texttt{OUT}^T$ tile is ready to be stored into the DRAM ⑤. After that, a new $\texttt{IN}^T$ tile is loaded, going first on the width ⑥ and then on the height ⑦ directions of the IN tensor – for each one of them, the same $\texttt{KS}^T$ tiles are reloaded again and again from DRAM.

Given the access pattern performed by IS when loading/storing data from/to the DRAM, one can use Equation 6.2 to determine, for each tile ($\texttt{IN}^T$, $\texttt{KS}^T$ and $\texttt{OUT}^T$ tile), the number of times it is required to load/store to cover the entire computation of a Conv-layer over the TLEs/TLTs. The α_{in} and α_{ks} symbols denote the number of times the TLEs/TLTs have to load the $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles from the DRAM to compute an entire Conv-layer. The α_{out} stands for the number of times the $\texttt{OUT}^T$ tiles are stored into DRAM.

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \dfrac{TLE_R}{T_R} \right\rceil \times \left\lceil \dfrac{C}{T_C} \right\rceil \times \left\lceil \dfrac{N}{T_N} \right\rceil \\ \alpha_{ks} = \#TLEs \times \left\lceil \dfrac{TLE_R}{T_R} \right\rceil \times \left\lceil \dfrac{C}{T_C} \right\rceil \times \left\lceil \dfrac{N}{T_N} \right\rceil \\ \alpha_{out} = \left\lceil \dfrac{R}{T_R} \right\rceil \times \left\lceil \dfrac{C}{T_C} \right\rceil \end{cases} \tag{6.2}$$

Output Stationary (OS) – is a scheduling strategy that prioritizes the generation of the $\texttt{OUT}^T$ tiles, no matter if the same $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles have to be loaded multiple

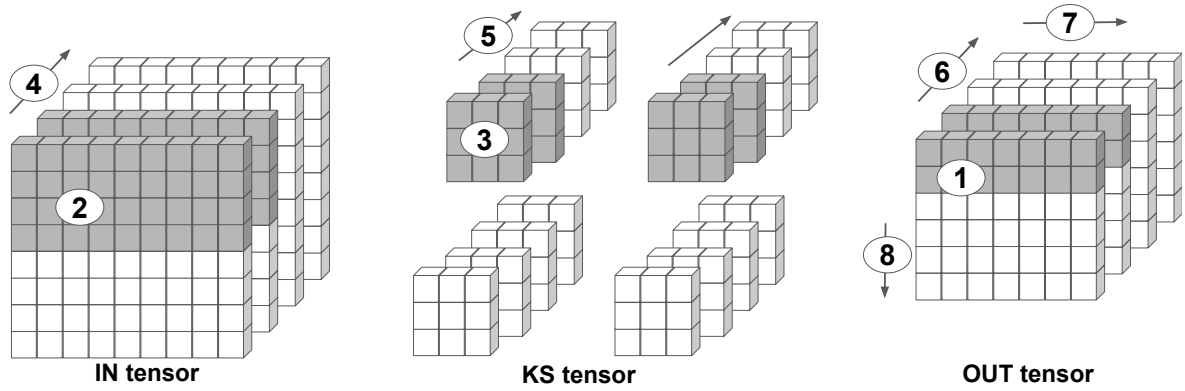

Figure 6.6: Output Stationary.

times from the DRAM into their respective on-chip memories. Figure 6.6 shows the execution flow of OS. First, based on the $\texttt{OUT}^T$ tile ① dimensions, the corresponding $\texttt{IN}^T$ ② and $\texttt{KS}^T$ ③ tiles are loaded from the DRAM into their respective on-chip memories to compute their convolution using the TLT's MAC Unit. Given that typically, the on-chip memories have not enough space to accommodate all the required input data, multiple $\texttt{IN}^T$ ④ and $\texttt{KS}^T$ ⑤ tiles have to be loaded using the channel (depth) direction. After finishing the computation of an $\texttt{OUT}^T$ tile, it is stored into the DRAM, and a new $\texttt{OUT}^T$ tile starts to be computed using the channels' (depth) direction ⑥. After that, the other $\texttt{OUT}^T$ tiles are computed by following first the OUT width dimension (C) ⑦ and then its height dimension (R) ⑧. The number of times the $\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$ tile have to be loaded/stored from/to the DRAM is defined by Equation 6.3.

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{ks} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{out} = \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{M}{T_M} \right\rceil \end{cases} \tag{6.3}$$

Weight Stationary (WS) – is a scheduling strategy that focuses on loading the filters from the DRAM only once and reusing them as the convolution tiles are computed. Figure 6.7 shows the execution flow of WS. First, the T_M filters in $\texttt{KS}^T$ tile are loaded from the DRAM into the MB1 on-chip memory ①. In this strategy, each loaded filter includes all its N channels. The loaded filters are then reused until the resulting output feature maps in OUT tensor are computed ②. Prior to executing the nmp_conv2d instruction, an $\texttt{IN}^T$ tile is loaded from the DRAM ③. Multiples loads of an $\texttt{IN}^T$ tile along the channels' depth may be required ④, each computing and storing partial results that will later be accumulated to form the final $\texttt{OUT}^T$ tile ⑤, so it can be stored back to the DRAM. This is followed by loading other $\texttt{IN}^T$ tiles in sequence over the width ⑥ and then over the height ⑦. This proceeds until all the output feature maps (dimension M in OUT tensor)

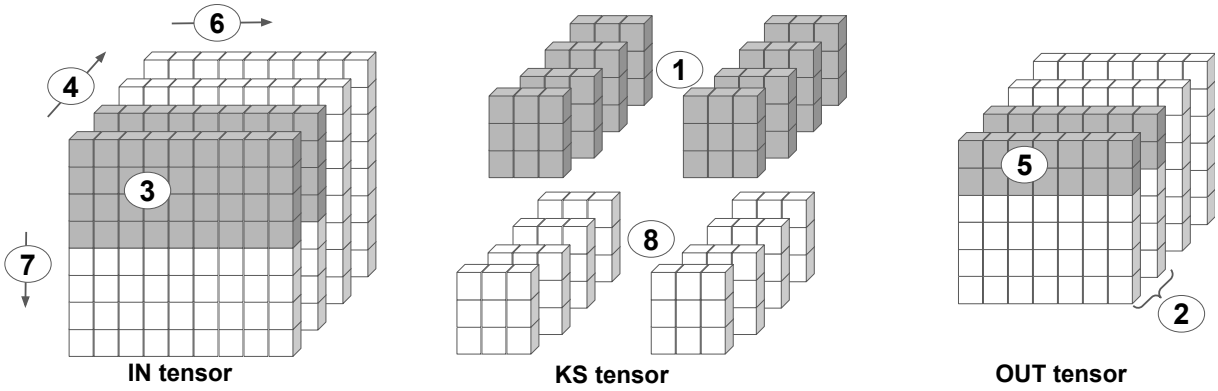

Figure 6.7: Weight Stationary.

of the respective filters in $\texttt{KS}^T$ tile are computed. After that, a new $\texttt{KS}^T$ tile with other filters may be required to be loaded ⑧ to compute their corresponding output feature maps – at this point, for each iteration, the same $\texttt{IN}^T$ tiles are again loaded. Equation 6.4 defines the number of required data transfers to/from the DRAM to cover the entire Conv-layer.

$$\begin{cases} \alpha_{in} = \#TLEs \times \left\lceil \frac{TLE_R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{ks} = \#TLEs \times \left\lceil \frac{TLE_W}{T_M} \right\rceil \\ \alpha_{out} = \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times \left\lceil \frac{M}{T_M} \right\rceil \end{cases} \tag{6.4}$$

Note that increasing the dimensions of tiling (T_R, T_C, and T_N) improves data reuse, thus reducing the need to load the same tiles from DRAM repeatedly. The TLT partitioning (Algorithm 3) generates multiple tile shapes, which are combined with scheduling strategies (IS, WS, and OS) and TLE slicing (KS, KS&OUT, and OUT). The combination that minimizes α_{in}, α_{ks}, and α_{out} is considered a potential solution. It is also important to mention that the IS and WS strategies prioritize the computation of the $\texttt{OUT}^T$ tiles, thus preventing the need for reloading them. Consequently, the next $\texttt{OUT}^T$ computation begins only after the previous one is fully computed, which includes the calculation of all partial sums.

6.5 Estimating Time

In order to decide, for each convolution, which slicing scheme is the best among those discussed in the sections above, TSO combines multiple solutions from the search space $<PART_{TLE}, PART_{TLT}, T_M, T_N, T_R, T_C>$, and estimates the time taken by each valid combination to select the one which provides the best performance. This estimate has the

following components, listed in increasing order of their contribution to the total convolution execution time: (a) the time required to run the RISC-V instructions at each TLT; (b) the time needed to perform the MAC unit operations on the slices; and (c) the time required to load/store data between the DRAM and the NPU on-chip memories. An estimate for the convolution execution time is calculated by the function *CalcTime* (defined in Algorithm 3 – line 11), which sums the time of each component of the execution according to Equation 6.5.

$$T_{CONV} = T_{MAC} + T_{DRAM} + T_{SW} \tag{6.5}$$

where T_{MAC}, T_{DRAM} and T_{SW} stands for the time taken by the MAC Unit, the time taken to transfer data between the NPU's on-chip memories and DRAM, and the time taken to execute the RISC-V instructions, respectively. Since T_{SW} is not significant (usually less than 5% of the total execution), we will not cover it in detail in this thesis. However, a brief discussion on this topic is available in Section 7.1.

The time the convolution spends computing the MAC operation (T_{MAC}) is calculated according to the number of Multiply-accumulate operations (MAC operations) of a Conv-layer. The first step to estimate it (see Equation 6.6) is to determine the number of MAC operations required to compute a single channel of the `IN` tile. This is then divided by the number of MAC operations that a MAC unit can execute at each cycle. After that, the other input feature maps (T_N) and the output feature maps (T_M) are then considered to compose the estimated time of the entire tile ($Tile_{MAC}$). Once the time taken by a single tile is determined, it is then possible to estimate the total time required to compute the entire Conv-layer's workload, which is distributed over all the TLTs cores ($\#TLEs \times \#TLTs$) in NMP (see Equation 6.7).

$$Tile_{MAC} = T_N \times T_M \times \left(\left\lceil \frac{T_R \times T_C \times K1 \times K2}{\#MACs} \right\rceil \right) \times \frac{1}{Freq} \tag{6.6}$$

$$T_{MAC} = \frac{1}{\#TLTs} \times \left\lceil \frac{M}{T_M} \right\rceil \times \left\lceil \frac{N}{T_N} \right\rceil \times \left\lceil \frac{R}{T_R} \right\rceil \times \left\lceil \frac{C}{T_C} \right\rceil \times Tile_{MAC} \tag{6.7}$$

TSO uses two approaches to estimate the time taken to load/store the $\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$ tiles between the on-chip memories and DRAM. They are: (a) *TSO-burst*, a burst-based model that estimates the tile's transfer time based on the number of memory bursts, which is used to estimate the DRAM CAS (Column Address Strobe) latency. Additionally, the burst-based model also estimates the DRAM transfer time; and (b) *TSO-noburst*, a data volume-based estimate, which determines the DRAM transfer time solely based on the size of the tile's data and the memory bandwidth. Algorithm 4 describes how the DRAM transfer time is computed using both approaches. For the sake of explanation of Algorithm 4 assume a DDR3 memory with CAS latency (us), BW bandwidth (Bytes/sec), and $BURST$ size in bytes (e.g., 128B). For simplicity, TSO assumes that each transaction (i.e. LOAD/STORE) from/to DRAM is processed one at a time. In addition, another data volume-based approach, *TSO-roof*, which estimates

the solutions in the TSO's search space using a cost model based on the Roofline model and prioritizes solutions that enhance data reuse, was adapted from [57] to work with TSO. This approach, in particular, selects the solutions for each convolution based on the Operational Intensity (OI) and an estimate of the Attainable Performance using the estimated T_{MAC} timing.

To estimate the time taken by data transfers (T_{DRAM}) in the TSO-burst and TSO-noburst approaches, TSO needs to consider the number of memory accesses (i.e., reloads) required by each tile. These memory accesses include the $\texttt{IN}^T$ tile (α_{in}), the $\texttt{KS}^T$ tile (α_{ks}), and the $\texttt{OUT}^T$ tile (α_{out}). Then, TSO calculates the data transfer time for each tile using the function *CalcDataTransfer*, which is defined in Algorithm 4. This function takes as input the *tile*, which can be an $\texttt{IN}^T$, $\texttt{KS}^T$, or $\texttt{OUT}^T$ tile, and holds information about the tile shape, which is used to determine factors such as tile size and, in the case of TSO-burst, the number of bursts. Additionally, the function includes information about the convolution (*conv*), which encompasses the shapes of the `IN`, `KS`, and `OUT` tensors, which are used to calculate the number of bursts for TSO-burst. Lastly, it also contains architectural details (*arch*), such as memory bandwidth and latency, which are essential for estimating the time required for data transfers, along with the data type in use. Once the data transfer times for all tiles are calculated, they are accumulated into T_{DRAM} (see Equation 6.8), which represents the total data transfer time for the chosen solution within the search space.

$$\begin{aligned} T_{DRAM} = \alpha_{in} * CalcDataTransfer(IN^T, conv, arch) + \\ \alpha_{ks} * CalcDataTransfer(KS^T, conv, arch) + \\ \alpha_{out} * CalcDataTransfer(OUT^T, conv, arch) \end{aligned} \tag{6.8}$$

Burst-based data transfer (TSO-burst) – The key idea behind TSO-burst is to use the number of bursts taken by each access to a tile row so as to compute an estimate for the DRAM access time of the tile. For instance, consider an $\texttt{IN}^T$ tile containing T_N (channels) $\times$ T_H (rows) $\times$ T_L (columns) where each entry has 16-bit (2B) elements. Given that the channel is laid out in row-major, loading the first element of a row takes time CAS, while loading the remaining elements takes $\sim (2 \times (T_L - 1))/BW$. Thus, an IN row takes $CAS+ \sim (2 \times (T_L - 1))/BW$ to load. This is true if the size of the row ($2 \times T_L$) is smaller than $BURST$ bytes. Otherwise, other memory bursts may occur when loading the row, and additional CAS penalties will impact the time.

Algorithm 4 is used to estimate the execution time when convolution *conv* is divided into tiles *tile* on architecture *arch*. Initially, the tile data size *type* (e.g., 16-bit fixed-point) (line 2), the DRAM memory bandwidth BW (line 3), the CAS latency CAS (e.g., 14ns) (line 4), and the BURST size (e.g. 128B) (line 5) are extracted from the *arch* data structure. Next (line 6), the algorithm selects the memory transfer model (e.g., TSO-burst) and uses a call to function *CalBurstCount* (line 7) to determine the number of bursts (*nburts*) required to load all the T_H rows of an ($\texttt{IN}^T$) tile. Then, the impact of the CAS (Column Address Strobe) latency is computed into *cas_latency* (line 8), and the size of the tile (*tile_size*) is determined in line 9 by calling function *GetTileSize*. The

Algorithm 4 Estimate the time taken by Data Transfer

```
1:  function CalcDataTransfer(tile, conv, arch)
2:      type ← arch.type
3:      BW ← arch.BW
4:      CAS ← arch.CAS
5:      BURST ← arch.BURST
6:      if model = TSO-burst then
7:          nbursts ← CalBurstCount(tile, conv, type, BURST)
8:          cas_latency ← nbursts * CAS
9:          tile_size ← GetTileSize(tile, type)
10:         transfer_time ← tile_size/BW
11:         total_time ← transfer_time + cas_latency
12:     else
13:         ▷ TSO-noburst
14:         tile_size ← GetTileSize(tile, type)
15:         total_time ← tile_size/BW
16:     return total_time
```

time to transfer all the data in a tile ($transfer\_time$) is then determined (line 10), and finally, the total time to load the tile is computed (line 11) and returned (line 16).

