



الجمهورية الجزائرية الديمقراطية الشعبية
People's Democratic Republic of Algeria
وزارة التعليم العالي والبحث العلمي
Ministry of Higher Education and Scientific Research
المدرسة الوطنية العليا للتكنولوجيات المتقدمة
National Higher School of Advanced Technologies



Department of Electrical Engineering and Industrial Computing

Final Year Project to Obtain the Diploma of Engineering

- Field -

Electronics

- Specialty -

Embedded Systems

- Subject -

Automatic Extraction of the Inference Structure of a CNN as a DAG for Temporal Analysis of Many-Core Systems

Realized by :

LOUNNACI Sarah

Presented publicly, the 11/09/2025

Members of the Jury :

REBAI Karima	MCA	ENSTA	President
BENDOUDA Djamila	MCB	ENSTA	Supervisor
HLADIK Pierre-Emmanuel	MCF HDR	Univ. Nantes/Lab. LS2N	Supervisor
MAIZA Claire	MCF HDR	Univ. Grenoble-Alpes/Lab. VERIMAG	Supervisor
ABBADI Belkacem	MCB	ENSTA	Examiner
ZELLAT Khadidja	MCA	ENSTA	Examiner

Academic year 2024-2025

Dedication

To myself —

I rose like a phoenix from a time of darkness. After losing an opportunity I had worked hard for, I found myself in the depths of despair; I nearly abandoned everything. But I stood back up. I rebuilt, I persevered, and I found strength within when the path ahead seemed lost. This project is, above all, ***a testament to that resilience.***

To my little family —

You know who you are. Your unwavering support has carried me through more than words can express. Thank you for your love and for always being there, even in the hardest times...

To my friends —

You know who you are. The ones who stood by me and believed in me, I sincerely appreciate each of you.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Mr. Pierre-Emmanuel Hladik for his trust in me. His acceptance to supervise the project online greatly encouraged me and gave me the motivation to progress confidently. His insightful guidance, technical expertise, and continued support have been essential to the successful completion of this work.

I would also like to thank Mr. Abderaouf Nassim Amalou for his kind words and encouragement during my very first difficult moments. His support helped me stay focused and resilient.

My gratitude extends as well to Mrs. Claire Maiza for her valuable advice and mentorship throughout this project. Her insightful feedbacks helped me stay on track and improve the quality of my work significantly.

At ENSTA school, I would like to give a special thanks to Mrs. Djamila Bendouda for her continuous assistance and encouragement. Her motivational emails and daily support gave me the energy and determination to push forward. Her orientation and presence made a real impact on both my morale and my progress.

To all the teachers at ENSTA who supported me, especially during challenging periods, I am deeply thankful. There are many whom I hold in high regard, and while I cannot name them all individually, they each hold a special place in my heart. They know perfectly who they are, and I sincerely thank them for their dedication and kindness.

I also wish to extend my appreciation specifically to the administration and staff of the GEII department only. Their professionalism and support played an important role in shaping my academic experience.

Finally, I thank every person who has contributed to my learning journey and helped shape the Sarah Lounnaci I am today.

A heartfelt thank you to all...

Abstracts

الملخص

تُعتبر الشبكات العصبية الالتفافية (CNN) أداة لا غنى عنها في تطبيقات الذكاء الاصطناعي بفضل قدرتها على استخراج الخصائص الهرمية بشكل آلي. ومع ذلك، فإن استخدامها في الأنظمة المدججة والزمن الحقيقي لا يقتصر على عامل الدقة فقط، بل يتطلب أيضاً قابلية التنبؤ الزمني وضمان احترام المهل الزمنية الصارمة. يعالج هذا العمل هذا التحدي من خلال ربط استدلال الشبكات العصبية بالتحليل الزمني عبر تحويلها بشكل منهجي إلى رسوم بيانية لا دورية موجهة (DAG).

المساهمة الرئيسية لهذا المشروع هي تصميم برنامج Designer Network Inference Model MIND الذي يستخرج تلقائياً بنية الاستدلال لشبكة CNN ويحوّلها إلى DAG متوافق مع أداة Analysis Interference Multicore MIA. ولتمثيل استراتيجيات تنفيذ مختلفة، قمنا بتطوير مجموعة من سياسات التجميع المبتكرة مثل: النمذجة الماكروسكوبية على مستوى الطبقات (MLWM)، التقسيم الفضائي المنظم (SSP)، الالتفاف المتكرر عبر تقسيم القنوات (RCCT)، والدمج الكثيف (DF). هذه السياسات توازن بين الحساب والوصول إلى الذاكرة وتكاليف الاتصال بشكل متوازٍ، مما يسمح بدراسة تأثيرها على معالجات MPPA Kalray متعددة النوى.

كما اقترحنا نماذج تكلفة مرتبطة بهندسة معالجات Bostan Kalray بـ 256 نواة و Coolidge بـ 80 نواة ومعالج مساعد TCA. من خلال صياغة دقيقة لأسوأ وقت تنفيذ (WCET) وأقصى وصول للذاكرة (WCMA) عند دقات مختلفة FP32، FP16، INT8، وفرنا نموذجاً حسابياً نسبياً دقيقاً يمكن من حساب أسوأ زمن استجابة (WCRT) باستخدام أداة MIA. وقد أثبتت مرحلة الاختبار، التي استخدم فيها نموذج CNN أولي مع عدة تغييرات في المعلمات، جدوى منهجيتنا وأظهرت التوازنات بين التسريع، كفاءة الحساب، التداخل، تكاليف النقل بين الذاكرة، إلخ.

يثبت هذا العمل إمكانية إجراء تحليل منهجي للشبكات العصبية الالتفافية في سياق أنظمة الزمن الحقيقي على معالجات متعددة النوى، من خلال سلسلة متكاملة: من التحويل CNN إلى DAG، مروراً بتطبيق سياسات التجميع، ونمذجة التكاليف، وصولاً إلى توليد ملفات YAML والتحليل الزمني باستخدام MIA. كما تؤكد النتائج صلاحية المنهجية المقترحة وتفتح آفاقاً مستقبلية لدعم نماذج أكبر وأكثر تعقيداً.

الكلمات المفتاحية: التعلم العميق، الشبكات العصبية الالتفافية، الرسم البياني غير الدوري الموجه، الأنظمة الزمن الحقيقي، Kalray، MPPA، WCET، WCMA، WCRT.

Abstract

Convolutional Neural Networks (CNNs) have become indispensable in Artificial Intelligence applications due to their ability to automatically extract hierarchical features. However, when deployed in real-time and embedded systems, accuracy is not the only metric of interest. In such environments, timing predictability and the guarantee of meeting strict deadlines are essential. This work addresses this challenge by bridging Deep Learning inference with real-time analysis through the systematic transformation of CNN models into Directed Acyclic Graphs (DAGs).

The primary contribution of this project is the design of MIND Model Inference Network Designer software, which automatically extracts the inference structure of a CNN and converts it into a DAG compatible with the MIA Multicore Interference Analysis tool. To capture different execution strategies, we developed a set of novel and innovative *regrouping policies*, such as Macroscopic Layer-Wise Modeling (MLWM), Structured Spatial Partitioning (SSP), Redundant Convolution via Channel Tiling (RCCT), and Dense Fusion (DF). These policies balance computation, memory access, and communication overheads in parallel, making it possible to analyze their impact on the many-core Kalray MPPA architectures.

Furthermore, we proposed architecture-aware cost for Kalray’s architectures: Bostan 256 cores and Coolidge 80 cores with TCA co-processor. By formalizing the Worst-Case Execution Time (WCET) and Worst-Case Memory Access (WCMA) at different precisions FP32, FP16, INT8, we provided a relatively accurate computation model that enables Worst-Case Response Time (WCRT) computation through MIA. The testing phase validates our methodology with a prototype CNN and multiple parameter variations, highlighting the trade-offs between speedup, compute efficiency, interference, memory transfer costs, etc.

This work demonstrates the feasibility of systematically analyzing CNNs for Real-Time Systems on many-core processors, achieving a complete workflow: from CNN to DAG extraction, to regrouping policy application, cost modeling, YAML generation, and MIA-based timing analysis. The outcomes not only validate the proposed approach but also open perspectives for extending analysis to larger models.

Keywords: Deep Learning, Convolutional Neural Networks, Directed Acyclic Graph, Real-Time Systems, Kalray MPPA, WCET, WCMA, WCRT.

Résumé

Les réseaux de neurones convolutifs (CNN) sont devenus indispensables dans les applications d'intelligence artificielle grâce à leur capacité à extraire automatiquement des caractéristiques hiérarchiques. Cependant, lorsqu'ils sont déployés dans des systèmes embarqués et temps réel, la précision n'est pas la seule métrique importante. Dans ce contexte, la prévisibilité temporelle et le respect strict des délais sont essentiels. Ce travail répond à ce défi en reliant l'inférence des CNN à l'analyse temps réel par la transformation systématique des modèles CNN en graphes acycliques dirigés (DAG).

La principale contribution de ce projet est la conception du logiciel MIND Model Inference Network Designer, qui extrait automatiquement la structure d'inférence d'un CNN et la convertit en un DAG compatible avec l'outil MIA Multicore Interference Analysis. Pour capturer différentes stratégies d'exécution, nous avons développé un ensemble de *politiques de regroupement* innovantes, telles que la modélisation macroscopique par couche (MLWM), le partitionnement spatial structuré (SSP), la convolution redondante par découpage de canaux (RCCT) et la fusion dense (DF). Ces politiques équilibrent le calcul, l'accès mémoire et les coûts de communication en parallèle, ce qui permet d'analyser leur impact sur les architectures pluri-cœurs Kalray MPPA.

Nous avons également proposé des modèles de coût adaptés aux architectures de Kalray : Bostan 256 cœurs et Coolidge 80 cœurs avec co-processeur TCA. En formalisant le pire temps d'exécution (WCET) et le pire accès mémoire (WCMA) à différentes précisions FP32, FP16, INT8, nous avons fourni un modèle de calcul relativement précis permettant l'estimation du pire temps de réponse (WCRT) à l'aide de MIA. La phase de test, réalisée sur un CNN prototype avec plusieurs variations de paramètres, valide notre méthodologie et met en évidence les compromis entre accélération, efficacité de calcul, interférences, coûts de transfert mémoire, etc.

Ce travail démontre la faisabilité d'une analyse systématique des CNN pour les systèmes temps réel sur processeurs pluri-cœurs, en proposant une chaîne complète : extraction CNN en DAG, application de politiques de regroupement, modélisation des coûts, génération de fichiers YAML et analyse temporelle via MIA. Les résultats valident l'approche proposée et ouvrent des perspectives pour l'extension à des modèles plus complexes.

Mots-clés : Apprentissage profond, Réseaux de neurones convolutifs, Graphe acyclique dirigé, Systèmes temps réel, Kalray MPPA, WCET, WCMA, WCRT.

Table of Contents

Dedication	i
Acknowledgements	ii
Abstracts	iii
General Introduction	1
1 Essential Concepts and Preliminaries	5
1.1 Introduction	6
1.2 Mathematical Formulation of CNN Layers	6
1.2.1 Convolution Layer (Conv2D)	6
1.2.2 Pooling Layer (Pool2D)	7
1.2.3 Flatten Layer	8
1.2.4 Dense/FC Layer	9
1.2.5 Activation Functions	9
1.3 CNN Model to DAG Topology	11
1.4 Overview of RTS	13
1.5 Kalray MPPA Architecture: Bostan vs. Coolidge	14
1.5.1 Kalray MPPA2-256 “Bostan” Architecture	15
1.5.2 Kalray MPPA3-80 “Coolidge” Architecture	17
1.6 Operating Principle of MIA Tool	20
1.6.1 Interference Origins	22
1.6.2 Explanation of MIA Algorithm	22
1.7 Conclusion	23
2 Methodology and Functional Description	25
2.1 Introduction	26
2.2 End-to-End Pipeline Overview	26
2.3 Used Format Selection	27
2.3.1 .JSON (JavaScript Object Notation)	27
2.3.2 .PY (Python)	27
2.3.3 .YAML (Yet Another Markup Language)	27
2.3.4 .PDF (Portable Document Format)	28
2.4 UML Use-Case Driven Methodology	28
2.4.1 First Step	29
2.4.2 Second Step	29
2.4.3 Third Step	30
2.5 MIND Software Demonstration	31
2.6 Conclusion	33

3	Introducing DAG Nodes Regrouping Policies	34
3.1	Introduction	35
3.2	Initial Methods for Representing CNNs as DAGs	35
3.2.1	Massively Merged Method (MMM)	35
3.2.2	Massively Decomposed Method (MDM)	36
3.2.3	Motivation for Advanced Regrouping Policies	36
3.3	Regrouping Policy 1: Macroscopic Layer-Wise Modeling (MLWM)	37
3.4	Regrouping Policy 2: Structured Spacial Partitioning (SSP)	37
3.4.1	Design Principle and Local Padding Strategy (LPS)	37
3.4.2	Overlap Padding Strategy (OPS)	38
3.5	Regrouping Policy 3: Redundant Convolution via Channel Tiling (RCCT)	40
3.5.1	Partitioning Principle	40
3.5.2	Channel Padding Strategy (CPS)	41
3.6	Regrouping Policy 4: Dense Fusion (DF)	41
3.6.1	Formal Description and Neuron Padding Strategy (NPS)	42
3.6.2	Group Computation	42
3.6.3	Masking Strategy (MS)	43
3.7	Graphical Representation of All the Advanced Regrouping Policies	44
3.7.1	Regrouping Policy 1: MLWM	45
3.7.2	Regrouping Policy 2: SSP	46
3.7.3	Regrouping Policy 3: RCCT	48
3.7.4	Regrouping Policy 4: DF	50
3.8	Regrouping Policies Workflow Rrepresentation	51
3.9	Conclusion	52
4	Architecture-Aware Cost Modeling on Kalray MPPA	53
4.1	Introduction	54
4.2	Cost Analysis of CNN Metrics	55
4.2.1	Convolution Layer	55
4.2.2	Pooling Layers	55
4.2.3	Flatten Layer	56
4.2.4	Dense Layer	56
4.2.5	Activation Functions	56
4.2.6	Modeling WCET in Isolation	57
4.2.7	Modeling WCMA	58
4.3	Cost Analysis of the Initial Methods	61
4.3.1	Modeling WCET in MMM	62
4.3.2	Modeling WCMA in MMM	63
4.4	Cost Analysis of the Advanced Regrouping Policies	64
4.4.1	MLWM	64
4.4.2	SSP	66
4.4.3	RCCT	66
4.4.4	DF	67
4.5	DDR Transfer to Clusters Analysis	70
4.5.1	Assumptions	70
4.5.2	Hardware Setup	71
4.5.3	Cluster Memory Organization	71
4.5.4	Per-Layer Parameter & Hyper-Parameter Counts	71

4.5.5	WCMA Cost for DDR Transfers	72
4.5.6	SRAM Recharge per Layer	72
4.5.7	Example: First Convolution Layer on Bostan	73
4.5.8	Important Remarks	74
4.6	Data Transfers Analysis Between Cores Within and Across Clusters . . .	74
4.6.1	Data Volume Calculation	75
4.6.2	Per-Layer Operations Volume	75
4.6.3	Transfer Cost Calculation	76
4.6.4	Average Inter-Cluster Latency	76
4.7	Establishing Task Execution Order for Deterministic Inference	77
4.7.1	Formalization of Intra-Core Task Ordering	77
4.7.2	Illustrative Example	78
4.8	Period and Deadline Configuration	79
4.8.1	Converting Cycles to Time in Both Architectures	79
4.8.2	Frequency Ranges	79
4.9	Conclusion	79
5	Test and Evaluation	81
5.1	Introduction	82
5.2	Pre-Testing Phase	82
5.3	Evaluation Metrics	84
5.3.1	Runtime Cost Performance	84
5.3.2	Structural Expansion	86
5.4	Testing with MMM_MMM	86
5.5	Testing with MLWM_MMM	87
5.6	Testing with SSP_MMM	88
5.7	Testing with RCCT_MMM	89
5.8	Testing Previous CPLs Using DF in the MLP	90
5.9	Testing with Input Tensor Variation Using MMM in the MLP	93
5.10	Testing with Convolution Hyper-Parameters Variation Using MMM in the MLP	94
5.10.1	Filters Variation	94
5.10.2	Kernels Variation	96
5.11	Projection on Coolidge Architecture	98
5.11.1	Using FP32 Data (TCA Exclusion)	98
5.11.2	Using FP16 Data (TCA Inclusion)	99
5.12	Evaluation and Discussion	100
5.12.1	Analysing in Bostan	101
5.12.2	Analysing in Coolidge & Coolidge+TCA	101
5.12.3	Policy Recommendation	102
5.12.4	Additional Practical Considerations	102
5.13	Conclusion	102
	General Conclusion	104
	References	107
	Appendices	109

A	Details of WCMA Calculation	110
A.1	Bostan WCMA Calculation	110
A.1.1	Convolution Operation	110
A.1.2	Pooling Operation	111
A.1.3	Dense Operation	111
A.1.4	Softmax Operation	111
A.2	Coolidge WCMA Calculation	112
A.2.1	Convolution Operation	112
A.2.2	Pooling Operation	112
A.2.3	Dense Operation	113
A.2.4	Softmax Operation	113

List of Figures

Figure 1.1	DAG representation of an FFNN CNN structure	12
Figure 1.2	Timeline of a periodic task	13
Figure 1.3	Deadline configuration in RTS	14
Figure 1.4	Simplified representation of Kalray MPPA2-256 Bostan architecture	15
Figure 1.5	Demonstration simple RR arbitration	16
Figure 1.6	Bostan request arbitration to an SMEM Bank	16
Figure 1.7	Simplified representation of Kalray MPPA3-80 Coolidge architecture	18
Figure 1.8	Demonstration simple SAP arbitration	19
Figure 1.9	Coolidge request arbitration to an SMEM Bank	19
Figure 1.10	Workflow of MIA tool process	21
Figure 1.11	Interference caused by bus arbitration for SMEM access in many- core systems	22
Figure 1.12	Cursor mechanism for task execution timeline	23
Figure 2.1	End-to-end workflow pipeline of our project	26
Figure 2.2	Global UML use-case overview of the three-step workflow	28
Figure 2.3	UML use-case representation of the First Step; CNN-to-DAG con- struction	29
Figure 2.4	UML use-case representation of the Second Step; Global WCRT computation	30
Figure 2.5	UML use-case representation of the Third Step; Evaluation & re- porting	30
Figure 2.6	Illustration of the MIND software	31
Figure 2.7	Visualization of the MIND environment showing the Coolidge architecture	32
Figure 2.8	Partial view of a DAG automatically generated from a CNN model using the MIND environment.	32
Figure 3.1	DAG representation of a CNN architecture in MMM method	35
Figure 3.2	DAG representation of a CNN architecture in MDM method	36
Figure 3.3	Simple example of a CNN architecture	45
Figure 3.4	DAG representation of the MLWM policy in CPL combined with the MDM policy in MLP	45
Figure 3.5	DAG representation of the MLWM policy in CPL combined with the MDM policy in MLP with separate activation layers	46
Figure 3.6	Example of a CNN architecture using the SSP policy	47
Figure 3.7	DAG representation of the SSP policy	47
Figure 3.8	Pool1–Conv2 layers transition under RCCT policy	49
Figure 3.9	DAG representation of Pool1–Conv2 layers transition under RCCT policy	50

Figure 3.10 DAG representation of the DF policy.	50
Figure 3.11 Workflow representation of the used regrouping policies	51
Figure 5.1 3D representation of the prototype CNN model's operation layers generated by MIND	83
Figure 5.2 3D representation of the prototype CNN model's feature map layers generated by MIND	84

List of Tables

Table 1.1	Comparison of key architectural properties relevant to our project between Bostan MPPA2-256 and Coolidge MPPA3-80	20
Table 4.1	Summary of required notation	54
Table 4.2	Calculation costs of activation functions	56
Table 4.3	Computational throughput per core : Bostan vs. Coolidge	57
Table 4.4	WCET in isolation per output element for CNN layers operation	57
Table 4.5	WCET in isolation per output element for activation functions	58
Table 4.6	Bus width from SMEM to core: Bostan vs. Coolidge	59
Table 4.7	WCMA transfers from SMEM to core	59
Table 4.8	WCMA transfers per output element for CNN layers operation	60
Table 4.9	WCMA transfers per output element for activation layers	61
Table 4.10	WCET in isolation for CNN layers operation in MMM	62
Table 4.11	WCET in isolation for activation functions in MMM	62
Table 4.12	WCMA transfers for CNN layers operation in MMM	63
Table 4.13	WCMA transfers for activation layers in MMM	64
Table 4.14	WCET in isolation for CNN layers operation in MLWM	65
Table 4.15	WCET in isolation for activation functions in MLWM	65
Table 4.16	WCMA transfers for CNN layers operation in MLWM	65
Table 4.17	WCMA transfers for activation layers in MLWM	66
Table 4.18	WCET and WCMA costs of a single Summation node in RCCT	67
Table 4.19	WCET in isolation for CNN layers operation in DF	68
Table 4.20	WCET in isolation for activation functions in DF	68
Table 4.21	WCET in isolation of masking operations in DF	68
Table 4.22	WCMA transfers for CNN layers operation in DF	69
Table 4.23	WCMA transfers for activation layers in DF	69
Table 4.24	WCMA transfers of masking operations in DF	69
Table 4.25	DDR-to-cluster bus transfer rate per architecture	71
Table 4.26	Average inter-cluster latency decomposition before 10% guard	77
Table 4.27	Clock frequencies of Kalray MPPA devices	79
Table 5.1	Summary of notations used in this chapter	84
Table 5.2	Headline performance metrics for MMM_MMM	86
Table 5.3	DAG node and edge of the MMM_MMM	87
Table 5.4	Headline performance metrics for MLWM_MMM	87
Table 5.5	DAG node and edge of the MLWM_MMM	88
Table 5.6	Headline performance metrics for SSP_MMM	88
Table 5.7	DAG node and edge growth of SSP_MMM	89
Table 5.8	Headline performance metrics for RCCT_MMM	90
Table 5.9	DAG node and edge growth of RCCT_MMM	90

Table 5.10 Headline performance metrics per policy after applying DF policy with $\iota(2)$	91
Table 5.11 DAG node and edge growth per policy after applying DF policy with $\iota(2)$	91
Table 5.12 Headline performance metrics per policy after applying DF policy with $\iota(16)$	92
Table 5.13 DAG node and edge growth per policy after applying DF policy with $\iota(16)$	92
Table 5.14 Headline performance metrics per policy for an input tensor of size $84 \times 84 \times 3$	93
Table 5.15 Headline performance metrics per policy for an input tensor of size $104 \times 104 \times 3$	93
Table 5.16 Headline performance metrics per policy for convolutional layers with 19 filters in Conv1 and 35 filters in Conv2	94
Table 5.17 Headline performance metrics per policy for convolutional layers with 27 filters in Conv1 and 37 filters in Conv2	95
Table 5.18 DAG node and edge growth per policy for convolutional layers with 19 filters in Conv1 and 35 filters in Conv2	95
Table 5.19 DAG node and edge growth per policy for convolutional layers with 27 filters in Conv1 and 37 filters in Conv2	96
Table 5.20 Headline performance metrics per policy for kernels of size 6×6 . .	97
Table 5.21 Headline performance metrics per policy for kernels of size 9×9 . .	97
Table 5.22 Headline performance metrics per policy for Coolidge architecture .	98
Table 5.23 Speedup metric per policy comparing between Coolidge and Bostan architectures	99
Table 5.24 Headline performance metrics per policy for Coolidge architecture with the TCA co-processor	99
Table 5.25 Speedup metric per policy comparing between Coolidge & TCA and Bostan architectures	100
Table 5.26 Speedup metric per policy comparing between Coolidge & TCA and Coolidge architectures	100
Table 5.27 Policy compliance across success criteria	102

List of Acronyms

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AXI	Advanced eXtensible Interface
CNN	Convolutional Neural Network
CPL	Convolution & Pooling Layers
CPS	Channel Padding Strategy
DAG	Directed Acyclic Graph
DC\$	Data Cache
DDR	Double Data Rate
DF	Dense Fusion
DL	Deep Learning
DN	Dummy Neuron
DPU	Data Processing Unit
DSU	Debug and System Unit
FC	Fully Connected
FFNN	Feed-Forward Neural Network
FIFO	First-In First-Out
FP	Fixed-Priority
FPU	Floating-Point Unit
GPR	General-Purpose Register
IC\$	Instruction Cache
JSON	JavaScript Object Notation
LPS	Local Padding Strategy

LRU	Least Recently Used
MAC	Multiply-Accumulate
MDM	Massively Decomposed Method
MIA	Multicore Interference Analysis
MIND	Model Inference Network Designer
MLP	Multi-Layer Perceptron
MLWM	Macroscopic Layer-Wise Modeling
MMM	Massively Merged Method
MPPA	Massively Parallel Processor Array
MS	Masking Strategy
NoC	Network-on-Chip
NPS	Neuron Padding Strategy
OPS	Overlap Padding Strategy
PDF	Portable Document Format
PE	Processing Element
PY	Python
R/W	Read/Write
RCCT	Redundant Convolution via Channel Tiling
ReLU	Rectified Linear Unit
RM	Resource Management
RR	Round-Robin
RTS	Real-Time Systems
SAP	Slot Allocation Policy
SMEM	Shared Memory
SRAM	Static Random Access Memory
SSP	Structured Spacial Partitioning
Tanh	Hyperbolic Tangent
TCA	Tensor Co-Processor Accelerator
UML	Unified Modeling Language

VLIW	Very Long Instruction Word
WCET	Worst-Case Execution Time
WCMA	Worst-Case Memory Access
WCRT	Worst-Case Response Time
YAML	Yet Another Markup Language

General Introduction

Introduction

The success of CNNs in AI technology is primarily attributed to their automatic extraction ability and achieving accuracy on a wide variety of classification, detection, and recognition tasks. Traditional research on CNNs has therefore focused largely on improving prediction quality, increasing precision, and even designing new architectural variants to achieve those goals [1].

However, in our context, the context of real-time and embedded systems, accuracy is not the relevant metric. When CNNs are put in environments with strict temporal constraints such as autonomous vehicles and aerospace, the primary performance metric becomes response time. In such scenarios, the ability of the system to meet real-time deadlines is crucial, often more important than the slightest improvements in accuracy. This observation motivated us into a fundamental shift in perspective: instead of analyzing CNNs solely as statistical models of high precision. In our case, CNNs are considered as computational mechanisms whose feasibility must be evaluated on the target hardware in terms of response time. This response time is calculated using MIA tool, which takes into consideration the interference caused by shared resource utilization, execution time, and other factors, in order to ensure timing predictability.

Our work therefore situates CNNs not in the classical context of accuracy-driven design, but in the context of temporal analysis and time optimization on massively parallel many-core embedded architectures. Specifically, we focus on the Kalray MPPA architectures, such as Bostan and Coolidge. These many-core processors represent a promising target for real-time CNN execution because they provide both high parallelism and fine-grained control over task mapping and data transfers. Nevertheless, exploiting such architectures efficiently requires rigorous analysis: how should CNN layers be decomposed into parallel tasks by avoiding overheads, how to map those tasks by order into each core, how to manage the data dependency and transfer, and how do memory constraints affect execution?

To address these challenges, first, we consider CNNs through the lens of DAGs, where each node corresponds to a computational task, *e.g.*, a convolution operation, and edges represent data dependencies and transfer between those tasks. By modeling CNNs in this way, it becomes possible to analyze parallelism opportunities, quantify core communication costs, and evaluate how different regrouping policies or partitioning strategies affect timing performance. Our work is applied only to FFNN CNNs, by considering them as already trained.

Our project does not address how to train CNNs nor how to improve their accuracy; instead, we focus solely on their topology and their inference, *i.e.*, the analysis of their topology, describing how operations are executed and how data flows from the input to the output. Through our perspective, CNNs become not only a DL scope but also a real-time scheduling and embedded utilization problem, bridging both domains together.

Background of the Project

This project is conducted within the ANR. COATIC (*Agence Nationale de la Recherche. Composants d’Ordonnancement et d’Analyse Temporelle pour l’Informatique Critique*) research project, an international collaboration that has for goal to collect and unify together the work of major French research institutions in the domain of real-time system

time analysis for multi/many core processors.

This project is proposed and supervised by Mr. Pierre-Emmanuel Hladik at the LS2N French research laboratory in Nantes, in collaboration with Mme. Claire Maiza at the VERIMAG French research laboratory in Grenoble. The work is carried out remotely from Algeria under the supervision of French research institutions.

Project Motivation

Our project was firstly inspired by the 2024 PhD research of Iryna De Albuquerque [2]. Her work focused on generating a formal representation of CNN architectures, using C programming, within the context of critical systems, where strict compliance with avionic and aerospace timing norms is required and respected, such as the DO-178C (Software Considerations in Airborne Systems) which imposes strict constraints on predictability, determinism, and schedulability of software components, including AI modules.

In her research, CNNs were modeled to respect these timing guarantees, thereby enabling their potential use in mission-critical avionics and aerospace systems. However, her work was limited to single-core architectures, which restricts scalability and does not fully exploit the computational capabilities of modern parallel hardware platforms.

In contrast, our project does not follow the same research direction, as we do not aim to define a formal language. Instead, our contribution lies in extending the study of CNNs to many-core architectures, where new challenges arise from highly parallel execution targets.

By extracting the topological structure of CNNs and their inference execution, and by converting them into DAGs, our approach enables timing analysis and schedulability studies on platforms with parallel execution resources, under timing and resource constraints.

Project Objectives

The main objective of this project is to deliver a final software product capable of automatically extracting the inference structure of a CNN and representing it as a DAG that is compatible with the input format of the MIA tool for evaluation purposes.

Main Objective

- **Analysis of CNN topology:** Inspecting CNNs to understand their structures and see how it may be represented as a DAG.
- **Development of an extractor tool:** Designing a tool that can automatically extract the structure of a CNN model and represent it as a DAG.
- **Merging with MIA:** Assuring the generated DAG is correct and compatible and can be integrated in MIA tool.
- **Validation and testing:** Testing the extractor tool with an existing embedded CNN model and analyse their timing results using MIA tool and other evaluation metrics.