For an illustrative example of how the burst count is calculated in the *CalBurstCount* function in Algorithm 4, refer to Figure 6.8 (a) – (c) and Equations 6.9 – 6.11. For the sake of simplicity, the burst count is only presented for the input tile ($\texttt{IN}^T$). However, the same process is also applicable to the other two tiles ($\texttt{KS}^T$ and $\texttt{OUT}^T$). In summary, three scenarios exist when calculating the number of bursts, and these scenarios depend on the tile shape. For the first scenario, as depicted in Figure 6.8 (a) and Equation 6.9, for every two consecutive rows (T_H) within the $\texttt{IN}^T$ tile, there exists a memory access stride of $L - T_L$ elements that separates them, where $L - T_L > 1$. Due to this memory access stride, it becomes necessary to individually load each row (T_H) from DRAM into the NPU's on-chip memory (MB0) through separate memory transactions. The number of memory transactions required to load each row (T_H) can be determined by dividing the sequential tile dimension (T_L) in memory by $BURST$, which specifies the number of consecutive elements that can be loaded from DRAM in a single memory access operation, and then multiplying the result by the other two dimensions (T_H and T_N). For the second scenario, as shown in Figure 6.8 (b) and Equation 6.10, the elements of all rows (T_H) within a single channel (T_N) are sequential in memory, since $T_L = L$. On the other hand, for every two channels (T_N) in the $\texttt{IN}^T$ tile, there exists a stride of $(H - T_H) \times L$ elements that separates them, where $(H - T_H) \times L > 1$. For this scenario, the two sequential dimensions are aggregated (T_L and T_H) prior to counting the number of memory transactions. Finally, for the third scenario, presented in Figure 6.8 (c) and Equation 6.11, all the $T_N \times T_H \times T_L$ elements in $\texttt{IN}^T$ are laid out sequentially in memory (i.e., there is no memory access

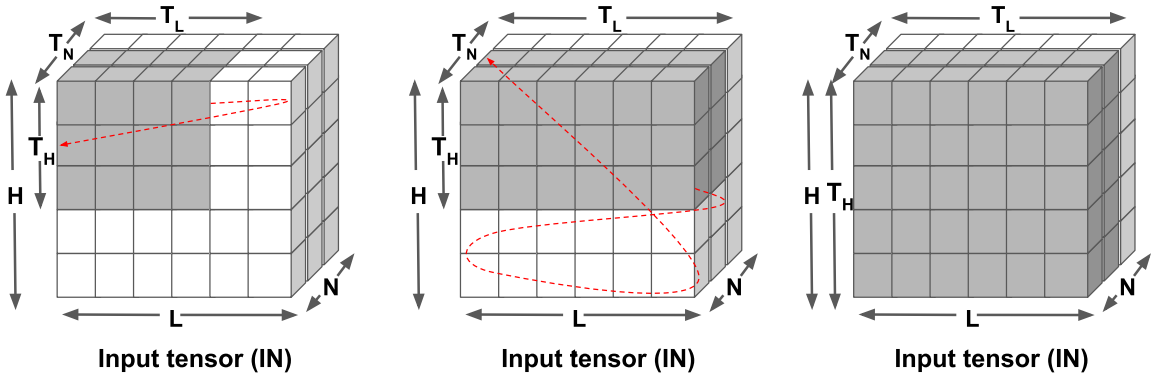

Figure 6.8: The three presented scenarios (a) - (c) demonstrate how the input tiles ($\texttt{IN}^T$) are represented in memory within the input tensor (IN), and the memory access stride applied to each of them (dashed red arrow). Note that the third scenario (c) has the elements of the $\texttt{IN}^T$ tile sequentially represented in memory.

stride). Therefore, all three dimensions are composed prior to counting the number of memory transactions. As a matter of comparison, the third scenario tends to outperform the other two scenarios. However, it is possible to achieve equivalence in the first two scenarios scenario (a) - (b) with respect to the third (c), in terms of performance, if the modulus of the sequential dimensions with the BURST size is zero or close to zero. Conversely, if the modulo is high, the memory bursts are underutilized, which may lead to more memory transactions, leading to performance degradation.

$$nbursts = \begin{cases} \left\lceil \frac{2 \times T_L}{BURST} \right\rceil \times T_H \times T_N, & if \;\; T_L \neq L \quad (6.9) \\ \left\lceil \frac{2 \times T_L \times T_H}{BURST} \right\rceil \times T_N, & if \;\; T_L = L \; and \; T_H \neq H \quad (6.10) \\ \left\lceil \frac{2 \times T_L \times T_H \times T_N}{BURST} \right\rceil, & if \;\; T_L = L \; and \; T_H = H \quad (6.11) \end{cases}$$

TSO-burst does not make any assumptions about the external DRAM or memory-controller designs besides the existence of burst-based accesses typically found in these memories. The memory controller found in the NMP board follows the ARM-bus protocol. Besides CAS latency, other DRAM parameters (e.g., Trcd/Trp/Tras) could also be included to improve the precision of data transfer modeling. Nevertheless, since CAS-latency is one of the most relevant of these DRAM parameters, Algorithm 4 focused only on it.

Volume-based data transfer (TSO-noburst) – This approach is typically used by all previous works which address this problem [30, 57, 81, 87]. As shown in lines 13-15

of Algorithm 4, it estimates the tile time by considering only the time to transfer the tile data (line 15) and not the impact of the CAS latency of the tile's memory bursts.

To compare TSO with an existing solution, we adapted the approach proposed in [57] to make it compatible with our ML compiler, TF-XLA, and also with the TSO's search space. See below for an extended description of how the adaption was done to our ML compiler.

Volume-based data transfer (TSO-roof) – Although the Roofline model [84] serves as a means to understand the balance between available hardware performance and the computational requirements of a specific application, allowing the user to identify bottlenecks and optimize them for peak performance. The Roofline model is employed in [57] as an analytical model to investigate a search space of tiling applied to FPGAs. This model estimates two metrics: (1) Attained Performance and (2) Operational Intensity for each solution in the search space, and then selects the one that maximizes both metrics, by first prioritizing (1) and then (2). To demonstrate how this compares to TSO, these two metrics were integrated into TSO's search space. The same TLE schemes, TLT partitioning, scheduling techniques, and the search space for tiling were embedded into the integrated solution, which is referred to as TSO-roof in this thesis. The integration process involved the adaption of the equations used to rank different solutions in the analytical model in [57] to work with TSO. First, in Equation 6.12, the number of operations performed by the convolution is calculated (note that the constant 2 indicates a multiplication followed by an accumulation). Comparatively, the same number of operations is used for both approaches (i.e., TSO and [57]). After that, the number of operations is used to estimate the Attained Performance (see Equation 6.13), which serves as a performance metric to evaluate the capacity of a solution for delivering high performance in OPS/s (operations per second). The estimated MAC timing used in [57] is equivalent to the one defined in this thesis in Equation 6.7 (T_{MAC}). Finally, the adaption of the Operational Intensity from [57] to TSO is shown in Equation 6.14. This metric plays a crucial role in measuring the number of operations executed per byte retrieved from the DRAM memory. Both the estimate of the Attained Performance and Operational Intensity are indicators of solution performance. Thus, a higher value in either of these metrics indicates a more optimal and efficient solution.

$$Number\ of\ Operations = 2 \times R \times C \times M \times N \times K1 \times K2 \tag{6.12}$$

$$Attained\ Performance = \frac{Number\ of\ Operations}{T_{MAC}} \tag{6.13}$$

$$Operational\ Intensity = \frac{Number\ of\ Operations}{\alpha_{in} \times |IN^T| + \alpha_{ks} \times |KS^T| + \alpha_{out} \times |OUT^T|} \tag{6.14}$$

Chapter 7

Main Experimental Results

This chapter presents the experimental results designed to address the three research questions (Q_1 – Q_3) outlined in Chapter 2. The results are provided, and the benefits of applying the TSO optimization are analytically demonstrated.

It begins (Section 7.1) by describing the testing environment, evaluating the impact of TSO on the ML models' accuracy, and providing a brief discussion of the percentage of the execution time taken to run the RISC-V instructions on NMP. This is followed (Section 7.2) by showing and analyzing the results of applying TSO to a set of representative convolutions, where it is detailed how the `IN`, `KS`, and `OUT` tensors are sliced among the TLEs, how these slices are partitioned among the TLTs, and finally, how these partitions are further tiled into small tiles ($\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$).

It then compares (Section 7.3) the performance of tiling when using the two TSO heuristics that estimate the time spent on DRAM access (i.e., *cost*): (1) TSO-burst, which incorporates the search space described in Section 6.5 along with a cost model based on burst analysis; and (2) TSO-noburst, which uses the same search space but applies a different cost model based on data-volume. The application of TSO-burst addresses the first research question (Q_1) presented in this thesis. A breakdown analysis, along with an end-to-end CNN model execution that compares both cases, is presented, as well as a convolution speedup analysis that shows the difference for each of the evaluated convolutions. In Section 7.4, two experiments are conducted: (1) an experiment that addresses the second research question (Q_2), where the different TLT scheduling strategies are optimally selected depending on the convolution operation and compared against fixed TLT scheduling strategies; and (2) an experiment that addresses the third research question (Q_3), where TSO selects the best TLE slicing scheme depending the convolution, and this is compared against a solution that fixes the same solution for all convolutions.

Unfortunately, not many NPUs are available in the literature for performance comparison, mainly due to intellectual property restrictions. Even when such NPUs are accessible, their architectural configurations differ in a way that makes it impossible to compare them against TSO on NMP (e.g. [40]). Given such restrictions, and in order to evaluate TSO effectiveness, an experiment was designed (Section 7.5) to compare TSO-burst performance against a solution adapted from [57], referred to as TSO-roof, which has an analytical model based on estimates derived from a Roofline model.

Model	Accuracy (TF-XLA)			
	NMP (%)		CPU FP (%)	
	Top-1	Top-5	Top-1	Top-5
InceptionV3	75.1	92	76.9	93.4
LeNet	99.9	100	99.9	100
MobileNetV2	70.2	89.6	70.5	89.8
ResNet-50	70.6	89.9	70.7	89.9
SqueezeNet	47.1	71	47.1	71
YOLO	-	-	-	-

Table 7.1: Model accuracy on CPU (FP32) and NMP (16-bit fixed point). YOLO uses a different metric for accuracy; it measures the precision of the detection, which is 93.53% on NMP while on CPU is 93.03%.

7.1 Experimental Setup and Accuracy Analysis

In order to validate the TSO approach, a set of experiments was executed on an NMP board equipped with 4 TLEs, each having 8 TLTs. Each TLT contains three 8KiB MB on-chip memories (MB0–MB2). Five CNN image classification models were used in the experiments: InceptionV3 [78], LeNet [47], MobileNetV2 [70], ResNet-50 [29], and SqueezeNet [32]. TSO was also evaluated on a real-world object detection application - a YOLO-based model [74] used to recognize car license plates. The selected models have a varied number of convolutions with different shapes of `IN` (input), `KS` (weight), and `OUT` (output) tensors. When measuring performance, the number of runs of each tested configuration was 5, with less than 0.5% observed standard deviation in all experiments. Beyond the mentioned CNN models, we have also attempted the VGG16 model [75]. However, due to the number of parameters exceeding 256MB even after quantization and surpassing the available DRAM memory space for the NMP, mapping this network for execution on NMP was not possible.

All models were compiled with a TSO-modified TF-XLA compiler using a quantization pass set to 16-bit fixed-point. The accuracy achieved by each image classification model on NMP is shown in Table 7.1. The Top-1 and Top-5 accuracies were measured by running all images from the validation datasets, MNIST [2] and ImageNet (ILSVRC2012) [1], for LeNet and the other image classification models, respectively. The same datasets were also used to measure the CPU's original floating-point models (FP 32-bit). The difference in terms of accuracy drop ranges from 0.1 up to 1.8%. For the YOLO-based model, NMP reaches a precision of detection of 93.53% on a car plate dataset [24] executed on 16-bit quantized data. The same model on CPU results in 93.03%.

The quantization scheme utilized in this thesis focuses on 16-bit fixed-point quantization for convolution data. However, when considering smaller precisions, such as 8-bit fixed-point, TSO may choose a different solution than that selected for the same convolution quantized to 16-bit fixed-point. For instance, if a given convolution is quantized to 8-bit and one of its components (e.g., $\texttt{KS}^T$) becomes smaller than its corresponding MBLOB, maintaining this component stationary would result in a single load of it and the other components' tiles ($\texttt{IN}^T$ and $\texttt{OUT}^T$) for the convolution computation. Conse-

quently, this would considerably reduce data transfer between DRAM and NMP. Hence, TSO would favor this solution due to its ability to reduce data transfer compared to other options significantly.

Regarding the time taken by RISC-V instructions (T_{SW}), a profile-guided analysis conducted for the TSO-burst heuristic, using the actual execution time on NMP, reveals that an average of 9.29% of the total execution time is dedicated solely to the execution of RISC-V instructions. For the LeNet model, the time taken to run the RISC-V instructions consumes 33.49% of the total time. This can be attributed to the small workload distributed to the TLTs for this CNN model, thus resulting in more time being spent on running RISC-V instructions. These instructions include loops iterating over the small tiles, handling semaphores, and executing branches to evaluate specific attributes for the given CNN operation, such as padding and stride presence. In the case of MobileNetV2, 10.52% of the total time is taken to run the RISC-V instructions. This high percentage is due to the fact that the convolutions in this model mainly consist of depthwise convolutions, which are simpler computationally compared to traditional convolutions. The RISC-V timings for the other CNN models are 4.61%, 3.23%, 2.36%, and 1.52% for the SqueezeNet, ResNet-50, InceptionV3, and YOLO models, respectively. It is important to highlight that these models require less time to execute the RISC-V instructions because the operations (e.g. convolution) in these CNN models are more computationally intensive, causing the NMP runtime execution to spend more time on other tasks such as LOAD/STORE and MAC operations.

Regarding the accuracy of the execution time estimates of the TSO-burst cost model compared to the actual execution time, we have compared the average percentage error of the most computationally intensive models. We observed errors of 13.3% for SqueezeNet, 13.8% for YOLO, 18.3% for InceptionV3, and 21.1% for ResNet50. These errors can be attributed to the fact that TSO does not account for many architectural features, such as the sequentialization of memory accesses in the DME unit, as a more precise cost model would be computationally much more expensive.

7.2 TSO Tiling Analysis and Results

To evaluate the effectiveness of the solutions chosen by TSO (with the burst heuristic activated), Table 7.2 shows the results of 11 different convolutions, each exhibiting distinct characteristics. The table illustrates the evolution of the `IN`, `KS`, and `OUT` tensors after undergoing several division steps between the TLEs and further between their corresponding TLTs. The table also includes information on TLE slicing, TLT partitioning, and scheduling, as well as $\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$ tiles, along with their corresponding shapes and sizes after each transformation.