Report Structure

This project report is organized as follows. Chapter 1 surveys the foundations required for this work: CNN layer formulations, an overview of RTS metrics, architecture characteristics of the Kalray MPPA relevant to cost modeling, and the essentials of MIA tool operating principles. Chapter 2 presents the overall methodology and provides the functional and logic description steps of our project’s workflow. It then explains the formats employed in the project, and concludes with a detailed UML use-case representation of the project. Chapter 3 introduces the proposed advanced DAG node regrouping policies, presents their mathematical formalization, and demonstrates how the resulting DAGs are obtained from our new policies. Chapter 4 develops an architecture-aware cost model for the Kalray MPPA, by formalizing mathematical equations to compute each DAG task cost such as memory, processor computation, data transfer costs, etc. Chapter 5 describes the tests we conducted and reports the evaluation results of our defined metrics and MIA tool output, followed by a discussion using the evaluation criteria we defined. Finally, we conclude our project report and outlines directions for future research.

Chapter 1

Essential Concepts and Preliminaries

1.1 Introduction

The purpose of this chapter is to present the theoretical and technical background necessary to understand the context of our work. We introduce the fundamental concepts that form the basis of our methodology and experimental analysis. Rather than providing an exhaustive survey, we focus on the notions most relevant to our study in this report, while referring the reader to complementary references such as our Master Degree state-of-the-art review in [3] for additional details, *e.g.*, on DAGs.

This chapter is organized into five sections. First, we present a detailed mathematical formulation of the principal CNN layers used in this work, highlighting their computational structure. Second, we describe the DAG representation topology of a CNN model with a focus on its inference. Third, we provide an overview of RTS concepts and constraints relevant to timing analysis. Fourth, we introduce the Kalray MPPA architecture, with particular focus on the Bostan and Coolidge DPUs, which are the hardware platforms targeted in this study. Finally, we explain the MIA framework, including its theoretical basis and a description of the tool.

1.2 Mathematical Formulation of CNN Layers

In this section, we introduce the mathematical equations corresponding to the CNN layers required in our project. It should be noted that we do not cover all existing CNN layers, but rather we focus only on layers used in our work. The mathematical formulations are taken from Goodfellow et al. in *Deep Learning* (MIT Press) [4], with slight restructuring to express them easily in a DAG form, enabling their direct use later on in the MLWM policy described in Chapter 3, Section 3.3.

For all CNN models considered in our project, we adopt a set of hypotheses regarding certain parameters. We assume a stride of 1, and a dilation rate equals 1 in convolution layers. By assuming this, we will obtain the densest results and the most compute-intensive case. This configuration generates the largest output size and the maximum number of computations since no positions are skipped, resulting in the maximum number of sliding windows. Also, for pooling layers we do not apply padding, both for simplicity and to ensure that pooling effectively reduces the size of the feature maps.

In the following CNN layer calculations, each operation is considered as being applied to the entire feature map (or to the input image in the case of the first layer). Throughout the remainder of this section, we denote the input feature map by I_l and the output of an operation made on a feature map by O_l .

The superscript l is an index to denote “at layer l ”. Given the spatial dimensions, height and width (H_l, W_l), and the channel count D_l , the following equations provide the mathematical representation of each operation in each layer:

1.2.1 Convolution Layer (Conv2D)

Convolutional layers form the fundamental building block of CNNs, as they are responsible for feature extraction from the input data. Each kernel, or filter, slides across the spatial dimensions of the input feature map, performing element-wise multiplications and summations to produce an output map. This operation allows the network to detect spatial patterns such as edges, textures, and higher-level structures. The number

of kernels determines the number of output channels, with each channel representing a different learned feature. Depending on the use of padding, the convolution can preserve the spatial resolution of the input or reduce it. We denote:

- $I_l \in \mathbb{R}^{H_l \times W_l \times D_l}$: input feature map.
- $K_l \in \mathbb{R}^{k_H \times k_W \times D_l \times D_{l+1}}$: kernel (filter) of shape $k_H \times k_W$, mapping D_l input channels to D_{l+1} output channels.
- $O_l = \text{Conv}(I_l, K_l) \in \mathbb{R}^{H_{l+1} \times W_{l+1} \times D_{l+1}}$: output feature map.

The convolution computation for each spatial position (i, j) and output channel d is:

$$O_l(i, j, d) = \sum_{i'=0}^{k_H-1} \sum_{j'=0}^{k_W-1} \sum_{d'=0}^{D_l-1} \underbrace{I_l(i+i', j+j', d')}_{\text{value from input feature map}} \times \underbrace{K_l(i', j', d', d)}_{\text{corresponding kernel}} \quad (1.1)$$

Where:

- The summation over d' iterates over all input channels D_l of the feature map.
- The number of output channels is equal to the number of filters (kernels) in the convolutional layer, which is d .
- If padding is used in the convolution operation, the spatial dimensions of the output will remain the same as those of the input. If padding is not used, the output size will be reduced relative to the input. The output height and width are given by:

$$\begin{cases} H_{l+1} = H_l - k_H + 1 \\ W_{l+1} = W_l - k_W + 1 \end{cases} \quad (1.2)$$

1.2.2 Pooling Layer (Pool2D)

Pooling layers are used in CNNs to progressively reduce the spatial resolution of feature maps while retaining the most significant information. It reduces the number of parameters and computations in the network, thereby improving efficiency and reducing overfitting. The most common pooling strategies are max-pooling and average-pooling, which differ in the way they calculate values inside each pooling window.

Max-Pooling

Max-pooling selects the maximum value within each pooling window. It preserves the most prominent value in the window while discarding weaker ones. It is widely used in CNN architectures because it provides robustness to noise and small variations in the input. The number of channels is preserved, as pooling is applied independently to each channel. We denote:

- $I_l \in \mathbb{R}^{H_l \times W_l \times D_l}$: input feature map.
- $P_l \in \mathbb{R}^{m \times n}$: pooling parameters of size $m \times n$ of the pooling window.

- $O_l = \text{MaxPool}(I_l, P_l) \in \mathbb{R}^{H_{l+1} \times W_{l+1} \times D_{l+1}}$: output feature map.

The formula for max-pooling over each channel d is:

$$O_l(i, j, d) = \max_{\substack{0 \leq i' < m \\ 0 \leq j' < n}} I_l(i + i', j + j', d) \quad (1.3)$$

Where:

- For each position (i, j) in the output, the maximum is taken over the $m \times n$ window in the input.
- The number of channels d remains unchanged, which means that $D_l = D_{l+1}$. The operation is applied independently to each channel.

Average-Pooling

Average-pooling computes the mean value of the elements inside each pooling window. Unlike max-pooling, which focuses on the most dominant value, average-pooling provides a smoother aggregation of local features by considering the overall intensity of values. This operation is less aggressive in feature selection and is particularly useful in scenarios where global information is more important than local peaks. As with max-pooling, the number of channels remains unchanged. We denote:

- $I_l \in \mathbb{R}^{H_l \times W_l \times D_l}$: input feature map.
- $P_l \in \mathbb{R}^{m \times n}$: pooling parameters of size $m \times n$ of the pooling window.
- $O_l = \text{AvgPool}(I_l, P_l) \in \mathbb{R}^{H_{l+1} \times W_{l+1} \times D_{l+1}}$: output feature map.

The formula for average-pooling over each channel d is:

$$O_l(i, j, d) = \frac{1}{m n} \sum_{i'=0}^{m-1} \sum_{j'=0}^{n-1} I_l(i + i', j + j', d) \quad (1.4)$$

Where:

- The average is computed over values inside a $m \times n$ window.
- The number of channels d also remains unchanged.

1.2.3 Flatten Layer

The flatten layer serves as a bridge between convolutional/pooling layers and dense, or FC layers. Its role is purely structural: it reshapes the multidimensional feature maps into a one-dimensional vector, preserving all values but discarding spatial arrangement. This vectorized representation allows the next dense layers to treat the extracted features as a standard input vector suitable for classification or regression tasks. We denote:

- $I_l \in \mathbb{R}^{H_l \times W_l \times D_l}$: input feature map.
- $F_l(\cdot)$: reshape operation; converts a 3D tensor into a 1D vector.

- $O_l = F_l(I_l) \in \mathbb{R}^{H_l \cdot W_l \cdot D_l}$: output vector.

The flattening operation rearranges channel dimensions into a single vector:

$$O_l = \text{vec}(I_l) \quad (1.5)$$

Where:

- The values are preserved; only the layout is changed.
- This operation prepares the data for input to a dense (FC) layer.

1.2.4 Dense/FC Layer

Dense layers, also known as FC layers, perform high-level reasoning based on the features extracted by previous layers. Each neuron in a dense layer receives input from every element of the preceding layer, making it capable of learning complex global relationships between all the received inputs. The linear combination of inputs, weights, and biases is followed directly by a non-linear activation function, enabling the network to approximate non-linear decision boundaries. In classification tasks, the final dense layer typically outputs class probabilities after applying a softmax activation. In regression tasks, by contrast, the final dense layer usually produces continuous values, often without an activation function (linear output) or with a suitable activation such as ReLU activation. We denote:

- $I_l \in \mathbb{R}^{H_l \cdot W_l \cdot D_l} = \mathbb{R}^{N_l}$: input vector from a flatten operation.
- $FC_l(\cdot)$: fully connected transformation at layer l , applying linear mapping followed by a non-linear activation function f .
- $O_l = FC_l(I_l) \in \mathbb{R}^{N_{l+1}}$: output vector.

The dense layer performs:

$$O_l(v) = f \left(\sum_{u=1}^{N_l} I_l(u) \cdot W_{gl}(v, u) + b_l(v) \right), \quad \text{for } v = 1, \dots, N_{l+1} \quad (1.6)$$

Where:

- $W_{gl}(v, u)$: scalar parameter of the weight matrix connecting input u to output v .
- $b_l(v)$: bias term for output neuron v .
- $f(\cdot)$: activation function.

1.2.5 Activation Functions

The activation functions are often used only after convolutional and dense layers. Therefore, when an activation function is used with a convolutional layer, the output becomes:

$$O_l = f(\text{Conv}(I_l, K_l)) \quad (1.7)$$

In the following, we let $z_l \in \mathbb{R}$ be the pre-activation value at layer l , and o_l its post-activation value. Common activation functions are:

ReLU

ReLU is one of the most widely used activation functions in CNNs. Its popularity comes from its simplicity and effectiveness, by setting all negative inputs to zero while keeping positive values unchanged. It introduces non-linearity into the network without significantly increasing computational cost. We denote:

- Input : $z_l \in \mathbb{R}$
- Operation : clips negative values to zero.
- Output : $o_l = \text{ReLU}(z_l)$

The ReLU equation performs as:

$$o_l = \max(0, z_l) \quad (1.8)$$

Leaky ReLU

Leaky ReLU was proposed as an extension of ReLU to address the “dying neuron” issue. Instead of assigning zero to all negative inputs, it allows defining a small negative slope controlled by the parameter α . This ensures that neurons continue to have small values even for negative inputs, thereby improving learning stability and reducing the chance of inactive neurons. We denote:

- Input : $z_l \in \mathbb{R}$.
- Parameter : assigns slope α for the negative branch (by default $\alpha \approx 10^{-2}$).
- Output : $o_l = \text{LeakyReLU}(z_l)$.

The Leaky ReLU equation performs as:

$$a = \begin{cases} z_l, & z_l \geq 0 \\ \alpha z_l, & z_l < 0 \end{cases} \quad (1.9)$$

Sigmoid

The sigmoid activation function maps inputs into the interval $]0, 1[$, producing outputs interpretable as probabilities. It was widely used in early neural networks and is still employed in specific contexts, such as binary classification. However, the sigmoid function suffers from saturation: for very large positive or negative inputs, the resulting output approaches zero, which can hinder learning in deep networks. We denote:

- Input : $z_l \in \mathbb{R}$.
- Operation : maps the input to $]0, 1[$ via the sigmoid function.
- Output : $o_l = \text{Sigmoid}(z_l) \in]0, 1[$.

The Sigmoid equation performs as:

$$o_l = \frac{1}{1 + e^{-z_l}} \quad (1.10)$$

Tanh

Tanh activation function is similar to the sigmoid but maps inputs to the range $] -1, 1[$. This symmetric output often leads to faster convergence compared to the sigmoid, as activations are centered around zero. Like the sigmoid, however, it can also suffer from saturation for large input magnitudes. Despite this, it remains useful in certain models where negative activations carry meaningful information. We denote:

- Input : $z_l \in \mathbb{R}$.
- Operation : maps the input to $] -1, 1[$ via the hyperbolic tangent.
- Output : $o_l = \text{Tanh}(z_l) \in] -1, 1[$.

The Tanh equation performs as:

$$o_l = \frac{e^{z_l} - e^{-z_l}}{e^{z_l} + e^{-z_l}} \quad (1.11)$$

Softmax

The softmax activation function is typically applied to the last dense layer of classification networks. It transforms a vector of arbitrary inputs into a probability distribution over the possible classes. This property makes softmax the standard choice for multi-class classification problems, where the highest-probability output corresponds to the predicted class. In regression tasks, however, softmax is generally not used, since regression requires continuous outputs rather than discrete probabilities. We denote:

- Input vector : $\mathbf{z}_l = (z_1, \dots, z_N)^\top \in \mathbb{R}^N$.
- Operation : normalises the inputs into a probability distribution over N classes.
- Output : $\mathbf{o}_l = \text{Softmax}(\mathbf{z}_l) \in [0, 1]^N$ with $\sum_{i=1}^N o_i = 1$.

The Softmax equation performs as:

$$o_i = \frac{e^{z_i}}{\sum_{j=1}^N e^{z_j}}, \quad \text{with } i = 1, \dots, N \quad (1.12)$$

1.3 CNN Model to DAG Topology

The main assumption of this project is that CNNs are represented as DAGs. To guarantee this representation, we restrict our study to FFNNs only, since recurrent connections or backward edges would violate the acyclicity property required for a DAG.

To enforce this assumption, we further consider CNNs that are already trained. During training, the backward propagation introduces dependencies that are not compatible with a DAG structure, particularly in dense layers. By focusing exclusively on trained models, we avoid such backward edges and ensure that the resulting representation remains acyclic.

This restriction allows us to concentrate solely on CNN inference and topology, without addressing training procedures, precision improvements, or performance optimization. A CNN ready for inference requires only:

- Layers such as convolution, pooling, and FC layers.
- Activation functions, *e.g.*, ReLU, sigmoid, softmax.
- Trained weights parameters associated with each layer.

These elements are sufficient to perform forward propagation, *i.e.*, generate outputs for given inputs, and to establish the data flow required for timing analysis. Figure 1.1 is an example of a DAG representation of an FFNN CNN structure where each node corresponds to a computational layer and edges capture the flow of data between layers. In this schematic, four convolutional layers are followed by pooling operations, whose outputs are sent to a flattening stage. The flatten node is then fully connected to three dense layers, which in turn feed into the final classification layers. The image highlights the left-to-right progression of data through the network.

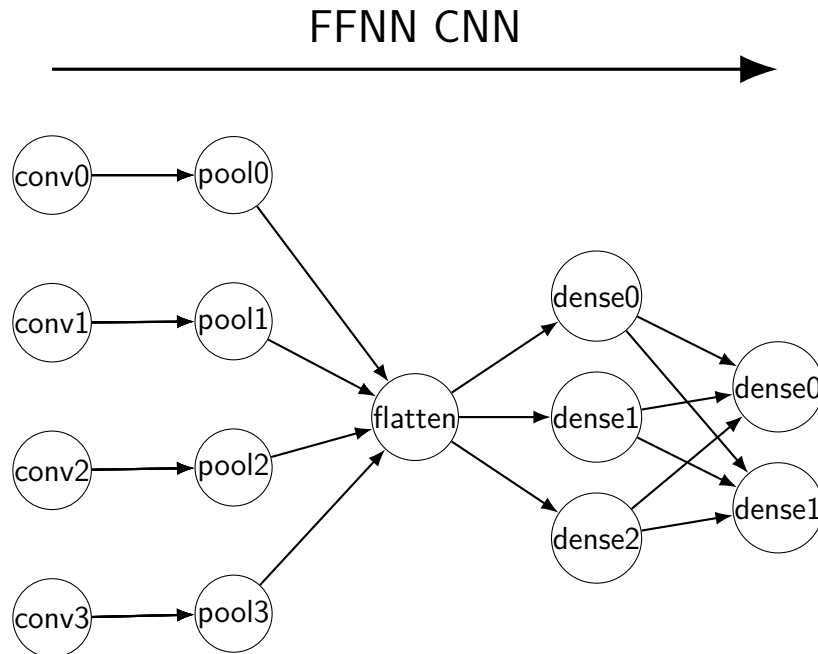


Figure 1.1: DAG representation of an FFNN CNN structure

However, for components related to training, such as:

- Loss functions, used to compute the error between predictions and targets.
- Optimizers, employed to update weights during training iterations.
- Training metrics, applied to monitor learning progress.

are irrelevant once the model is trained and are not part of the inference pipeline. Whether the models are well-trained or achieve high accuracy is outside the scope of our analysis. We will not focus on improving a CNN performance, but we will focus on its topology and model their computational structure for real-time execution.

1.4 Overview of RTS

In this section, we review foundational notions in RTS that we will be using throughout this work. We focus on timing-related quantities that support timing analysis. Figure 1.2 shows the key metrics on a single core and a single temporal axis for one periodic task. All timing quantities are reported in cycles. However, for time unit conversion, we use the target core frequency.

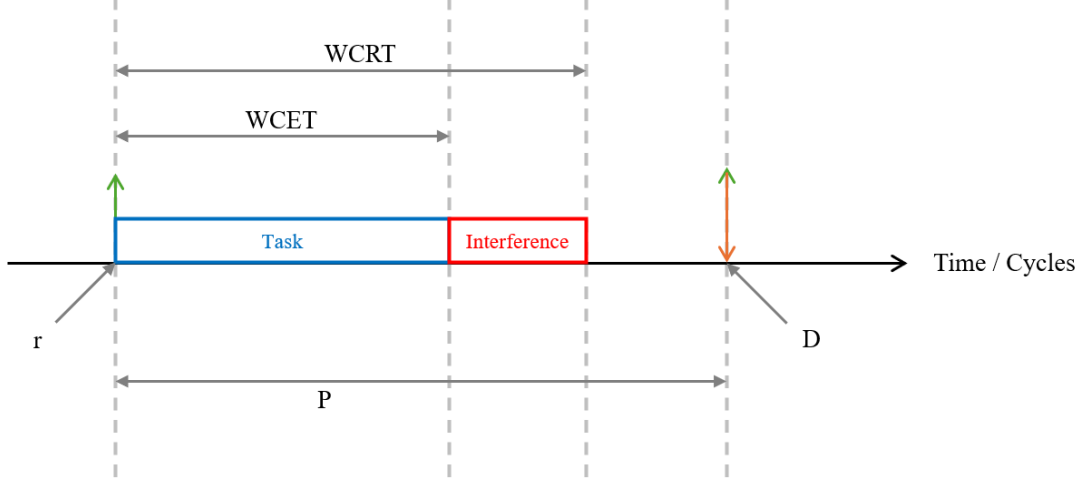


Figure 1.2: Timeline of a periodic task

The explanation of the metrics in Figure 1.2 is presented as follows:

Release Time (r). The release time is the instant at which a task becomes ready for execution. For periodic tasks, successive releases occur every period P .

Worst-Case Execution Time (WCET). WCET is an upper bound on the processor time required to execute the task's code in isolation, *i.e.*, with no interference, for a given hardware/software configuration. In this thesis, WCETs are expressed in cycles.

Interference. Interference denotes the delay added to the isolated execution due to resource contention, *e.g.*, shared buses/memories. In Figure 1.2, the interference is presented as the red segment appended to the blue task segment which is executed in the WCET duration, producing the WCRT.

Worst-Case Response Time (WCRT). The response time of a task is the interval between its release and the instant it completes by including the interference calculation into that interval. The WCRT is the maximum such value over all possible scheduling scenarios. It satisfies:

$$\text{WCRT} > \text{WCET} \quad (1.13)$$

Period (P). For a periodic task, the period is the inter-arrival time between consecutive releases. In our study, it represents the time interval at which the complete execution of the CNN model is repeated.

Deadline (D). The deadline is the latest/maximum allowable completion time for a task after its release by which its WCRT must be completed. If it exceeds the deadline, this constitutes an undesirable case. Different types of deadline configurations exist in RTS as it is shown in Figure 1.3.

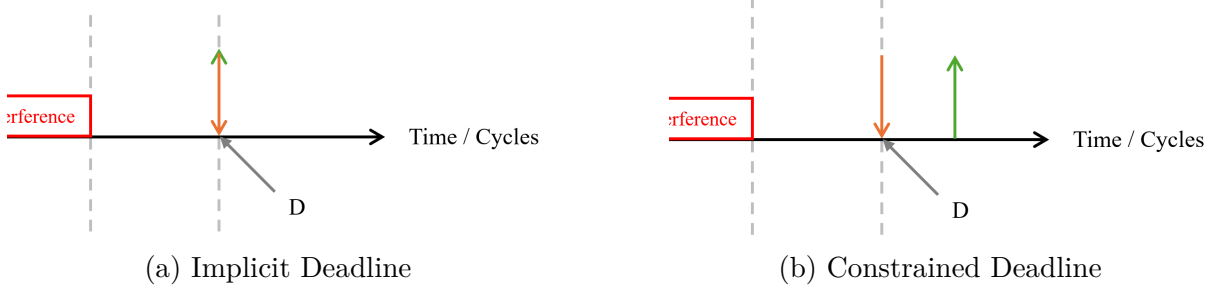


Figure 1.3: Deadline configuration in RTS

- **Implicit Deadline:** In Figure 1.3a, the deadline is equal to the period, *i.e.*, $D = P$. This is the most common assumption in many systems.
- **Constrained Deadline:** In Figure 1.3b, the deadline is less than the period, *i.e.*, $D < P$.
- **Arbitrary Deadline:** The deadline is independent of the period, with no fixed relationship between D and P .

In our study, we focus exclusively on implicit and constrained deadlines only. The arbitrary deadline model is not considered.

We should also note that our real-time analysis in this project is entirely static: all parameters are fixed and defined in advance, with no dynamic behavior considered.

1.5 Kalray MPPA Architecture: Bostan vs. Coolidge

Kalray MPPA chips like Bostan and Coolidge are not like traditional microprocessors with a limited number of cores. They are classified as many-core parallel processors, *i.e.*, DPUs, specifically designed for high-performance embedded computing.

In this section, we present both the Bostan and Coolidge architectures, focusing only on the aspects relevant to this project. The complete architectures are highly complex, so our description is simplified and does not cover every detail. Likewise, the accompanying figures are drawn with certain simplifications to facilitate understanding.

Further architectural and technical details can be found on the official Kalray website [5]. For a detailed description of the Bostan architecture, we refer to [6], while the main reference for the Coolidge is [7].

In the context of the architectures, we assume that there are no memory leaks nor corrupted transfer data. In other words, we consider an idealized system where all transfers are correct, avoiding the need for error-correction or data-reloading protocols. For the sake of simplicity, caches are also not considered in our study, thereby eliminating the effects of cache misses.

These aspects are excluded to reduce complexity and to focus the analysis on the core mechanisms of CNN execution. Nevertheless, cache behavior and fault-tolerance

mechanisms remain important topics which can be identified as interesting directions for future work.

1.5.1 Kalray MPPA2-256 “Bostan” Architecture

The second generation of Kalray MPPA processor named Kalray MPPA2-256, code-named Bostan, is composed of 16 compute clusters, each containing 16 user processing cores ($16 \times 16 = 256$). Each compute cluster has a private SMEM implemented as a 2 MB SRAM, organized into 16 parallel banks of 128 KB each ($128 \text{ KB} \times 16 = 2 \text{ MB}$).

The 16 cores within a cluster are homogeneous in-order VLIW processors, each capable of executing 1 MAC per cycle and up to five operations per cycle, *i.e.*, a 5-way VLIW. It can theoretically do all those operations simultaneously: 1 ALU, 1 Load, 1 Store, 1 Branch, 1 Multiply.

Every core is equipped with a private set of 64 GPRs, each 32-bit register, together with instruction and data L1 caches: the instruction cache (IC\$) handles program fetches, and the data cache (DC\$) services load/store operations, each 8 KB in size.

Transfer between a core and SMEM memory, we suppose it is carried out over buses of 64-bit width. In this work, we neglect transfers between cores and their caches, as well as between caches and SMEM. These aspects are left as perspectives for future work.

In addition to the 16 user cores, each cluster contains an RM core, responsible for orchestration by managing shared data and coordinating the distribution and collection of updated data among the cores. Each cluster also includes a DSU, which provides debugging and monitoring capabilities by enabling the observation of execution behavior, memory usage, and event tracing.

Beyond the inside-cluster level, clusters are interconnected with each other and with I/O via a dual 2D-torus NoC (Rx/Tx).

A simplified representation of the Bostan architecture is provided in Figure 1.4.

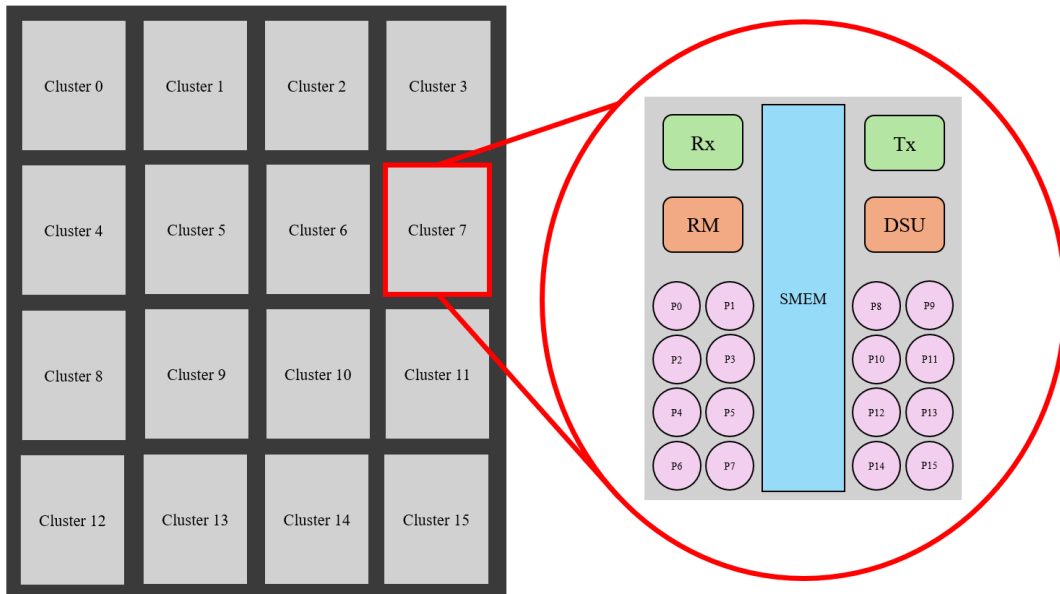


Figure 1.4: Simplified representation of Kalray MPPA2-256 Bostan architecture

Memory Banks and Arbitration

As recommended in prior work [8, 9], we adopt the blocked addressing mode, *i.e.*, contiguous 128 KB blocks mapped to each bank, with the 2 MB cluster memory partitioned into 16 banks. This banked organization is essential for predictability, as it enables multiple cores to access different memory banks in parallel without interference.

Within each SMEM bank, an arbiter resolves contention among concurrent requesters, *e.g.*, cores. One commonly used policy is RR: the arbiter maintains a cyclic order over the active requesters and grants access to the memory bank to each requester in turn with one slot only equally between all the requesters. If a requester has no pending request when its turn arrives, the arbiter will simply skip the requester without waiting.

Figure 1.5 shows an RR arbiter for three initiators A, B, C with equal weights $w_A = w_B = w_C = 1$, where the weight w_i is the number of grant slots given to initiator i per round. A round is the minimal sequence that grants exactly one slot to every active initiator; here its length is $N = 3$.

Round 1 occupies slots 1–3 in order $A \rightarrow B \rightarrow C$. Round 2 repeats the same order on slots 4–6. With equal weights, each initiator receives one grant slot per round in RR arbitration. We consider in this example that no request is skipped.

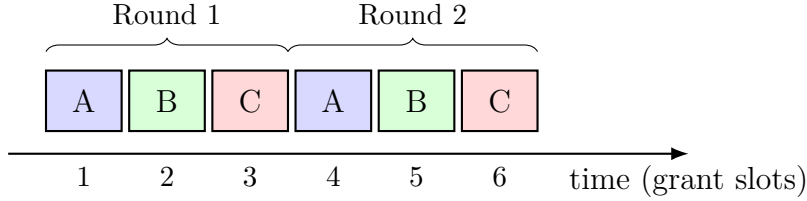


Figure 1.5: Demonstration simple RR arbitration

In Bostan, a multi-level bus arbitration policy is implemented to fairly distribute memory access slots among cores as it is shown in Figure 1.6. The figure is taken from [9].

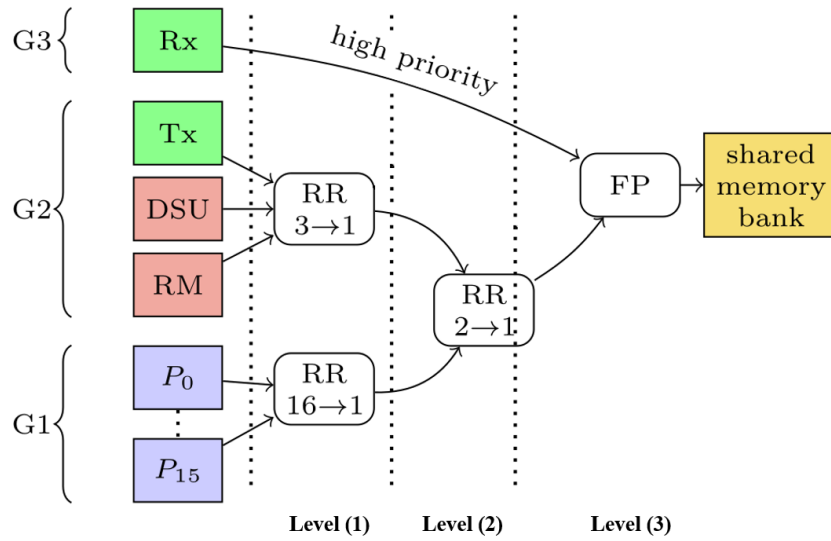


Figure 1.6: Bostan request arbitration to an SMEM Bank

The explanation of the arbitration levels in the Figure 1.6 is presented as follows:

- Level (1): Requests from the 16 processing cores (G1 group) go through a 16-to-1 RR arbiter to the SMEM bank direction. In parallel, another system (G2 group) that consists of: RM core, DSU and Tx of the NoC, requests arbitration among themselves via another 3-to-1 RR.
- Level (2): The winners in G1 and G2 then arbitrate against each other with a 2-to-1 RR. Arbitration is performed between the “core group” and the “system group”.
- Level (3): Finally, incoming NoC traffic Rx, *e.g.*, data arriving from another cluster or external memory, is given highest priority via an FP arbiter at the last stage. The Rx requests (G3 group) preempt others to avoid network congestion, as the design prioritizes incoming data.