For the TLE slicing selection, TSO chooses the TLE OUT slicing scheme for five of the convolutions, indicating that the combined size of the `IN` and `OUT` tensors of those convolutions is significantly bigger than that of the `KS` tensor. In the case of YOLO_C6, the combined size of the `IN` and `OUT` tensors is 5x bigger than the `KS` tensor, whereas, for YOLO_C1, these two tensors are 8027x larger than the `KS` tensor. Conversely, there are

Convolution	IN tensor KS tensor OUT tensor		TSO (with burst activated)							
			TLE Slicing			TLT Partition & Schedule			IN^T tile KS^T tile OUT^T tile	
	Shape	Size (KiB)	Scheme	Slice (per TLE)	Size (KiB)	Strategy	Partition (per TLT)	Size (KiB)	Shape	Size (KiB)
InceptionV3_C4	64x73x73 80x64x1x1 80x73x73	666.1 10.5 832.6	OUT	64x19x73 80x64x1x1 80x19x73	173.4 10.5 216.7	WS	64x19x73 10x64x1x1 10x19x73	173.4 1.2 27.1	32x1x73 10x64x1x1 10x1x73	4.5 1.2 1.4
InceptionV3_C39	768x17x17 192x768x1x1 192x17x17	433.5 288 108.3	KS&OUT	768x9x17 96x768x1x1 96x9x17	229.5 144 28.7	IS	768x9x17 12x768x1x1 12x9x17	229.5 18 3.5	64x3x17 12x64x1x1 12x3x17	6.37 1.5 1.2
InceptionV3_C91	448x8x8 384x448x3x3 384x8x8	56 3024 48	KS	448x8x8 96x448x3x3 96x8x8	56 756 12	IS	448x8x8 12x448x3x3 12x8x8	56 94.5 1.5	35x10x10 12x35x3x3 12x8x8	6.8 7.4 1.5
ResNet-50_C1	3x224x224 64x3x7x7 64x112x112	294 18.4 1568	OUT	3x56x224 64x3x7x7 64x28x112	73.5 18.4 392	WS	3x56x224 8x3x7x7 8x28x112	73.5 2.3 49	3x10x118 8x3x7x7 8x2x56	6.9 2.3 1.7
ResNet-50_C2	64x56x56 64x64x1x1 64x56x56	392 8 392	OUT	64x14x56 64x64x1x1 64x14x56	98 8 98	WS	64x14x56 8x64x1x1 8x14x56	98 1 12.2	32x2x56 8x64x1x1 8x2x56	7 1 1.7
ResNet-50_C23	128x28x28 128x128x3x3 128x14x14	196 288 49	KS&OUT	128x14x28 64x128x3x3 64x7x14	98 144 12.2	IS	128x14x28 8x128x3x3 8x7x14	98 18 1.5	7x16x30 8x7x3x3 8x7x14	6.6 1 1.5
ResNet-50_C27	512x14x14 1024x512x1x1 1024x14x14	196 1024 392	KS&OUT	512x7x14 512x512x1x1 512x7x14	98 512 98	OS	512x7x14 64x512x1x1 64x7x14	98 64 12.2	64x4x14 18x64x1x1 18x4x14	7 2.2 1.97
ResNet-50_C53	512x7x7 2048x512x1x1 2048x7x7	49 2048 196	KS	512x7x7 512x512x1x1 512x7x7	49 512 49	OS	512x7x7 64x512x1x1 64x7x7	49 64 6.1	64x7x7 20x64x1x1 20x7x7	6.1 2.5 1.9
YOLO_C1	3x416x416 16x3x3x3 16x416x416	1014 0.8 5408	OUT	3x104x416 16x3x3x3 16x104x416	253.5 0.8 1352	WS	3x104x416 2x3x3x3 2x104x416	253.5 0.1 169	2x11x122 2x3x3x3 2x9x120	5.2 0.1 4.22
YOLO_C5	1024x13x13 1024x1024x3x3 1024x13x13	338 18432 338	KS	1024x13x13 256x1024x3x3 256x13x13	338 4608 84.5	OS	1024x13x13 32x1024x3x3 32x13x13	338 576 10.6	14x15x15 6x14x3x3 6x13x13	6.15 1.5 2
YOLO_C6	1024x13x13 35x1024x1x1 35x13x13	338 70 11.5	OUT	1024x4x13 35x1024x1x1 35x4x13	104 70 3.5	IS	1024x4x13 5x1024x1x1 5x4x13	104 10 0.5	64x4x13 5x64x1x1 5x4x13	6.5 0.6 0.5

Table 7.2: The selection performed by TSO-burst for TLE/TLT slicing, partitioning, scheduling, and tile size.

instances where the KS tensor is considerably bigger than the IN and OUT tensors, leading TSO to choose the TLE KS scheme. For example, in ResNet-50_C53, the KS tensor is 5x bigger than the other two tensors together (IN and OUT), while in InceptionV3_C91, it is 29x bigger. In addition, TSO may also select another TLE scheme called KS&OUT, which applies to cases where both IN and OUT tensors have similar sizes compared to the KS tensor. This scheme applies to three of the convolutions, with the size difference between the tensors ranging from 1.17x (ResNet-50_C23) to 1.88x (InceptionV3_C39). By selecting the appropriate TLE scheme, the sizes of the IN, KS, and OUT tensors are optimally reduced into smaller chunks in the TLE slices, leading to reduced data transfers at runtime. As an example, consider YOLO_C6, which employs the TLE scheme KS. In this case, the KS tensor is optimally reduced from 18432KiB to slices of 4608KiB.

When it comes to scheduling selection, TSO tends to favor WS scheduling when the size of the KS slice becomes smaller than its corresponding MBLOB after passing through the TLT partitioning. As an example, consider ResNet-50_C1, which produces partitions of 8x3x7x7 (2.3 KiB) from a slice of 64x3x7x7 (18.4KiB). In general, among the convolutions presented in Table 7.2, TSO selects four of them to run with WS scheduling due to the reduced size of the partitioned KS slice, which ranges from 2x3x3x3 (0.1KiB) to 8x3x7x7 (2.3KiB). One of the benefits of selecting WS scheduling for those cases is that TSO can incorporate the partitioned KS slice into a single $\texttt{KS}^T$ tile, which, when kept stationary, allows for a unique load of the $\texttt{KS}^T$ tile along with the $\texttt{IN}^T$ tiles from DRAM. Besides the WS scheduling, TSO selects the IS strategy for four convolutions and the OS strategy for three other convolutions. The IS strategy is typically preferred for convolutions with a small number of filters and compact spatial dimensions (after the TLT partitioning). For instance, in the case of InceptionV3_C39, the $\texttt{IN}^T$ tile has a shape of 64x3x17, and both $\texttt{KS}^T$ and $\texttt{OUT}^T$ tiles have all the filters assigned to the TLT (12 in total). For this convolution, in particular, since the $\texttt{IN}^T$ tile does not comprise the entire IN slice assigned to the TLT, other three $\texttt{IN}^T$ tiles are formed, which requires, at runtime, three full reloads of all $\texttt{KS}^T$ tiles. Since reloading is unavoidable, keeping the larger-sized tiles stationary effectively reduces data transfer. Conversely, in the case of YOLO_C6, the partitioned IN slice forms a single $\texttt{IN}^T$ tile, thus resulting in a single corresponding $\texttt{OUT}^T$ tile, thus enabling the execution with just one load of the $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles. Finally, for the selected OS scheduling strategies, note that the number of filters increases up to 20 in the $\texttt{KS}^T$ and $\texttt{OUT}^T$ tiles for ResNet-50_C53. In contrast, for YOLO_C5, only 6 filters are selected to compose the $\texttt{KS}^T$ tile, but there is a significant increase in the spatial dimensions of the $\texttt{IN}^T$ and $\texttt{OUT}^T$ tiles, which accommodates 15x15 and 13x13 elements, respectively. Comparatively, this convolution does not perform well in the IS strategy, since the partitioned KS slice has 32 filters. Likewise, it also does not perform well on the WS strategy, since the partition of the KS slice assigned to the TLT is excessively large, with a size of 576KiB. Comparing the IS and OS scheduling strategies, there are cases in which both schedulings lead to theoretically equivalent reuse factors. Nevertheless, TSO often favors the IS strategy in such scenarios, as its implementation results in reduced RISC-V execution time due to the fewer number of loops it runs at runtime.

The tile selection in TSO aims to optimize data reuse by maximizing the tile size, whenever possible, up to the capacity of the MBLOB. To illustrate this, consider Incep-

tionV3_C4 as an example. This convolution's $\texttt{KS}^T$ tile includes the entire tensor after the TLT partition. For the $\texttt{IN}^T$ tile, TSO selects a shape of 32x1x73, occupying 4.5KiB. Attempting to increase the number of rows (T_H) in this tile by one would result in a $\texttt{IN}^T$ tile with a shape of 32x2x73, which would exceed the MBLOB capacity (9.1KiB). Another possibility to increase the $\texttt{IN}^T$ tile size is by increasing the number of channels (T_N) to up to 56x1x73. However, this would introduce edge cases, as it would require an additional uneven $\texttt{IN}^T$ tile of size 8x1x73 to compute the final output of the $\texttt{OUT}^T$ tile. It is worth noting that the $\texttt{OUT}^T$ tile cannot be increased further, as it already contains all the filters assigned to the TLT (after the partition). Moreover, NMP has a particularity that when computing a $\texttt{OUT}^T$ tile that requires multiple $\texttt{IN}^T$ and $\texttt{KS}^T$ tiles to be computed, the hardware enters a stacked/accumulation mode, which reduces the MBLOB (MB2) by $1/4^{th}$ as it computes the accumulations in quad-precision. When analyzing various tile shapes within the search space, TSO considers this constraint and excludes tiles that do not meet the hardware requirements. On the other hand, YOLO_C1 exhibits a larger $\texttt{OUT}^T$ tile compared to the other convolutions (4.22KiB). This is possible because this convolution only requires a single $\texttt{IN}^T$ and $\texttt{KS}^T$ tile to compute the corresponding $\texttt{OUT}^T$ tile, which enables the MB2 (OUT) on-chip memory to have its original size. Finally, in the case of InceptionV3_C4, TSO is already selecting an efficient solution for the tile sizes. It maximizes data reuse within the NMP constraints.

In order to improve access to memory, TSO tends to favor tiles with an increased width to enhance burst accesses. Among the convolutions presented in Table 7.2, nine of them (82%) maximize the width dimension of the $\texttt{IN}^T$ (T_L) and also of the $\texttt{OUT}^T$ (T_C) to their maximum achievable values. Conversely, for the two other convolutions (ResNet-50_C1 and YOLO_C1), TSO also increases the width of the tiles but not to the maximum size, as doing so would result in small tiles, making it impossible to add channels and rows to them as they would exceed the MBLOB capacity.

7.3 TSO-burst vs TSO-noburst Tiling

This section compares the models' performance when TSO uses the two heuristics that estimate the time convolutions spend on DRAM access: (1) TSO-burst, based on burst modeling, proposed in this thesis (Section 6.5); and (2) TSO-noburst, that relies on data-volume without burst modeling, a common approach found in most previous works. The speedup of TSO-burst over TSO-noburst ranges from 7.4%, for YOLO, up to 21.7%, for InceptionV3, as shown in Figure 7.1. The main improvements from using TSO-burst come when the $\texttt{IN}$ are divided into $\texttt{IN}^T$ tiles. This happens because TSO tends to select larger tiles in the width (row-major) direction. By selecting larger tiles, TSO minimizes the number of required bursts, thus reducing the impact of the CAS latency on the memory access time. By prioritizing bursts on the width direction, TSO maximizes the usage of the bursts, as it improves memory access coalescing. For the case of TSO-noburst, the tiles are selected so as to reduce the number of bytes loaded from the DRAM to NMP. Contrary to those, the TSO-burst technique proposed in this thesis takes into consideration DRAM access coalescing to estimate the time taken to LOAD/STORE data from memory, thus

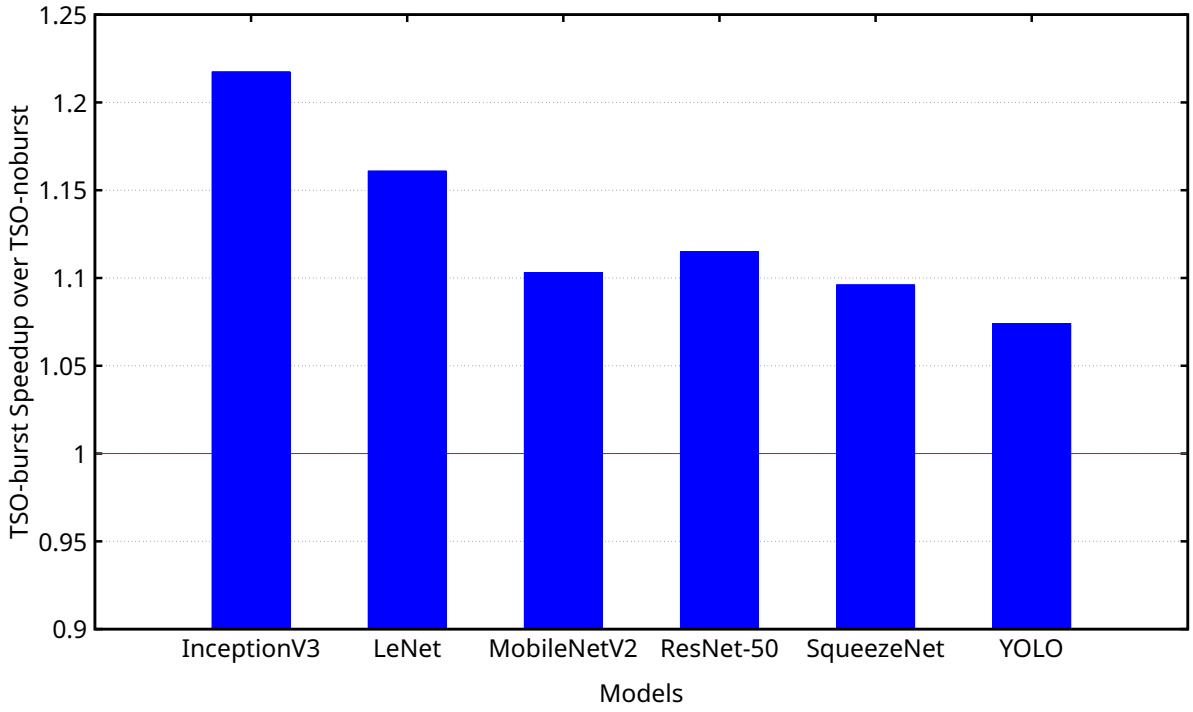

Figure 7.1: TSO-burst speedup over TSO-noburst on end-to-end model execution time. The baseline used in this experiment is the TSO-noburst.

Model	**Model TSO (us)**		**TLE slicing fixed (us)**			**TLT scheduling fixed (us)**		
	Burst	**No Burst**	**KS**	**KS& OUT**	**OUT**	**IS**	**OS**	**WS**
InceptionV3	72686	88478	82370	81367	91408	73706	93731	101641
LeNet	199	231	202	224	231	199	233	228
MobileNetV2	14030	15470	17765	16696	17571	14084	17387	15155
ResNet-50	55927	62375	62077	63118	78844	59714	71905	77535
SqueezeNet	12504	13713	16579	14134	12993	12908	15840	13452
YOLO	53271	57225	56591	55713	63550	68530	58592	55375

Table 7.3: Model execution time of TSO-burst, TSO-noburst, fixed TLE slicing, and fixed TLT scheduling.

resulting in better partitioning and improved performance. The resulting execution time for the various models is shown in Table 7.3. Notice in the table that TSO-burst always produces the shortest execution time.

As an example, consider the 5th Conv-layer of InceptionV3, which has 80 input feature maps (i.e., channels) of size 73x73 each, and 192 filters of size 80x3x3. The shape of the $\texttt{IN}^T$ tile selected by TSO-burst has a size 14x4x73 ($T_N \times T_H \times T_L$). Since the $\texttt{IN}^T$ tile of the TSO-burst takes the whole width L (73) of a channel, it results in a sequence of 584 bytes aligned sequentially on the DRAM ($4 \times 73 \times 2B$), which requires 5 memory bursts for each tile's channel. When considering all the 14 channels of that tile, the 5th

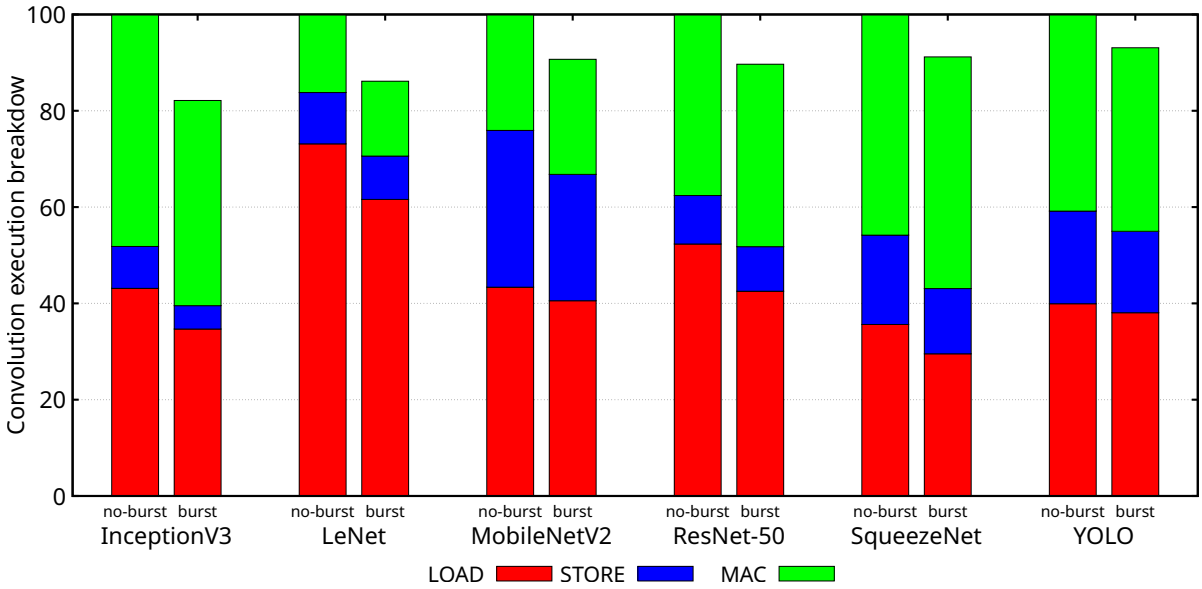

Figure 7.2: Convolution execution breakdown with TSO-burst as a relative proportion of TSO-noburst. The time taken to run the RISC-V instructions is not included; instead, it is distributed into the LOAD, STORE, and MAC timings.