Globally, an SMEM request initiated by a core in Bostan must traverse three stages of RR arbitration plus one level of FP before accessing the SMEM bank. This arbitration is work-conserving, *i.e.*, no idle cycles, but it implies that in the worst case, a core’s memory access can be delayed by interference from other cores’ or system requests.

Timing Predictability

The architecture of Bostan was conceived to support time-predictable execution. The cores are in-order, *i.e.*, no speculative out-of-order execution, and caches are relatively small, which is advantageous in static analysis. For future work focusing on cache analysis, which is outside the scope of our study, the replacement policy of the L1 caches is a simple FIFO or pseudo-LRU to facilitate worst-case cache analysis. The cache replacement policy is valid in Coolidge architecture.

Additionally, by taking into consideration that Bostan provides a total of 32 MB of on-chip SMEM (2 MB per cluster $\times$ 16 clusters), this capacity is extremely small for our application. Such a memory constraint represents a significant challenge, since CNN inference typically requires larger memory capacity. To address this limitation, we assume that all pre-trained parameters and hyper-parameters of the CNN model are stored in the external DDR3 memory of the Bostan. This assumption is pessimistic from a WCRT analysis perspective, as it necessitates the creation of additional DAG nodes representing DDR transfers, thereby increasing the interference, which in turn leads to longer WCRT.

It should be noted that 2D torus NoC transfers, whether between clusters or from DDR3 to clusters, generate variable latency and are not highly predictable, as they depend on the number of hops in the mesh routing Bostan NoC architecture. In this work, however, we assume a fixed transfer latency and proceed by estimating an approximate average number of cycles for these transfers.

1.5.2 Kalray MPPA3-80 “Coolidge” Architecture

The third generation of Kalray MPPA processor named Kalray MPPA3-80, code-named Coolidge, retains the same clustered many-core design of its predecessors but with several enhancements for performance and predictability. Coolidge has 80 cores comprised of 5 compute clusters, each with 16 user cores ($16 \times 5 = 80$).

Every core is equipped with private 64 GPRs 64-bit registers with a wider issue width of 6-way VLIW. It can theoretically execute those operations simultaneously: 2 ALU, 1 Load or Store, 1 Branch, 1 Multiply, 1 slot used for TCA.

Regarding TCA, Coolidge adds a co-processor to each core called TCA, specialized for neural network computations, *e.g.*, fast MAC operations, by performing up to 16 MAC per cycle for FP16/INT16 and INT8 data precision, significantly boosting the WCET computation in CNN convolution and dense layers. Moreover, each Coolidge core has larger L1 caches than Bostan; both (IC\$) and (DC\$) are 16 KB.

Identical to Bostan architecture, the SMEM in Coolidge is an SRAM organized as 16 banks; each bank has a large capacity of 256 KB compared to Bostan's bank, resulting in 4 MB total per cluster. For addressing mode, we will keep choosing the blocked approach for the same reasons. Regarding bus width, we assume transfers from a core to SMEM are made with a 128-bit bus, neglecting intermediate cache transfers as we did previously in Bostan.

The DSU, RM core, and the 2D-torus NoC (Rx/Tx) are preserved in Coolidge, as in Bostan. However, a major difference is that Coolidge introduces a new crossbar-based R/W interconnect inside the cluster, called AXI, which is used for global communication between clusters and with the other I/O. Unlike Bostan, which relies solely on the NoC that generates variable latency due to hop-dependent routing, the AXI interconnect provides stable transfers with fixed latency. This results in reduced variability and improved predictability, as AXI offers more deterministic communication compared to the NoC.

A simplified representation of the Coolidge architecture is provided in Figure 1.7.

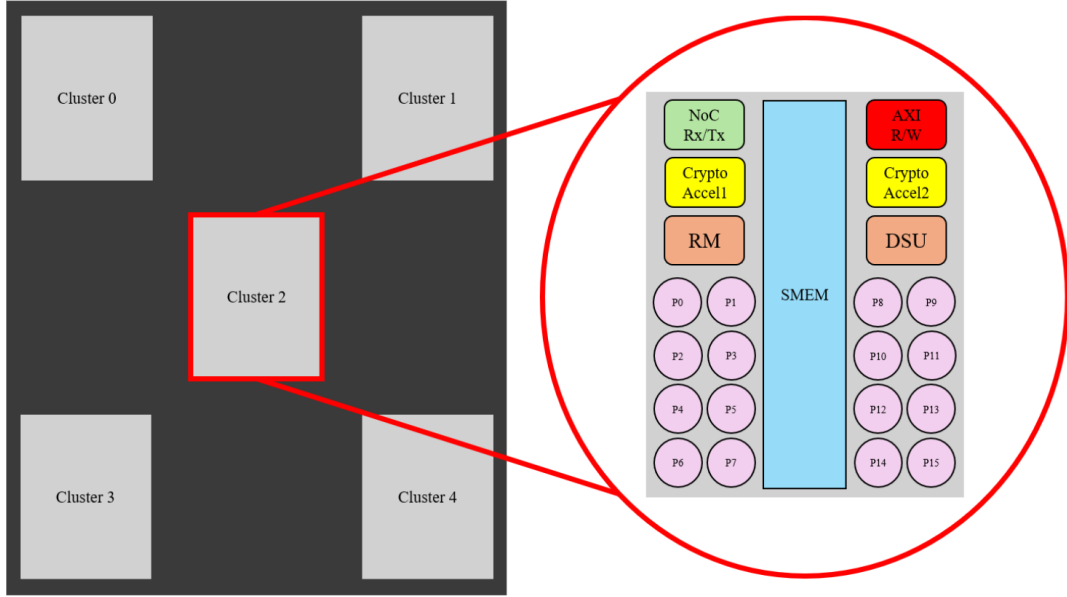


Figure 1.7: Simplified representation of Kalray MPPA3-80 Coolidge architecture

Memory Arbitration and Predictability

At the cluster level, SMEM access is not managed by a simple RR policy. Instead, an advanced RR, called SAP arbiter, is employed. The SAP arbiter implements a configurable with variable weights RR policy: rather than allocating one slot per initiator in each round, initiator i is assigned an integer weight w_i and receives w_i consecutive grant slots per round.

In Figure 1.8, we consider three initiators A, B, C with weights $w_A = 2$, $w_B = 1$,

$w_C = 3$. One round thus has $2 + 1 + 3 = 6$ grant slots in the fixed order:

$$\underbrace{A, A}_{w_A}, \underbrace{B}_{w_B}, \underbrace{C, C, C}_{w_C}$$

Round 1 occupies slots 1–6. Round 2 occupies slots 7–12, repeating the same pattern. We consider in this example that no request is skipped.

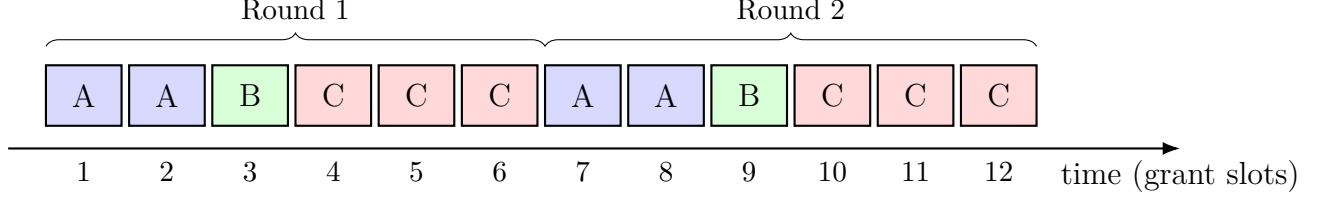


Figure 1.8: Demonstration simple SAP arbitration

In Coolidge, SMEM bank access requests issued by various sources are managed through a SAP arbitration scheme as shown in Figure 1.9. The figure is taken from [10].

At arbitration Level (1), all initiators, including the 16 PE (from PE_0 – PE_{15}), the RM core, the DSU, cryptographic accelerators, NoC interfaces, and AXI read/write ports, compete for access to the SMEM bank.

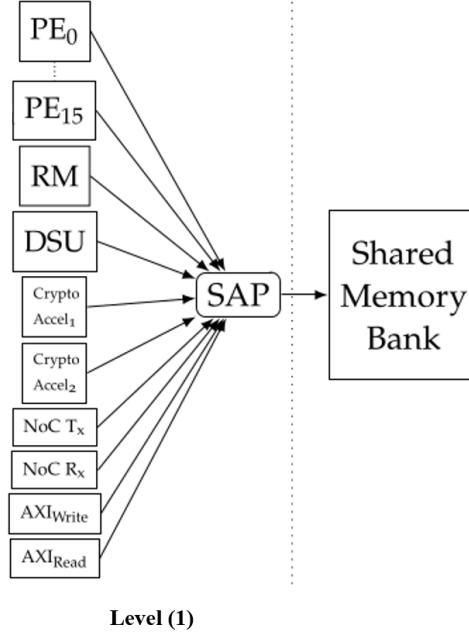


Figure 1.9: Coolidge request arbitration to an SMEM Bank

Timing Predictability

The Coolidge architecture was conceived to improve time-predictable execution compared to Bostan.

Considering that Coolidge provides a total of 20 MB of on-chip SMEM (4 MB per cluster $\times$ 5 clusters), this capacity, which is less than Bostan's capacity, remains also

limited for our application. We further assume that all pre-trained parameters and hyper-parameters of the CNN model are stored in the external DDR4 memory of Coolidge.

The AXI interconnect has constant cycles for data transfer between any two clusters or a cluster and DDR4. This is a major improvement for predictability: it means that if two clusters communicate or if a cluster accesses external DDR4, the network latency is fixed and does not depend on routing or contention.

After presenting the characteristics of each architecture separately, we summarize in Table 1.1 the key architectural properties that are most relevant to our project. This comparative view highlights the essential differences between the Bostan (MPPA2-256) and Coolidge (MPPA3-80) platforms.

Property	Bostan (MPPA2-256)	Coolidge (MPPA3-80)
Clusters $\times$ cores	$16 \times 16 = 256$	$5 \times 16 = 80$
SMEM / cluster	2 MB (16 banks $\times$ 128 KB)	4 MB (16 banks $\times$ 256 KB)
Core $\leftrightarrow$ SMEM bus	64-bit (assumed)	128-bit (assumed)
Special unit	–	TCA (up to ~ 16 MAC/cycle, FP16/INT16/INT8)
Interconnect	2D-torus NoC (variable latency)	AXI crossbar (fixed latency) + NoC

Table 1.1: Comparison of key architectural properties relevant to our project between Bostan MPPA2-256 and Coolidge MPPA3-80

1.6 Operating Principle of MIA Tool

The Multicore Interference Analysis (MIA) tool is a collection of Python scripts, developed by the VERIMAG laboratory at Grenoble University, France, computes interference-aware timing for a DAG of tasks mapped onto a many-core platform. For each task in the DAG, the following information must be provided:

- WCET in isolation,
- WCMA,
- a static mapping of tasks to cores,
- a static execution order of tasks, *i.e.*, no dynamic preemption, everything is fixed,
- dependencies on preceding tasks, and
- successor tasks, along with the amount of data to be transferred to them.

MIA then computes the release dates and WCRT of all tasks, taking into consideration the interference produced through the arbitration scheme of the target architecture when accessing SMEM banks by each task, thereby resulting in an interference-aware global completion time for the entire DAG.

By default, we set all release dates to zero, allowing MIA to compute them without any artificial provoked delays for certain tasks. Furthermore, the tool requires the specification of architectural parameters: for Bostan, the number of cores and clusters is set to 16 each, while for Coolidge it is set to 16 cores and 5 clusters. We will therefore use both

architectures for evaluation and tests: Bostan, which has a large number of cores and can offer high parallelism but lacks advanced architectural optimizations, and Coolidge, which has fewer cores and thus lower parallelism compared to Bostan, but provides several architectural optimizations.

The user may freely define the period and the deadline according to the desired scheduling constraints. All quantities, computations, and costs are expressed in cycles, based on the characteristics of the target architecture. The equations for calculating these values will be presented later on in Chapter 4.

Figure 1.10 summarizes the main stages of the MIA process. Inputs consist of the DAG description (validated for acyclicity), real-time constraints (validated for determinism, which ensures that the DAG is predictable), and hardware parameters. After input acquisition, the MIA engine computes interference under SMEM-bank arbitration, then calculates per-task WCRT and updates release dates, provides the global WCRT, and finally issues a schedulability verdict by comparing the global WCRT against the specified deadline.

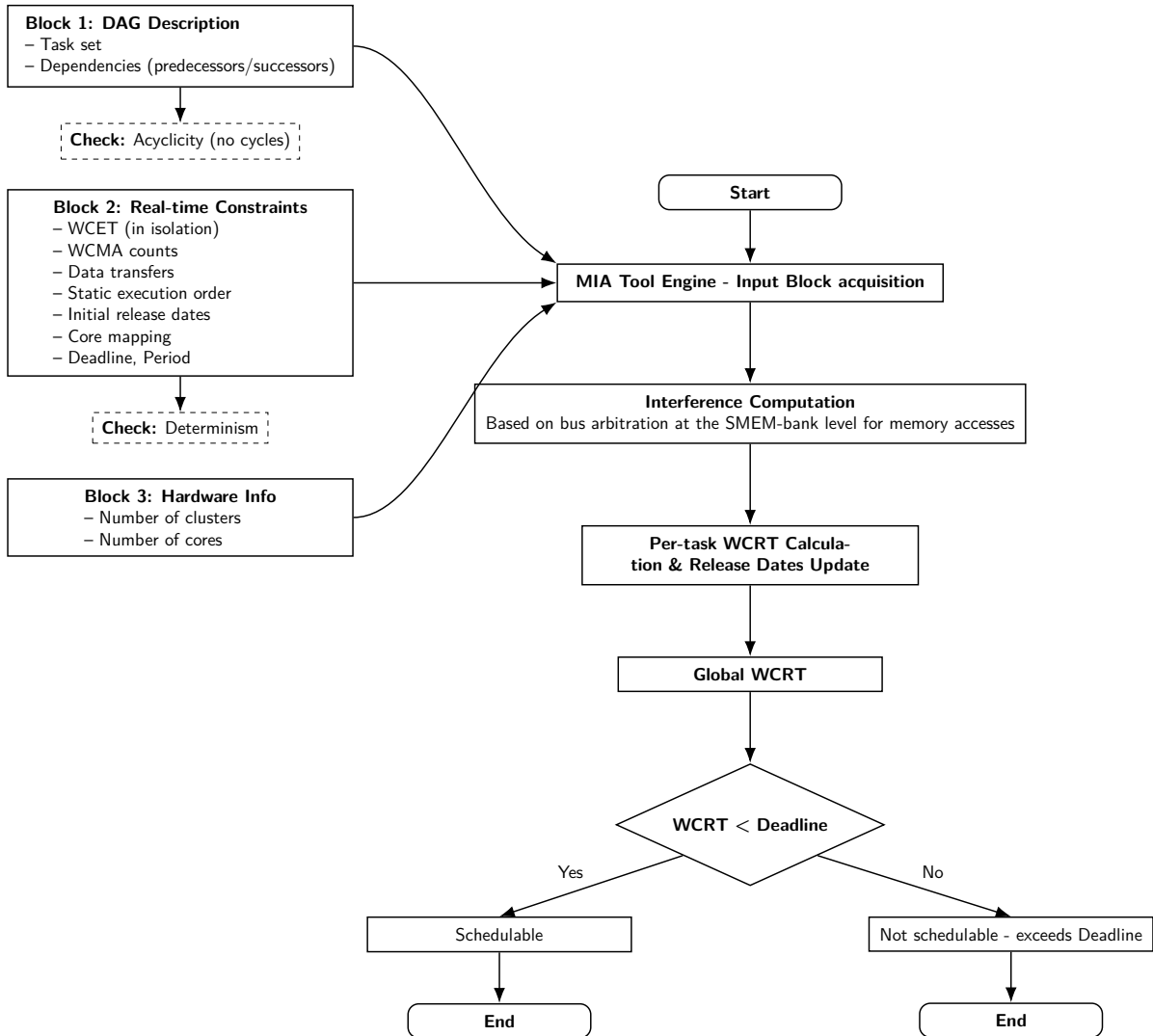


Figure 1.10: Workflow of MIA tool process

1.6.1 Interference Origins

In multi-core architectures, several PE (cores) must often share access to common resources, such as the main memory. An interference occurs when two or more cores attempt to access the same shared resource simultaneously. Since the resource cannot handle multiple requests in parallel, an arbitration mechanism, *e.g.*, RR arbiter, is required to regulate access. The arbiter receives incoming requests and decides which one will proceed to execution. While this mechanism avoids collisions on the bus, it also introduces delays for the requests that are not immediately granted access.

When a core issues a memory request, the delay it experiences depends not only on the intrinsic access time of the memory system in isolation, but also on the additional waiting time caused by competing requests from other cores; this delay is known as interference. The worst-case delay may accumulate when multiple requests overlap, which will make timing analysis more complex and less predictable.

Figure 1.11 illustrates this concept. We see several cores ($PE_0, PE_1, PE_2, \dots$) issue requests through a shared bus arbiter before reaching the memory. The arbiter enforces exclusive access, which means that while one core is being served, the others must wait. Consequently, memory latency becomes variable and depends on both the number of active requesters and the arbitration policy employed. This shared-access bottleneck is one of the fundamental origins of timing interference in multi-core platforms.

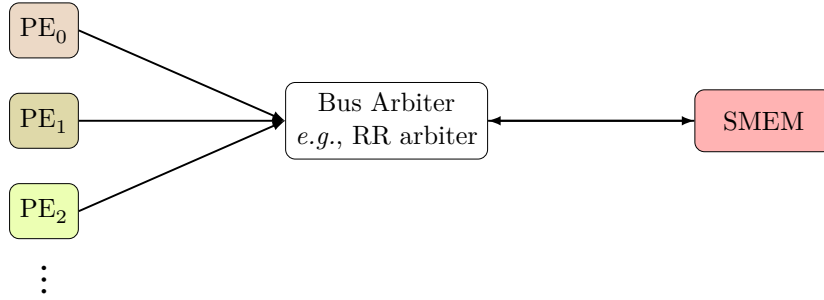


Figure 1.11: Interference caused by bus arbitration for SMEM access in many-core systems

1.6.2 Explanation of MIA Algorithm

The old MIA algorithm, detailed in Rihani's thesis [8], computes interference by a fixed-point iteration over release dates and task interference; this method has time complexity $O(n^4)$ and does not scale to large DAGs. Later on, an optimized algorithm described in [11] replaced the global fixed point with an incremental, cursor-driven construction of the time-triggered schedule and applies interference updates to the set of alive tasks, reducing the time complexity to $O(n^2)$.

The principle of the updated algorithm is to maintain a time cursor t and partition tasks into *Alive* (A : currently executing), *Closed* (C : finished), and *Future* (F : not yet released). At each step, advance t to the next event τ ; either the earliest finish time among A or the smallest release date among F :

$$t \leftarrow \min \left\{ \min_{\tau \in A} \text{finish}_{\tau}, \min_{\tau \in F} \text{rel}_{\tau} \right\} \quad (1.14)$$

When tasks open at time t , the algorithm recomputes only the pairwise interferences among tasks in A that share an arbitrated resource. Tasks in C are excluded since they are already *closed*, *i.e.*, their execution has finished, and they can no longer interfere. Similarly, tasks in F are not yet considered because they will only be handled once their activation time is reached and they become *open*.

In the example of Figure 1.12 from [11], we consider a set of tasks $\{n_0, \dots, n_{10}\}$ mapped onto $PE = 4$ cores as follows: $n_0, n_1, n_2 \mapsto \text{PE0}$, $n_3, n_4 \mapsto \text{PE1}$, $n_5, n_6, n_7 \mapsto \text{PE2}$, and $n_8, n_9, n_{10} \mapsto \text{PE3}$.

The time cursor is initialized at $t = 0$, with the set of currently alive tasks A empty. The algorithm then keeps updating the cursor to the next event time until all tasks are scheduled:

- **Closed tasks.** It is obtained by scanning the alive set A and checking whether $\text{rel}_\tau + \text{WCRT}_\tau = t$. These tasks are removed enabling successors to start. *Example:* $C = \{n_6\}$.
 - **Update Alive.** Remove closed tasks from the alive set. *Example:* $A = \{n_0, n_4, n_9\}$.
 - **Opening tasks.** Let O denote the set of tasks opening at time t . A task is added if its dependencies are satisfied and its minimal release date. *Example:* $O = \{n_7\}$.
 - **Update Alive.** Add newly opened tasks. *Example:* $A = \{n_0, n_4, n_7, n_9\}$.
 - **Interference update.** For each pair of tasks in A accessing the same memory bank, compute the interference and update the interference list.
- Interferences are computed per bank access, and the total interference of a task τ is the sum across banks.
- **Future tasks.** Tasks that have not yet been reached by the cursor and are therefore not handled at the moment. *Example:* $F = \{n_1, n_2, n_{10}\}$.

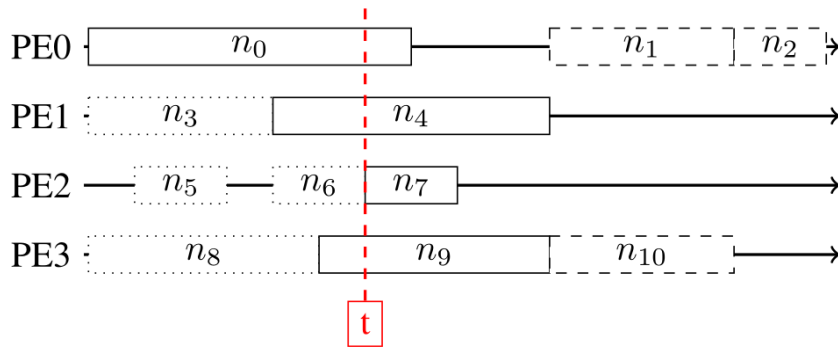


Figure 1.12: Cursor mechanism for task execution timeline

1.7 Conclusion

In this first chapter, we have reviewed the fundamental concepts that are necessary for understanding the remainder of this work.

We presented the theoretical foundations of CNNs in detail and their representation as DAGs, along with some notions of the RTS. We also examined the main aspects of the target Kalray MPPA hardware architecture, and introduced the MIA tool as a framework for interference-aware scheduling analysis.

This chapter of essential concepts and preliminaries provides us with the essential notions required for the foundation of our project. Having established a clearer vision, we are now prepared to move on to Chapter 2, where we describe the methodology and functioning of the proposed project.

Chapter 2

Methodology and Functional Description

2.1 Introduction

This chapter presents the step-by-step methodology we followed to complete our project. We begin with an end-to-end overview of the execution pipeline to introduce the main stages and to understand what is produced at each step. We then justify the choice of data formats used throughout the software development. Finally, we detail every stage of the process and clarify responsibilities using UML use-case diagrams.

For clarity of analysis, the CNN is partitioned into two zones:

- **CPL zone:** all convolutional and pooling layers.
- **MLP zone:** all FC (dense) layers.

The **Flatten** layer serves as an intermediate layer: it reshapes spatial feature maps into a vector and thus bridges CPL and MLP. For methodological completeness, we assume that it is included partially in both zones.

2.2 End-to-End Pipeline Overview

This section provides a high-level view of the workflow pipeline and its main steps. The pipeline has three main steps as it is shown in Figure 2.1:

- **First Step:** CNN-to-DAG conversion,
- **Second Step:** WCRT timing analysis,
- **Third Step:** Evaluation and reporting environment.

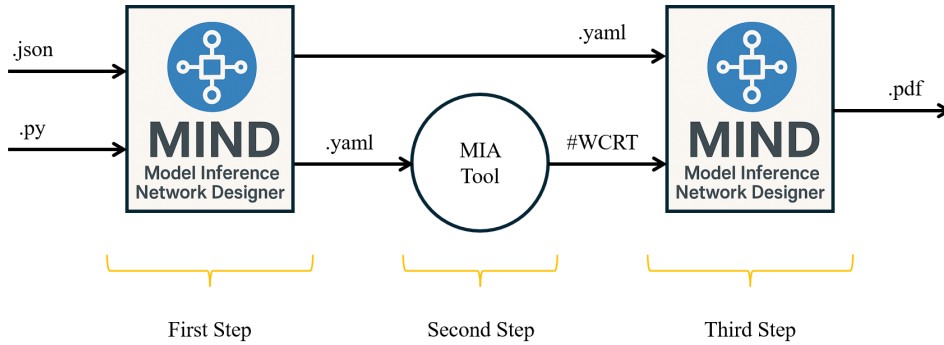


Figure 2.1: End-to-end workflow pipeline of our project

The MIND (Model Inference Network Designer), that we have developed during our project, is a Java software that supports two operating modes:

1. **CNN-to-DAG conversion (First Step):** MIND takes as input the CNN model provided either as `.json` or `.py`. It performs validation of the input file and conversion, and emits a single `.yaml` file representing the DAG.
2. **Evaluation & reporting environment (Third Step):** MIND takes as input its generated `.yaml` file together with the WCRT returned by MIA. It then computes evaluation metrics, displays some visualizations, and produces a summarized `.pdf` report of all the calculation made.

Regarding the **WCRT timing analysis (Second Step)**, the generated `.yaml` by MIND is passed to MIA, which computes the WCRT value. The computed WCRT value is then fed back into MIND software for use in the Third Step. The explanation of the MIA tool has been introduced previously in Section 1.6 in Chapter 1.

2.3 Used Format Selection

We justify in this section the input/output formats adopted in the pipeline, based on the following criteria:

- **Interoperability:** The ability for different tools and programming languages to R/W the files and use them correctly without custom converters or adapters.
- **Machine readability:** Ease with parsing and transforming the data.
- **Human readability:** How easily a person can inspect and understand the content without tooling.

2.3.1 .JSON (JavaScript Object Notation)

Used as an input to MIND for easy CNN model description.

- Widely supported and easily parsed in Java programs.
- Compact, unambiguous, human-readable hierarchical text representation.

2.3.2 .PY (Python)

Used simultaneously as an input to MIND for CNN model description.

- Contains only the CNN model (layers and activation functions) without all the Python code.
- Supports scripted model of AI Keras/PyTorch when a JSON file is unavailable.

2.3.3 .YAML (Yet Another Markup Language)

Used as an input/output of MIND and as an input of MIA for formal DAG representation.

- Human-readable configuration with comments.
- Clean structure for DAG representation fields and calculations used by MIA.

The .YAML choice was selected primarily for compatibility with MIA input format.

2.3.4 .PDF (Portable Document Format)

Used for reporting computed metrics in the final step.

- Portable format for the final evaluation report (figures, tables, diagrams, etc.).
- Universally viewable.

All this combination of formats balances ease of tooling, validation and testing, and supports long-term reproducibility for future upgrades.

2.4 UML Use-Case Driven Methodology

In this section, we present the methodology in detail using UML use-case diagrams. We begin with a global overview and then describe each step.

Recall that, in UML use-case diagrams, `<<include>>` denotes a mandatory subroutine, while `<<extend>>` denotes an optional, conditional behavior added to a base use-case. The label `<<uses>>` is a stereotype we apply to a dependency to make the data usage explicit, *i.e.*, not a special UML keyword, it simply annotates that a use-case uses a given artifact.

In the Figure 2.2, there are three subsystems: MIND (Step 1), MIA (Step 2), and MIND (Step 3). The user directly triggers one use-case per subsystem. Four artifacts structure the data flow: CNN model source (`.json/.py`), merged DAG (`<CPL>_<MLP>.yaml`), the WCRT value, and the final report (`.pdf`). Dashed dependencies labeled `<<uses>>` indicate which artifacts each use-case consumes.

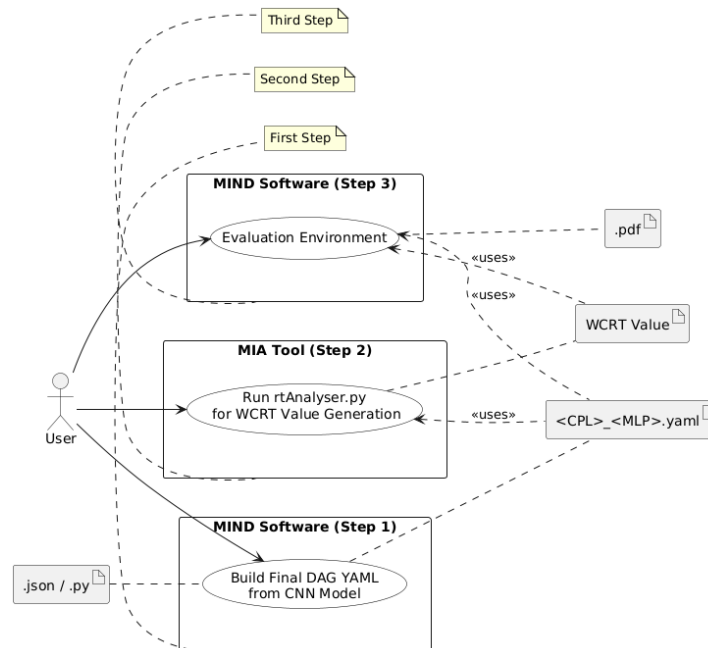


Figure 2.2: Global UML use-case overview of the three-step workflow

2.4.1 First Step

As we see in Figure 2.3, the User initiates five primary actions: loading input, configuring the model, exploring 3D/metrics, choosing architecture, and choosing the regrouping policy. The execution order is enforced by a chain of `<<include>>`: “Validate & Normalize” depends on “Load Input”; “Configure Model” includes “Validation”; “3D Views & Arithmetic/MAC Calculation” includes “Configuration”; “Select Target Architecture” includes the “3D/Metrics” step; “Choose Regrouping Policy” includes “Architecture Selection”; “Generate YAML Parts” includes “Policy Selection”; finally “Merge for Final YAML” includes “Parts Generation”. The “Error Handler” extends several steps when formats are invalid.

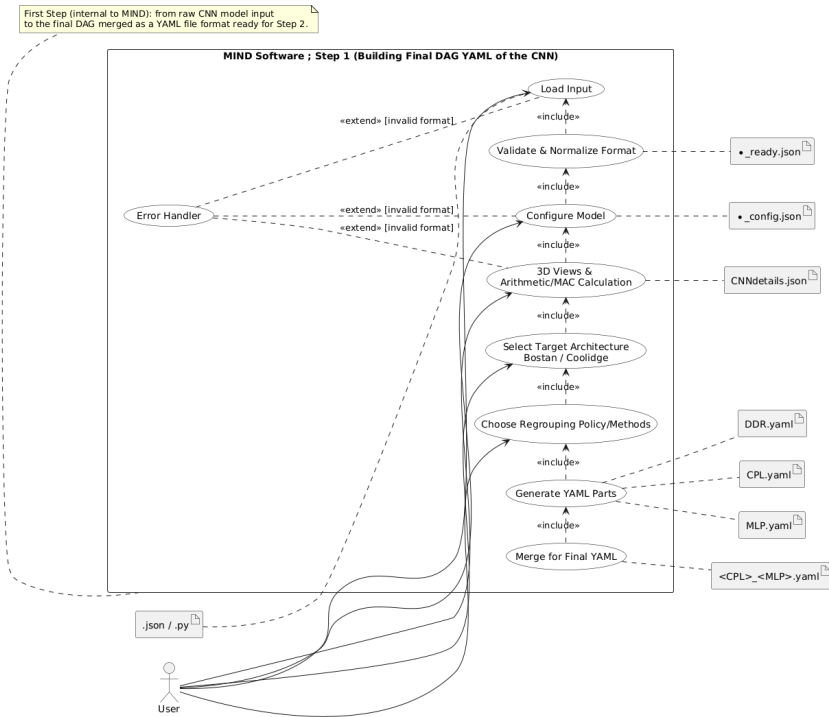


Figure 2.3: UML use-case representation of the First Step; CNN-to-DAG construction

Artifacts document each milestone: `.json/.py` (raw input), `*_ready.json` (normalized), `*_config.json` (configured), `CNNdetails.json` (computed metrics), `DDR.yaml` / `CPL.yaml` / `MLP.yaml` (partials), and the merged `CPL_MLP.yaml` is the final file for Step 1 ready for Step 2.

2.4.2 Second Step

Next, in Figure 2.4, we notice that the pipeline of MIA tool execution is linear and fully mandatory via `<<include>>`: “Check YAML Conformance” includes “Load Final YAML”; “Run rtAnalyser” includes “Validation”; “Compute Global WCRT” includes the “Run”; “Show WCRT Result” includes the “Computation”.

It is the User who triggers the run and export.

Artifacts clarify execution and outputs: the merged `.yaml` (input), `rtAnalyser.py` (executable), and the WCRT Value (output metric) that we will be needing in Step 3.

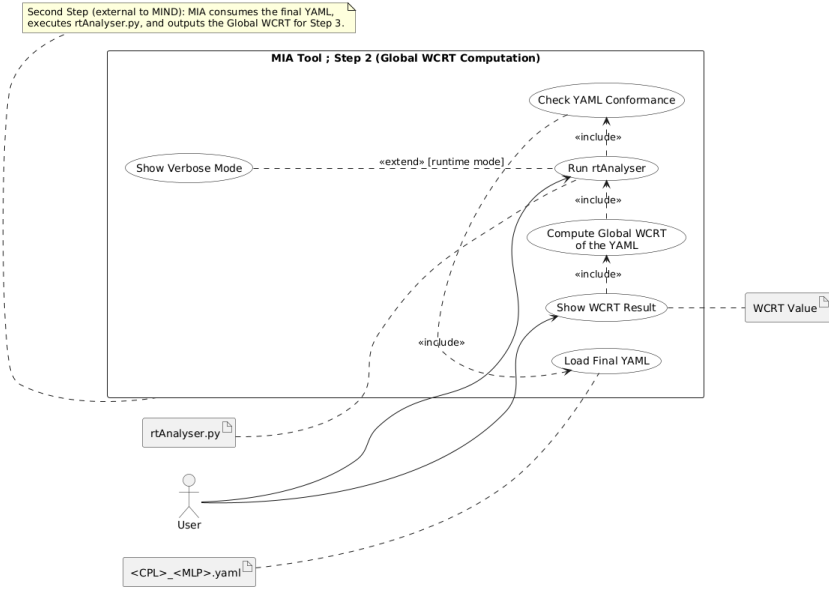


Figure 2.4: UML use-case representation of the Second Step; Global WCRT computation

2.4.3 Third Step

Finally, as shown in Figure 2.5, the two inputs are required to prepare the dashboard: “Load Final YAML Path” and “Provide the Global WCRT Value”; both are included by “Select & Open Evaluation Environment”.

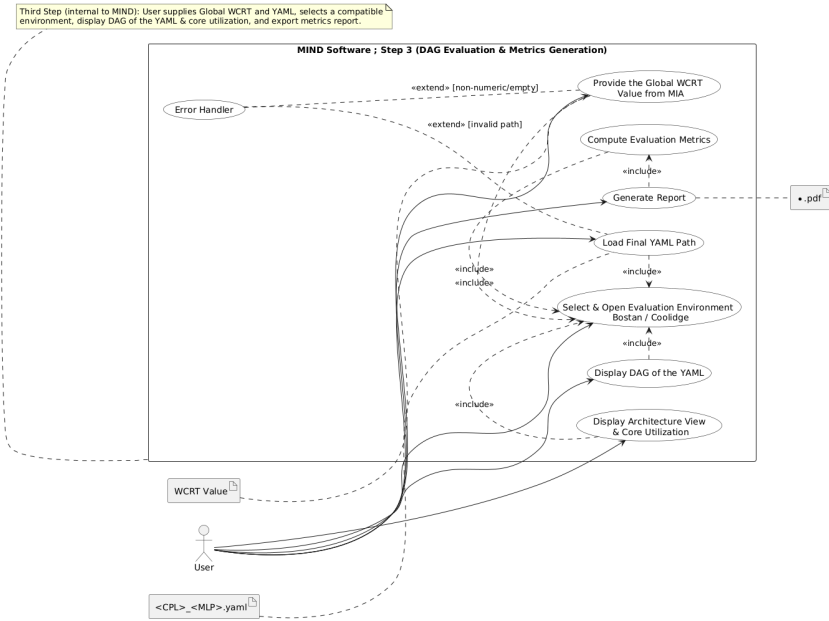


Figure 2.5: UML use-case representation of the Third Step; Evaluation & reporting

From this dashboard, the User includes three analysis tools: “Display DAG”, “Display Architecture View & Core Utilization”, and “Compute Evaluation Metrics”.

The use-case “Generate Report” includes the “Computed Metrics” to produce a .pdf file. And for managing errors, “Error Handler” extends input steps on invalid paths or non-numeric or empty WCRT Value.

Artifacts make dependencies explicit: the merged `.yaml`, the numeric WCRT Value, and the generated `.pdf` report.

2.5 MIND Software Demonstration

After presenting the functionalities of the MIND software using UML diagrams, this section illustrates its workspace windows. Figure 2.6 illustrates the main interface of this software.

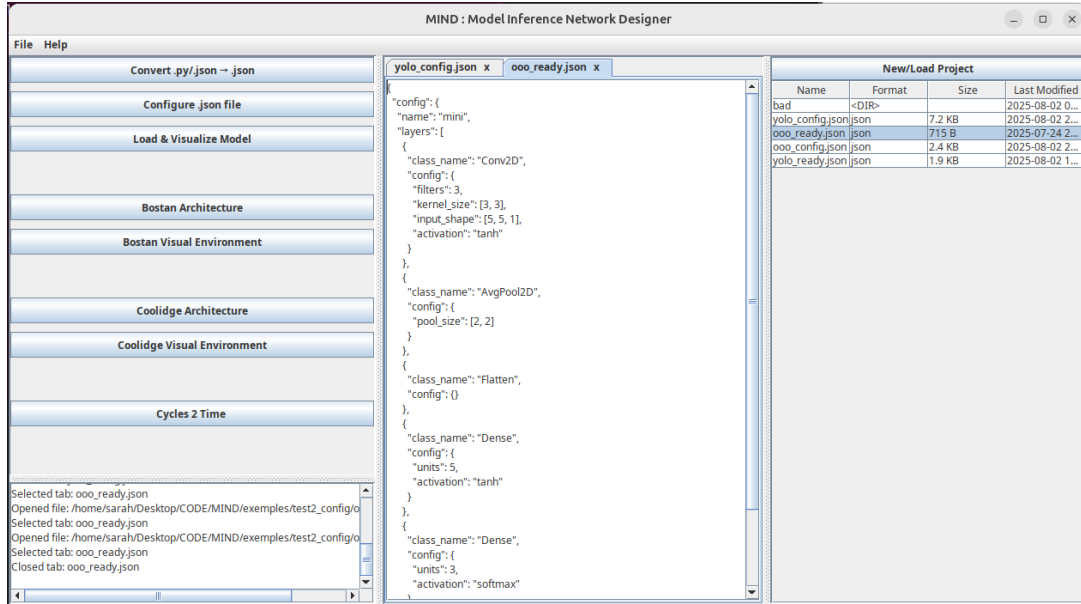


Figure 2.6: Illustration of the MIND software

On the right-hand side, the panel *New/Load Project* allows users to either create a new project or reopen existing ones. Once a project is loaded, its associated files can be inspected through different tabs inside MIND. On the left, a set of dedicated buttons provides the core functionalities of the software: the **Convert Python/JSON to JSON** button generates a normalized and standard JSON description of the CNN model that can be exploited inside MIND; the **Configure JSON file** button enables the configuration of the standardized JSON as well as the CNN parameters; and the **Load & Visualize Model** button allows the 3D visualization of the model together with the computation of arithmetic metrics. An example of the 3D model visualization is presented in Chapter 5.

The two buttons **Bostan Architecture** and **Coolidge Architecture** are used to select the target architecture and, within each, to specify the regrouping policies at the CPL and MLP zones. The buttons **Bostan Visual Environment** and **Coolidge Visual Environment** provide access to the corresponding architectural environments. In Figure 2.7, the Coolidge environment view is shown. The **Cycles 2 Time** button converts cycles into time according to each architecture's frequency. At the bottom of the interface, a terminal panel displays MIND system exchanges and execution traces.

All calculations performed within this program are discussed theoretically in detail in the following chapters.

Next, Figure 2.7 presents the Coolidge architecture environment in MIND, showing its clusters, cores, DDR, and SRAM memories.

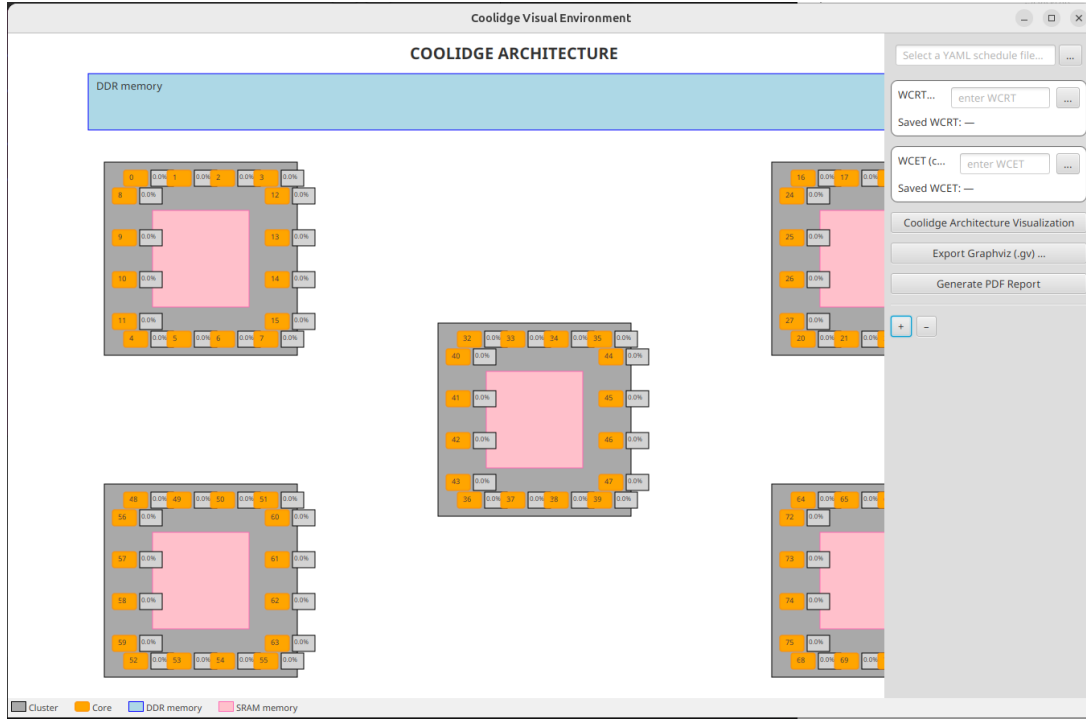


Figure 2.7: Visualization of the **MIND environment** showing the Coolidge architecture

When a YAML file is loaded, the environment parses it and assigns each task to a specific core within this architectural visualization. In this view, the WCRT (together with the interference calculated by MIA) can be specified, whereas the WCET corresponds only to a value without interference. Moreover, the software allows the generation of a DAG file in Graphviz format (.gv) and the export of a PDF report summarizing all calculations, which will be analyzed and discussed in the final chapter, Chapter 5.

Finally, Figure 2.8 shows the Graphviz view of a DAG generated from a CNN model example using the MIND environment, where each node corresponds to an operation, black edges represent dependencies, and red edges indicate transferred data.

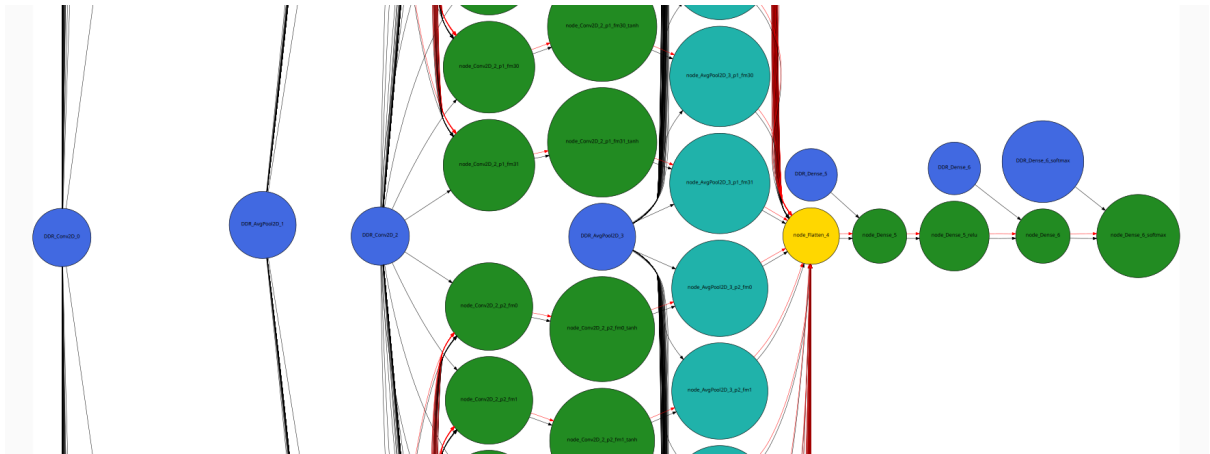


Figure 2.8: Partial view of a DAG automatically generated from a CNN model using the MIND environment.

For clarity, the shown figure displays only a cropped portion of the complete DAG. As mentioned earlier, this DAG can be generated directly within the architectural MIND

environment. It should be noted that this Graphviz DAG window itself is an interactive window.

2.6 Conclusion

This chapter has established the methodology and functional flow of our toolchain. We presented the end-to-end pipeline $\text{MIND} \rightarrow \text{MIA} \rightarrow \text{MIND}$, justified the choice of exchange formats, and specified responsibilities using UML use-case diagrams for the three steps, and then we presented several screenshots of the developed program, each accompanied by an explanation.

The central challenge that appears is the “faithful conversion of a CNN model into an analysable DAG” that includes: extracting a correct dependency structure, propagating shapes through layers, placing activations consistently, handling the Flatten transition, and crucially choosing an appropriate node granularity so that each task can be annotated with the suitable RTS constraints on the target architectures.

To address this challenge, the next chapter, named Introducing DAG Node Regrouping Policies, formalizes the regrouping strategies that we created to control node granularity and structure. These policies ensure semantic fidelity, while improving analyzability and mapping efficiency on Bostan and Coolidge.

Chapter 3

Introducing DAG Nodes Regrouping Policies

3.1 Introduction

In this chapter, we address the CNN-to-DAG conversion challenge by introducing a family of mathematically defined “DAG node regrouping policies”, specified per CNN layer type. Our approach follows a **Top-Down Modeling** strategy, *i.e.*, we begin with high-level semantics-preserving groupings, and progressively refine node granularity to expose the computation required for timing analysis in the following chapter, Chapter 4.

We first present the baseline initial representations that motivated our designs: **MMM** and **MDM**, which are both used in the CPL and MLP zones.

In the following sections, we then detail the advanced regrouping policies: **MLWM**, **SSP**, **RCCT**, and **DF**. Note that **MLWM**, **SSP**, and **RCCT** target the CPL zone only, whereas **DF** is specific to the MLP zone.

The chapter concludes with graphical illustrations of each advanced policy to clarify node/edge semantics and facilitate understanding.

3.2 Initial Methods for Representing CNNs as DAGs

We begin by introducing two extreme DAG representations for CNNs: **MDM**, very high per-operation granularity, which models each primitive operation as its own node, and **MMM**, very low merged granularity, which combines computations at the layer level. The following subsections detail each method separately.

3.2.1 Massively Merged Method (MMM)

This method consists of considering each CNN model layer as one DAG node, as it is illustrated in Figure 3.1.

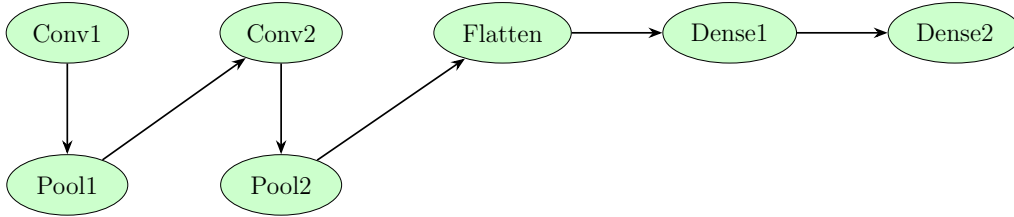


Figure 3.1: DAG representation of a CNN architecture in MMM method

Limitations of the MMM Method. By merging each layer into a single node, MMM exposes no parallelism. This makes it not suited to our many-core target, and it generates a pessimistic (maximal) WCRT in cycles. Such representations is useful as a mono-core baseline rather than parallel architectures.

Reason for Keeping MMM in our Study. We will not remove MMM in our toolchain and compute it in MIND software, because it provides an upper-bound reference for WCRT. This pessimistic baseline will serve helpful as a comparison point in the Test and Evaluation chapter, Chapter 5.

3.2.2 Massively Decomposed Method (MDM)

This method consists of considering each operation per element in a CNN layer as one DAG node, as it is illustrated in Figure 3.2.

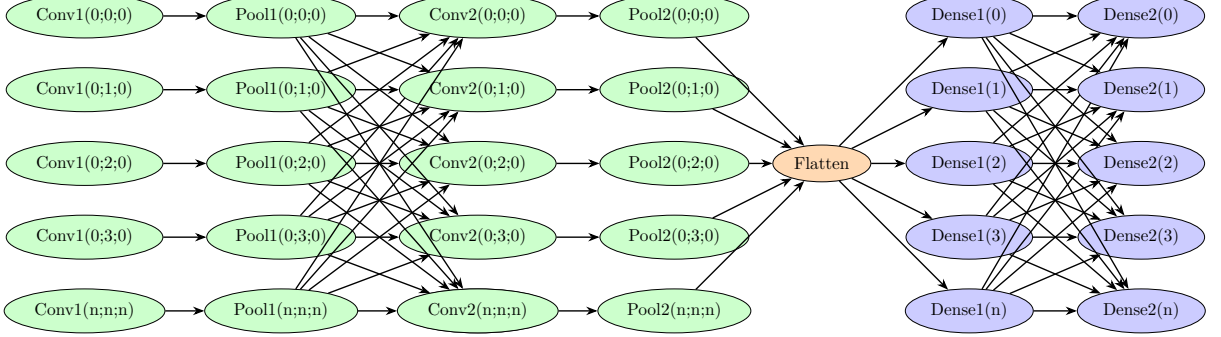


Figure 3.2: DAG representation of a CNN architecture in MDM method

Limitations of the MDM Method. MDM exposes an extremely fine-grained level of parallelism per output element operations. In practice, this generates far more tasks than available cores even on many-core systems, making both mapping and per-core ordering complex and costly. Such fine granularity effectively represents an optimistic (ideal minimal) lower bound on WCRT cycles, because under an idealized setting with infinite cores, each task could be mapped to its own core, which will lead to simultaneous execution.

Reason for Not Keeping MDM in our Study. We will not implement MDM because the task explosion is prohibitive. For example, in LeNet-5 CNN model [12] with a grayscale input of $32 \times 32 \times 1$ and a first convolutional layer of $5 \times 5 \times 6$ (stride 1, dilation 1, with valid padding), the result of this layer according to MDM method will be $28 \times 28 \times 6 = 4,704$ output elements convolution nodes for the first layer alone. The next layers then, can easily exceed 8,000 nodes.

On Bostan (256 cores), we notice that in this method, the number of tasks vastly exceeds the number of cores, making mapping and ordering computationally heavy and sometimes impossible. We therefore treat MDM as a conceptual perfect/ideal unattainable lower-bound reference, and exclude it from our implemented evaluations, focusing instead on practicable regrouping policies between MDM and MMM.

3.2.3 Motivation for Advanced Regrouping Policies

The baseline initial methods, MDM and MMM, sit at opposite ends of the granularity spectrum and each presents practical limitations for many-core timing analysis: MDM induces a prohibitive task explosion that complicates mapping and scheduling, whereas MMM hides exploitable parallelism and generates pessimistic WCRT estimates. To bridge this gap, we introduce a set of advanced DAG regrouping policies in the following sections that balance semantic exploitation and fidelity with analyzability node counts.

These policies arise from our own analysis of CNN topology and execution characteristics; to the best of our knowledge, they are not adapted or inspired from prior work. Our two main design goals are: (i) preserve the computational semantics of each layer, thereby

guaranteeing semantic fidelity, (ii) expose useful parallelism without overwhelming the scheduler.

3.3 Regrouping Policy 1: Macroscopic Layer-Wise Modeling (MLWM)

Given the spatial dimensions, respectively height and width (H_l, W_l) , the channel count D_l , and a layer index l . The first advanced MLWM regrouping policy models each DAG node n_l as the complete operation applied to an entire feature map or, in the case of the first layer, it is applied to the input image. We denote:

$$n_l = (I_l, O_l), \quad \text{where} \quad \begin{cases} I_l \in \mathbb{R}^{H_l \times W_l \times D_l} & \text{(input feature map)} \\ O_l \in \mathbb{R}^{H_{l+1} \times W_{l+1} \times D_{l+1}} & \text{(output feature map)} \end{cases} \quad (3.1)$$

The exact per-layer calculations for the MLWM policy were detailed earlier in Section 1.2 in Chapter 1.

3.4 Regrouping Policy 2: Structured Spatial Partitioning (SSP)

We define a second regrouping policy based on structured spatial subdivision of the input image. The partitioning is applied once at the input image, and the same partitioning structure is then propagated across all the rest of the layers. Each partition is treated as an independent block throughout the CNN inference. The convolution, pooling, and activation operations are applied independently within each partition block. This policy decomposes each macroscopic node n_l from MLWM policy into several spatially local sub-nodes, enabling finer-grained task decomposition to better exploit parallelism. We then, introduce a partitioning operator denoted by $\lambda : \mathbb{N}^* \times \mathbb{N}^*$.

3.4.1 Design Principle and Local Padding Strategy (LPS)

Given a tuple $(h, w) \in \mathbb{N}^{*2}$, the operator $\lambda(h, w)$ divides the spatial dimensions of the input feature map I_l into a uniform grid of $k = h \cdot w$ blocks. This division will result in a set of sub-nodes denoted by $n_{l,i}$.

Applying $\lambda(h, w)$ to the input I_l :

$$\{n_{l,i} = (\lambda(h, w)I_l, O_l) = (I_{l,i}, O_{l,i})\}_{i=0}^{k-1} \quad \text{with} \quad k = h \cdot w \quad (3.2)$$

Each sub-node $n_{l,i}$ operates on a spatial region $I_{l,i}$ of the input and produces an output $O_{l,i}$. Their new tensor shapes will be then:

$$\begin{cases} I_{l,i} \in \mathbb{R}^{H_{l,i} \times W_{l,i} \times D_l} \\ O_{l,i} \in \mathbb{R}^{H_{l+1,i} \times W_{l+1,i} \times D_{l+1}} \end{cases} \quad (3.3)$$

And the new spatial dimensions $H_{l,i}$ and $W_{l,i}$ of the input feature map are given by:

$$\begin{aligned} H_{l,i} &= \left\lceil \frac{H_l}{h} \right\rceil, \quad \text{with} \quad H_l \in \mathbb{N}^* \quad \text{and} \quad 1 \leq h < H_l \\ W_{l,i} &= \left\lceil \frac{W_l}{w} \right\rceil, \quad \text{with} \quad W_l \in \mathbb{N}^* \quad \text{and} \quad 1 \leq w < W_l \end{aligned} \quad (3.4)$$

When the resulting values of $H_{l,i}$ and $W_{l,i}$ are not integers, instead of ignoring the left-over regions, we take the higher integer value of the division with rounding up by adding zero-padding row or column by default to the input image along the corresponding borders (applying LPS). This LPS method ensures that the residual regions are handled without loss of spatial information, thereby introducing our first fidelity-preserving operation.

Example. Let the spatial dimensions of the input feature map at layer l be:

$$H_l = 7, \quad W_l = 5$$

Assume the partitioning operator is chosen as $\lambda(3, 2)$. Then, the new spatial dimensions $H_{l,i}$ and $W_{l,i}$ of the partitioned input tiles are given by:

$$\begin{aligned} H_{l,i} &= \left\lceil \frac{H_l}{h} \right\rceil = \left\lceil \frac{7}{3} \right\rceil = \lceil 2.3 \rceil = 3 & (2 + \text{zero padding}) \\ W_{l,i} &= \left\lceil \frac{W_l}{w} \right\rceil = \left\lceil \frac{5}{2} \right\rceil = \lceil 2.5 \rceil = 3 & (2 + \text{zero padding}) \end{aligned}$$

Note that since the original dimensions are not perfectly divisible by the SSP operator, zero-padding is applied along the bottom and right edges to ensure complete coverage.

3.4.2 Overlap Padding Strategy (OPS)

Due to the non-local nature of operations in CNN layers such as convolution and pooling, we must preserve, for each sub-block, a valid neighbourhood overlap, *i.e.*, the shared region between adjacent spatial blocks. Otherwise, the computations may be incorrect. To address this, we introduce a local padding strategy by calculating the needed overlap in order to ensure correct computation:

Calculating the Overlap

Let a convolution or pooling kernel (window) be of size $r_H \times r_W$. We split the required overlap asymmetrically:

$$\begin{aligned} \delta_H^{\text{top}} &= \left\lfloor \frac{r_H - 1}{2} \right\rfloor, & \delta_H^{\text{bottom}} &= (r_H - 1) - \delta_H^{\text{top}} \\ \delta_W^{\text{left}} &= \left\lfloor \frac{r_W - 1}{2} \right\rfloor, & \delta_W^{\text{right}} &= (r_W - 1) - \delta_W^{\text{left}} \end{aligned} \quad (3.5)$$

Thus:

$$\delta_H^{\text{top}} + \delta_H^{\text{bottom}} = r_H - 1, \quad \delta_W^{\text{left}} + \delta_W^{\text{right}} = r_W - 1 \quad (3.6)$$

We assign the lower integer value of the overlap to the “top” and “left” sides, and place the remainder on the “bottom” and “right”, so that both odd and even kernels are handled without gaps, ensuring accurate coverage.

Example. In the case of an odd kernel, *e.g.*, $r_H = 3$:

$$\delta_H^{\text{top}} = \left\lfloor \frac{3-1}{2} \right\rfloor = \left\lfloor \frac{2}{2} \right\rfloor = 1, \quad \delta_H^{\text{bottom}} = (3-1) - 1 = 2 - 1 = 1$$

This results in symmetric padding:

$$\delta_H^{\text{top}} + \delta_H^{\text{bottom}} = 1 + 1 = 2$$

In the case of an even kernel, *e.g.*, $r_H = 4$:

$$\delta_H^{\text{top}} = \left\lfloor \frac{4-1}{2} \right\rfloor = \left\lfloor \frac{3}{2} \right\rfloor = 1, \quad \delta_H^{\text{bottom}} = (4-1) - 1 = 3 - 1 = 2$$

This results in asymmetric padding:

$$\delta_H^{\text{top}} + \delta_H^{\text{bottom}} = 1 + 2 = 3$$

Calculating the Exact Padded Input

Here we can clearly see the difference between the unpadded input tensor $I_{l,i}$ of a sub-node $n_{l,i}$, and the padded input tensor $I'_{l,i}$ after padding has been applied:

$$I_{l,i} \in \mathbb{R}^{H_{l,i} \times W_{l,i} \times D_l} \xrightarrow{\text{padding}} I'_{l,i} \in \mathbb{R}^{(H_{l,i} + \delta_H^{\text{top}} + \delta_H^{\text{bottom}}) \times (W_{l,i} + \delta_W^{\text{left}} + \delta_W^{\text{right}}) \times D_l} \quad (3.7)$$

The extra rows and columns are replicated from neighboring blocks only when the SSP partitioning operator $\lambda(h, w)$ is applied. If applied, the two partitioned blocks (sub-nodes) share a common boundary.