Model	LOAD Time (us)			STORE Time (us)			MAC Time (us)		
	Burst	No Burst	Speed up	Burst	No Burst	Speed up	Burst	No Burst	Speed up
InceptionV3	30679	38155	1.24	4303	7701	1.79	37703	42623	1.13
LeNet	142	169	1.19	21	25	1.19	36	37	1.03
MobileNetV2	6273	6707	1.07	4059	5037	1.24	3697	3725	1.01
ResNet-50	26521	32641	1.23	5775	6278	1.09	23632	23456	0.99
SqueezeNet	4050	4885	1.21	1858	2544	1.37	6597	6284	0.95
YOLO	21783	22849	1.05	9676	11006	1.14	21813	23370	1.07

Table 7.4: Breakdown of how the execution time is distributed for LOAD, STORE, and MAC operations on NMP. In this experiment, the time taken to execute RISC-V instructions is assigned to the operation (e.g., LOAD) that requires those instructions.

Conv-layer requires a total of 70 memory bursts. On the other hand, TSO-noburst selects a tile of size 16x11x20, which corresponds to only 20 bytes aligned on the DRAM, thus resulting in one memory burst for each row. Given that the $\texttt{IN}^T$ tile has 16 channels, each with 11 rows, it requires a total of 176 memory bursts, which is more than double what is needed to load the TSO-burst tile. For that specific 5th Conv-layer, TSO-burst reduces the convolution execution time by 28%.

7.3.1 Execution Breakdown Analysis

The solution provided by TSO-burst aims to reduce the total time taken for data transfer operations. To illustrate that, refer to Figure 7.2, which shows for each model two bars representing the breakdown of the percentage of computation time spent in LOAD, STORE, and MAC operations with respect to the total execution time of TSO-noburst.

Model	Convolution TSO (us)			Convolution Time Share	
	Burst	No Burst	Speed up	Burst	No Burst
InceptionV3	66943	82727	1,24	0,92	0,94
LeNet	64	89	1,39	0,32	0,39
MobileNetV2	10015	11433	1,14	0,71	0,74
ResNet-50	46187	52359	1,13	0,83	0,84
SqueezeNet	11444	12658	1,11	0,92	0,92
YOLO	35711	39471	1,11	0,67	0,69

Table 7.5: Convolution execution time on each CNN model, along with the respective percentage of time allocated to running these convolutions (time share) for TSO-burst and TSO-noburst.

Note that for the TSO-burst bars, the percentage of execution time is calculated with respect to the total time of TSO-noburst. As shown in Figure 7.2, when TSO-noburst is used, the percentage of the LOAD+STORE transfer time ranges from 51.83%, for InceptionV3, up to 83.80% for LeNet. On the other hand (refer to Table 7.4), when TSO-burst is used, the time taken by LOAD+STORE operations decreases from 7.08% to 23.71% for YOLO and InceptionV3, respectively.

Another observation that can be derived from Figure 7.2 and Table 7.4 is a comparison of time taken by MAC operations between TSO-burst and TSO-noburst. These two approaches may select different tile shapes for the convolutions due to distinct cost models, leading to varying MAC utilization. For instance, TSO-burst outperforms TSO-noburst in specific models, showing speedups ranging from 1% in MobileNetV2 to 13% in InceptionV3. Conversely, TSO-noburst exhibits superior performance compared to TSO-burst in ResNet-50 and SqueezeNet, causing slowdowns ranging from 1% to 5%. It is important to highlight that, even though TSO-noburst surpasses TSO-burst in MAC operations for ResNet-50 and SqueezeNet, it incurs penalties in loading time (LOAD+STORE), where TSO-burst demonstrates significant improvement.

7.3.2 Convolution Time Analysis

Table 7.5 shows the convolution-only speedup, comparing TSO-burst with TSO-noburst. TSO-burst outperforms TSO-noburst in every model on NMP, with convolution speedups ranging from 11% to 39% for YOLO and LeNet, respectively. Additionally, it also presents the fraction of time spent on convolutions for the evaluated CNN models, which varies from 32% (TSO-burst) for LeNet to 94% (TSO-noburst) for InceptionV3.

A more in-depth performance analysis is necessary to examine the performance changes in individual convolutions. Figure 7.3 presents the results for all individual convolutions, sorted by speedup. TSO-burst outperforms TSO-noburst in 73% of the convolutions (174 out of 238), achieving a maximum speedup of 3.69x. Among the 27% of convolutions that experience a slowdown, representing 64 out of 238 convolutions, 34 of them (i.e.,

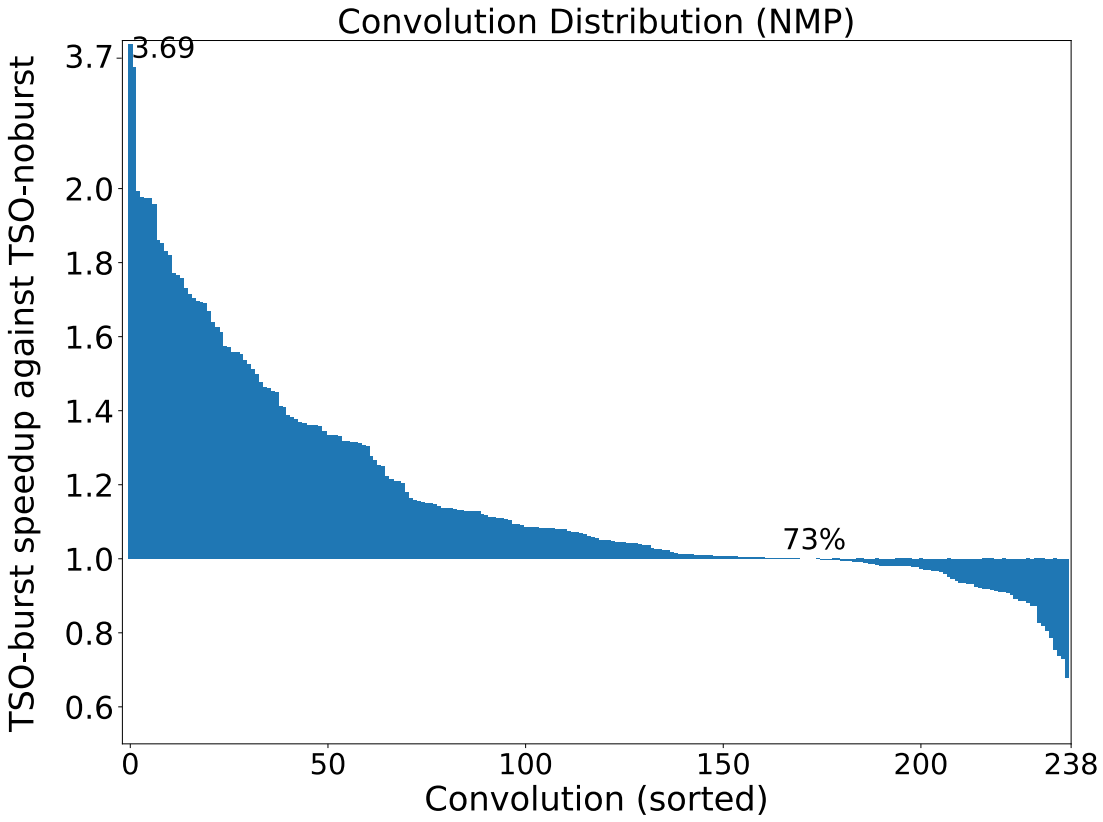

Figure 7.3: TSO-burst performance speedup against TSO-noburst for each of the 238 individual convolutions from the evaluated CNN models. The x-axis is sorted by speedup (y-axis). TSO-noburst performance is normalized to 1. The graphs also indicate the percentage of convolutions in which TSO-burst outperforms TSO-noburst.

53%) exhibit a slowdown of less than 3%. This slowdown can be attributed to certain limitations in TSO-burst: (a) in cases (8 out of 34) where the spatial dimensions are small, and the convolution involves padding, more memory transactions may be required because NMP loads one row at a time to organize the data internally in the on-chip memories (e.g. MB0), and this process is not accurately estimated by TSO; (b) in some cases (8 out of 34), TSO-noburst, which is a data-volume based cost model, may select more channels for some tiles compared to TSO-burst, which does not find this addition beneficial (e.g., due to introduction of edge cases); (c) in some other cases (4 out of 34), the select TLT schedule in TSO-burst is not the most suitable for the convolution; (d) in the remaining 14 convolutions (out of 34), both TSO-burst and TSO-noburst choose the same solution from the search space. However, due to a very slight difference ($\sim$ 0.05%), they are categorized as exhibiting a slowdown. For the 30 convolutions with a slowdown exceeding 3% (representing 30 out of 64), other limitations in TSO-burst are responsible for the performance degradation, as follows: (a) the TSO-burst heuristic tends to select more rows in a channel (e.g. T_H in a $\texttt{IN}^T$ tile) before proceeding with the channel direction, and this issue applies to 22 out of 30 convolutions. This choice aims to enhance memory burst accesses by creating more sequential data within a tile and, consequently, maximizing the burst usage (refer to Equations 6.9–6.11 in Section 6.5). However, due to this greedy

selection, the number of channels available for constructing the tiles is constrained by the on-chip memory sizes. In this same scenario, TSO-noburst opts for fewer rows (e.g. T_H) and more channels across the tensor's depth when constructing the tiles. This results in larger tiles and, consequently, more data reuse; and (b) for the other convolutions (8 out 30), the TLT scheduling selected by TSO-burst is not the most suitable, leading to more memory access due to the selected scheduling.

7.4 TLT Scheduling and TLE Slicing Analysis

This section aims to evaluate the sensitivity of models' performance to TLT Scheduling and TLE Slicing. It shows the impact of consistently selecting a fixedTLT scheduling strategy (IS, OS, and WS) and TLE slicing (KS, KS&OUT, and OUT) when mapping the convolutions. It is important to note that TSO explores all possible solutions within its search space, which encompasses different combinations of TLT scheduling strategies and TLE slicing schemes, and aims to identify the solution with the best-estimated timing for better performance. In this experiment, the search space is limited at compile-time either by restricting the TLT schedule to a single solution (Subsection 7.4.1) or by constraining the TLE slicing to a single solution (Subsection 7.4.2). The other solutions in the TSO's search space are preserved, and the candidate solutions are evaluated through TSO with the burst modeling activated (TSO-burst). Refer to Table 7.3 for the timings.

7.4.1 Fixed TLT Scheduling

In this experiment, the compiler is set to generate code that fixes one of the three TLT scheduling strategies (IS, OS, and WS) for all the model's Conv-layers. The compiler applies the fixed scheduling strategy to all possible TLE slicing options (KS, KS&OUT, and OUT) and TLT tiling ($\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$) to search for the solution that improves performance over the others. Figure 7.4 shows the speedup of TSO when compared to the best fixed TLT strategy. By fixing IS during TF-XLA Code Generation, TSO speedup reaches 28,6% on YOLO. For the YOLO network, IS does not perform well since this network has a varied number of Conv-layers with IN tensors varying from 102×102 ($H \times L$) to 416×416, which results in multiple loads of the filters in the $\texttt{KS}^T$ tiles since multiple $\texttt{IN}^T$ tiles are created. TSO speedup with respect to fixed OS reaches up to 28.6% on ResNet-50. When compared to fixed WS, TSO speedup is 39.8% on InceptionV3. In the case of InceptionV3, when WS is used, multiples $\texttt{KS}^T$ tiles are required to cover the TLT assigned data, and thus multiple loads of the $\texttt{IN}^T$ tiles become necessary for each $\texttt{KS}^T$ tile, leading to an increase in data transfers. Here again, TSO outperforms the best fixed TLT scheduling strategy on every CNN model.

7.4.2 Fixed TLE Slicing

In this experiment, the compiler was set to generate code that fixes each TLE slicing scheme described in Section 6.2 for all Conv-layers of a model. The experiment works as follows. The compiler identifies, for the fixed TLE slicing, a solution that encompasses

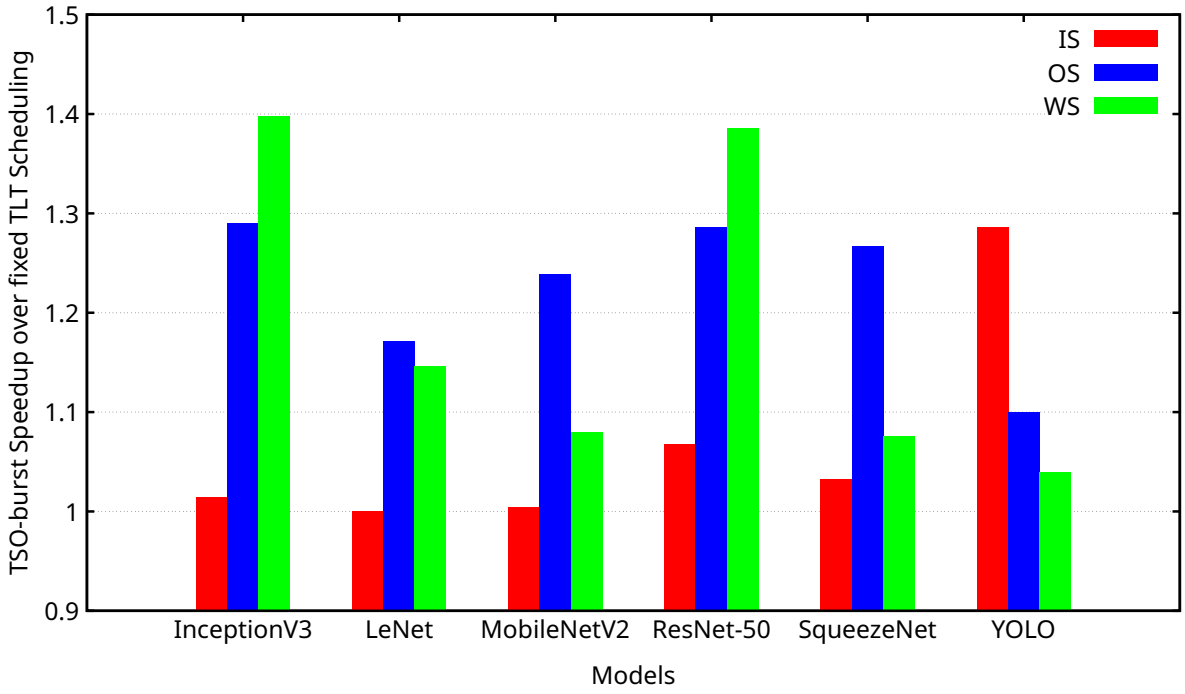

Figure 7.4: The three TLT schedulings (IS, OS, and WS) remain fixed during code generation. Although the TLT scheduling itself is fixed, TSO attempts to find the best TLE scheme and optimal tiling for that fixed TLT scheduling.

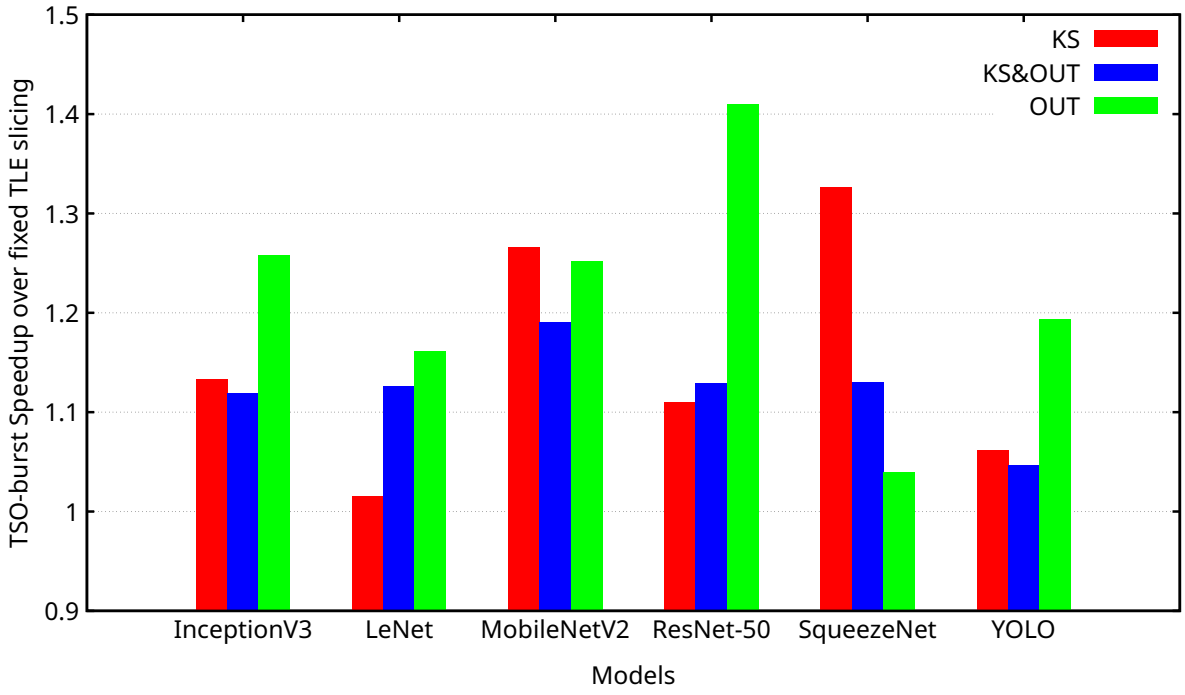

Figure 7.5: The three TLE slicing schemes (KS, KS&OUT, and OUT) remain fixed during code generation. Although the TLE scheme itself is fixed, TSO attempts to find the best TLT scheduling and optimal tiling for that fixed TLE scheme.