Next, we define a boolean indicator function $\mathbb{P}(n_{l,i}, \text{dir})$, which returns 1 if the sub-node $n_{l,i}$ shares a boundary with another sub-node in the direction $\text{dir} \in \{\text{top}, \text{bottom}, \text{left}, \text{right}\}$, and 0 otherwise. Then, the final local padding applied to sub-block $n_{l,i}$ becomes:

$$\begin{cases} \Delta_H^{\text{top}}(i) &= \mathbb{P}(n_{l,i}, \text{top}) \cdot \delta_H^{\text{top}} \\ \Delta_H^{\text{bottom}}(i) &= \mathbb{P}(n_{l,i}, \text{bottom}) \cdot \delta_H^{\text{bottom}} \\ \Delta_W^{\text{left}}(i) &= \mathbb{P}(n_{l,i}, \text{left}) \cdot \delta_W^{\text{left}} \\ \Delta_W^{\text{right}}(i) &= \mathbb{P}(n_{l,i}, \text{right}) \cdot \delta_W^{\text{right}} \end{cases} \quad (3.8)$$

And the final padded input tensor shape, which represents our second fidelity-preserving operation in the SSP policy, is thus:

$$\tilde{I}_{l,i} \in \mathbb{R}^{(H_{l,i} + \Delta_H^{\text{top}} + \Delta_H^{\text{bottom}}) \times (W_{l,i} + \Delta_W^{\text{left}} + \Delta_W^{\text{right}}) \times D_l} \quad (3.9)$$

Both used LPS and OPS padding strategies ensure that every position within the sub-blocks has a complete independent field without boundary leakage, regardless of:

- Whether the input image dimension is even or odd; we use LPS.
- Whether the kernel dimensions are even or odd; we the OPS.

Those approaches guarantee the correctness of computations near partition boundaries without any information loss.

3.5 Regrouping Policy 3: Redundant Convolution via Channel Tiling (RCCT)

The RCCT policy reduces the massive resulting network within convolution operations caused by all the required input feature maps, since each convolution must consume all input feature maps to compute a single output feature map. This policy addresses and resolves this problem by partitioning the input channels into independent blocks. Each block will be computed each time redundantly by the full kernel set, allowing partial convolutions to be computed in parallel which will produce an intermediate layer. The final output layer is obtained by summing these partial results per output channel, thereby preserving correct convolution semantics.

3.5.1 Partitioning Principle

The input tensor and the convolutional kernel tensor at layer l are defined previously:

$$I_l \in \mathbb{R}^{H_l \times W_l \times D_l} \quad \text{and} \quad K_l \in \mathbb{R}^{k_H \times k_W \times D_l \times D_{l+1}}$$

We apply a channel tiling operator $\zeta(c)$, with $c \in \mathbb{N}^*$, which partitions the input channels into c disjoint blocks indexed by i . The partitioning is expressed as:

$$D_{l,i} = \frac{D_l}{c} \in \mathbb{N}^*, \quad \text{where} \quad \sum_{i=1}^c D_{l,i} = D_l \quad \text{and} \quad I_l = \bigcup_{i=1}^c I_{l,i} \quad (3.10)$$

For each tile $I_{l,i}$ we apply the calculation of the full kernel set K_l . And for each output channel j in block i , we define the partial output $O_{l,i}^j \in \mathbb{R}^{H_l \times W_l \times D_{l+1,i}}$, resulting from the convolution between the tile input $I_{l,i} \in \mathbb{R}^{H_l \times W_l \times D_{l,i}}$ and each kernel K_l^j , where $j \in D_{l+1,i}$. These correspond to the first resulting DAG nodes $n_{l,i}^j$. We compute:

$$O_{l,i}^j = I_{l,i} * K_l^j \quad (3.11)$$

The total number of resulting DAG nodes $n_{l,i}^j$, which corresponds to the total number of intermediate output channels $D_{l+1,i}$, is given by:

$$Total(n_{l,i}^j) = \sum_{i=1}^c \sum_{j=1}^{D_{l+1,i}} n_{l,i}^j = \sum_{i=1}^c D_{l+1,i} = c \cdot D_{l+1,i} \quad (3.12)$$

The final output feature map, which corresponds to the final set of nodes $n_{l,i}$ in the DAG, is reconstructed by summing the corresponding partial outputs across all c tiles. For each output channel $j \in D_{l+1,i}$:

$$O_l(j) = \sum_{i=1}^c O_{l,i}^j \quad (3.13)$$

The number of the final output channels D_{l+1} is equal to the number of existing kernels K_l in the convolution layer. Thus:

$$D_{l+1} = \frac{1}{c} \sum_{i=1}^c D_{l+1,i} \quad (3.14)$$

3.5.2 Channel Padding Strategy (CPS)

When the number of input channels D_l is not divisible by the number of tiles c as defined earlier in the Equation (3.10):

$$D_{l,i} = \frac{D_l}{c} \notin \mathbb{N}^* \quad (3.15)$$

As a solution, we apply a zero-channel padding strategy; we introduce adding dummy channels filled with zeros of the same dimension as the existing feature maps, forming a tensor of shape $H_l \times W_l \times \delta$, with:

$$\delta = c \cdot \left\lceil \frac{D_l}{c} \right\rceil - D_l \quad (3.16)$$

The padded input tensor becomes:

$$\bar{I}_l \in \mathbb{R}^{H_l \times W_l \times (D_l + \delta)} \quad \text{with} \quad \bar{I}_l = \text{Concat}(I_l, \mathbf{0}_\delta) \quad (3.17)$$

Example. Let $H_l = W_l = 28$, $D_l = 13$. We choose $\zeta(4)$ with $c = 4$ channel tiles. Then we calculate $D_{l,i}$:

$$D_{l,i} = \left\lceil \frac{D_l}{c} \right\rceil = \left\lceil \frac{13}{4} \right\rceil = \lceil 3, 25 \rceil = 4$$

Since the result is not an integer, we compute δ as follows:

$$\delta = c \cdot \left\lceil \frac{D_l}{c} \right\rceil - D_l = 4 \cdot 4 - 13 = 3$$

We append 3 zero-channels to obtain:

$$\bar{D}_l = D_l + \delta = 13 + 3 = 16 \quad \text{with} \quad \bar{I}_l = \text{Concat}(I_l, \mathbf{0}_3) \in \mathbb{R}^{28 \times 28 \times 16}$$

The channels now can be divided by $c = 4$ blocks, with 4 channels in each block:

$$D_{l,i} = \left\lceil \frac{\bar{D}_l}{c} \right\rceil = \left\lceil \frac{16}{4} \right\rceil = 4$$

This ensures that $\bar{D}_l = D_l + \delta$ is divisible by c , allowing balanced tiling into c blocks. The zero-padded channels do not affect the final result. However, if this padding strategy is not applied and the remainder from operations is neglected, there is a risk of removing a significant number of feature maps, potentially resulting in information loss. The CPS strategy is our fidelity-preserving operation in the RCCT policy.

3.6 Regrouping Policy 4: Dense Fusion (DF)

Dense layers differ from convolutional or pooling layers; they produce an output vector of neurons rather than operating on spatial feature maps. In this DF regrouping policy, we reduce granularity by clustering the output neurons into fused execution groups.

We introduce a fusion operator $\iota(g)$, denoted by:

$$\iota(g) : \mathbb{N}^* \quad \text{which leads to} \quad \iota(g) : \{1, \dots, N_{l+1}\} \longrightarrow \{1, \dots, g\}$$

which assigns the N_{l+1} output neurons of the dense layer l into g ; the desired number of fused execution groups.

3.6.1 Formal Description and Neuron Padding Strategy (NPS)

Let i be the number of parts after the fusion operation, the resulting number of neurons per group $N_{l+1,i}$ is:

$$N_{l+1,i} = \iota(g) \cdot N_{l+1} = \left\lceil \frac{N_{l+1}}{g} \right\rceil \quad (3.18)$$

In some cases, N_{l+1} may not be divisible by g , we then introduce Dummy Neurons ϵ :

$$N_{l+1}^{\text{pad}} = g \cdot N_{l+1,i}, \quad \epsilon = N_{l+1}^{\text{pad}} - N_{l+1} \quad (3.19)$$

Thus, the layer is padded with ϵ Dummy Neurons with zero weights and biases. This approach represents our first fidelity-preserving operation named NPS strategy:

$$W_l^{\text{pad}} = \begin{bmatrix} W_l \\ \mathbf{0}_{\epsilon \times N_l} \end{bmatrix} \in \mathbb{R}^{N_{l+1}^{\text{pad}} \times N_l}, \quad \mathbf{b}_l^{\text{pad}} = \begin{bmatrix} \mathbf{b}_l \\ \mathbf{0}_\epsilon \end{bmatrix} \in \mathbb{R}^{N_{l+1}^{\text{pad}}} \quad (3.20)$$

We note that in the case $g = 1$, no fusion is applied: the neurons are not grouped in that layer and each one is calculated individually following the MDM method. However, when $g \rightarrow +\infty$, the layer is heavily fused and the computation tends to follow the MMM approach with a single node.

3.6.2 Group Computation

For each part of the fused node $i \in \{1, \dots, g\}$, we define the new neuron fusion $\mathcal{D}_i$ as:

$$\mathcal{D}_i = \{(i-1)N_{l+1,i} + 1, \dots, i \cdot N_{l+1,i}\} \quad (3.21)$$

The final layer computation of each group i is then:

$$O_{l,i} = f(W_l^{\text{pad}}[\mathcal{D}_i, :] \cdot \mathbf{x}_l + \mathbf{b}_l^{\text{pad}}[\mathcal{D}_i]) \quad (3.22)$$

Example. Let $N_{l+1} = 7$ and we choose to fuse this layer into 2 parts; $g = 2$. Then:

$$N_{l+1,i} = \left\lceil \frac{7}{2} \right\rceil = 4, \quad N_{l+1}^{\text{pad}} = 8, \quad \epsilon = 1$$

The padded weight matrix has one dummy neuron. We obtain:

$$\mathcal{D}_1 = \{1, 2, 3, 4\}, \quad \mathcal{D}_2 = \{5, 6, 7, 8\}$$

Each fused node computes:

$$\begin{aligned} O_{l,1} &= f(W_l^{\text{pad}}[\mathcal{D}_1, :] \cdot \mathbf{x}_l + \mathbf{b}_l^{\text{pad}}[\mathcal{D}_1]) \in \mathbb{R}^4, \\ O_{l,2} &= f(W_l^{\text{pad}}[\mathcal{D}_2, :] \cdot \mathbf{x}_l + \mathbf{b}_l^{\text{pad}}[\mathcal{D}_2]) \in \mathbb{R}^4 \end{aligned}$$

3.6.3 Masking Strategy (MS)

This subsection is inspired by the general masking approaches presented in [13], where Hadamard-product masks are applied in DL models to suppress irrelevant entries. In our work, this is not a direct copy of their approach; rather, it is an adaptation of the general masking principle to a new innovative context, which is the treatment of Dummy Neurons (DNs) generated by applying NPS strategy in the DF policy. We reformulate their idea so that masking can be applied within dense layers to safely eliminate the influence of these DNs while preserving the integrity of the original coherent computation. This MS strategy is our second fidelity-preserving operation used in the DF policy.

As discussed previously in Equation (3.20), when the number of output neurons N_{l+1} is not divisible by the chosen group size g , we introduce ϵ Dummy Neurons. These DNs have zero weights and biases, but if left unchecked, they may generate undesired values (noise) in activation functions:

- For ReLU, LeakyReLU or Tanh activations, the dummy output is $f(0) = 0$, which is safe in this case.
- For Sigmoid, however, $f(0) = \sigma(0) = 0.5 \neq 0$. This means that a dummy neuron would propagate a constant non-zero value into the next layer, which is incorrect and may introduce noise and errors in calculation which will affect the accuracy.
- Also for Softmax, dummy indices would steal probability mass, disturbing the entire distribution which will give a completely wrong output result of the CNN model.

To eliminate this problem without changing the vector size, we define a binary mask vector $m \in \{0, 1\}^{N_{l+1}^{\text{pad}}}$:

$$m_i = \begin{cases} 1, & 1 \leq i \leq N_{l+1} \\ 0, & N_{l+1} < i \leq N_{l+1}^{\text{pad}} \end{cases}$$

This mask simply marks the “real” neurons with 1 and the “dummy” neurons with 0.

Step 1: Pre-activations. The dense layer produces the pre-activation vector $\mathbf{y}_l$ of dimension N_{l+1}^{pad} :

$$\mathbf{y}_l = W_l^{\text{pad}} \mathbf{x}_l + \mathbf{b}_l^{\text{pad}}$$

This pre-activated equation either goes for Sigmoid calculation (Step 2.1) or Softmax calculation (Step 2.2):

Step 2.1: Masked Sigmoid. Normally, we would compute:

$$\mathbf{O}_l = f(\mathbf{y}_l)$$

But this still includes dummy outputs. To cancel them, we apply the mask element-wise:

$$\widehat{\mathbf{O}}_l = \mathbf{O}_l \odot \mathbf{m} \quad (3.23)$$

Where $\odot$ denotes the Hadamard¹ product. This ensures that all dummy positions are exactly zero, regardless of the activation function. For example, even if $\sigma(0) = 0.5$ for a Sigmoid, multiplying by $m_i = 0$ sets it back to zero.

¹The Hadamard product, also called the element-wise product, is an operation between two vectors (or matrices) of the same size. If $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, then their Hadamard product is $\mathbf{a} \odot \mathbf{b} = (a_1 b_1, a_2 b_2, \dots, a_n b_n)^\top$. Unlike the usual classical matrix product. In this case, each component is multiplied independently.

Step 2.2: Masked Softmax. For the special case of Softmax, the masking must be applied inside the normalization function to avoid allocating probability mass to dummy indices. The correct formula is:

$$\mathbf{o}_{l,i} = \frac{e^{\mathbf{y}_i} m_i}{\sum_{j=1}^{N_{l+1}^{\text{pad}}} e^{\mathbf{y}_j} m_j} \quad \text{for} \quad i = 1, \dots, N_{l+1}^{\text{pad}} \quad (3.24)$$

- If $m_i = 1$ (a real neuron), the numerator is unchanged, and the term participates normally.
- If $m_i = 0$ (a DN), the numerator is zero and the denominator ignores that index. Thus, the DN is completely excluded from the probability distribution.

The same thing for m_j , if we mask the numerator only but not the denominator, the DNs still contribute a positive term $e^{\mathbf{y}_j}$ to the normalization. This artificially inflates the denominator and reduces the probability assigned to the real neurons. By multiplying with m_j in the denominator, we guarantee that only the valid neurons ($m_j = 1$) are included in the normalization term.

3.7 Graphical Representation of All the Advanced Regrouping Policies

In this section, first, we present a simple example of a CNN architecture and represent it graphically. We then apply various regrouping policies to illustrate their effects and to facilitate comprehension.

The illustration in Figure 3.3 represents a small CNN architecture designed for demonstrative purposes only. The input is an image of size $6 \times 6 \times 3$, representing a typical RGB image. It first passes through a convolutional layer using $3 \times 3 \times 5$ kernels and “same” padding, preserving spatial dimensions and producing 5 output feature maps of size 6×6 . These are then downsampled using a 2×2 pooling operation, reducing each map to 5×5 .

Following this, a second convolutional layer using $2 \times 2 \times 3$ kernels with “valid” padding is applied, resulting in 3 feature maps of size 4×4 . Another pooling layer is applied using a 3×3 pooling window that results in 2×2 spatial dimensions. The resulting feature maps are then flattened into a 1D vector of 12.

This flattened representation is fully connected to the first dense layer with 3 neurons, followed by an output dense layer with 2 neurons.

In the following subsections, we will be presenting the modified CNN architecture after applying each advanced regrouping policy by showing the corresponding DAG representation to illustrate the resulting inference structure.



Figure 3.3: Simple example of a CNN architecture

3.7.1 Regrouping Policy 1: MLWM

In this regrouping policy, the CNN architecture remains unchanged and is identical to the one presented in Figure 3.3. The corresponding DAG representing the inference structure of MLWM is shown in Figure 3.4.

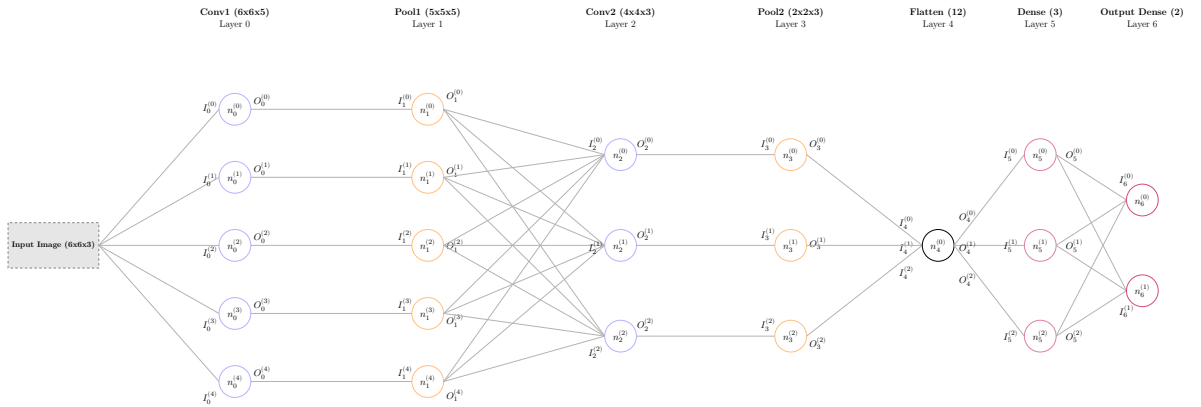


Figure 3.4: DAG representation of the MLWM policy in CPL combined with the MDM policy in MLP

We observe in Figure 3.4 that each node n_l is associated with an input I_l and an

output O_l . Within some inner layers, *e.g.*, between Conv1 and Pool1, we may encounter:

$$I_{l+1} = O_l \quad (3.25)$$

And in some other inner layers, *e.g.*, between Pool1 and Conv2, we may encounter:

$$I_{l+1} = \sum O_l \quad (3.26)$$

The summation in (3.26) results in a fully connected network between both layers, which reduces parallelism since each node must wait for all preceding operations to complete before proceeding.

For activation layers, if they are used in a CNN model, the DAG structure can remain unchanged, with activation operations integrated into their corresponding nodes.

Alternatively, as illustrated in Figure 3.5, activation layers (nodes in green) can be separated from both convolutional and dense layers and be represented as distinct nodes following these operations.

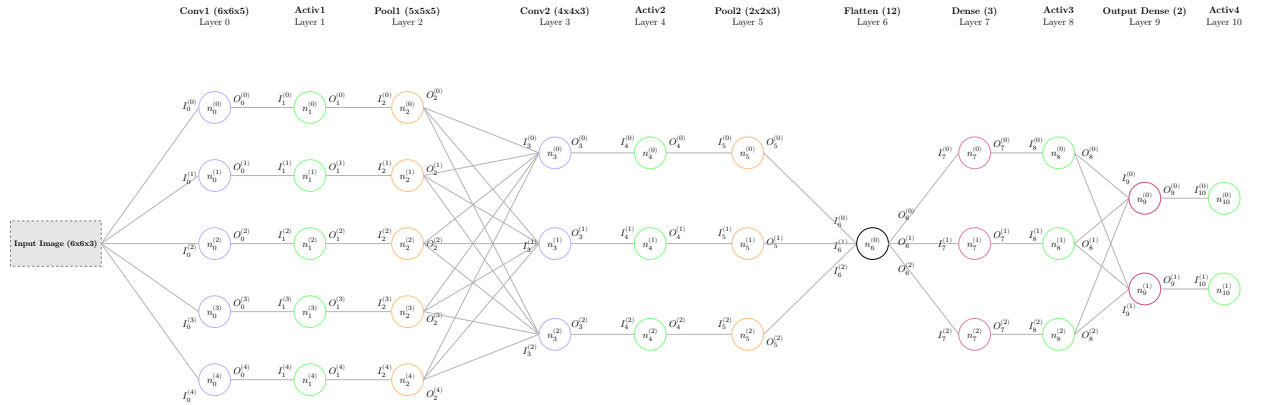


Figure 3.5: DAG representation of the MLWM policy in CPL combined with the MDM policy in MLP with separate activation layers

3.7.2 Regrouping Policy 2: SSP

In the SSP policy, we begin by examining the modifications applied to the CNN architecture previously shown in Figure 3.3 with a slight adjustment to the input image dimensions so that both height and width are odd.

In this example, we will consider the input image of dimension $5 \times 5 \times 3$ and we adjust also the pooling window of the layer Pool2 to a 2×2 window.

After the adjustment made, we consider a simple SSP partitioning operation defined by $\lambda(1, 2)$, which indicates that for the input image, the height (rows) remains unchanged while the width (columns) is divided into two blocks, resulting in a vertical partitioning. The outcome of this transformation is illustrated in Figure 3.6.

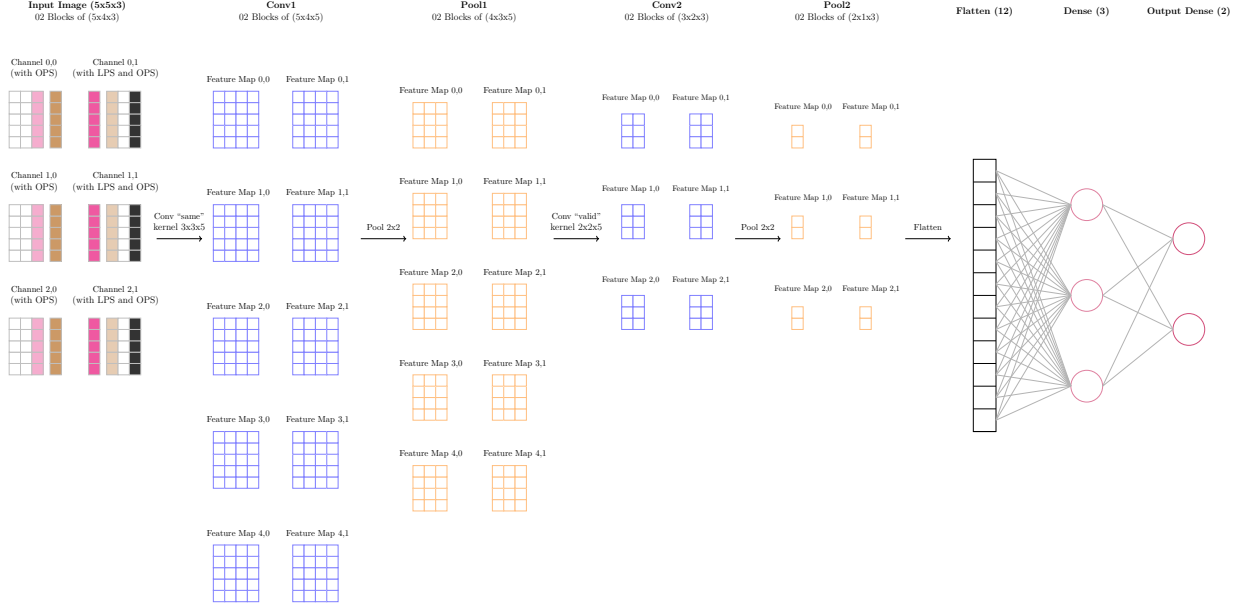


Figure 3.6: Example of a CNN architecture using the SSP policy

After introducing the SSP policy into the CNN architecture, as illustrated in Figure 3.6, we observe an increase in the number of operations across the layers. This is a direct result of splitting the input image into two parts, which produces more feature maps and thus doubles the number of operations, *i.e.*, each convolution and each pooling is applied twice. Furthermore, the vertical partitioning remains the same throughout the layers, maintaining the same division structure. The corresponding DAG of this architecture is introduced in Figure 3.7.

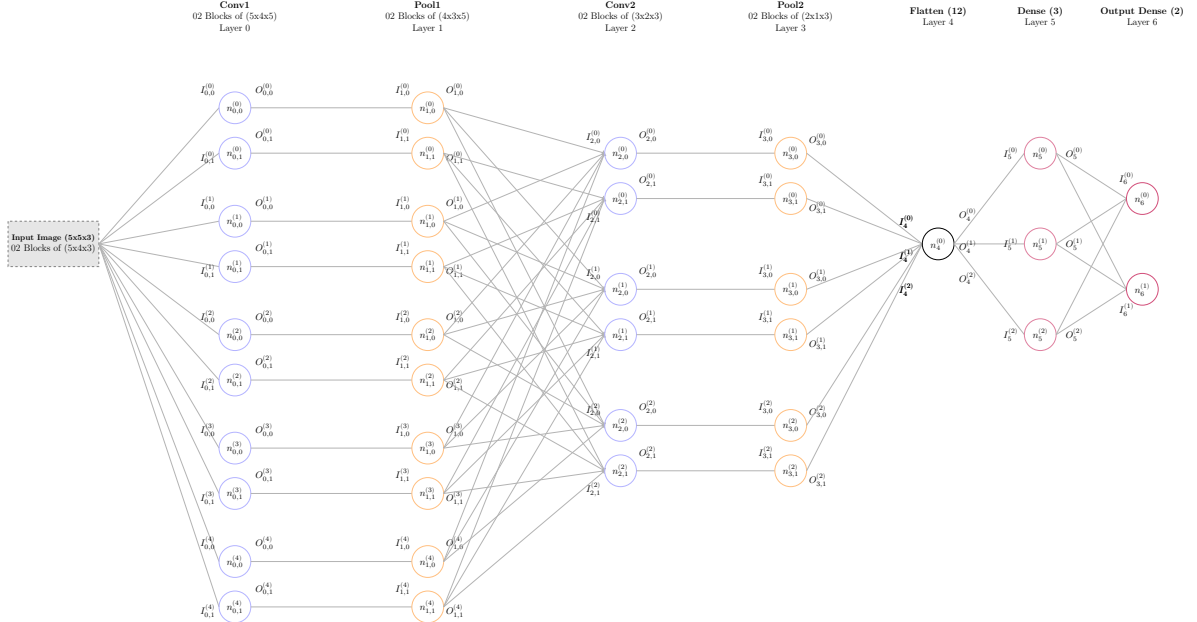


Figure 3.7: DAG representation of the SSP policy

For the input image, before applying the first convolution operation, both LPS and OPS fidelity operation adjustments are required.

Given that $\lambda(1, 2)$, *i.e.*, a vertical division of the width by 2, and the original input image width is 5, a spatial partition of the input image into two halves would not result in an equally sized integer partition. To resolve this, we apply the first fidelity operation LPS by adding a single zero-padding column to exactly one of the two blocks (represented in black in the Figure 3.6) to ensure divisibility. This results in two symmetric blocks of the input tensor, each of size 5×3 , across all its 3 channels. Note that the first block (0) is shown in the Figure 3.6 as Channel $\forall N, 0$, and the second block (1) as Channel $\forall N, 1$.

Following this adjustment, we apply also the second fidelity operation OPS mechanism to account for the receptive field of the convolutional kernels, *i.e.*, including the inputs neighborhood each convolution output needs.

Since the convolution kernel size is 3×3 , the overlap ensures that each partition has access to neighbouring pixels, which are essential for correct convolutional computation. In the Figure 3.6, we observe in the shared boundary location that one column from block 1 (in light brown) is duplicated and added to block 0 (in dark brown), while one column from block 0 (in light pink) is duplicated and added to block 1 (in dark pink). This guarantees that boundary computations remain accurate without missing context.

After these two fidelity operation preprocessing steps, the final modified input image becomes suitable for convolution operations, with a block size of 5×4 .

For the resulting DAG representation in Figure 3.7, we may observe that between the Pool1 and Conv2 layers, the fully connected network becomes significantly massive compared to the DAG of the MLWM policy, which represents one of the main drawbacks of the SSP policy.

Additionally, before applying the flattening operation, we must ensure that the ordering of all resulting operations is respected per block. To this end, it is beneficial to merge each partitioned block (by grouping all the paired blocks in this example) to obtain $I_4^{(0)}, I_4^{(1)}, I_4^{(2)}$, thereby preserving structural ordering coherence during the dimensional transformation.

3.7.3 Regrouping Policy 3: RCCT

In the RCCT policy, we focus on convolutional layers only. This policy reduces massive cross-channel connectivity and improves parallelism by partitioning the input feature map channels into blocks; each block is then computed independently by all convolution kernels, replicated per block, *i.e.*, redundantly.

We reconsider the example of Figure 3.3, focusing on the Pool1 $\rightarrow$ Conv2 layer transition where the fully connected subgraph clearly appears. We examine the structural differences before and after applying the RCCT policy, as shown in Figure 3.8.

The convolutional layer Conv2 contains three kernels applied to the five input feature maps produced by the Pool1 layer. The output of this convolution is three feature maps, as shown in Figure 3.8a.

We consider the RCCT partitioning operator $\zeta(3)$, which divides the channel dimension of all feature maps into three blocks, as illustrated in Figure 3.8b. As the 5 input feature maps are not divisible by 3, we add one zero-padded dummy feature map (the feature map in black) to make the total 6, thereby ensuring divisibility by 3. This procedure of adding a dummy feature map constitutes the CPS fidelity-preserving operation.

For each block, the same set of convolution kernels is applied, resulting in three feature maps per block. We treat this as an intermediate layer (it is not the final desired output).



Figure 3.8: Pool1-Conv2 layers transition under RCCT policy

Finally, the next layer (Summation Layer of Conv2) sums according to channel ordering the corresponding feature maps across the previous Intermediate Conv2 Layer that contains the three blocks as follows:

$$(\text{Feature Map } 0)_{\text{Final Layer}} = (\text{Feature Map } 0,0 + \text{Feature Map } 0,1 + \text{Feature Map } 0,2)_{\text{All Prev Layer}}$$

To generalize the preceding example, for channel index X , and the number of the available blocks i , we denote:

$$(\text{Feature Map } X)_{\text{Final Layer}} = (\text{Feature Map } X, \sum_{i=0}^2 i)_{\text{All Prev Layers}} \quad (3.27)$$

The corresponding DAG for this representation is illustrated in Figure 3.9. In the DAG, the added dummy feature map (the black rectangle), emerged from applying CPS strategy, is not the result of a computation from the preceding Pool1 layer; therefore, it is shown not as a node but as a rectangle labeled D , required for the calculation.

For the final Summation Layer, we notice that each node sums according to the channel ordering of the preceding Intermediate Layer.

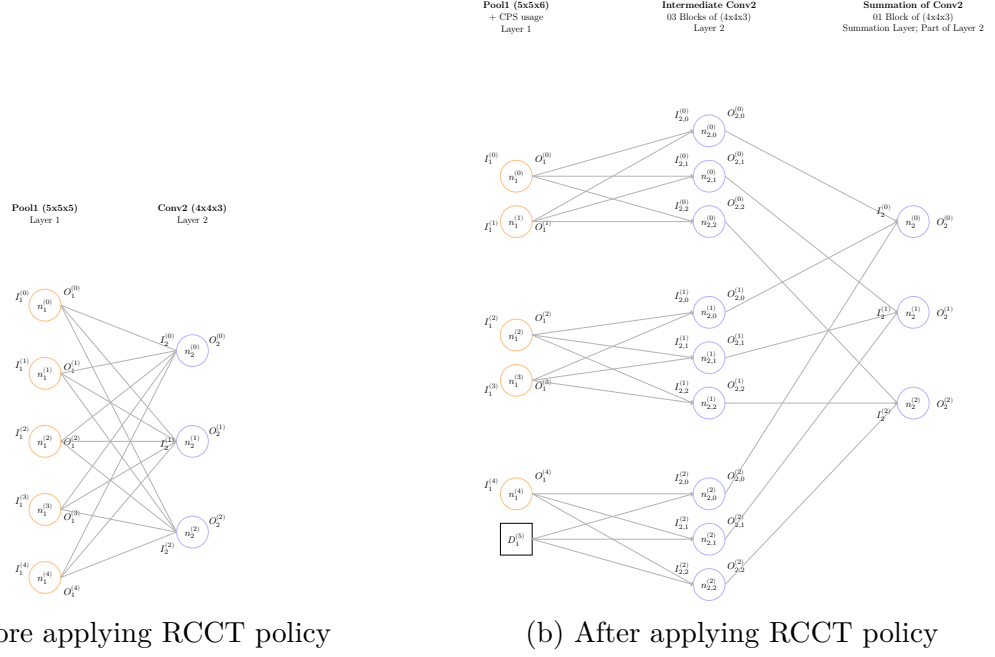


Figure 3.9: DAG representation of Pool1–Conv2 layers transition under RCCT policy

By comparing the DAG representations before applying the RCCT policy (Figure 3.9a, equivalent to the MLWM policy) and after applying RCCT policy (Figure 3.9b), we observe a reduction of the massive connectivity and an increase in parallelism/independence of computations, leading to shorter execution time compared to MLWM.

3.7.4 Regrouping Policy 4: DF

In the DF policy, we focus exclusively on dense layers; therefore, we present directly the DAG representation in Figure 3.10.

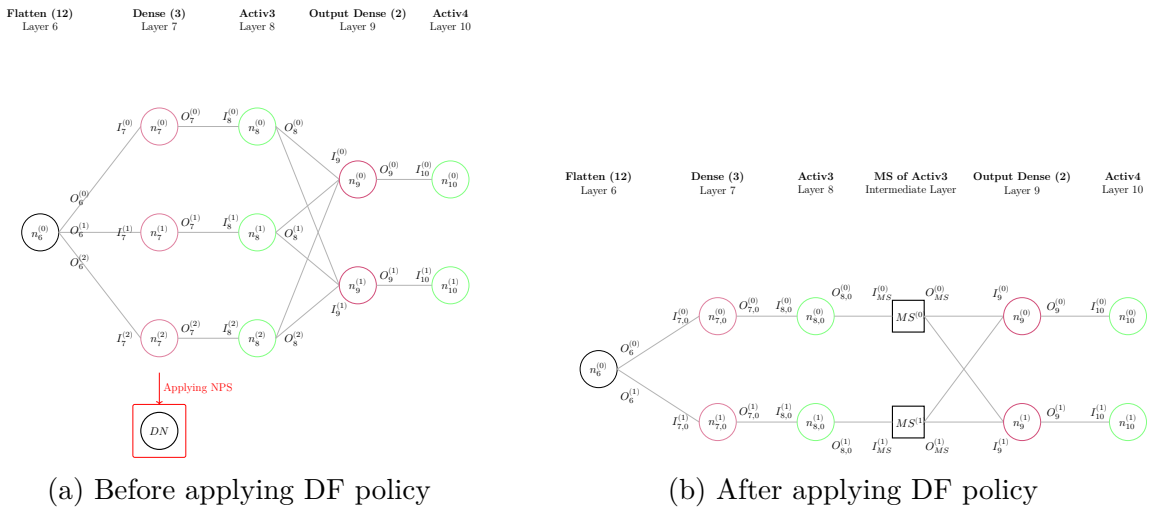


Figure 3.10: DAG representation of the DF policy.

The Figure 3.10a, that we took from the detailed MLWM policy (Figure 3.5), shows the DAG node distribution before applying DF.

In this example, we select the first dense layer and apply the DF operator $\iota(2)$, *i.e.*, we regroup the first dense layer into two blocks.

Because the first dense layer has 3 neurons and we wish to divide it into 2 blocks, the quotient is not an integer. We therefore apply the first fidelity-preserving operation NPS, which adds a dummy neuron DN so that the count becomes divisible; adding one DN will give us a total of 4 neurons, which is divisible by 2. The DN neuron appears in Figure 3.10a in black colour inside the red rectangle.

After regrouping, the activation functions in this layer are applied per block. The resulting activations then pass through an Intermediate Layer implementing the second fidelity operation MS, which applies masks to suppress noise and erroneous values produced by activating the DN introduced by NPS, before forwarding them as inputs to the next dense layer. In Figure 3.10b, the masked operations are shown as black squares labeled MS .

We should note that the MS is necessary only when dense layers use Sigmoid, or when the final layer uses Softmax; otherwise, MS is not required for the other activation functions considered in this report.

3.8 Regrouping Policies Workflow Rrepresentation

In this section, we summarize all regrouping policies needed in this project into workflow diagrams to clarify their logic and execution flow. Figure 3.11 presents all regrouping policies used from this point onward.

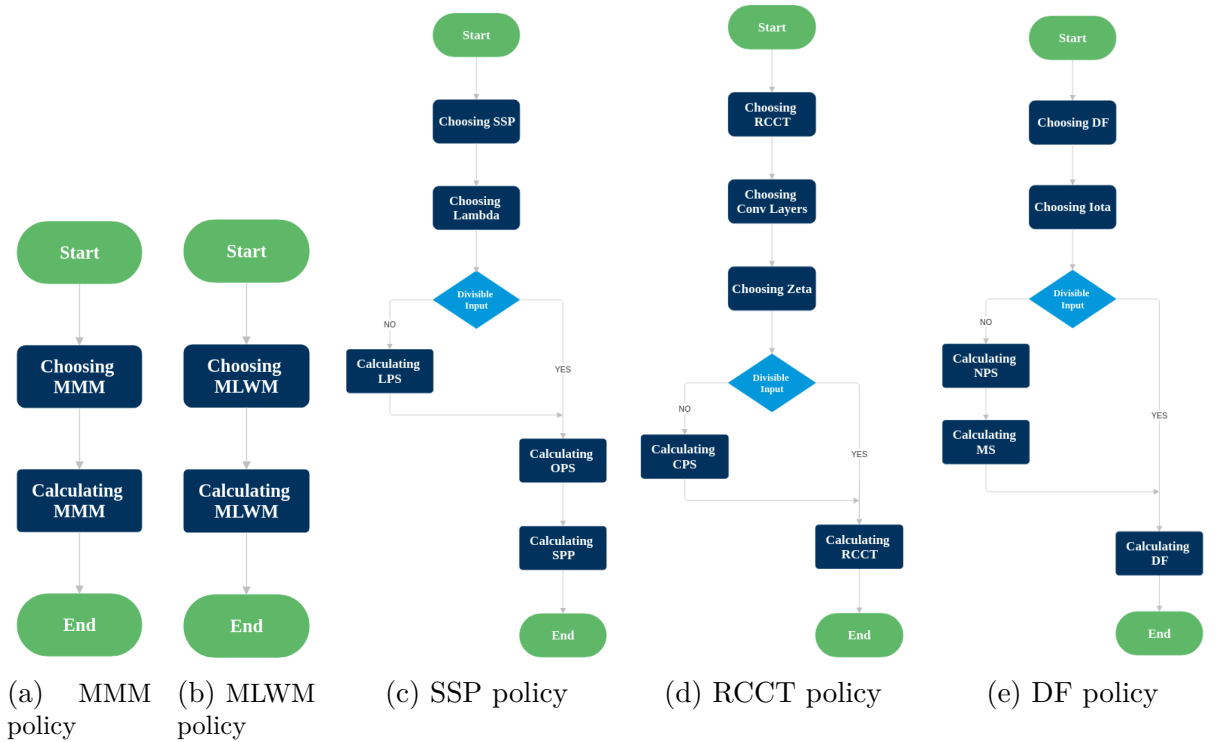


Figure 3.11: Workflow representation of the used regrouping policies

We notice that the first two policies, MMM (Figure 3.11a) and MLWM (Figure 3.11b), have no operator parameter; they are straightforward computations and do not require fidelity-preserving protocols.

By contrast, the remaining policies, SSP (Figure 3.11c), RCCT (Figure 3.11d), and DF (Figure 3.11e), are parameterized by operators and a divisibility condition of the inputs by the operator’s parameter. For these three policies, fidelity-preserving protocols are essential to ensure correctness and to avoid information loss during data propagation in the CNN DAG.

3.9 Conclusion

This chapter introduced and formalized the DAG node regrouping policies that we will be using throughout the analysis pipeline of our project, highlighting first the gap between the extremal granularities of MDM (conceptual ideal/perfect lower bound) and MMM (pessimistic upper bound retained as a reference). Then, we defined MLWM for the whole feature maps node DAG representation and also the operator-driven policies SSP, RCCT, and DF, together with fidelity-preserving operation protocols that ensure the regrouped DAGs compute the same input–output as the original CNN without information loss.

These policies expose useful parallelism while keeping node counts manageable for many-core mapping. In the next chapter, we develop architecture–aware cost equations on Kalray MPPA (Bostan/Coolidge), quantifying RTS parameters needed for timing analysis.

Chapter 4

Architecture-Aware Cost Modeling on Kalray MPPA

4.1 Introduction

In this chapter, we will be analysing first the costs of CNN layers, and then project those costs to the regrouping policies discussed in Chapter 3 under simplified assumptions for the Kalray MPPA architectures (Bostan and Coolidge). We also present all RTS quantities needed as inputs to the MIA tool.

We begin by detailing per-layer costs: we define the arithmetic cost and then derive, per output element, the WCET-in-isolation and the WCMA. These per-output-element formulas serve as a reference for the next calculations in each regrouping policy.

Next, we discuss DDR costs and the transfer of the CNN model parameters and hyper-parameters from the DDR to each SMEM cluster. The assumption is about exchanges between external and internal memory.

We then analyze data transfers between cores and distinguish inter/intra-cluster communications in the cost modeling.

We also describe the task ordering policy used in our project for task-to-core mapping management. Finally, we will end this chapter by presenting the configuration of period and deadline.

Throughout this chapter, we adopt FP32/INT32 as the maximum data precision supported by Kalray MPPA architectures.

For our study, and for compatibility with the MIA tool's `.yaml` input, the fields are interpreted as follows:

- `procDemand`: WCET-in-isolation,
- `memDemand`: WCMA,
- `accessTo`: contains successor nodes and their data transfers,
- `assignedTo`: core mapping,
- `order`: execution order of tasks on the same core.

Before proceeding, Table 4.1 summarizes notation; some are used earlier in this report.

Notation	Representation
H_l, W_l	Input height, width
D_l, D_{l+1}	Input, output channel depth
k_H, k_W	Kernel height, width
p_H, p_W	Padding height, width
s_H, s_W	Stride height, width
d_H, d_W	Dilation rate height, width
H_{l+1}, W_{l+1}	Output height, width
m, n	Pooling window height, width
N_l, N_{l+1}	Input, output vector length of a dense layer which is also equivalent to the available neuron counts in the previous layer N_l and in the current layer N_{l+1}
V	Vector length for softmax which is equal to the N_{l+1} of the final dense layer

Table 4.1: Summary of required notation

4.2 Cost Analysis of CNN Metrics

In this section, we first analyse, per output element, the relevant metrics which are: arithmetic operations, parameter counts, and hyper-parameters, for each CNN layer. We then estimate, also per output element, the processor cost WCET, and the memory cost WCMA of each operation under the Bostan and Coolidge architectural constraints.

In the following section, we should mention that the arithmetic operations: “Add” denotes addition, “Mult” denotes multiplication, “MACs” denotes multiply-accumulate, “Sub” denotes subtraction, “Div” denotes division, “Cmp” denotes comparison, and “Exps” denotes calls to the exponential function.

4.2.1 Convolution Layer

Per output element arithmetic for the convolution operation is given by:

$$\begin{aligned} \text{Mult.} &= k_H k_W D_l, & \text{Add.} &= k_H k_W D_l - 1 + 1_{bias}, & \text{MACs} &= k_H k_W D_l \cdot 1_{bias} \\ \text{Sub.} &= 0, & \text{Div.} &= 0, & \text{Cmp.} &= 0 \end{aligned}$$

Where:

$$1_{bias} = \begin{cases} 1 & \text{if a bias vector is used,} \\ 0 & \text{otherwise.} \end{cases}$$

We observe that when the bias exists, the number of addition operations equals the number of multiplications. In this case, the computation can be interpreted directly as a MAC operation. However, if the bias is not present, the number of additions is less than the number of multiplications, and thus additions and multiplications must be computed separately using ALU.

The number of parameters and hyper-parameters of the convolution layer is:

$$\text{Parameters} = k_H k_W D_l D_{l+1} + 1_{bias} \cdot D_{l+1}$$

$$\text{Hyper-parameters} = \{k_H, k_W, p_H, p_W, s_H, s_W, d_H, d_W, D_{l+1}\}$$

We may notice also that the total number of parameters is reduced when the bias is omitted.

4.2.2 Pooling Layers

Per output element arithmetic, the number of parameters and hyper-parameters of the pooling operation are calculated in the following Max-Pooling Layer and Average-Pooling Layer titles:

Max-Pooling Layer

Per output element arithmetic:

$$\begin{aligned} \text{Mult.} &= 0, & \text{Add.} &= 0, & \text{MACs} &= 0 \\ \text{Sub.} &= 0, & \text{Div.} &= 0, & \text{Cmp.} &= m n - 1 \end{aligned}$$

The number of parameters and hyper-parameters:

$$\text{Parameters} = 0$$

$$\text{Hyper-parameters} = \{m, n, s_H, s_W\}$$

Average-Pooling Layer

Per output element arithmetic:

$$\begin{aligned} \text{Mult.} &= 1, & \text{Add.} &= m n - 1, & \text{MACs} &= 0 \\ \text{Sub.} &= 0, & \text{Div.} &= 1, & \text{Cmp.} &= 0 \end{aligned}$$

The number of parameters and hyper-parameters:

$$\text{Parameters} = 0$$

$$\text{Hyper-parameters} = \{m, n, s_H, s_W\}$$

4.2.3 Flatten Layer

Note that the Flatten layer has no arithmetic operations, parameters, or hyper-parameters. Its role is a pure reshape; hence, all metrics are zero.

4.2.4 Dense Layer

Per output neuron element arithmetic for the dense operation is given by:

$$\begin{aligned} \text{Mult.} &= N_l, & \text{Add.} &= N_l - 1 + 1_{bias}, & \text{MACs} &= N_l \cdot 1_{bias} \\ \text{Sub.} &= 0, & \text{Div.} &= 0, & \text{Cmp.} &= 0 \end{aligned}$$

Where

$$1_{bias} = \begin{cases} 1 & \text{if a bias vector is used,} \\ 0 & \text{otherwise.} \end{cases}$$

The number of parameters and hyper-parameters of the dense layer is:

$$\text{Parameters} = N_l N_{l+1} + 1_{bias} \cdot N_{l+1}$$

$$\text{Hyper-parameters} = \{N_{l+1}\}$$

4.2.5 Activation Functions

The costs of the activation functions are defined in Table 4.2. Counts are per element except for Softmax, where the row covers the entire vector length V .

Activation	Adds	Subs	Mults	Divs	Cmps	Exps	Hyper-params
ReLU	0	0	0	0	1	0	—
Leaky-ReLU	0	0	1	0	1	0	α
Sigmoid	1	1	0	1	0	1	—
TanH	1	2	0	1	0	2	—
Softmax	V-1	0	0	1	0	V	V

Table 4.2: Calculation costs of activation functions

4.2.6 Modeling WCET in Isolation

For each output element, we estimate the number of cycles based on the architectural information about the cores given in Section 1.5 in Chapter 1.

As a reminder, Bostan features a 5-way VLIW that issues 1 ALU, 1 load, 1 store, 1 multiply, and 1 branch per cycle. However, Coolidge features a 6-slot VLIW, with 2 ALUs, 1 multiply, 1 load/store, 1 branch, with a sixth slot for the TCA co-processor that supports only MAC operations. The computational throughput of each core in Bostan and Coolidge is summarized in Table 4.3.

Operation Type	Data Precision	Operation per Cycle		
		Bostan	Coolidge	Coolidge TCA
MAC	FP32	1	1	not supported
MAC	FP16/INT16/INT8	1	1	16
ALU	FP32/INT32	1	2	not supported

Table 4.3: Computational throughput per core : Bostan vs. Coolidge

We estimate the WCET in isolation in Table 4.4; the calculations are made per output element for each layer operation in a CNN model, based on arithmetic costs, *e.g.*, multiplication, addition, division, comparison, and the VLIW execution capabilities of the Bostan and Coolidge architectures from Table 4.3.

Operations	Type	Bostan (cycles)	Coolidge (cycles)
Convolution	Add + Mul	$2k_H k_W D_l - 1$	$k_H k_W D_l - 1$
Convolution+ 1_{bias}	MAC	$k_H k_W D_l$	$\begin{cases} k_H k_W D_l & \text{FP32} \\ \left\lceil \frac{k_H k_W D_l}{1 + 16_{TCA}} \right\rceil & \text{FP16/INT16/INT8} \end{cases}$
Max-Pool	Cmp	$mn - 1$	$\left\lceil \frac{mn-1}{2} \right\rceil$
Avg-Pool	Add + Mul + Div	$mn + 1_{div}$	$\left\lceil \frac{mn}{2} \right\rceil + 1_{div}$
Flatten	Reshaping	1	1
Dense	Add + Mul	$2N_l - 1$	$N_l - 1$
Dense+ 1_{bias}	MAC	N_l	$\begin{cases} N_l & \text{FP32} \\ \left\lceil \frac{N_l}{1 + 16_{TCA}} \right\rceil & \text{FP16/INT16/INT8} \end{cases}$

Table 4.4: WCET in isolation per output element for CNN layers operation

In Table 4.4, the term 1_{div} shows one division operation in an Average Pooling layer. We assume for Bostan, one division operation costs 20 cycles while in Coolidge it costs only 10 cycles.

This assumption is based on architectural differences between the two generations of Kalray MPPA processors; Bostan lacks an FPU, which typically results in higher latency, and Coolidge features an improved FPU pipeline that supports more efficient handling of arithmetic operations, including division.

Remark. Division is a multi-cycle operation. The values reported are conservative upper bound estimates only. Because Kalray does not publicly specify exact division latencies, we adopt these modeling constants to avoid underestimation and also to avoid them becoming a bottleneck in our WCET analysis.

For the Flatten layer, no arithmetic operations are performed; the layer simply reshapes the memory layout, converting spatial tensors to vectors and updating memory pointers.

Analyzing Table 4.4, we observe that the Coolidge architecture reduces WCET for certain operations, for example, ALU operations are halved, and also when MAC operations use FP16/INT16/INT8 data precision, the TCA co-processor is employed and offloads 16 MACs per cycle. Together with one MAC available on the core, this generates an effective $17\times$ reduction.

Next, for the activation functions, we analyze their per output element WCET in Table 4.5.

Activation	Type	Bostan (cycles)	Coolidge (cycles)
ReLU	Cmp	1	1
Leaky-ReLU	Cmp + Mult	2	$\frac{2}{2} = 1$
Sigmoid	Add + Sub + Div + Exp	$2 + 1_{div} + 1_{exp}$	$1 + 1_{div} + 1_{exp}$
TanH	Add + Sub + Div + Exp	$3 + 1_{div} + 2_{exp}$	$2 + 1_{div} + 2_{exp}$
Softmax	Add + Sub + Div + Cmp + Exp	$V - 1 + 1_{div} + V_{exp}$	$\lceil \frac{V-1}{2} \rceil + 1_{div} + V_{exp}$

Table 4.5: WCET in isolation per output element for activation functions

In Table 4.5, the term 1_{exp} shows exponential operation. We assume for Bostan, one exponential operation costs 60 cycles, while in Coolidge it costs 30 cycles.

We note that exponential operations are generally implemented via software-based polynomial approximations. Because the results are real numbers, they rely on floating-point data support. As with division, the absence of a FPU on Bostan generates higher latency than on Coolidge, which benefits from improved FPU pipelines. We therefore adopt the same conservative estimation policy described earlier for division operations.

4.2.7 Modeling WCMA

After establishing the WCET in isolation per output element for each operation, we now turn to the cost metric that quantifies the total data-movement pressure on the internal shared memory: WCMA.

For each element operation in a CNN layer, we estimate the number of memory transactions directly between SMEM and the core (see Section 1.5 in Chapter 1), intentionally neglecting cache effects for simplicity purposes and to obtain a conservative upper-bound approximation of WCMA.

Table 4.6 presents the per-cycle bus transfer width from SMEM to core registers on the Bostan and Coolidge architectures in bits per cycle and bytes per cycle.

	Bits per Cycle			Bytes per Cycle		
	Bostan	Coolidge	Coolidge TCA	Bostan	Coolidge	Coolidge TCA
Bus width	64	128	128	8	16	16

Table 4.6: Bus width from SMEM to core: Bostan vs. Coolidge

We note that the Coolidge architecture, together with its TCA co-processor, features a wider bus and higher overall flow than Bostan.

Therefore, in Table 4.7, the number of WCMA is represented as the number of transfers per cycle, depending on the amount and data precision of the transfer.

Data Precision	Ops	Transfers per Cycle		
		Bostan (cycles)	Coolidge (cycles)	Coolidge TCA (cycles)
FP32 / INT32	2	1	1	—
FP32 / INT32	4	2	1	—
FP16 / INT16	8	2	1	1
INT8	16	2	1	1

Table 4.7: WCMA transfers from SMEM to core

To understand how WCMA is computed in Table 4.7, the following equations present the calculation used to obtain a result expressed in cycles:

$$\text{WCMA}_{\text{bit}} = \left\lceil \text{Ops} \cdot \frac{D_p \text{ (bits)}}{B \text{ (bits/cycle)}} \right\rceil \quad (\text{cycles}) \quad (4.1)$$

If we convert the calculation to bytes, the Equation 4.1 will become:

$$\text{WCMA}_{\text{byte}} = \left\lceil \text{Ops} \cdot \frac{D_p \text{ (bytes)}}{8B \text{ (bytes/cycle)}} \right\rceil \quad (\text{cycles}) \quad (4.2)$$

Where:

- Ops is the number of load or store operations in a single computation.
- D_p is the data precision, *e.g.*, 32 bits for FP32 or INT32.
- B is the bus width, *e.g.*, 64 bits/cycle for Bostan.

Then, the WCMA transfer per cycle for each operation per output element in every CNN layer is calculated as shown in Table 4.8, taking into account data precision and the number of load and store operations, by assuming that each operation produces exactly one store 1_{store} per output. We should note that the load and store data should first be calculated separately, and only afterwards combined to obtain the total.

Operation	Data Precision	Bostan (cycles)	Coolidge (cycles)
Convolution	FP32/INT32	$k_H k_W D_l + 1$	$\lceil \frac{k_H k_W D_l}{2} \rceil + 1$
Convolution	FP16/INT16	$\lceil \frac{k_H k_W D_l}{2} \rceil + 1$	$\lceil \frac{k_H k_W D_l}{4} \rceil + 1$
Convolution	INT8	$\lceil \frac{k_H k_W D_l}{4} \rceil + 1$	$\lceil \frac{k_H k_W D_l}{8} \rceil + 1$
Convolution+1 _{bias}	FP32/INT32	$k_H k_W D_l + 2$	$\lceil \frac{k_H k_W D_l}{2} \rceil + 2$
Convolution+1 _{bias}	FP16/INT16	$\lceil \frac{k_H k_W D_l}{2} \rceil + 2$	$\lceil \frac{k_H k_W D_l}{2^9} \rceil + 2$
Convolution+1 _{bias}	INT8	$\lceil \frac{k_H k_W D_l}{4} \rceil + 2$	$\lceil \frac{k_H k_W D_l}{2^{10}} \rceil + 2$
Max/Avg ¹ -Pool	FP32/INT32	$\lceil \frac{mn}{2} \rceil + 1$	$\lceil \frac{mn}{4} \rceil + 1$
Max/Avg ¹ -Pool	FP16/INT16	$\lceil \frac{mn}{4} \rceil + 1$	$\lceil \frac{mn}{8} \rceil + 1$
Max/Avg ¹ -Pool	INT8	$\lceil \frac{mn}{8} \rceil + 1$	$\lceil \frac{mn}{16} \rceil + 1$
Flatten	-	1	1
Dense	FP32/INT32	$N_l + 1$	$\lceil \frac{N_l}{2} \rceil + 1$
Dense	FP16/INT16	$\lceil \frac{N_l}{2} \rceil + 1$	$\lceil \frac{N_l}{4} \rceil + 1$
Dense	INT8	$\lceil \frac{N_l}{4} \rceil + 1$	$\lceil \frac{N_l}{8} \rceil + 1$
Dense+1 _{bias}	FP32/INT32	$N_l + 2$	$\lceil \frac{N_l}{2} \rceil + 2$
Dense+1 _{bias}	FP16/INT16	$\lceil \frac{N_l}{2} \rceil + 2$	$\lceil \frac{N_l}{2^9} \rceil + 2$
Dense+1 _{bias}	INT8	$\lceil \frac{N_l}{4} \rceil + 2$	$\lceil \frac{N_l}{2^{10}} \rceil + 2$

Table 4.8: WCMA transfers per output element for CNN layers operation

The detailed calculation of all operation results for various data precisions on the Bostan and Coolidge architecture presented in Table 4.8 is written in the Appendix A. We assume in this per element output calculation case only, that the pooling operations have the same costs.

For flatten layer calculation, it is lightweight in terms of memory traffic: it simply re-orientes the tensor's base pointer with a single access to memory, allowing the next layers to consume the data as a 1D vector instead of a 2D spatial array.

For activation layers, we analyse their WCMA per output element in Table 4.9.

¹For average pooling we assume the multiplication operation of the window size $m \cdot n$ is already calculated, pre-loaded and kept in each core register. We consider the pre-loading cost, which is equal to 1, only when computing the total number of operation per layer. So, we will be considering this preloading value in the following WCMA calculation of each regrouping policy.

Activation	Data Precision	Load	Store	Bostan (cycles)	Coolidge (cycles)
ReLU	$\forall FP/\forall INT$	1	1	2	2
Leaky-ReLU	$\forall FP/\forall INT$	1	1	2	2
Sigmoid ¹	$\forall FP/\forall INT$	1	1	2	2
TanH ¹	$\forall FP/\forall INT$	1	1	2	2
Softmax	FP32/INT32	V	1	$\lceil \frac{V}{2} \rceil + 1$	$\lceil \frac{V}{4} \rceil + 1$
Softmax	FP16/INT16	V	1	$\lceil \frac{V}{4} \rceil + 1$	$\lceil \frac{V}{8} \rceil + 1$
Softmax	INT8	V	1	$\lceil \frac{V}{8} \rceil + 1$	$\lceil \frac{V}{16} \rceil + 1$

Table 4.9: WCMA transfers per output element for activation layers

To better understand the computation of the Softmax results, we refer to the detailed mathematical formulations for various data precisions on the Bostan and Coolidge architectures provided in Appendix A.

4.3 Cost Analysis of the Initial Methods

In this section, we present the cost analysis for the initial regrouping methods. We begin by modeling the WCET in isolation for the CNN layer operations, followed by the activation functions. Next, we model the WCMA for the CNN layer operations and then for the activation functions. The results of these calculations will be summarized in the corresponding tables.

The MMM approach models each layer as a single operation: for instance, a convolutional layer as one operation, a pooling layer as another, and so on. The WCET and WCMA are then computed at the layer level. In this whole-layer calculation, all formulas are multiplied by the number of output elements, $H_{l+1}W_{l+1}D_{l+1}$.

By defining $\rho = H_{l+1} \cdot W_{l+1}$, this multiplication can be expressed as ρD_{l+1} .

Regarding padding, if “valid” padding is used, no padding is applied, and consequently, the dimensions of the output feature map are reduced, as previously described in Equation (1.2). In contrast, if “same” padding is used, padding is applied such that the spatial dimensions are preserved; in this case, the input and output dimensions remain equal, *i.e.*, $H_l = H_{l+1}$ and $W_l = W_{l+1}$.

For MDM, all computations follow the same structure as those presented earlier in Section 4.2. Since MDM adopts a fine-grained approach, where each node performs exactly one operation per output element, the analysis remains identical. Therefore, it is not necessary to redefine the cost formulas in this subsection.

We should note that per element MDM calculations serve as the unit calculation henceforth, which is then scaled and adjusted according to the regrouping methods employed.