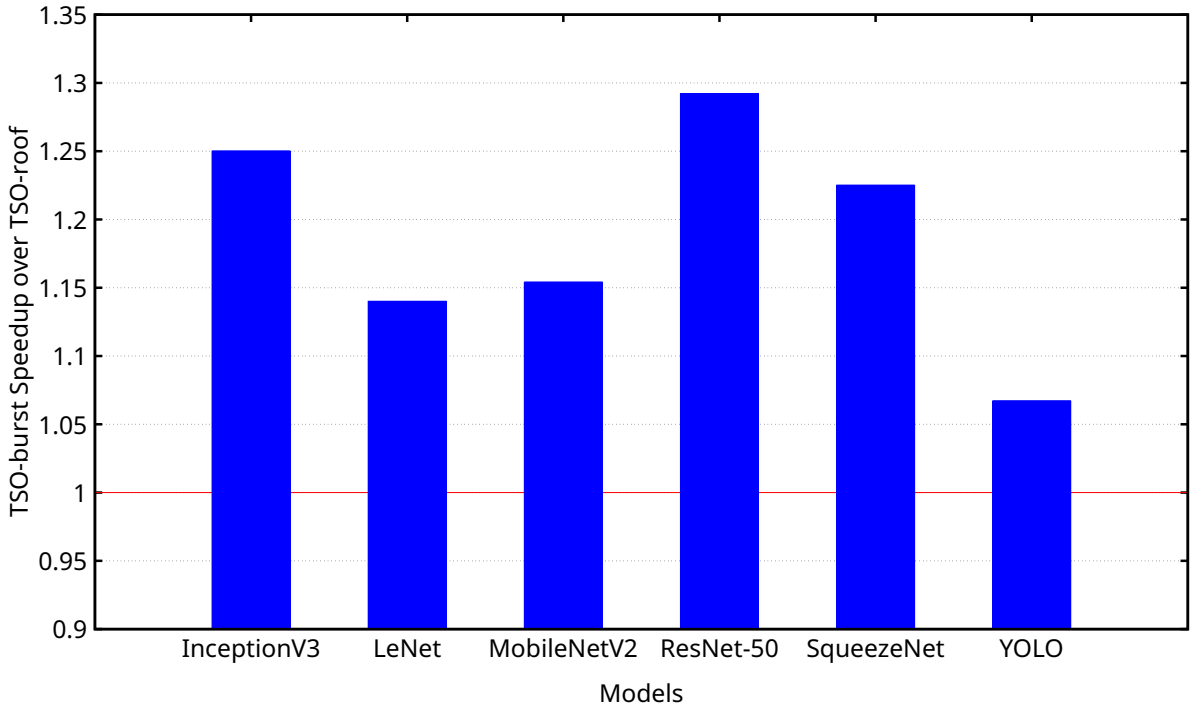

Figure 7.6: TSO-burst speedup over TSO-roof (an adaption from [57]) on end-to-end model execution time.

the best TLT scheduling strategy (IS, OS, and WS) and TLT tiling ($\texttt{IN}^T$, $\texttt{KS}^T$, and $\texttt{OUT}^T$). The result of this experiment is shown in Figure 7.5, which reports the speedup of the model compiled with TSO (burst mode) when compared to the model compiled with the fixed TLE slicing. For the *KS* slicing case, TSO achieves a speedup of up to 32.6%, for SqueezeNet. SqueezeNet does not perform well with the TLE KS slicing fixed since most of its Conv-layers have the `IN` tensors larger than the size of the filter set (`KS`). For the *KS*&*OUT* slicing, TSO speedup reaches up to 19%, for MobileNetV2. This TLE scheme usually works better for Conv-layers with similar sizes for both `IN` and weights (`KS`). For the *OUT* slicing, TSO speedup is 41.0%, for ResNet-50. For most of ResNet-50's Conv-layers, the size of the weight set (`KS`) is larger than the size of the `IN` data tensor. As a result, TSO outperforms the best fixed TLE slicing scheme in every CNN model.

7.5 TSO-burst vs TSO-roof Tiling

The analytical model in [57] was adapted to work with the TSO's search space and then encapsulated into a solution called TSO-roof, as described in Section 6.5. TSO-roof is a cost model that estimates the Attainable Performance and Operational Intensity of every solution generated by the TSO's search space and selects the optimal solution from this solution space. For every evaluated solution, TSO-roof prioritizes those with the highest Attainable Performance, and then, from these selected solutions, it selects the one with the highest Operational Intensity. In other words, TSO-roof first prioritizes solutions that increase the number of operations performed per second (OPS/s), which results in a set of solutions, and then chooses the one that maximizes the number of operations performed

for each byte retrieved from DRAM.

When comparing TSO-burst against TSO-roof in Figure 7.6, it is notable that TSO-burst outperforms every CNN model in the comparison, achieving performance improvements ranging from 6.7% for YOLO up to 29.2% for ResNet-50. This enhancement can be attributed to the fact that TSO-burst prioritizes tiling layouts that maximize burst memory access, thereby reducing the impact of LOAD and STORE operations due to optimized data access patterns. On the other hand, TSO-roof exclusively focuses on data reuse and does not take into account the potential benefits of an efficient tiling layout, which is where burst access optimization is explored.

When analyzing each convolution individually, TSO-burst outperforms 198 convolutions, accounting for 84% of the total 236 convolutions. The maximum speedup achieved is 5.19x, while the convolution with the slowest slowdown is only 30%. The 38 convolutions out of the total 236 that experienced slowdowns exhibit similar behavior to those described in Subsection 7.3.2, where the same limitations in TSO-burst were noted.

Chapter 8

Additional Experimental Results

This chapter explores additional analysis and experimental results of TSO to show its effectiveness. In Section 8.1, an analysis of the impact of applying loop fusion optimization in TF-XLA is presented. This optimization involves combining multiple CNN operations into a single operation to reduce DRAM accesses. Next, Section 8.2 compares how NPUs fare with respect to CPUs for the same models. Section 8.3 demonstrates how multi-threading can be used to reduce the time TSO spends exploring the tiling/scheduling solution space. Section 8.4 presents a discussion on the impact of a decision tree applied to TSO. Section 8.5 shows in a Roofline model the actual execution time extracted from TSO-burst and indicates how close it is to the peak performance and bandwidth. This enables one to evaluate how far TSO with burst modeling enabled is from exploiting all the potential speedup available in the NMP hardware. Finally, Section 8.6 shows that TSO can also be incorporated into other compilers, such as Glow and ONNX-MLIR, without any major impact on models' performance and accuracy.

8.1 Impact of Loop Fusion on NMP

In addition to the TSO optimization, another optimization technique with strong potential for reducing execution time when running CNN models on Multicore NPU accelerators, as in the case of NMP, is loop fusion. Loop fusion involves combining one or more operations into a single operation. This eliminates the need to save intermediate tensors into DRAM for subsequent loading to compute the next operation. Instead, the data undergoes multiple operations while residing in the NPU's on-chip memory, and only after completing all of them, the result is stored in DRAM (for more details, refer to Subsection 5.1.1).

The impact of fusing multiple operations is significant in optimizing performance. Figure 8.1 shows a speedup analysis where TSO-burst is compiled through the TF-XLA compiler with the loop fusion optimization enabled in the Pass Manager (PM) and is compared against another compilation flow where the loop fusion optimization is disabled in the PM. All CNN models running with loop fusion enabled in the compiler are faster than the other flow without that optimization, with speedup ranging from 1% for MobileNetV2 up to 48% for LeNet. In MobileNetV2, most of the convolutions, including depthwise con-

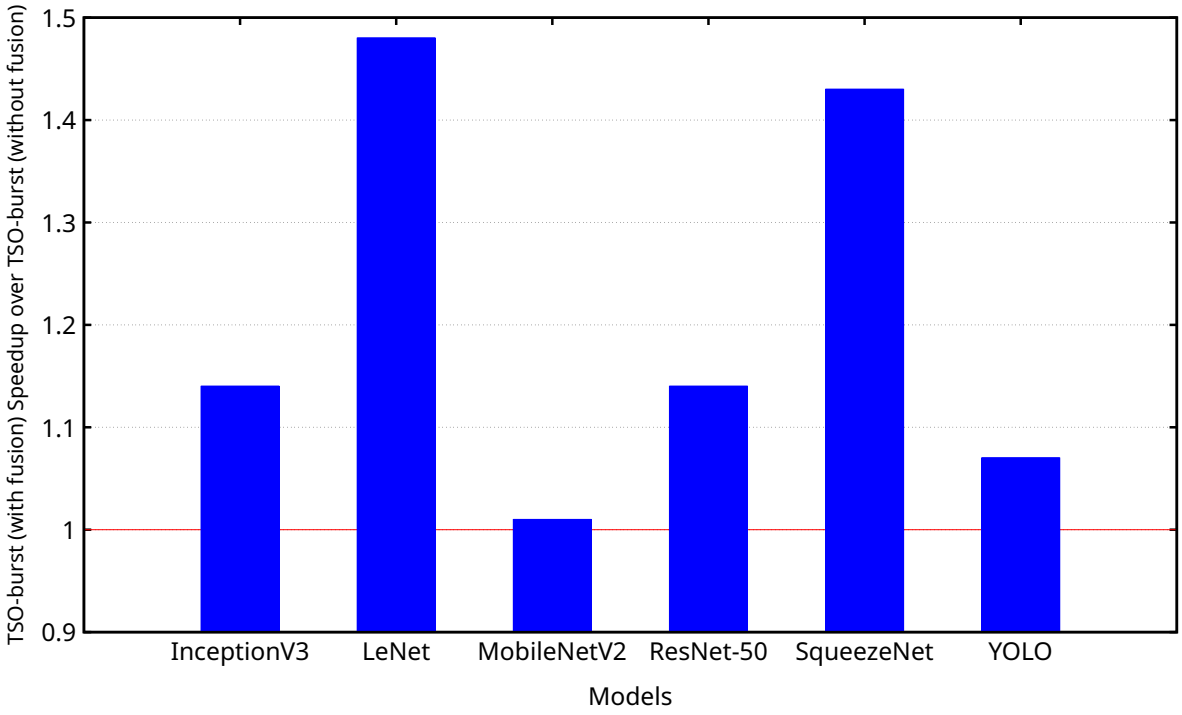

Figure 8.1: TSO-burst with loop fusion enabled compared to TSO-burst without loop fusion enabled.

volutions, are followed by ReLU6. Due to the complexity of computing this operation on NMP, it is not fused into the convolutions but is computed separately as an independent operation. However, a few convolutions, which do not significantly contribute to the overall execution time, are followed by bias summation and ReLU, where the speedup is achieved. In the case of YOLO, a similar scenario applies, where LeakyReLU could be fused, but it is not, even when loop fusion is enabled in the Pass Manager (PM) in TF-XLA. However, for most of the evaluated CNN models, the convolutions are followed by bias addition and ReLU, and not fusing them has a significant impact on performance. For instance, in SqueezeNet, only for the first convolution, when bias addition and ReLU are not fused, an additional 9.2MiB of data movement is required to/from DRAM. If one considers all convolutions in SqueezeNet, an additional 34 MiB of data transfer is needed.

8.2 NMP-TSO vs CPU-Eigen

In this section, a comparison is made between the performance of TSO-optimized models running on NMP, with Eigen-optimized models running on a CPU. Eigen [26] is an optimization library for linear algebra operations like GEMM, that is extensively used in the design of convolution operations. The CPU experiment relies on TF-XLA to parallelize convolutions using a multi-threaded Eigen-based parallelization approach on a Xeon-CPU-E5-265L @1.8GHz with 16 physical cores (32 threads with hyper-threading activated). In order to be fair, during the CPU experiment, the first execution of the model was ignored since the whole CPU cache hierarchy misses to load the model input and weight data from the disk to DRAM and then to the caches (in some cases, the model

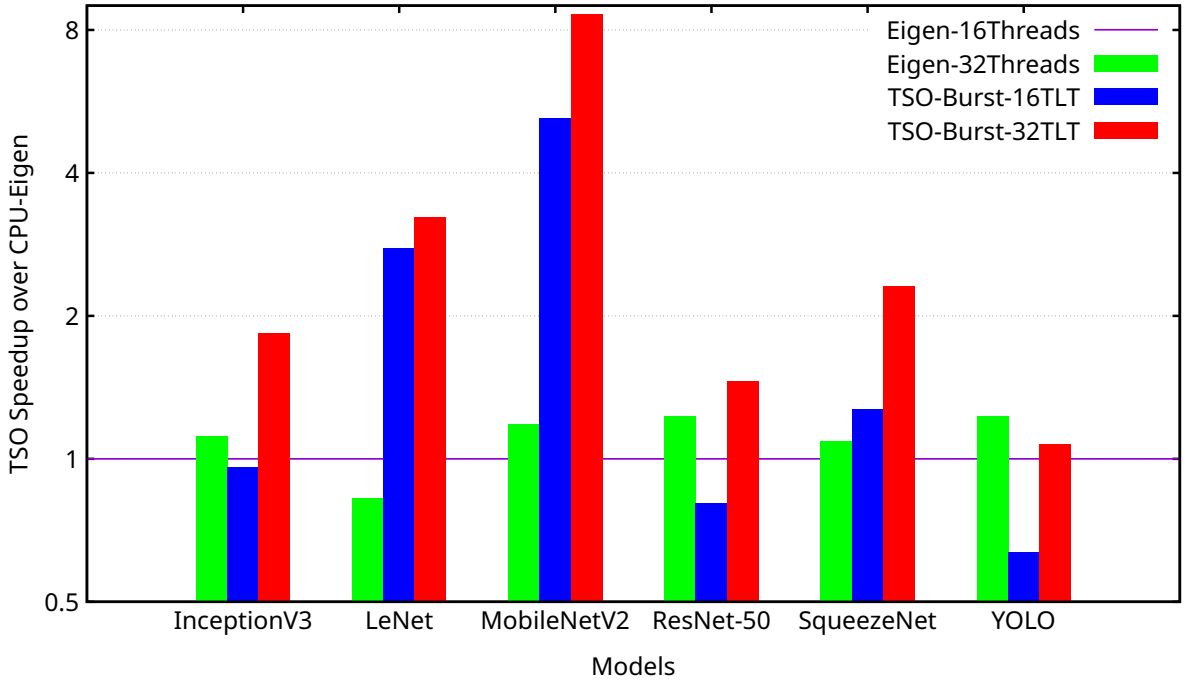

Figure 8.2: NMP-TSO-burst vs CPU-Eigen performance speedup, where NMP was experimented with 16 and 32 TLTs, and the CPU with 16 and 32 threads.

weights reside in L3 cache between two consecutive inferences). This does not occur during the NMP execution, which is the reason why that time was discharged in the CPU case.

Figure 8.2 shows the performance speedups of NMP-TSO with respect to CPU-Eigen for each one of the models discussed so far (refer to Table 8.1 for the execution time of each experiment). For the CPU experiment, TF-XLA was compiled to use a thread pool composed of 16- (Eigen-16Threads) and 32-threads (Eigen-32Threads). For NMP, NMP-code was generated for the same number of TLTs as the number of CPU cores, i.e., 16- (TSO-Burst-16TLT) and 32-RISCV (TSO-Burst-32TLT) cores each running @1GHz. The results of each execution were then compared by considering the CPU-Eigen with 16 threads (Eigen-16Threads) as the baseline. The reader should notice that the clock of each TLT RISCV core (1GHz) is almost half of the clock of the powerful Xeon CPU cores (1.8GHz).

As shown in Figure 8.2, it is possible to see that the CPU does not scale very well as the number of threads doubles for the Eigen-32Threads experiments (green bars). For example, in LeNet, a slowdown can be observed given that the model and its computational intensity are very small, and an increase in the number of threads directly impacts the communication cost. For the other models, Eigen-32Threads shows a slight speedup that ranges from 8.3% up to 23.1% for SqueezeNet and ResNet-50, respectively.