¹The additional operations, *e.g.*, exponential approximation function, are assumed to reside in the L1 cache of each core and therefore do not generate further external memory traffic.

4.3.1 Modeling WCET in MMM

In Table 4.10, we present the WCET in isolation costs for CNN layer operations under the MMM method. These values are obtained by counting the arithmetic operations involved in each layer type, *e.g.*, additions, multiplications, etc., while taking into account the presence or absence of bias. The results are expressed per output element and distinguished by data precision, *e.g.*, FP32, FP16, etc. Each entry in the table corresponds to the maximum number of execution cycles of the given operation.

Layer	Bostan (cycle)	Coolidge (cycle)
Convolution	$(2k_H k_W D_l - 1)\rho D_{l+1}$	$(k_H k_W D_l - 1)\rho D_{l+1}$
Convolution+1 _{bias}	$(k_H k_W D_l)\rho D_{l+1}$	$\begin{cases} (k_H k_W D_l)\rho D_{l+1} & \text{FP32} \\ \left\lceil \frac{k_H k_W D_l}{1 + 16_{TCA}} \right\rceil \rho D_{l+1} & \text{FP16/INT16/INT8} \end{cases}$
Max-Pooling	$(mn - 1)\rho D_{l+1}$	$\left\lceil \frac{mn-1}{2} \right\rceil \rho D_{l+1}$
Average-Pooling	$(mn + 1_{div})\rho D_{l+1}$	$\left(\left\lceil \frac{mn}{2} \right\rceil + 1_{div} \right) \rho D_{l+1}$
Flatten	1	1
Dense	$(2N_l - 1)N_{l+1}$	$(N_l - 1)N_{l+1}$
Dense+1 _{bias}	$N_l N_{l+1}$	$\begin{cases} N_l N_{l+1} & \text{FP32} \\ \left\lceil \frac{N_l}{1 + 16_{TCA}} \right\rceil N_{l+1} & \text{FP16/INT16/INT8} \end{cases}$

Table 4.10: WCET in isolation for CNN layers operation in MMM

The WCET in isolation cost calculations for activation functions under the MMM method are presented in Table 4.11. We should note that ρD_{l+1} is equal to N_{l+1} when applying activation functions in dense layers.

Activation	Bostan (cycles)	Coolidge (cycles)
ReLU	ρD_{l+1}	$\left\lceil \frac{\rho D_{l+1}}{2} \right\rceil$
Leaky-ReLU	$2\rho D_{l+1}$	ρD_{l+1}
Sigmoid	$(2 + 1_{div} + 1_{exp})\rho D_{l+1}$	$(1 + 1_{div} + 1_{exp})\rho D_{l+1}$
TanH	$(3 + 1_{div} + 2_{exp})\rho D_{l+1}$	$\left\lceil \frac{3\rho D_{l+1}}{2} \right\rceil + (1_{div} + 2_{exp})\rho D_{l+1}$
Softmax	$(V - 1 + 1_{div} + V_{exp})N_{l+1}$	$\left\lceil \frac{(V-1)N_{l+1}}{2} \right\rceil + (1_{div} + V_{exp})N_{l+1}$

Table 4.11: WCET in isolation for activation functions in MMM

4.3.2 Modeling WCMA in MMM

Table 4.12 presents the WCMA costs of CNN layer operations under the MMM method. The values are derived from the number of memory loads and stores required by each layer type, with adjustments for the presence or absence of bias. Results are differentiated by data precision, *e.g.*, FP32, FP16, etc. Each table entry represents the maximum number of memory transfer cycles associated with the corresponding operation.

Layer	Precision	Bostan (cycles)	Coolidge (cycles)
Convolution	FP32/INT32	$\lceil (k_H k_W D_l + \frac{1}{2}) \rho D_{l+1} \rceil$	$\lceil (\frac{k_H k_W D_l}{2} + \frac{1}{4}) \rho D_{l+1} \rceil$
	FP16/INT16	$\lceil (\frac{k_H k_W D_l}{2} + \frac{1}{4}) \rho D_{l+1} \rceil$	$\lceil (\frac{k_H k_W D_l}{4} + \frac{1}{8}) \rho D_{l+1} \rceil$
	INT8	$\lceil (\frac{k_H k_W D_l}{4} + \frac{1}{8}) \rho D_{l+1} \rceil$	$\lceil (\frac{k_H k_W D_l}{8} + \frac{1}{16}) \rho D_{l+1} \rceil$
Convolution+1 _{bias}	FP32/INT32	$(k_H k_W D_l + 1) \rho D_{l+1}$	$\lceil (\frac{k_H k_W D_l}{2} + \frac{1}{2}) \rho D_{l+1} \rceil$
	FP16/INT16	$\lceil (\frac{k_H k_W D_l}{2} + \frac{1}{2}) \rho D_{l+1} \rceil$	$\lceil (\frac{k_H k_W D_l}{2^9} + \frac{1}{2^9}) \rho D_{l+1} \rceil$
	INT8	$\lceil (\frac{k_H k_W D_l}{4} + \frac{1}{4}) \rho D_{l+1} \rceil$	$\lceil (\frac{k_H k_W D_l}{2^{10}} + \frac{1}{2^{10}}) \rho D_{l+1} \rceil$
Max-Pool	FP32/INT32	$\lceil (\frac{mn}{2} + \frac{1}{2}) \rho D_l \rceil$	$\lceil (\frac{mn}{4} + \frac{1}{4}) \rho D_l \rceil$
	FP16/INT16	$\lceil (\frac{mn}{4} + \frac{1}{4}) \rho D_l \rceil$	$\lceil (\frac{mn}{8} + \frac{1}{8}) \rho D_l \rceil$
	INT8	$\lceil (\frac{mn}{8} + \frac{1}{8}) \rho D_l \rceil$	$\lceil (\frac{mn}{16} + \frac{1}{16}) \rho D_l \rceil$
Avg-Pool	FP32/INT32	$\lceil (\frac{mn}{2} + \frac{1}{2}) \rho D_l \rceil + 1_{pl}$	$\lceil (\frac{mn}{4} + \frac{1}{4}) \rho D_l \rceil + 1_{pl}$
	FP16/INT16	$\lceil (\frac{mn}{4} + \frac{1}{4}) \rho D_l \rceil + 1_{pl}$	$\lceil (\frac{mn}{8} + \frac{1}{8}) \rho D_l \rceil + 1_{pl}$
	INT8	$\lceil (\frac{mn}{8} + \frac{1}{8}) \rho D_l \rceil + 1_{pl}$	$\lceil (\frac{mn}{16} + \frac{1}{16}) \rho D_l \rceil + 1_{pl}$
Flatten	—	1	1
Dense	FP32/INT32	$\lceil (N_l + \frac{1}{2}) N_{l+1} \rceil$	$\lceil (\frac{N_l}{2} + \frac{1}{4}) N_{l+1} \rceil$
	FP16/INT16	$\lceil (\frac{N_l}{2} + \frac{1}{4}) N_{l+1} \rceil$	$\lceil (\frac{N_l}{4} + \frac{1}{8}) N_{l+1} \rceil$
	INT8	$\lceil (\frac{N_l}{4} + \frac{1}{8}) N_{l+1} \rceil$	$\lceil (\frac{N_l}{8} + \frac{1}{16}) N_{l+1} \rceil$
Dense+1 _{bias}	FP32/INT32	$(N_l + 1) N_{l+1}$	$\lceil (\frac{N_l}{2} + \frac{1}{2}) N_{l+1} \rceil$
	FP16/INT16	$\lceil (\frac{N_l}{2} + \frac{1}{2}) N_{l+1} \rceil$	$\lceil (\frac{N_l}{2^9} + \frac{1}{2^9}) N_{l+1} \rceil$
	INT8	$\lceil (\frac{N_l}{4} + \frac{1}{4}) N_{l+1} \rceil$	$\lceil (\frac{N_l}{2^{10}} + \frac{1}{2^{10}}) N_{l+1} \rceil$

Table 4.12: WCMA transfers for CNN layers operation in MMM

The preloaded value 1_{pl} shown in Table 4.12 corresponds to the window size dimension $m \cdot n$. It is computed only once and stored in the core registers during the average pooling operation, thereby avoiding its recalculation for ρD_{l+1} times.

The WCMA cost calculations for activation functions under the MMM method are presented in Table 4.13. We should note that ρD_{l+1} is equal to N_{l+1} when applying activation functions in dense layers.

Activation	Precision	Loads	Stores	Bostan (cycles)	Coolidge (cycles)
ReLU / Leaky-ReLU / Sigmoid / TanH	FP32/INT32	ρD_{l+1}	ρD_{l+1}	$\lceil \frac{1}{2} \rho D_{l+1} \rceil + \lceil \frac{1}{2} \rho D_{l+1} \rceil$	$\lceil \frac{1}{4} \rho D_{l+1} \rceil + \lceil \frac{1}{4} \rho D_{l+1} \rceil$
	FP16/INT16	ρD_{l+1}	ρD_{l+1}	$\lceil \frac{1}{4} \rho D_{l+1} \rceil + \lceil \frac{1}{4} \rho D_{l+1} \rceil$	$\lceil \frac{1}{8} \rho D_{l+1} \rceil + \lceil \frac{1}{8} \rho D_{l+1} \rceil$
	INT8	ρD_{l+1}	ρD_{l+1}	$\lceil \frac{1}{8} \rho D_{l+1} \rceil + \lceil \frac{1}{8} \rho D_{l+1} \rceil$	$\lceil \frac{1}{16} \rho D_{l+1} \rceil + \lceil \frac{1}{16} \rho D_{l+1} \rceil$
Softmax	FP32/INT32	V	V	$\lceil V \cdot \frac{1}{2} \rceil + \lceil V \cdot \frac{1}{2} \rceil$	$\lceil V \cdot \frac{1}{4} \rceil + \lceil V \cdot \frac{1}{4} \rceil$
	FP16/INT16	V	V	$\lceil V \cdot \frac{1}{4} \rceil + \lceil V \cdot \frac{1}{4} \rceil$	$\lceil V \cdot \frac{1}{8} \rceil + \lceil V \cdot \frac{1}{8} \rceil$
	INT8	V	V	$\lceil V \cdot \frac{1}{8} \rceil + \lceil V \cdot \frac{1}{8} \rceil$	$\lceil V \cdot \frac{1}{16} \rceil + \lceil V \cdot \frac{1}{16} \rceil$

Table 4.13: WCMA transfers for activation layers in MMM

4.4 Cost Analysis of the Advanced Regrouping Policies

In this section, we compute both the WCET in isolation and the WCMA of each CNN layer operation, including the activation layers, for the advanced regrouping policies MLWM, SSP, RCCT, and DF.

It should be noted that MLWM, SSP, and RCCT apply the WCET and WCMA analysis only within the CPL zone, whereas the DF policy applies the analysis exclusively within the MLP zone. Thus, each policy is evaluated in its respective domain.

4.4.1 MLWM

In this policy, we calculate both the WCET in isolation and the WCMA costs for convolutional and pooling layers only, excluding dense layers (CPL zone). For the activation layers, Softmax is omitted since it is not required for this analysis.

For the padding calculation, we follow the same logic previously established in the MMM method, however, we notice that the D dimension is omitted, since the analysis is performed on a per-feature-map basis. Consequently, all formulas in MLWM are multiplied by the number of output elements $H_{l+1}W_{l+1}$ only, which corresponds to ρ .

Modeling WCET

The WCET in isolation results for layer operations under the MLWM regrouping policy are presented in Table 4.14.

Operations	Type	Bostan (cycles)	Coolidge (cycles)
Convolution	Add+Mul	$(2k_H k_W D_l - 1)\rho$	$(k_H k_W D_l - 1)\rho$
Convolution+1 _{bias}	MAC	$k_H k_W D_l \cdot \rho$	$\begin{cases} (k_H k_W D_l)\rho & \text{FP32} \\ \left\lceil \frac{k_H k_W D_l}{1 + 16_{TCA}} \right\rceil \rho & \text{FP16/INT16/INT8} \end{cases}$
Max-Pool	Cmp	$(mn - 1)\rho$	$\left\lceil \frac{mn-1}{2} \right\rceil \rho$
Avg-Pool	Add+Mul+Div	$(mn + 1_{div})\rho$	$(\left\lceil \frac{mn}{2} \right\rceil + 1_{div})\rho$

Table 4.14: WCET in isolation for CNN layers operation in MLWM

The WCET in isolation calculation for activation functions is presented in Table 4.15.

Activation	Type	Bostan (cycles)	Coolidge (cycles)
ReLU	Cmp	ρ	$\left\lceil \frac{\rho}{2} \right\rceil$
Leaky-ReLU	Cmp + Mult	$2 \cdot \rho$	ρ
Sigmoid	Add + Sub + Div + Exp	$(2 + 1_{div} + 1_{exp}) \cdot \rho$	$(1 + 1_{div} + 1_{exp}) \cdot \rho$
TanH	Add + Sub + Div + Exp	$(3 + 1_{div} + 2_{exp}) \cdot \rho$	$\left\lceil \frac{3\rho}{2} \right\rceil + (1_{div} + 2_{exp}) \cdot \rho$

Table 4.15: WCET in isolation for activation functions in MLWM

Modeling WCMA

The WCMA results for layer operations under the MLWM regrouping policy are presented in Table 4.16.

Operation	Data Precision	Bostan (cycles)	Coolidge (cycles)
Convolution	FP32/INT32	$\left\lceil (k_H k_W D_l + \frac{1}{2}) \rho \right\rceil$	$\left\lceil (\frac{k_H k_W D_l}{2} + \frac{1}{4}) \rho \right\rceil$
	FP16/INT16	$\left\lceil (\frac{k_H k_W D_l}{2} + \frac{1}{4}) \rho \right\rceil$	$\left\lceil (\frac{k_H k_W D_l}{4} + \frac{1}{8}) \rho \right\rceil$
	INT8	$\left\lceil (\frac{k_H k_W D_l}{4} + \frac{1}{8}) \rho \right\rceil$	$\left\lceil (\frac{k_H k_W D_l}{8} + \frac{1}{16}) \rho \right\rceil$
Convolution+1 _{bias}	FP32/INT32	$(k_H k_W D_l + 1)\rho$	$\left\lceil (\frac{k_H k_W D_l}{2} + \frac{1}{2}) \rho \right\rceil$
	FP16/INT16	$\left\lceil (\frac{k_H k_W D_l}{2} + \frac{1}{2}) \rho \right\rceil$	$\left\lceil (\frac{k_H k_W D_l}{2^9} + \frac{1}{2^9}) \rho \right\rceil$
	INT8	$\left\lceil (\frac{k_H k_W D_l}{4} + \frac{1}{4}) \rho \right\rceil$	$\left\lceil (\frac{k_H k_W D_l}{2^{10}} + \frac{1}{2^{10}}) \rho \right\rceil$
Max-Pool	FP32/INT32	$\left\lceil (\frac{mn}{2} + \frac{1}{2}) \rho \right\rceil$	$\left\lceil (\frac{mn}{4} + \frac{1}{4}) \rho \right\rceil$
	FP16/INT16	$\left\lceil (\frac{mn}{4} + \frac{1}{4}) \rho \right\rceil$	$\left\lceil (\frac{mn}{8} + \frac{1}{8}) \rho \right\rceil$
	INT8	$\left\lceil (\frac{mn}{8} + \frac{1}{8}) \rho \right\rceil$	$\left\lceil (\frac{mn}{16} + \frac{1}{16}) \rho \right\rceil$
Avg-Pool	FP32/INT32	$\left\lceil (\frac{mn}{2} + \frac{1}{2}) \rho \right\rceil + 1_{pl}$	$\left\lceil (\frac{mn}{4} + \frac{1}{4}) \rho \right\rceil + 1_{pl}$
	FP16/INT16	$\left\lceil (\frac{mn}{4} + \frac{1}{4}) \rho \right\rceil + 1_{pl}$	$\left\lceil (\frac{mn}{8} + \frac{1}{8}) \rho \right\rceil + 1_{pl}$
	INT8	$\left\lceil (\frac{mn}{8} + \frac{1}{8}) \rho \right\rceil + 1_{pl}$	$\left\lceil (\frac{mn}{16} + \frac{1}{16}) \rho \right\rceil + 1_{pl}$

Table 4.16: WCMA transfers for CNN layers operation in MLWM

The WCMA calculation for activation functions is presented in Table 4.17.

Activation	Data Precision	Load	Store	Bostan (cycles)	Coolidge (cycles)
ReLU / Leaky-ReLU / Sigmoid / TanH	FP32/INT32	ρ	ρ	$\lceil \frac{1}{2}\rho \rceil + \lceil \frac{1}{2}\rho \rceil$	$\lceil \frac{1}{4}\rho \rceil + \lceil \frac{1}{4}\rho \rceil$
	FP16/INT16	ρ	ρ	$\lceil \frac{1}{4}\rho \rceil + \lceil \frac{1}{4}\rho \rceil$	$\lceil \frac{1}{8}\rho \rceil + \lceil \frac{1}{8}\rho \rceil$
	INT8	ρ	ρ	$\lceil \frac{1}{8}\rho \rceil + \lceil \frac{1}{8}\rho \rceil$	$\lceil \frac{1}{16}\rho \rceil + \lceil \frac{1}{16}\rho \rceil$

Table 4.17: WCMA transfers for activation layers in MLWM

4.4.2 SSP

The WCET and WCMA modeling for the SSP method is similar to that of MLWM, with the exception of the computation of the parameter ρ . In SSP, ρ_i varies across parts; for each part i resulting from the spatial partitioning, it is defined using Equation (3.9) as a function of $H_{l,i}$ and $W_{l,i}$, including the additional padding introduced by fidelity protocols:

$$\rho_i = (H_{l,i} + \Delta_H^{\text{top}} + \Delta_H^{\text{bottom}}) \times (W_{l,i} + \Delta_W^{\text{left}} + \Delta_W^{\text{right}}) \quad (4.3)$$

We note that the WCET and WCMA calculations must be performed separately for each part i .

4.4.3 RCCT

For the RCCT policy, we focus exclusively on the convolution operation. The calculation follows the same approach as the convolution in the MLWM policy for intermediate layers, but includes an additional final Summation Layer step for the last layer.

The WCET and WCMA are computed for each partition individually. The final Summation Layer then performs a pure element-wise summation across the same kernel outputs from all partitions of the intermediate layer. The number of summation nodes is therefore equal to the number of filters in the convolution layer.

Table 4.18 presents both the WCET in isolation and WCMA calculations for one Summation Layer node in the RCCT policy. These summation operations are performed exclusively by ALU units in both the Bostan and Coolidge architectures.

Cost	Data Precision	Bostan (cycles)	Coolidge (cycles)
WCET	$\forall FP/\forall INT$	$(c-1) \cdot \rho'_i$	$\left\lceil ((c-1) \cdot \rho'_i) \frac{1}{2} \right\rceil$
WCMA	$FP32/INT32$	$\lceil (c \cdot \rho'_i + \rho'_i) \frac{1}{2} \rceil$	$\lceil (c \cdot \rho'_i + \rho'_i) \frac{1}{4} \rceil$
WCMA	$FP16/INT16$	$\lceil (c \cdot \rho'_i + \rho'_i) \frac{1}{4} \rceil$	$\lceil (c \cdot \rho'_i + \rho'_i) \frac{1}{8} \rceil$
WCMA	$INT8$	$\lceil (c \cdot \rho'_i + \rho'_i) \frac{1}{8} \rceil$	$\lceil (c \cdot \rho'_i + \rho'_i) \frac{1}{16} \rceil$

Table 4.18: WCET and WCMA costs of a single Summation node in RCCT

As defined earlier in Equation (3.10) of Chapter 3, the parameter $\zeta(c)$ denotes the channel/dimension partition operator, where c is the number of channel partitions. The spatial dimension of the resulting feature map in partition i of the intermediate layer is given by:

$$\rho'_i = H_{l+1,i} \cdot W_{l+1,i}$$

After the summation operation, the final output regains the channel dimension which is equal to the number of filters of the convolution layer. The spatial dimension ρ'_i is maintained for the final layer.

4.4.4 DF

In the DF policy, we focus exclusively on dense layers in the CNN model.

Under this policy, granularity is reduced by grouping the N_{l+1} neurons of a dense layer into g execution groups, referred to as “fused nodes.”

If the number of neurons is divisible by the chosen parameter g of the DF operator $\iota(g)$, the fusion is applied directly without introducing any additional computation.

However, if the neurons are not divisible by g , the NPS protocol, which consists of adding DNs, is introduced to ensure divisibility in the chosen layer. In this case, another protocol, MS, is applied by adding an additional masking calculation layer. This masking layer operates only after fusion and activation functions have been executed. If the activation function is Sigmoid or Softmax, the masking layer is applied; otherwise, for other activation functions, the masking layer is omitted.

Modeling WCET

In Table 4.19, we present the WCET in isolation calculations for dense layers only under the DF policy. We remind that $N_{l+1,i}$ denotes the total number of neurons contained within one fused block.

Operation	Bostan (cycles)	Coolidge (cycles)
Dense	$(2N_l - 1)N_{l+1,i}$	$(N_l - 1)N_{l+1,i}$
Dense+1 _{bias}	$N_l N_{l+1,i}$	$\begin{cases} N_l N_{l+1,i} & \text{FP32} \\ \left\lceil \frac{N_l N_{l+1,i}}{1 + 16_{TCA}} \right\rceil N_{l+1,i} & \text{FP16/INT16/INT8} \end{cases}$

Table 4.19: WCET in isolation for CNN layers operation in DF

The WCET in isolation for activation layers under the DF policy is presented in Table 4.20. In this policy, we note that V in the Softmax calculation corresponds exclusively to $N_{l+1,i}$.

Activation	Bostan (cycles)	Coolidge (cycles)
ReLU	$N_{l+1,i}$	$\left\lceil \frac{N_{l+1,i}}{2} \right\rceil$
Leaky-ReLU	$2N_{l+1,i}$	$N_{l+1,i}$
Sigmoid	$(2 + 1_{div} + 1_{exp})N_{l+1,i}$	$(1 + 1_{div} + 1_{exp})N_{l+1,i}$
TanH	$(3 + 1_{div} + 2_{exp})N_{l+1,i}$	$\left\lceil \frac{3N_{l+1,i}}{2} \right\rceil + (1_{div} + 2_{exp})N_{l+1,i}$
Softmax	$(V - 1 + 1_{div} + V_{exp})N_{l+1,i}$	$\left\lceil \frac{(V-1)N_{l+1,i}}{2} \right\rceil + (1_{div} + V_{exp})N_{l+1,i}$

Table 4.20: WCET in isolation for activation functions in DF

The WCET in isolation for the Masked Layer, applied only after Sigmoid and Softmax activations, is presented in Table 4.21.

Operation	Bostan (cycles)	Coolidge (cycles)
Masking Mult. by m	$N_{l+1,i}$	$\left\lceil \frac{N_{l+1,i}}{2} \right\rceil$

Table 4.21: WCET in isolation of masking operations in DF

Modeling WCMA

The Table 4.22 reports the WCMA calculations for dense layers under DF policy, where $N_{l+1,i}$ also represents the number of neurons contained in a single fused block.

Operation	Precision	Bostan (cycles)	Coolidge (cycles)
Dense	FP32/INT32	$\lceil (N_l + \frac{1}{2}) N_{l+1,i} \rceil$	$\lceil (\frac{N_l}{2} + \frac{1}{4}) N_{l+1,i} \rceil$
	FP16/INT16	$\lceil (\frac{N_l}{2} + \frac{1}{4}) N_{l+1,i} \rceil$	$\lceil (\frac{N_l}{4} + \frac{1}{8}) N_{l+1,i} \rceil$
	INT8	$\lceil (\frac{N_l}{4} + \frac{1}{8}) N_{l+1,i} \rceil$	$\lceil (\frac{N_l}{8} + \frac{1}{16}) N_{l+1,i} \rceil$
Dense+1 _{bias}	FP32/INT32	$(N_l + 1) N_{l+1}$	$\lceil (\frac{N_l}{2} + \frac{1}{2}) N_{l+1,i} \rceil$
	FP16/INT16	$\lceil (\frac{N_l}{2} + \frac{1}{2}) N_{l+1,i} \rceil$	$\lceil (\frac{N_l}{2^9} + \frac{1}{2^9}) N_{l+1,i} \rceil$
	INT8	$\lceil (\frac{N_l}{4} + \frac{1}{4}) N_{l+1,i} \rceil$	$\lceil (\frac{N_l}{2^{10}} + \frac{1}{2^{10}}) N_{l+1,i} \rceil$

Table 4.22: WCMA transfers for CNN layers operation in DF

The WCMA calculation for activation layers under the DF policy is presented in Table 4.23. In this policy, we note that V in the Softmax calculation corresponds exclusively to $N_{l+1,i}$.

Activation	Precision	Bostan (cycles)	Coolidge (cycles)
ReLU / Leaky-ReLU / Sigmoid / TanH	FP32/INT32	$\lceil \frac{1}{2} N_{l+1,i} \rceil + \lceil \frac{1}{2} N_{l+1,i} \rceil$	$\lceil \frac{1}{4} N_{l+1,i} \rceil + \lceil \frac{1}{4} N_{l+1,i} \rceil$
	FP16/INT16	$\lceil \frac{1}{4} N_{l+1,i} \rceil + \lceil \frac{1}{4} N_{l+1,i} \rceil$	$\lceil \frac{1}{8} N_{l+1,i} \rceil + \lceil \frac{1}{8} N_{l+1,i} \rceil$
	INT8	$\lceil \frac{1}{8} N_{l+1,i} \rceil + \lceil \frac{1}{8} N_{l+1,i} \rceil$	$\lceil \frac{1}{16} N_{l+1,i} \rceil + \lceil \frac{1}{16} N_{l+1,i} \rceil$
Softmax	FP32/INT32	$\lceil V \cdot \frac{1}{2} \rceil + \lceil V \cdot \frac{1}{2} \rceil$	$\lceil V \cdot \frac{1}{4} \rceil + \lceil V \cdot \frac{1}{4} \rceil$
	FP16/INT16	$\lceil V \cdot \frac{1}{4} \rceil + \lceil V \cdot \frac{1}{4} \rceil$	$\lceil V \cdot \frac{1}{8} \rceil + \lceil V \cdot \frac{1}{8} \rceil$
	INT8	$\lceil V \cdot \frac{1}{8} \rceil + \lceil V \cdot \frac{1}{8} \rceil$	$\lceil V \cdot \frac{1}{16} \rceil + \lceil V \cdot \frac{1}{16} \rceil$

Table 4.23: WCMA transfers for activation layers in DF

The WCMA for the Masked Layer, applied only after Sigmoid and Softmax activations, is presented in Table 4.24.

Operation	Precision	Bostan (cycles)	Coolidge (cycles)
Memory transfer after Masking	FP32/INT32	$\lceil \frac{1}{2} N_{l+1,i} \rceil + \lceil \frac{1}{2} N_{l+1,i} \rceil$	$\lceil \frac{1}{4} N_{l+1,i} \rceil + \lceil \frac{1}{4} N_{l+1,i} \rceil$
	FP16/INT16	$\lceil \frac{1}{4} N_{l+1,i} \rceil + \lceil \frac{1}{4} N_{l+1,i} \rceil$	$\lceil \frac{1}{8} N_{l+1,i} \rceil + \lceil \frac{1}{8} N_{l+1,i} \rceil$
	INT8	$\lceil \frac{1}{8} N_{l+1,i} \rceil + \lceil \frac{1}{8} N_{l+1,i} \rceil$	$\lceil \frac{1}{16} N_{l+1,i} \rceil + \lceil \frac{1}{16} N_{l+1,i} \rceil$

Table 4.24: WCMA transfers of masking operations in DF

4.5 DDR Transfer to Clusters Analysis

First, all model parameters and hyper-parameters of the CNN are assumed to reside in off-chip DDR memory. We analyze the transfer of these values into each cluster's SRAM for execution.

For all data transfers of parameters and hyper-parameters from DDR to a given cluster, we assume a WCET of 1 cycle. This simplification accounts for the data refreshing and synchronization cost of each incoming transfer into the cluster's SRAM.

All DDR-related transfers are assigned to the first core of each cluster (core 0), which is therefore reserved exclusively for this purpose. Consequently, in both the Bostan and Coolidge architectures, each cluster contains 16 cores in total, but only 15 remain available for general-purpose computation.

For the WCMA modeling, as discussed in Section 1.5 in Chapter 1, the NoC is the used the access on Bostan, while AXI defines the used access on Coolidge. NoC accesses on Bostan has variable latency per transfer, depending on factors such as routing path length. In contrast, AXI accesses on Coolidge are modeled with a fixed latency per transfer, independent of the data size. In this study, however, we assume a fixed transfer rate for both architectures. The impact of variable latency transfer scenarios is left as future work.

The total SRAM size across all clusters is 32 MB in Bostan and 20 MB in Coolidge. Per cluster, this corresponds to 2 MB for Bostan and 4 MB for Coolidge. In our DDR-to-cluster transfer analysis, we assume that each cluster's SRAM is divided into two equal parts: the first half is reserved for storing CNN results that may be required by successor tasks, while the second half is allocated to parameters and hyper-parameters. This second region is the target of DDR transfers, which are triggered based on both the DDR-to-SRAM bus bandwidth and the size of the required parameters and hyper-parameters. These two factors determine the number of recharges, which in turn are used to compute the WCMA.

For hyper-parameters, since they remain constant during the computations within a given layer, we assume they are loaded once at the beginning and then stored in registers or local caches inside the cores. This approach avoids allocating fixed SRAM space for hyper-parameters and eliminates the need for repeated reloading during execution.

In the `.yaml` input file of MIA, no `accessesTo` field is specified for DDR nodes. Consequently, MIA automatically applies its unknown-bank rule: it virtually distributes the entire `memDemand` across all memory banks, as explained in [10]. This ensures that all cores can access the required parameters within each cluster's SRAM.

4.5.1 Assumptions

In order to formalize the WCMA analysis, we first establish a set of assumptions regarding memory transfers of CNN parameters and hyper-parameters from DDR to the cluster SRAM. These assumptions define the quantities, units, and precision conventions used throughout the modeling. Specifically, we denote the sizes of parameters and hyper-parameters, the effective DDR bandwidth, and the uniform data precision adopted across all transfers:

- S_{param} is the size (in bytes) of all parameters, *e.g.*, weights and biases.
- S_{hyper} is the size (in bytes) of all hyper-parameters, *e.g.*, kernel size, stride...