Now consider a comparison of two architectures with the same number of physical cores, i.e., TSO-Burst-16TLT (blue bars) and the (Eigen-16Threads) baseline. As shown in Figure 8.2, a slowdown of 4.12%, 19.54%, and 36.36% occurs for InceptionV3, ResNet-50 and YOLO, respectively. The slowdown that is observed in TSO-Burst-16TLT when

Model	TSO-burst (us)		CPU Eigen (us)	
	16 TLTs	**32 TLTs**	**16 threads**	**32 threads**
InceptionV3	139150	72686	133416	119657
LeNet	231	199	642	776
MobileNetV2	23243	14030	120852	102216
ResNet-50	101239	55927	81455	66169
SqueezeNet	22698	12504	28875	26572
YOLO	90013	53271	57285	46695

Table 8.1: Execution time of TSO with 16- and 32-TLTs, and CPU Eigen with 16- and 32-threads.

compared to the Eigen-16Threads baseline comes from the high pressure imposed on the TLE DMA engines due to the large amount of data required by these models and also due to the fact that more work is assigned to the 16 TLTs. This does not occur with CPUs due to the larger caches and faster memory controllers. For the remaining models, a speedup from 1.27x up to 5.2x is achieved for SqueezeNet and MobilenetV2, respectively. In this case, the TLEs DMA engines can accommodate the data transfer workloads of the models.

Finally, consider now the case when the number of TLTs is 32 (TSO-Burst-32TLT), represented by the red bar in the graph. In this case, all the models perform better when compared to Eigen-16Threads, with speedups ranging from 1.075x up to 8.61x, for YOLO and MobileNetV2, respectively. Besides that, an average speedup of 83.72% is achieved when comparing TSO with 16- and 32-RISCV cores.

8.3 Speeding-up TSO Search Space Exploration

As discussed before, TSO explores a large solution space that combines all possible variations of TLE Slicing, TLT Partitioning, and Scheduling. This results in a large solution space to explore, and finding a small-cost (execution time) solution takes a long time. Given that estimating the cost of each solution can be performed in parallel, TSO uses OpenMP task parallelism to speed up its execution. This experiment aims to evaluate the impact of the OpenMP task-parallelism annotations in Algorithm 1 (lines 2, 3, and 5) on the overall time of the TSO space exploration. The experiment was performed on an Intel Xeon E5-2620 with 16-physical cores and 64GiB of memory. The results are shown in Figure 8.3 where each line corresponds to one model, the y-axis is speedup with respect to the TSO sequential execution as the number of threads used by OpenMP grows (x-axis). Note that the multi-threading execution has almost a linear improvement when compared to the serial execution for most of the models. For InceptionV3, which has 94 Convolutions, the multi-threading execution is almost linear. On the other hand, for LeNet, which has only 2 Convolutions, 2 threads are enough to accelerate the execution, and thus a slowdown shows up if the number of threads increases from that point on. In terms of

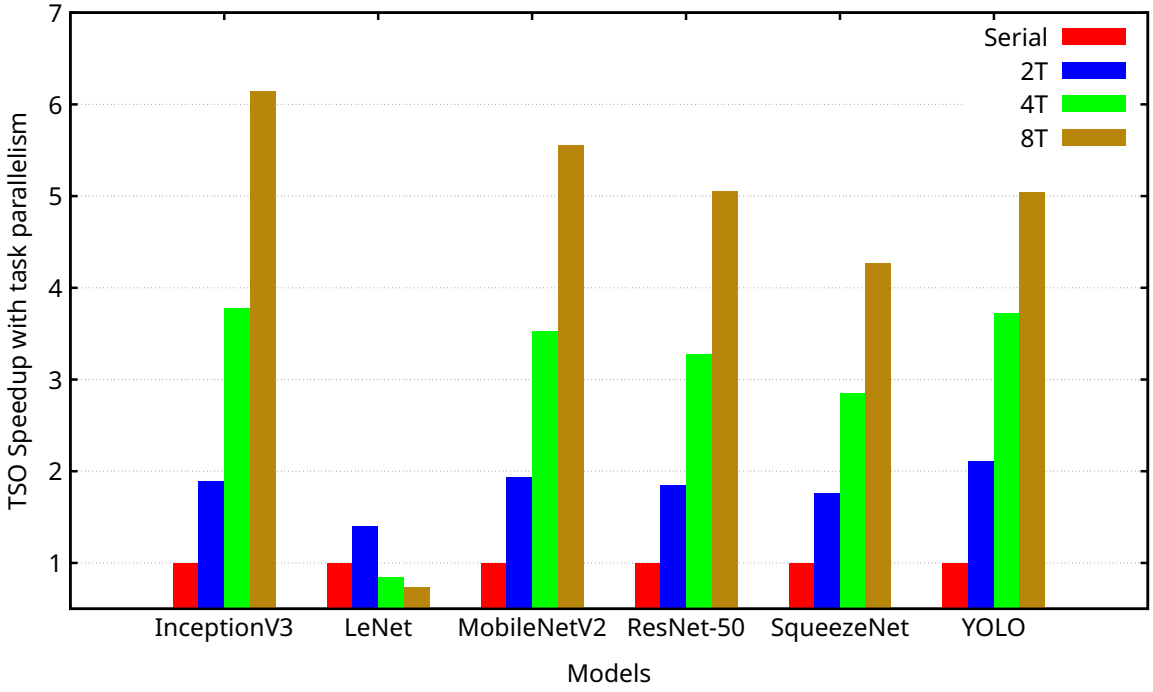

Figure 8.3: Evaluating OpenMP parallelization of TSO solution space exploration.

time, the serial execution of TSO varies from 28 ms, for the LeNet network, to 6 min for the InceptionV3 network, while with the multi-threading execution, this time is reduced to 17 ms and 59 sec, respectively. The total time spent by the compiler, from the beginning to the generation of the binaries, varies from 20 seconds to 7 minutes for the same models, respectively, whereas most of the time is consumed by the calibration/quantization step. TSO typically completes its computations within a couple of minutes, whereas other frameworks, such as TVM [13], may require several hours to accomplish the same task. For instance, compiling the InceptionV3 model with the TVM's auto-tuning feature takes approximately 8 hours, using the same CPU.

8.4 Using a Decision Tree in the TSO Algorithm

The TSO algorithm exhaustively explores the search space by combining the TLE slicings (KS, KS&OUT, OUT) with the TLT schedulings (IS, WS, OS) and subsequently running the tiling process for each combination. Since most of the TSO solutions are predictable, it is possible to construct a decision tree to simplify the search space.

Figure 8.4 shows the TSO algorithm using a decision tree. The node in the column marked with ① identifies whether the `KS` tensor is twice the size of the `IN` tensor; if true, the TLE KS slicing is selected. If the `IN` tensor is twice the size, then the TLE OUT slicing is chosen (see ②). If neither condition holds, indicating similar sizes for `IN` and `KS` tensors, the TLE KS&OUT slicing is selected. For each combination of TLE KS, KS&OUT, and OUT, the same TLT schedulings apply. The column marked with ③ presents the condition to determine if WS should be selected. The decision is based on whether the `KS` slice (after being partitioned among the TLEs/TLTs) fits in MB1. If not,

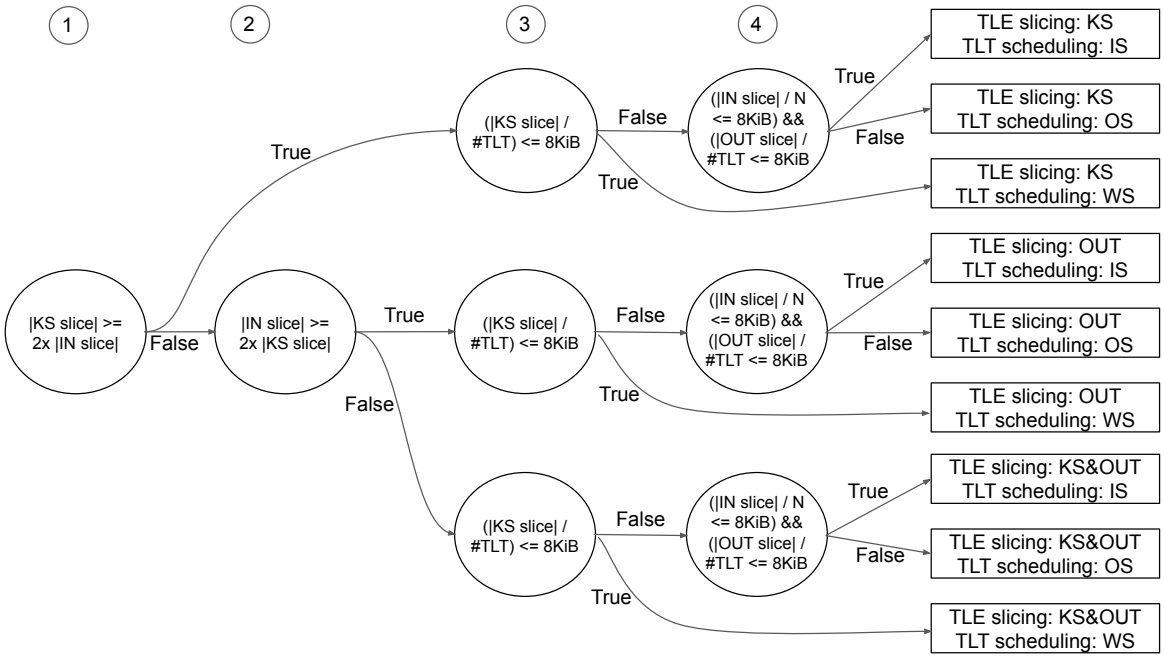

Figure 8.4: Using a Decision Tree in TSO for TLE slicing and TLT scheduling.

the conditions in the column denoted with ④ check if IS should be selected. This is done by validating if a single channel of the IN slice fits in MB0 and if the entire OUT slice, after being partitioned among the TLEs/TLTs, can be integrated into a single $\texttt{OUT}^T$ tile that fits in MB2. If both conditions are valid, the TLT IS is selected; otherwise, the TLT OUT is chosen.

The TLE slicing and TLT scheduling selected from the decision tree solution for the 238 convolutions exhibit an 80% correspondence with the exhaustive TSO algorithm. While TSO with the decision tree runs 9 times faster, the CNN models compiled with the exhaustive solution run, on average, 8% faster. The decision tree could be enhanced with additional nodes to make decisions (e.g., by taking hardware details into account), but we will leave this as future work.

8.5 Plotting Actual NMP Run time on a Roofline Model

To evaluate the performance of the code resulting from using TSO on the Conv-layers of each model executed on NMP (TSO-burst), we used the Roofline Model, shown in Figure 8.5, with the **actual** execution times obtained from running the CNN models on NMP. The graph's y-axis represents the Multiply-and-accumulate (MAC) throughput (in GMACs/sec) achieved by the architecture and the convolution execution. In the x-axis is the Operational Intensity, which represents the number of MAC operations executed for each byte loaded from the DRAM. The blue lines in the graph represent the theoretical roofs for both the MAC throughput (horizontal line) and DRAM bandwidth (sloped line) that can be respectively achieved by the NMP engine and the memory system. To better evaluate the system's real performance, two additional experiments were undertaken to measure these parameters. This is required, given that other architecture components

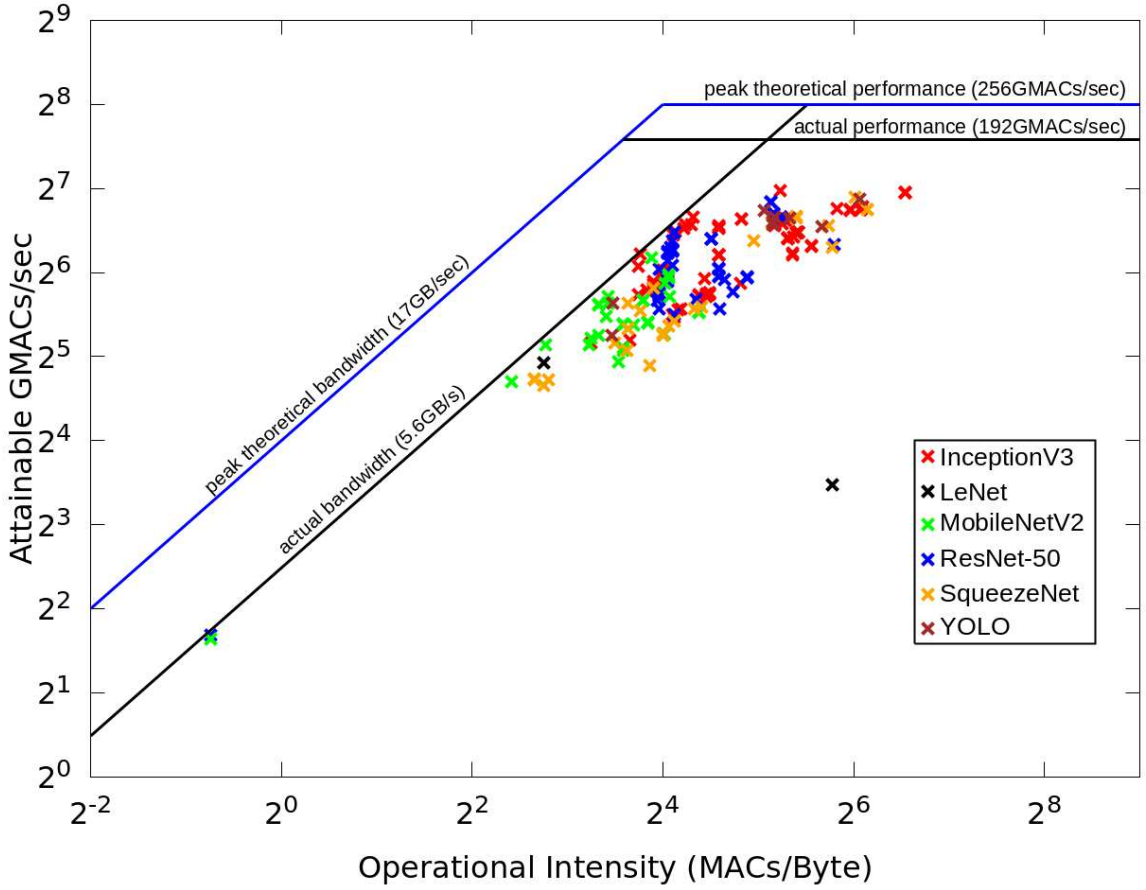

Figure 8.5: Roofline model for NMP architecture (TSO-burst) with actual execution time collected from running the CNN models on NMP.

can impact their values. The black lines in the graph represent these measurements. As shown, the measured MAC throughput reaches a roof of 192 GMACs/sec, represented by the horizontal black line. A number of issues can explain this reduction. For example, the NMP device used in this work has single-ported on-chip memories, and thus TLTs that are waiting for data get idle without using their MAC Unit. As for the memory bandwidth (the sloped black line), the measured value is also reduced. This can be explained by the fact that the DRAM bandwidth is constrained by a single DME engine per TLE, which has to serve all 8 TLT cores simultaneously. In order to evaluate the performance resulting from TSO, we plotted one point in Figure 8.5 for all convolutions in the models. As shown, most of the convolutions reach either the roof limited by the (measured) memory bandwidth (sloped black line) or approach the roof defined by the MAC throughput (horizontal black line). This makes it clear that TSO produces code that approaches the maximum performance of the architecture.

8.6 Evaluating TSO on Glow Compiler & ONNX-MLIR

A set of experiments was also performed to evaluate the portability of TSO to other Machine Learning compilers. The Glow compiler from Facebook was selected for this evalu-

Model	Arch.	Top-1	Top-5
MNIST	CPU (FP)	98.9%	100%
	NMP	99.2%	99.8%
LeNet	CPU (FP)	94.8%	99.9%
	NMP	95.2%	99.9%
ResNet-18	CPU (FP)	69.9%	89.3%
	NMP	66.4%	87.3%
SqueezeNet	CPU (FP)	47.1%	71%
	NMP	47.1%	71%
MobileNetV2	CPU (FP)	70.2%	89.6%
	NMP	70.9%	89.4%

Table 8.2: Model accuracy on CPU (FP32) and NMP (16-bit fixed point) from Glow.

ation. Although an analysis of inference accuracy has been performed with respect to the selected CNN models, a comprehensive performance comparison with TensorFlow/XLA was not conducted, as Glow utilizes, in some cases, a slightly different set of CNN models from the ONNX Model Zoo.

In the accuracy analysis conducted using Glow, several CNN models were either converted from the Google TFLite Hub to the ONNX format (e.g., LeNet, SqueezeNet, and MobileNetV2) or obtained directly from the ONNX Model Zoo (e.g., MNIST and ResNet-18). Accuracies were compared to those achieved on a CPU with 32-bit FP, as listed in their respective repositories (reference numbers for Top-1 and Top-5). On the NMP architecture, inferences for each model were executed using the ImageNet and MNIST validation datasets, depending on the model, and Top-1 and Top-5 accuracies were measured. As depicted in Table 8.2, the accuracies of the code generated by Glow on NMP closely approach those produced by the TensorFlow/XLA compiler (see Table 7.1) for models like SqueezeNet and MobileNetV2, as both are the same CNN models in both compilers. A slight difference in accuracy may occur in both cases as the compilers do not employ the same quantization tool, potentially leading to different Q-point selections. Furthermore, in terms of performance analysis, both compilers exhibit similar performance characteristics since they generate identical code using the same tiling factors (via TSO-burst).

For the ONNX-MLIR compiler, we have only integrated the LeNet model for compilation as a proof of concept of the TSO algorithm. The LeNet used is the same as the one used in TF-XLA but converted into the ONNX model. One difference applied to this compiler was the addition of 8-bit quantization support, which, when enabled for LeNet, reduced the overall execution time from 199 to 130 us (53% faster).

Chapter 9

Related Works

This chapter provides an overview of the related works and compares them with the TSO algorithm. These works are classified into multiple sections, including those focusing on specific target devices (FPGAs, NPUs, CPUs, and GPUs), compiler-based solutions, and, finally, other commonly used techniques such as sparsity and quantization.

9.1 Improving Memory Access and Data Reuse

9.1.1 On FPGAs

Maestro [44] and Timeloop [60] uses analytical modeling that evaluates different mapping configurations – dataflow strategy, data reuse, tile size, etc.; to estimate the run time for different configurations. While Maestro designs some annotations to classify the loops either as temporal or spatial, Timeloop analyzes the nested loops to apply the transformations to them. Given a DNN layer (e.g., a Convolution and its information), hardware configuration (number of PEs, on-chip memory size, etc), and the dataflow strategy, these approaches estimate the runtime performance, energy, and power. Given the easy use of Maestro, different solutions have adopted its annotations to estimate computation [11,38]. As an example, Marvel [11] uses the Maestro notations and has for its main goal the reduction of the search space by decoupling the analysis of the cost model of the accesses to the on-chip/off-chip sub-spaces. Timeloop and Maestro model Spatial DNN Accelerators, i.e., FPGA-based architectures, in which the inner-loops of a Convolution are unrolled and then synthesized into PE array (MAC units) which run in a synchronized fashion to leverage on data-sharing between them through inter-PE communication. Similar to Marvel and Timeloop, TSO also performs cost modeling and design space exploration, but contrary to them, the target architectures are multicore NPUs and not FPGA designs.

To choose a mapping solution from a search space that encompasses millions to billions of possible solutions, cost models based on the theoretical roofline model have been employed for this purpose, as demonstrated in previous works [57, 61]. In the context of [57], a roofline-based cost model is utilized during compile-time to classify candidate solutions based on two key performance metrics: (1) Operational Intensity, which is a metric that determines how data is reused during runtime by analyzing how many operations are performed on each byte retrieved from DRAM, and (2) Attained Performance,

which is a metric that estimates convolution performance in operations per second, based on both convolution complexity and hardware capabilities. The primary objective of the cost model is to select the solution that maximizes both of these metrics among all the verified solutions. In contrast to TSO, the roofline model used in [57] focuses exclusively on the number of memory accesses, without considering memory bursts. This approach aligns with another volume-based solution used in this work, referred to as TSO-noburst. Furthermore, the roofline model presented in [57] was integrated into TSO's search space in a solution called TSO-roof to enable a comparison with TSO-burst.

Tu *et al.* [81] and Hu *et al.* [30] proposed an FPGA-based accelerator capable of reconfiguring its resources to increase data reuse. They used the concepts of Input Stationary (IS), Weight Stationary (WS), and Output Stationary (OS). Besides that, they propose a novel approach called Hybrid Stationary (HS) that leverages on these concepts to find an optimal configuration for each Conv-layer. Although their work has some similarities to TSO, instead of mapping the operations to an array of PEs, TSO considers an architecture with multiple cores (NMP) where each core has an accelerator that runs independently of the others. Besides that, the TSO search space exploration algorithm considers different tile shapes based on memory bursts, not just square shapes that fit into the hardware topology.

Chen *et al.* [14, 15] discusses different ways to map convolutional and fully-connected layers on spatial architectures, which is a class of accelerators that exploit high compute parallelism while allowing direct communication between processing engines (PE). They created a novel approach called Row Stationary (RS) that minimizes data movement and consequently energy consumption by exploiting data reuse (weights and input feature map) through inter-PE communication. It is achieved by respectively sharing each filter row and input feature map row horizontally and diagonally on an array of PEs. Even being a solution tightly connected to spatial architecture, their solution is more energy-efficient (up to 2.5x on Conv-layers) than other strategies of mapping (Weight Stationary (WS), Output Stationary (OS), and No Local Reuse). Their solution shows to be tightly connected to their accelerator.

Peemen *et al.* [64] developed a template accelerator that varies the number of PEs, buffer size, etc., for each layer. Their solution applies optimizations such as tiling and interchange loop transformations to maximize data reuse. The results they achieve are notable: a 13x reduction in memory size while maintaining the same performance and an 11x improvement in execution time. In comparison to their solution, TSO also focuses on improving data reuse. However, instead of generating a different FPGA configuration for each layer, it generates an optimized mapping for each layer.

9.1.2 On NPUs

To improve data reuse, Zhang *et al.* use polyhedral-based optimization techniques [87] for tiling and other related loop transformations. Ma *et al.* [56] describe a performance model that depends solely on the Output Stationary (OS) scheduling to run the convolutions. Stoutchinin *et al.* [77], on the other hand, use a technique called reuse distance, which aims to identify the memory footprint required to accommodate the Convolution's data into

the on-chip memory, which varies up to 512KiB. Through the reuse distance technique, it is possible to identify the amount of data required to keep in the on-chip memories between two accesses to the same data (e.g., tile). They only consider data reuse over the on-chip memories, without taking into consideration DRAM accesses.

King *et al.* [40] introduced an algorithm designed to evaluate the scheduling of multiple Conv-layers, covering from the initiation of a branch to its merge within a CNN model. This approach aims to optimize on-chip memory usage by preserving the input/output data of each layer as much as possible, eliminating the need to store them in DRAM and retrieve them when required later. In their work [40], the authors employed an NPU equipped with 1 MiB of on-chip memory. While this capacity is suitable for numerous applications, it may pose cost challenges for edge inference AI accelerators.

Jung *et al.* [37] developed a mapping solution aimed at reducing synchronization among the multicore NPUs. Their solution also takes into account the TLE slicings explained in this thesis (TLE KS and OUT). However, instead of storing the data in DRAM, it attempts to keep the data in the local memory of the three NPU cores and employs two techniques: (a) halo-exchange, which minimizes accesses to external memory (DRAM) by retaining the required data to compute the next layer in the local on-chip memory and sharing the halo data (boundary data) necessary for the other cores, (b) stratum construction, which computes redundantly to minimize synchronization, working similarly to a pyramid of computation. Their work also considers the construction of a chain of stratum, selected based on a heuristic. They also apply tiling, but for double buffering to overlap external memory access with computation. Additionally, the halo-exchange approach applies overlapping since, while the data is being exchanged between the processing cores, the weight data is loaded from external memory.

The work proposed by Kwon *et al.* [45] shows how to enhance the throughput of CNN models by integrating an NPU into a system-on-chip (SoC) to work together with a CPU and GPU, where the computation on these devices is deployed using linear algebra libraries, such as OpenBLAS. To optimize the training process using multicore NPUs, Kim et. al. [42] shows that the backward pass requires a significant amount of communication with the DRAM, and in order to resolve the issue, a new data reuse pattern is proposed, where the on-chip memories are used for this purpose.

9.1.3 On CPUs

Patabandi *et al.* [63] describes an analytical model that aims to identify from a search space a solution that minimizes data movement overheads for convolutions. Their solution is composed of a cost model that quantifies the number of accesses to the convolution data, starting from the innermost loop and going all the way to the outermost loop. TSO also works by quantifying the memory accesses to the convolution data to decide on a solution through a cost model. But instead of analyzing each loop individually, TSO estimates costs from a tile perspective, where the number of accesses to each tile is calculated based on the scheduling strategy, and the solution that is prioritized is the one that maximizes both data reuse and memory bursts.

Li *et al.* [52] propose a tiling algorithm for CPU that initially assesses how different

permutations (loop orders) of the nested tiling loops may affect data movement. This analysis results in eight different permutations that are evaluated through a cost model that uses a non-linear optimization problem (min-max) to produce tile sizes and tries to reduce the data movement between each cache level. TSO goes in the same direction for NPU, but it works by reducing data movement between the NPU and DRAM, which involves a different architecture when modeling the cost model and data partitioning.

Tollenaere *et al.* [79] designed a solution for CPUs that mitigates edge cases during tiling by employing two distinct micro-kernels, which are referred to as *scheduling* in this work. While the solution proposed in [79] randomly selects which convolution data (e.g., the `IN` tensor) is tiled at each memory hierarchy level, it minimizes the memory transfer overhead for computing edge cases. In contrast, TSO uses a single scheduling approach when tiling the data but identifies, within its search space, which convolution data is more suitable to remain stationary and how to efficiently distribute the data across the TLE/TLTs. Regarding edge cases, TSO aims to minimize their occurrence when running the cost model, classifying solutions with edge cases as inefficient.

Goto and de Gejin [25] proposed an optimized algorithm for General Matrix Multiplication (GEMM) for CPU. The authors introduced the concept of exploiting an optimized micro-kernel (referred to as the *inner-kernel*) by tiling the problem in an external macro-kernel, which comprises a set of loops that iterate over the tiles. These tiles are then packed into a layout suitable for the micro-kernel, which results in the data being laid out in memory for proper loading using SIMD instructions. Tiling is employed to maximize data reuse in L1, L2, and L3 caches. The TSO algorithm employs a similar strategy, but it applies it to convolution instead of GEMM. Additionally, it tiles the convolution data to fit and maximize reuse in the on-chip memories of the NPU multicore accelerator. Unlike GEMM, TSO does not require data packing since it accesses the data without modifying the data layout, which is also known as direct convolution.

9.1.4 On GPUs

Li *et al.* [49] proposed a study for GPUs that explores different data layouts (e.g., NHWC), with the aim of improving memory access. They demonstrated that selecting the proper data layout for GPUs has the potential to improve memory access performed by a warp (i.e., a collection of 32 threads that run simultaneously the same instruction). The authors described the existence of up to 24 data layouts and showed how some of them (e.g., CHWN) can improve memory access by coalescing the accesses within a warp. In the case of NMP, the data layout is NCHW, since the custom RISC-V instructions expect the data to be in this format.

9.2 Compiler-based Solutions

To select different tile sizes, loop order, unroll factor, etc., TVM [13] uses a machine-learning cost model, which does not require hardware information, and periodically learns from previous predictions to search for an improved data tiling and code generation. TVM

depends on Halide [67] for performing loop transformations and other optimizations. Internally, it optimizes the runtime execution for FPGAs by applying optimizations that hide communication within computation. The cost model, which is based on a machine learning model, explores candidate solutions for running measurements, and the results are used to update the model. If the model has not been trained yet, it explores random candidates for measurement; otherwise, the cost model provides solutions for measurement. To the best of our knowledge and from the available public literature, TVM has not shown any results for multicore NPUs like NMP, generating code only for FPGAs, embedded CPUs and server CPUs.

Hummingbird [59] presents a method for mapping operators—both algebraic and algorithmic—onto tensor computations. The execution flow works as follows: (a) a pipeline parser identifies the operators (e.g., LogisticRegression, SVC, etc.) and creates a DAG. Next comes a (b) optimizer, with a list of optimizations for each operator, and also counts with runtime-independent optimizations. Finally, in the last phase, the (c) tensor DAG compiler takes place, and the DAG is traversed in topological order to generate PyTorch's neural network module, which can be subsequently exported to the target format. For the GEMM operator, Hummingbird supports three different strategies, each with different memory and runtime requirements.

9.3 Other Optimization Techniques

The work proposed by Alwani *et al.* [7] and Xiao *et al.* [86] focuses on data-flow across multiple Conv-layers. Instead of processing a layer at a time, as usual, they focus on processing multiple layers at once without generating intermediate data between them. Their solution works by fusing multiple layers resulting in a computation pyramid across those layers. They use some complex data structures to keep the intermediate data of each pyramid. In general, even reducing the memory transfers between the FPGA and host as they do, their accelerator still requires a huge amount of memory to store all the intermediate data from different pyramids. From the perspective of an NPU accelerator with limited on-chip memory, this optimization is not applicable.

Caffeine [88] is a library that can convert Fully Connected Layers (FC-Layers) into Conv-Layer. The conversion considers modifications in the data-layout to reduce the number of accesses to the DRAM to increase the burst length. Qiu *et al.* [66] also modifies the data layout and applies quantization to improve memory access. Putra *et al.* [65] maps the data in the DRAM to reduce row buffer conflicts. TSO uses a similar idea to increase the burst length, but does not rearrange the data layout. The process of modifying the layout proposed in [88] creates a certain complexity when writing a layer's output, given that the layer's output data has to be rearranged again to be accommodated to the next layer's input configuration (e.g., tile size).

It is also possible to reduce data transfers by applying data compression. NullHop [6] does this in hardware, and Han *et al.* [28] does it by applying Huffman Coding. Sparsity is another technique used to avoid computing zero elements and therefore reducing data transfer of unnecessary data besides avoiding unnecessary computation. Such technique

Paper	Q_1 (Tiling)	Q_2 (Scheduling)	Q_3 (Parallelism)	Cost model	Target
[MOTAMEDI et. al., 2016]	Only Tm and Tn are explored	OS	Divides the OFMs over the PEs	Roofline model	FPGA
[TU et. al., 2017]	Square tiling (up to 16x16)	IS, OS and WS	Divides the OFMs over the PEs to avoid idle PEs (POOM strategy)	Dual-objective optimization problem	FPGA
[HU et. al., 2019]	Square tiling (up to 14x14)	IS, OS and WS	Divides the OFMs over the PEs to avoid idle PEs (POOM strategy)	Roofline model	FPGA
[KIM et. al., 2019]	Entire convolution data, (NPU has enough on-chip memory)	OS	Divide the OFMs over 16 processing units	N/A	Multicore NPU
[CHEN et. al., 2019]	Uses Halide	Uses Halide	Uses Halide	Auto-tuning based on ML model (XGBoost)	CPU, GPU and FPGA
[KWON et. al., 2020]	N/A	WS and OS (based on annotations)	Uses annotations to define the loops as either spatial or temporal	MAESTRO (latency, energy, buffer requirement, etc)	FPGA
[SOUSA et. al., 2021]	Burst-based selection	IS, OS and WS	For TLE (input and output parallelism) and TLT (output parallelism)	A burst-based cost model	Multicore NPU

Table 9.1: The TSO algorithm compared to similar works that also seek to minimize DRAM memory access.

is used by several works [27, 28, 50]. Additionally, there are other ways to reduce data transfer. For instance, one can significantly reduce bandwidth and storage by applying quantization (e.g., 8-bit Integer) to the neural network data. Jacob *et al.* [33] present a technique that converts the network data from floating-point to integer format, demonstrating that this enables computing the entire network using only integers (8-bit) with a negligible loss of accuracy. All these techniques could also be used to improve the approach proposed in this thesis, although they are not the focus herein.

9.4 Comparative Analysis of Relevant Works

Among the works presented in this section, those that are closest to the TSO algorithm are summarized in Table 9.1 and compared with respect to the research questions (Q_1-Q_3) presented in Chapter 2.

Chapter 10

Future Works and Conclusion

This chapter discusses the conclusions of the TSO algorithm, an additional contribution inspired by TSO, and finally, provides a brief discussion on future works.

10.1 TSO Algorithm Conclusions

Given the limited on-chip memory capacity in NPU architectures, efficient data partitioning and scheduling techniques are crucial to minimize the cost of accessing DRAM. This thesis introduces TSO, an optimization pass applied during compile-time that identifies the best combination of data tiling, scheduling, parallelism, and MAC operations, which, when combined, minimizes the convolution execution time of CNN models. To achieve this, TSO first focuses on addressing Q_1 (refer to Chapter 2) and implements an efficient cost model based on memory bursts, resulting in a speedup of up to 21.7% for typical CNN models compared to non-burst modeling. Concerning Q_2, TSO identifies the best TLT scheduling strategy (out of 3) that leads to a speedup of up to 39.8% compared to a fixed TLT scheduling strategy. Finally, regarding Q_3, compared to a fixed TLE slicing scheme, TSO searches for the best scheme (out of 3) that improves performance, resulting in a speedup of up to 41.0%. When comparing individual convolutions (238 in total), TSO with burst modeling shows speed improvements in 73% of the cases compared to non-burst modeling, with the highest achieved speedup of 3.69x. Furthermore, when compared to an existing solution adapted from an analytical model based on Roofline [57], TSO with the burst modeling optimization outperforms all CNN models with speedups of up to 29.2%, where 84% of the convolutions perform faster, with the highest speedup achieved being 5.19x. Although TSO was initially deployed in the TF-XLA compiler, its generality was also evaluated by porting and running it on the Glow and ONNX-MLIR compilers.

10.2 SConv: Inspired by TSO

The TSO algorithm served as an inspiration for a CPU-based solution known as SConv [21]. The SConv algorithm introduces a series of optimizations, starting with the Convolution Slicing Analysis (CSA). CSA is a tiling algorithm that initially determines the tiles,

along with their sizes, and subsequently identifies the optimal distribution of the input, weight, and output tiles within a memory hierarchy consisting of three levels of cache and a DRAM, all aimed at enhancing data reuse. Instead of retrieving data from a more distant cache, CSA prioritizes data reuse within the L1 cache. When this is not feasible, it seeks to optimize data reuse within the L2 cache. CSA also considers the L3 cache and DRAM in its modeling. Additionally, SConv introduces Convolution Slicing Optimization (CSO), which defines the macro-kernel, which stands for a set of loops designed to iterate over the tiles that are distributed across the memory hierarchy, as determined by CSA. Additionally, SConv defines various data packing strategies that aim to improve memory access by reorganizing the data layout before running the micro-kernel. The micro-kernel is a highly optimized routine designed for the target architecture, leveraging all available architectural capabilities, such as SIMD AVX512, for x86, and MMA, for POWER10. SConv was evaluated on x86 and POWER10 systems, achieving a speedup of up to 34% when compared to the well-known im2col+GEMM solution, and a speedup of up to 25% when compared to highly optimized libraries, such as oneDNN and oneMKL.

10.3 Future Works with Potential to Improve TSO

Several potential factors that can enhance TSO performance when running CNN models on NMP are listed below:

1. **Extended TLT Partitioning** – The TLT partitioning involves distributing small partitions formed from a TLE slice among the TLT cores associated with that TLE, and this is done by dividing the filters in the weight (`KS`) slice evenly among the TLT cores (for detailed information, refer to Section 6.3). This approach is generally efficient for most convolutions, as after the TLT partitioning, the number of filters assigned to the cores becomes small, and in some cases, small enough to fit in a single tile ($\texttt{KS}^T$) that fits in the TLT's MB1 on-chip memory. Thus, even having every TLT core loading their $\texttt{KS}^T$ tiles independently of the others, the loading time is not significantly impacted due to the size of the tiles. As for the TLE slice of the input (`IN`) tensor, the loads from DRAM to bring the $\texttt{IN}^T$ tiles onto the on-chip memory use multicast loads, thus reducing the impact of the loading time. On the other hand, there are cases where even after the TLT partition, the weight tiles ($\texttt{KS}^T$) distributed to the cores remain significantly larger than that of the input tiles ($\texttt{IN}^T$) so that applying multicast loads to load the weight tiles (rather than the input tiles) would considerably improve performance. Therefore, extending TSO's search space to include the division of the input slice in the TLT partitioning step, allowing for multicast loads from the weight tiles, would substantially enhance performance. Achieving this would require an extension of TSO, along with the addition of other kernel implementations designed for convolution on NMP.

2. **Overlap Communication with Computation** – The kernels responsible for executing the CNN operations on NMP follow a pipeline approach. Initially, the input tile ($\texttt{IN}^T$) is loaded into the on-chip memory, followed by the loading of the weight

tile ($\texttt{KS}^T$). Subsequently, the operation computation takes place, resulting in the generation of an output tile ($\texttt{OUT}^T$), which is then stored in DRAM. This process repeats for each remaining tile of the input and weight tensors. Although the NMP board used in this work is single-ported, one potential optimization to reduce a few cycles of the overall execution time involves initiating the loading of the next tile (the non-stationary one) while the current tile is being computed by the MAC unit. To implement this, the on-chip memory serving the non-stationary tile could be divided into two parts. The first part would be used for the current tile, while the other would be used to load the next tile. However, it is important to note that while this approach could offer some benefits, it may not significantly improve performance due to the pressure imposed on the internal data bus within NMP. Additionally, adapting such a solution to TSO would require changes to the internal kernels responsible for implementing the operations.

3. **A More Precise Cost Model** – The cost model used in TSO-burst estimates the transfer time for loading/storing data from/to DRAM, respectively, and also estimates CAS (Column Address Strobe) latency by counting the number of memory bursts (refer to Section 6.5 for more details). This estimation could be significantly improved by incorporating other critical DRAM latencies, such as Trcd (Row Address to Column Address Delay), Trp (Row Precharge Time), and Tras (Row Active Time). Additionally, TSO does not make any assumption of any data alignment in DRAM, potentially leading to underutilized memory bursts, as the DRAM's row buffer might not be filled as expected. To implement these changes, one could modify the TSO-burst's cost model to incorporate these additional DRAM timings, taking into account the alignment of tensors in memory. Furthermore, one could optimize the allocation of weight tensors in memory (determined at compile-time) to fully leverage these improvements.

4. **Heuristics to Improve the Tiling Selection** – The TSO optimization involves an exhaustive search through every single solution within its search space. TSO evaluates these solutions using its cost model while obeying the restrictions imposed by NMP (e.g., for stacked convolution, only $1/4^{th}$ of the on-chip memory MB2 for the output ($\texttt{OUT}^T$) tile is made available for use). However, within this vast search space, some solutions may be highly inefficient and, therefore, unnecessary to test. For example, selecting tiles with spatial dimensions of 1x1 is a solution that TSO evaluates, despite its known inefficiency. Moreover, evaluating these inefficient solutions, especially in cases where TSO's search space is extended to combine more solutions, such as the ones described above, may result in more combinations, and consequently in an inefficient exploration. To enhance the exploration of solutions within the TSO's search space, heuristics can be applied to exclude solutions that would never be selected, thus resulting in a more streamlined optimization process. An example of a heuristic that could be applied is Simulated Annealing [18], which could reduce the exploration of candidates in the search space with small tiles.

5. **Full Code Generation for NMP in MLIR** – The most costly operations in NMP are the data transfers between on-chip memories and DRAM. However, RISC-V instructions, responsible for tasks such as iterating over the tiles through a set of loops, managing semaphores, and making critical runtime decisions – such as determining padding or stride – also consume a significant portion of the execution time (refer to Section 7.1 for more detail). To mitigate the time spent on RISC-V instructions, a potential approach is to implement the kernels that execute the CNN operations in MLIR/LLVM. This would enable all decisions to be made at compile-time, which would allow, for example, the elimination of unnecessary loops. In order to have this implemented, it is essential to define the NMP Backend in the LLVM compiler. The NMP library used in this work was previously compiled with a compatible gcc compiler before running the ML compiler with the TSO algorithm enabled.

6. **Towards a future version of NMP** – The NMP architecture used in this thesis contains only three MBLOB on-chip memories (MB0, MB1 and MB2). Expanding the architecture with additional on-chip memory (e.g., MB4) on each TLT would enable the overlap of communication with computation. This means that it would be possible to load the next non-stationary tile (such as the next $\texttt{IN}^T$ tile in the WS scheduling) while the current one is being computed. Once the computation of the current tile is complete, NMP could switch the computation to the next tile, which is already loaded and available in the extended MBLOB (e.g., MB4), and then the same process could repeat in a ping-pong manner with the other MBLOB that was previously being used to place the non-stationary tile. Implementing this approach would significantly reduce loading time, but it would also require the addition of a new port to enable double buffering. Another possible approach involves incorporating shared memory between the TLEs, as there are TLEs that require at runtime either the same `IN` or `KS` slices as the others, and employing shared memory would allow a similar behavior as the multicast loads between the TLTs of a single TLE. Finally, another extension could involve adding a DME on each TLT. The current DME on NMP, which is available on each TLE, serializes the loads of the TLTs within that TLE through a queue.

Bibliography

[1] ImageNet. http://www.image-net.org/challenges/LSVRC/2012/, 2023. Accessed: 2023-09-20.

[2] MNIST database. http://yann.lecun.com/exdb/mnist/, 2023. Accessed: 2023-09-20.

[3] Xla: Optimizing compiler for machine learning. https://www.tensorflow.org/xla, 2023. Accessed: 2023-09-20.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Osdi*, volume 16, pages 265–283. Savannah, GA, USA, 2016.

[5] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

[6] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, et al. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE transactions on neural networks and learning systems*, 30(3):644–656, 2018.

[7] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 22. IEEE Press, 2016.

[8] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[9] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. Origami: A convolutional network accelerator. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 199–204. ACM, 2015.

[10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 247–257. ACM, 2010.

[11] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric compiler for dnn operators on spatial accelerators. *arXiv preprint arXiv:2002.07752*, 2020.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[14] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 367–379. IEEE Press, 2016.

[15] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2016.

[16] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[17] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International conference on artificial neural networks*, pages 281–290. Springer, 2014.

[18] Lawrence Davis. Genetic algorithms and simulated annealing. 1987.

[19] ONNX Runtime developers. Onnx runtime. `https://onnxruntime.ai/`, 2021.

[20] Naznin Fauzia, Louis-Noël Pouchet, and P Sadayappan. Characterizing and enhancing global memory data coalescing on gpus. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 12–22. IEEE, 2015.

[21] Victor Ferrari, Rafael Sousa, Marcio Pereira, João PL de Carvalho, José Nelson Amaral, José Moreira, and Guido Araujo. Advancing direct convolution using convolution slicing optimization and isa extensions. *arXiv preprint arXiv:2303.04739*, 2023.

[22] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[23] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. Snowflake: A model agnostic accelerator for deep convolutional neural networks. *arXiv preprint arXiv:1708.02579*, 2017.

[24] Gabriel Resende Gonçalves, Sirlene Pio Gomes da Silva, David Menotti, and William Robson Schwartz. Benchmark for license plate character segmentation. *Journal of Electronic Imaging*, 25(5):053034, 2016.

[25] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), may 2008.

[26] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 3(1), 2010.

[27] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[28] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[30] Xianghong Hu, Yuhang Zeng, Zicong Li, Xin Zheng, Shuting Cai, and Xiaoming Xiong. A resources-efficient configurable accelerator for deep convolutional neural networks. *IEEE Access*, 2019.

[31] Hai Huang, Kang G Shin, Charles Lefurgy, and Tom Keller. Improving energy efficiency by making dram less randomly accessed. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 393–398, 2005.

[32] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[33] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[34] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2010.

[35] Tian Jin, Gheorghe-Teodor Bercea, Tung D Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, et al. Compiling onnx neural network models using mlir. *arXiv preprint arXiv:2008.08272*, 2020.

[36] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[37] Hanwoong Jung, Hexiang Ji, Alexey Pushchin, Maxim Ostapenko, Wenlong Niu, Ilya Palachev, Yutian Qu, Pavel Fedin, Yuri Gribov, Heewoo Nam, et al. Accelerating deep neural networks on mobile multicore npus. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 236–248, 2023.

[38] Sheng-Chun Kao and Tushar Krishna. Gamma: automating the hw mapping of dnn models on accelerators via genetic algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[39] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[40] Doyun Kim, Kyoung-Young Kim, Sangsoo Ko, and Sanghyuck Ha. A simple method to reduce off-chip memory accesses on convolutional neural networks. *arXiv preprint arXiv:1901.09614*, 2019.

[41] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. *ACM SIGARCH Computer Architecture News*, 44(3):380–392, 2016.

[42] Jungwoo Kim, Seonjin Na, Sanghyeon Lee, Sunho Lee, and Jaehyuk Huh. Improving data reuse in npu on-chip memory with interleaved gradient order for dnn training. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 438–451, 2023.

[43] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[44] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[45] Jinse Kwon, Jemin Lee, and Hyungshin Kim. Pipelining of a mobile soc and an external npu for accelerating cnn inference. *IEEE Embedded Systems Letters*, 2023.

[46] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[47] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[48] Sung-Jin Lee, Sang-Soo Park, and Ki-Seok Chung. Efficient simd implementation for accelerating convolutional neural network. In *Proceedings of the 4th International Conference on Communication and Information Processing*, pages 174–179, 2018.

[49] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing memory efficiency for deep convolutional neural networks on gpus. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 633–644. IEEE, 2016.

[50] Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, and Xiaowei Li. Squeezeflow: A sparse cnn accelerator exploiting concise convolution rules. *IEEE Transactions on Computers*, 68(11):1663–1677, 2019.

[51] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[52] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. Analytical characterization and design space exploration for optimization of cnns. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 928–942, 2021.

[53] Siwu Liu, Ji Hwan Park, and Shinjae Yoo. Efficient and effective graph convolution networks. In *Proceedings of the 2020 SIAM International Conference on Data Mining*, pages 388–396. SIAM, 2020.

[54] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Hai Li, Yiran Chen, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, et al. Reno: A high-efficient reconfigurable neuromorphic computing accelerator design. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[55] Shuai Lu, Jun Chu, and Xu T Liu. Im2win: Memory efficient convolution on simd architectures. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2022.

[56] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-Sun Seo. Performance modeling for cnn inference accelerators on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4):843–856, 2019.

[57] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580. IEEE, 2016.

[58] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.

[59] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 899–917, 2020.

[60] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[61] Chan Park, Sungkyung Park, and Chester Sungchung Park. Roofline-model-based design space exploration for dataflow techniques of cnn accelerators. *IEEE Access*, 8:172509–172523, 2020.

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[63] Tharindu R Patabandi, Anand Venkat, Rajkishore Barik, and Mary Hall. Swirl++: Evaluating performance models to guide code transformation in convolutional neural networks. In *Languages and Compilers for Parallel Computing: 32nd International Workshop, LCPC 2019, Atlanta, GA, USA, October 22–24, 2019, Revised Selected Papers 32*, pages 108–126. Springer, 2021.

[64] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 13–19. IEEE, 2013.

[65] Rachmad Vidya Wicaksana Putra, Muhammad Abdullah Hanif, and Muhammad Shafique. Drmap: A generic dram data mapping policy for energy-efficient processing of convolutional neural networks. *arXiv preprint arXiv:2004.10341*, 2020.

[66] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

[67] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[68] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. *ACM SIGARCH Computer Architecture News*, 28(2):128–138, 2000.

[69] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[70] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[71] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A massively parallel coprocessor for convolutional neural networks. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 53–60. IEEE, 2009.

[72] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[73] Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3626–3633, 2013.

[74] Sergio Montazzolli Silva and Claudio Rosito Jung. Real-time brazilian license plate detection and recognition using deep convolutional neural networks. In *2017 30th SIBGRAPI conference on graphics, patterns and images (SIBGRAPI)*, pages 55–62. IEEE, 2017.

[75] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[76] Mingcong Song, Yang Hu, Yunlong Xu, Chao Li, Huixiang Chen, Jingling Yuan, and Tao Li. Bridging the semantic gaps of gpu acceleration for scale-out cnn-based big data processing: Think big, see small. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 315–326. ACM, 2016.

[77] Arthur Stoutchinin, Francesco Conti, and Luca Benini. Optimally scheduling cnn convolutions for efficient memory access. *arXiv preprint arXiv:1902.01492*, 2019.

[78] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[79] Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P Sadayappan, and Fabrice Rastello. Autotuning convolutions is easier than you think. *ACM Transactions on Architecture and Code Optimization*, 20(2):1–24, 2023.

[80] Fengbin Tu, Weiwei Wu, Shouyi Yin, Leibo Liu, and Shaojun Wei. Rana: towards efficient neural acceleration with refresh-optimized embedded dram. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 340–352. IEEE Press, 2018.

[81] Fengbin Tu, Shouyi Yin, Peng Ouyang, Shibin Tang, Leibo Liu, and Shaojun Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2220–2233, 2017.

[82] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. 2011.

[83] Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*, pages 19–24. IEEE, 2017.

[84] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[85] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. *ACM SIGPLAN Notices*, 48(8):57–68, 2013.

[86] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[87] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[88] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[89] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. In *International Conference on Machine Learning*, pages 5776–5785. PMLR, 2018.

[90] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Advances in neural information processing systems*, pages 487–495, 2014.