- B_{DDR} is the effective DDR-to-cluster transfer bandwidth (in bytes per cycle).
- We assume a uniform data precision $\mathcal{P} \in \{\text{FP32}, \text{INT32}, \text{FP16}, \text{INT16}, \text{INT8}\}$ across all parameters and hyper-parameters.

The total size of data transferred from DDR to the SMEM of one cluster is given by:

$$S_{\text{total}} = S_{\text{param}} + S_{\text{hyper}} \quad (4.4)$$

The size (in bytes) of each data element depends on its precision $\mathcal{P}$, as defined below:

$$s_{\mathcal{P}} = \begin{cases} 4 & \text{for FP32 or INT32} \\ 2 & \text{for FP16 or INT16} \\ 1 & \text{for INT8} \end{cases} \quad (4.5)$$

4.5.2 Hardware Setup

For the sake of simplification, our study relies solely on the fixed parameters presented previously in Table 4.6, and we keep the same values when deriving Table 4.25, without considering additional architectural details.

Architecture	DDR Bus Width	Bus Bandwidth (B_{DDR})
MPPA-256 Bostan	64 bits = 8 bytes	8 bytes/cycle
MPPA3-80 Coolidge	128 bits = 16 bytes	16 bytes/cycle

Table 4.25: DDR-to-cluster bus transfer rate per architecture

4.5.3 Cluster Memory Organization

As mentioned earlier, each compute cluster has SRAM capacity of:

- 2 MB per cluster in Bostan (128 KB per bank $\times$ 16 banks).
- 4 MB per cluster in Coolidge (256 KB per bank $\times$ 16 banks).

We assume that each SRAM is divided into two halves as:

- 50% reserved for storing intermediate results (feature maps).
- 50% used for storing parameters and hyper-parameters coming from the DDR.

We denote C_{SRAM} the portion of DDR space allocated per cluster for incoming data:

MPPA2-256 Bostan: $C_{\text{SRAM}} = 1 \text{ MB} = 1 \cdot 2^{20} \text{ bytes} = 1\,048\,576 \text{ bytes}$

MPPA3-80 Coolidge: $C_{\text{SRAM}} = 2 \text{ MB} = 2 \cdot 2^{20} \text{ bytes} = 2\,097\,152 \text{ bytes}$

4.5.4 Per-Layer Parameter & Hyper-Parameter Counts

We define P_l as the number of parameters and Hp_l as the number of hyper-parameters for each layer l . For each layer and activation function, the parameters and hyper-parameters are given by:

Convolution.

$$P_l^{\text{conv}} = k_H^{(l)} k_W^{(l)} D_l D_{l+1} + 1_{\text{bias}}^{(l)} D_{l+1}$$

$$Hp_l^{\text{conv}} = 9 \quad \text{with} \quad Hp_l^{\text{conv}} \in (k_H, k_W, p_H, p_W, s_H, s_W, d_H, d_W, D_{l+1})$$

Dense/FC.

$$P_l^{\text{FC}} = N_l N_{l+1} + 1_{\text{bias}}^{(l)} N_{l+1}$$

$$Hp_l^{\text{FC}} = 1 \quad \text{with} \quad Hp_l^{\text{FC}} \in (N_{l+1})$$

Pooling. For both pooling layers, we let:

$$P_l^{\text{pool}} = 0$$

$$Hp_l^{\text{pool}} = 4 \quad \text{with} \quad Hp_l^{\text{pool}} \in (m, n, s_H, s_W)$$

Flatten.

$$P_l^{\text{flat}} = Hp_l^{\text{flat}} = 0$$

Activation Functions. For those activation functions, the hyper-parameter values are:

$$Hp_l^{\text{act:LeakyReLU}} = 1 \quad \text{with} \quad Hp_l^{\text{act:LeakyReLU}} \in \alpha$$

$$Hp_l^{\text{act:Softmax}} = V$$

Otherwise:

$$P_l^{\text{act}} = Hp_l^{\text{act}} = 0$$

4.5.5 WCMA Cost for DDR Transfers

Let T_{DDR} be the WCMA cost in cycles for transferring data from DDR to one cluster:

$$T_{\text{DDR}} = \left\lceil \frac{S_{\text{total}}}{B_{\text{DDR}}} \right\rceil \quad (\text{cycles}) \quad (4.6)$$

Assuming all data is of type of precision $\mathcal{P}$, the parameter size (in bytes) becomes:

$$\begin{cases} S_{\text{param}} = P_l \cdot s_{\mathcal{P}} \\ S_{\text{hyper}} = Hp_l \cdot s_{\mathcal{P}} \end{cases} \Rightarrow S_{\text{total}} = S_{\text{param}} + S_{\text{hyper}} = (P_l + Hp_l) \cdot s_{\mathcal{P}} \quad (4.7)$$

4.5.6 SRAM Recharge per Layer

Each layer l may require one or more DDR recharges depending on its total memory footprint S_l . The per-node limit is C_{SRAM} defined earlier.

Number of Recharges per Layer. We denote:

$$\nu_l = \left\lceil \frac{S_l}{C_{\text{SRAM}}} \right\rceil \quad (4.8)$$

We let N_{nodes} be the total number of recharge nodes for all the CNN network that contains L layers:

$$N_{\text{nodes}} = \sum_{l=0}^{L-1} \nu_l \quad (4.9)$$

Full Recharge WCMA. For each full recharge node of size C_{SRAM} , the WCMA value cost is:

$$T_{\text{DDR}}^{\text{full}} = \left\lceil \frac{C_{\text{SRAM}}}{B_{\text{DDR}}} \right\rceil \quad (\text{cycles}) \quad (4.10)$$

The exact maximum value calculation is:

$$T_{\text{DDR}}^{\text{Bostan}} = \left\lceil \frac{2^{20}}{8} \right\rceil = 131,072 \text{ cycles}, \quad T_{\text{DDR}}^{\text{Coolidge}} = \left\lceil \frac{2^{21}}{16} \right\rceil = 131,072 \text{ cycles}$$

Note: Although memory and bus sizes differ, their ratios are equal, which leads to the same WCMA duration in cycles.

Last Recharge WCMA (if partial). The size of the last recharge for layer l is:

$$S_l^{\text{last}} = S_l - (\nu_l - 1) \cdot C_{\text{SRAM}} \quad (4.11)$$

This applies if S_l is not an exact multiple of C_{SRAM} , which means:

$$\text{If } S_l \bmod C_{\text{SRAM}} \neq 0 \Rightarrow \text{last node is partial, with } 0 < S_l^{\text{last}} < C_{\text{SRAM}}$$

The WCMA for this last node is:

$$T_{\text{DDR}}^{\text{last}} = \left\lceil \frac{S_l^{\text{last}}}{B_{\text{DDR}}} \right\rceil \quad (\text{cycles}) \quad (4.12)$$

4.5.7 Example: First Convolution Layer on Bostan

We consider those parameters with a total size for the first convolution layer $l = 0$:

$$P_0 + Hp_0 \approx 4.7 \times 10^6$$

Total Memory Size. Using $s_{\mathcal{P}} = 4$ bytes for FP32:

$$S_0 = (P_0 + Hp_0) \cdot s_{\mathcal{P}} = 4.7 \times 10^6 \cdot 4 = 18.8 \text{ MB}$$

In bytes:

$$S_0 = 18.8 \times 2^{20} = 19,729,408 \text{ bytes}$$

SRAM Allocation. The SRAM space allocated per recharge node is:

$$C_{\text{SRAM}} = 1 \text{ MB} = 2^{20} = 1,048,576 \text{ bytes}$$

Number of Recharges.

$$\nu_0 = \left\lceil \frac{S_0}{C_{\text{SRAM}}} \right\rceil = \left\lceil \frac{19,729,408}{1,048,576} \right\rceil = 19$$

Full vs. Partial Recharges.

- Number of full-size recharges: $\nu_0^{\text{full}} = \nu_0 - 1 = 18$
- Size of the last (partial) node:

$$S_0^{\text{last}} = S_0 - (\nu_0^{\text{full}} \times C_{\text{SRAM}}) = 19,729,408 - 18 \times 1,048,576 = 855,040 \text{ bytes}$$

WCMA for full and last nodes.

$$T_{\text{DDR}}^{\text{full}} = \left\lceil \frac{C_{\text{SRAM}}}{B_{\text{DDR}}} \right\rceil = \left\lceil \frac{1,048,576}{8} \right\rceil = 131,072 \text{ cycles}$$

$$T_{\text{DDR}}^{\text{last}} = \left\lceil \frac{S_0^{\text{last}}}{B_{\text{DDR}}} \right\rceil = \left\lceil \frac{855,040}{8} \right\rceil = 106,880 \text{ cycles}$$

Total WCMA.

$$T_{\text{DDR}}^{\text{net}} = \nu_0^{\text{full}} \times T_{\text{DDR}}^{\text{full}} + T_{\text{DDR}}^{\text{last}} = 18 \times 131072 + 106880 = 2,466,176 \text{ cycles}$$

4.5.8 Important Remarks

To guarantee predictability in DDR-to-cluster transfers, we follow these key assumptions and constraints:

- **Transfer behavior:** All DDR transfers are treated as non-preemptive and strictly serialized on core 0 to avoid arbitration unpredictability and shared-resource interference.
- **Recharge node partition:** The parameters of a layer are decomposed into multiple recharge sub-nodes once the recharge limit is reached. Each sub-node is transferred and executed sequentially, ensuring that on-chip memory constraints are respected.
- **Recharging dependency:** A strict dependency is enforced between each recharge node and the operations in the corresponding layer. We ensure that recharging is completed before initiating the operation computations of each layer, in order to maintain execution correctness.

4.6 Data Transfers Analysis Between Cores Within and Across Clusters

In the MIA tool, this type of data movement is referred to as **accessTo**. It corresponds to the transfer of output data produced by one core to serve as input data for another core, enabling the receiving core to proceed with its execution.

We make the following assumptions regarding intra/inter-cluster communication:

- If two cores communicate within the same cluster, the associated transfer cost is limited to the number of cycles required for the actual data transfer.
- If a core communicates with another core located in a different cluster, the transfer cost includes both the data transfer cycles and an additional certain latency.

It is important to recall that on the Bostan architecture, inter-cluster communication takes place via the NoC through Rx and Tx ports, whereas on the Coolidge architecture, inter-cluster communication is handled through the AXI bus. For the sake of simplicity in this study, we will assume a fixed inter-cluster latency penalty for both platforms.

We begin by defining the needed notations below:

- $l, l + 1$: producer layer index, and consumer layer index.
- $s_{\mathcal{P}}$: data precision (in bytes).
- $\mathcal{A}_{l \rightarrow l+1}$: subset of the output data of l actually required by $l + 1$.
- $|\mathcal{A}_{l \rightarrow l+1}|$: number of scalar elements in that subset.
- $B_{\text{arch}}^{\text{intra}}$: intra-cluster bandwidth (bytes/cycle).
- $B_{\text{arch}}^{\text{inter}}$: inter-cluster bandwidth (bytes/cycle).
- $L_{\text{arch}}^{\text{inter}}$: fixed inter-cluster latency penalty (cycles).
- arch: the architecture used; $\text{arch} \in \{\text{Bostan}, \text{Coolidge}\}$.
- $\psi_{\text{cluster}}(l, l + 1)$: inter-cluster communication operator.

We assume $B_{\text{arch}}^{\text{intra}} = B_{\text{arch}}^{\text{inter}}$ and reuse the bus bandwidth values reported in Table 4.25, avoiding additional complexity in the study.

For inter-cluster communication operator $\psi_{\text{cluster}}(l, l + 1)$ it takes:

$$\psi_{\text{cluster}}(l, l + 1) = \begin{cases} 1, & \text{if } l \text{ and } l + 1 \text{ map to different clusters,} \\ 0, & \text{otherwise.} \end{cases} \quad (4.13)$$

4.6.1 Data Volume Calculation

We compute the amount of data transferred $\mathcal{V}_{l \rightarrow l+1}$ from one core to another by:

$$\mathcal{V}_{l \rightarrow l+1} = |\mathcal{A}_{l \rightarrow l+1}| \cdot s_{\mathcal{P}} \quad (4.14)$$

Knowing that $\mathcal{A}_{l \rightarrow l+1}$ differs from a layer to another layer, then:

$$\mathcal{A}_{l \rightarrow l+1} = \begin{cases} \text{only the corresponding output of } l, & \text{if } l + 1 \text{ is Pool/Activation,} \\ \text{all output data of } l, & \text{if } l + 1 \text{ is Conv/Flatten/Dense.} \end{cases} \quad (4.15)$$

If the output of layer l is partitioned into i parts, the equation becomes:

$$\mathcal{V}_{l, i \rightarrow l+1, i} = |\mathcal{A}_{l, i \rightarrow l+1, i}| \cdot s_{\mathcal{P}}. \quad (4.16)$$

4.6.2 Per-Layer Operations Volume

We denote the transfer size associated with one operation of layer l to the next layer $l + 1$ as:

Conv: Needs all previous data.

$$\mathcal{V}_{l \rightarrow l+1}^{\text{Conv}} = \sum_{d=1}^{D_{l+1}} H_{l+1} W_{l+1} s_{\mathcal{P}} \quad (4.17)$$

Pool: Only the corresponding data.

$$\mathcal{V}_{l \rightarrow l+1}^{\text{Pool}} = H_{l+1} W_{l+1} s_{\mathcal{P}} \quad (4.18)$$

Flatten: Vector of length $N_l = H_l W_l D_l$.

$$\mathcal{V}_{l \rightarrow l+1}^{\text{Flat}} = N_{l+1} s_{\mathcal{P}} \quad (4.19)$$

Dense: Consumes the full output vector N_{l+1} of the previous layer.

$$\mathcal{V}_{l \rightarrow l+1}^{\text{FC}} = \sum_{n=1}^{N_{l+1}} s_{\mathcal{P}} \quad (4.20)$$

Activation: Only the corresponding data.

$$\mathcal{V}_{l \rightarrow l+1}^{\text{Act}} = H_{l+1} W_{l+1} s_{\mathcal{P}} \quad (4.21)$$

Except for Softmax:

$$\mathcal{V}_{l \rightarrow l+1}^{\text{Softmax}} = \sum_{v=1}^V s_{\mathcal{P}} \quad (4.22)$$

4.6.3 Transfer Cost Calculation

If the transfer $Tr_{l \rightarrow l+1}$ between cores occurs within the same cluster, we assume no additional latency; otherwise, a penalty is added. We model this as:

$$Tr_{l \rightarrow l+1} = \begin{cases} \left\lceil \frac{\mathcal{V}_{l \rightarrow l+1}}{B_{\text{arch}}^{\text{intra}}} \right\rceil, & \psi_{\text{cluster}}(l, l+1) = 0, \\ L_{\text{arch}}^{\text{inter}} + \left\lceil \frac{\mathcal{V}_{l \rightarrow l+1}}{B_{\text{arch}}^{\text{inter}}} \right\rceil, & \psi_{\text{cluster}}(l, l+1) = 1. \end{cases} \quad (4.23)$$

4.6.4 Average Inter-Cluster Latency

The average inter-cluster latency $L_{\text{arch}}^{\text{inter}}$ is decomposed into three components: a setup cost before the transfer, the average per-hop traversal latency across the NoC or AXI fabric, and a synchronization cost to ensure memory consistency. This decomposition allows us to capture the key sources of delay. We denote:

$$L_{\text{arch}}^{\text{inter}} = L_{\text{arch}}^{\text{setup}} + Hop_{l \rightarrow l+1}^{\text{avg}} L_{\text{arch}}^{\text{hop}} + L_{\text{arch}}^{\text{sync}} \quad (4.24)$$

Then apply a 10% safety margin as a better estimation for a worst-case:

$$L_{\text{arch}}^{\text{inter,avg}} = \left\lceil 1.1 (L_{\text{arch}}^{\text{setup}} + Hop_{l \rightarrow l+1}^{\text{avg}} L_{\text{arch}}^{\text{hop}} + L_{\text{arch}}^{\text{sync}}) \right\rceil \quad (4.25)$$

We then bound each term in Table 4.26 with average-case approximate values before adding 10% guard:

Terms	Bostan (NoC)	Coolidge (AXI)	Explanation
$L_{\text{arch}}^{\text{setup}}$	64 cycles	48 cycles	Approximate cost for configuring NoC Tx/Rx in Bostan, and AXI descriptors in Coolidge.
$Hop_{l \rightarrow l+1}^{\text{avg}}$	3 hops	2 hops	Average cluster distance.
$L_{\text{arch}}^{\text{hop}}$	8 cycles/hop	4 cycles/hop	Per router (NoC) / Crossbar depth (AXI).
$L_{\text{arch}}^{\text{sync}}$	32 cycles	16 cycles	Memory fence before consumer node reads to guarantee the total observation of the data.

Table 4.26: Average inter-cluster latency decomposition before 10% guard

The decomposition in Table 4.26 highlights the main architectural differences. On Bostan, the NoC setup and synchronization dominate the total latency, while per-hop costs are moderate. In contrast, Coolidge benefits from a lower setup time and lighter synchronization overhead thanks to its AXI crossbar fabric. The final numerical application of the total latency, after applying the 10% guard margin, is:

$$\text{Bostan base} = 64 + 3 \times 8 + 32 = 120 \text{ cycles,}$$

$$L_{\text{Bostan}}^{\text{inter,avg}} = \lceil 1.1 \times 120 \rceil = \boxed{132 \text{ cycles}}$$

$$\text{Coolidge base} = 48 + 2 \times 4 + 16 = 72 \text{ cycles,}$$

$$L_{\text{Coolidge}}^{\text{inter,avg}} = \lceil 1.1 \times 72 \rceil = \boxed{80 \text{ cycles}}$$

4.7 Establishing Task Execution Order for Deterministic Inference

We previously assumed that our generated DAG is both deterministic and acyclic. However, despite satisfying these properties, the MIA compiler rejects the generated .yaml file produced by the MIND software, reporting non-determinism issues whose root cause we could not precisely identify. To address this, we enforce an explicit static execution order for tasks mapped to the same core. Specifically, for each core, we assign increasing order values to the tasks it handles, thereby linearizing their execution.

This mechanism does not imply any form of task prioritization, as we consider all tasks equally important. Instead, it just guides MIA to interpret the schedule deterministically by maintaining functional correctness without changing the semantics of the CNN model.

4.7.1 Formalization of Intra-Core Task Ordering

Let $\mathcal{T} = \{\tau_0, \tau_1, \dots, \tau_{N-1}\}$ denote the set of tasks in the DAG, and $\mathcal{C} = \{c_0, c_1, \dots, c_{M-1}\}$ represent the set of available processing cores. We define a mapping function μ to indicate that task τ_i is mapped to core c_j :

$$\mu : \mathcal{T} \rightarrow \mathcal{C} \quad \text{such that} \quad \mu(\tau_i) = c_j \quad (4.26)$$

We let $\mathcal{T}_j$ be the subset of tasks mapped to core c_j :

$$\mathcal{T}_j = \{\tau_i \in \mathcal{T} \mid \mu(\tau_i) = c_j\} \quad (4.27)$$

To enforce a static and deterministic execution sequence for tasks assigned to the same core, we define a core-local total order ord_j for each core c_j , which respects the order in which tasks are declared. We define the bijective¹ function ord_j :

$$ord_j : \mathcal{T}_j \rightarrow \{0, 1, \dots, |\mathcal{T}_j| - 1\} \quad \text{such that} \quad ord_j(\tau_{i-1}) < ord_j(\tau_i) < ord_j(\tau_{i+1}) \quad (4.28)$$

The function ord_j assigns a unique integer index to each task on core c_j , effectively linearizing intra-core task execution while preserving the DAG semantics. This index is then directly inserted into the generated `.yaml` file as an `order` field for each task node and interpreted by the MIA compiler as the enforced execution sequence.

4.7.2 Illustrative Example

Consider a simplified case in which four tasks from the DAG are all mapped to the same core, say c_0 . The task-to-core mapping function μ is defined as:

$$\begin{aligned} \mu(\tau_0) &= c_0 \\ \mu(\tau_1) &= c_0 \\ \mu(\tau_2) &= c_0 \\ \mu(\tau_3) &= c_0 \end{aligned}$$

From Equation (4.27), we construct the subset of tasks assigned to core c_0 :

$$\mathcal{T}_0 = \{\tau_0, \tau_1, \tau_2, \tau_3\}$$

Applying the intra-core ordering function $ord_0 : \mathcal{T}_0 \rightarrow \{0, 1, 2, 3\}$ gives:

$$\begin{aligned} ord_0(\tau_0) &= 0 \\ ord_0(\tau_1) &= 1 \\ ord_0(\tau_2) &= 2 \\ ord_0(\tau_3) &= 3 \end{aligned}$$

Discussion. This ordering does not affect the global dependencies constraint imposed by the DAG edges. It just serves as a disambiguation mechanism in the case of concurrent and unordered tasks. Which means that even if the DAG allows these tasks to execute in any order (because of their dependencies), the explicit ordering we introduced earlier removes any ambiguity.

This ensures that the execution sequence is always the same across multiple runs or analyses, which is essential for deterministic and static behavior and also compatibility with MIA tool.

¹The function ord_j is bijective because it is both injective, *i.e.*, each task gets a unique order value (no duplicates). And surjective, *i.e.*, all order values from 0 to $|\mathcal{T}_j| - 1$ are used.

4.8 Period and Deadline Configuration

There is no universally defined equation for determining the period and deadline of a task. These parameters are typically specified by the user based on system requirements. However, their values must satisfy real-time constraints to ensure correct scheduling and execution in MIA tool.

4.8.1 Converting Cycles to Time in Both Architectures

To convert from cycles to actual time in seconds, we use the following formula:

$$Time = \frac{T_{\text{cycles}}}{f_{\text{clock}}} \quad (4.29)$$

where:

- T_{cycles} is the number of cycles,
- f_{clock} is the operating clock frequency of an architecture, in Hz,
- $Time$ is the transfer duration in seconds.

4.8.2 Frequency Ranges

Table 4.27 presents the clock frequencies of the Kalray MPPA architectures.

Family	Silicon rev.	Typical / Nominal	Max. turbo
MPPA-256 “Bostan”	TSMC 28 nm	500 MHz	600 MHz
MPPA3-80 “Coolidge”	TSMC 16 nm	900 MHz	1 GHz

Table 4.27: Clock frequencies of Kalray MPPA devices

In our project, we do not consider turbo mode operation and will use only the standard typical/nominal values.

4.9 Conclusion

This chapter presented an architecture-aware cost model for CNN inference on Kalray MPPA devices, Bostan and Coolidge. We established per output element formulas for calculation, from which we derived WCET in isolation and WCMA under the execution constraints of each architecture. These unit costs were then lifted to layer level and systematically propagated through the regrouping policies, covering both the initial methods (MMM) and the advanced policies (MLWM, SSP, RCCT, DF). In parallel, we modeled DDR → SRAM recharges, intra/inter-cluster data movements, introduced a static intra-core ordering to ensure deterministic schedules compatible with the MIA tool, and we ended this chapter with the period and deadline configuration needed to have a completed input `.yaml` file.

The analysis highlights some practical takeaways. First, we observe that Coolidge WCET in isolation and WCMA calculations are reduced compared to the Bostan due to architectural optimizations. Second, the regrouping policies change the counting unit: MMM aggregates at the layer granularity; MDM preserves per-element granularity; MLWM omits the channel dimension and scales by $\rho = H_{l+1}W_{l+1}$; SSP varies ρ_i across spatial parts due to padding; RCCT requires a final element-wise Summation Layer; and DF fuses neurons into groups, thereby reducing the number of nodes in a dense layer while preserving the computations within each fused node. Under DF policy, Masking Layers are applied only for Sigmoid and Softmax activations when DNs are introduced.

Our modeling in this chapter deliberately adopts conservative assumptions to safeguard worst-case reasoning. For this reason, cache effects are neglected to avoid underestimating WCMA and, of course, for simplification purposes.

After defining each regrouping policy and its architectural costs, the next final chapter, Test and Evaluation Chapter, quantitatively compares these policies using evaluation metrics that we will formalize. We apply each regrouping strategy to a prototype CNN model under multiple configurations, enabling a parameter-by-parameter analysis and supporting the identification of the most advantageous and optimized policy.

Chapter 5

Test and Evaluation

5.1 Introduction

In this final chapter, we present the testing and evaluation by analyzing the results obtained from applying the regrouping policies to a prototype CNN model, as well as the output of the MIA tool, particularly the WCRT values on which our analysis is based.

We begin by introducing the prototype CNN model in the pre-testing phase. The chosen CNN serves as a testbench throughout all sections of this chapter. For simplicity, only the layer types cited in this project are used, and the same assumptions as in Chapter 1, Section 1.2 are adopted.

Next, we define the evaluation metrics that will be employed, followed by the testing of the regrouping policies. This chapter concludes with an evaluation and discussion section, where we establish success criteria to determine which behaviors are considered beneficial and which are not.

During testing, we explore variations of selected network parameters to observe the behavior of WCRT and other metrics under these changes. Examples include varying the input image size, the number of kernels, and filters in convolutional layers. We do not vary pooling parameters, since their effect is mainly the reduction of feature map size, which would not generate meaningful insights for our evaluation. For the same reason, we keep the number of dense neurons fixed, as DF already explores neuron-count variation.

Our detailed analysis is carried out primarily on the Bostan architecture, given its large core count (256 cores) and FP32 data precision. We then extend some of the study to the Coolidge architecture to highlight differences on a more optimized platform, including tests with FP16 precision to observe the contribution of the TCA co-processor.

By default, all tests are performed in cycles unit with the Bias term enabled in both convolutional and dense layers, thereby ensuring the use of MAC operations only in both layers. The naming convention for test cases follows the format `CPL_MLP`, where each zone indicates the regrouping policy applied.

5.2 Pre-Testing Phase

Before beginning the testing phase, we first select a CNN model to serve as a prototype for the next tests and analyses. We propose a lightweight CNN used solely for testing and comparative purposes; it is not intended for real use. Its topology is:

- **Input:** an RGB picture with a dimension $64 \times 64 \times 3$.
- **Backbone:** Two stages of convolution that use 3×3 kernels, Tanh activation, stride $[1, 1]$, dilation $[1, 1]$, and “same” padding.
 - s_1 : $\text{Conv}(3 \times 3 \times 16, \text{Tanh}) \rightarrow \text{AvgPool}(2 \times 2)$
 - s_2 : $\text{Conv}(3 \times 3 \times 32, \text{Tanh}) \rightarrow \text{AvgPool}(2 \times 2)$
- **Pooling:** After each stage, a AvgPool with pool size 2×2 and stride $[1, 1]$ is applied. We should note that padding in pooling layers is omitted; each pooling reduces height and width by exactly one pixel.
- **Classifier:** $\text{Flatten} \rightarrow \text{FC}(64, \text{ReLU}) \rightarrow \text{FC}(10, \text{Softmax})$

- **Feature Maps:** Convolutions keep the spatial size without reducing it (“same” padding and stride 1), while pooling layers reduce it by 1 in each dimension. With two poolings after each stage, the spatial progression is:

$$64 \rightarrow 63 \rightarrow 62$$

The output of the last pooling layer has shape $62 \times 62 \times 32$, therefore the Flatten length is $62 \times 62 \times 32 = 123,008$

Figure 5.1 illustrates the 3D representation of the proposed prototype CNN model in terms of its operation layers made by MIND software. Each layer type is visualized, providing a structural overview of how the network processes data sequentially.

This representation highlights the computational building blocks of the prototype model.

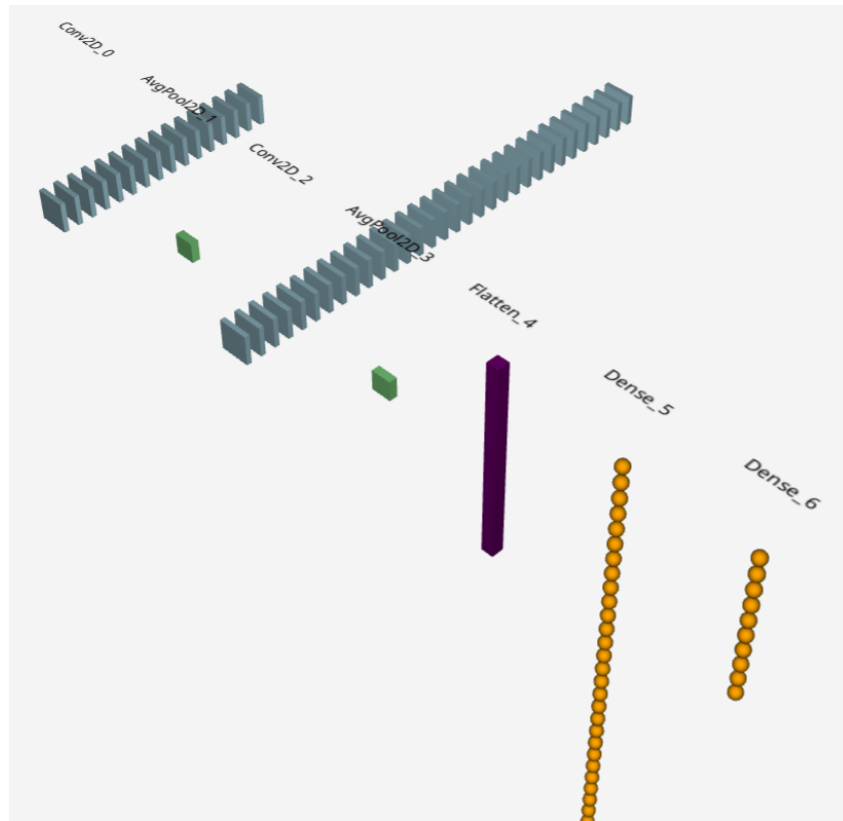


Figure 5.1: 3D representation of the prototype CNN model’s operation layers generated by MIND

Figure 5.2 complements the previous Figure 5.1 by showing the 3D representation of the corresponding feature map layers produced at each stage of the prototype CNN. In this view, the spatial dimensions and channel evolution across the network are visualized.

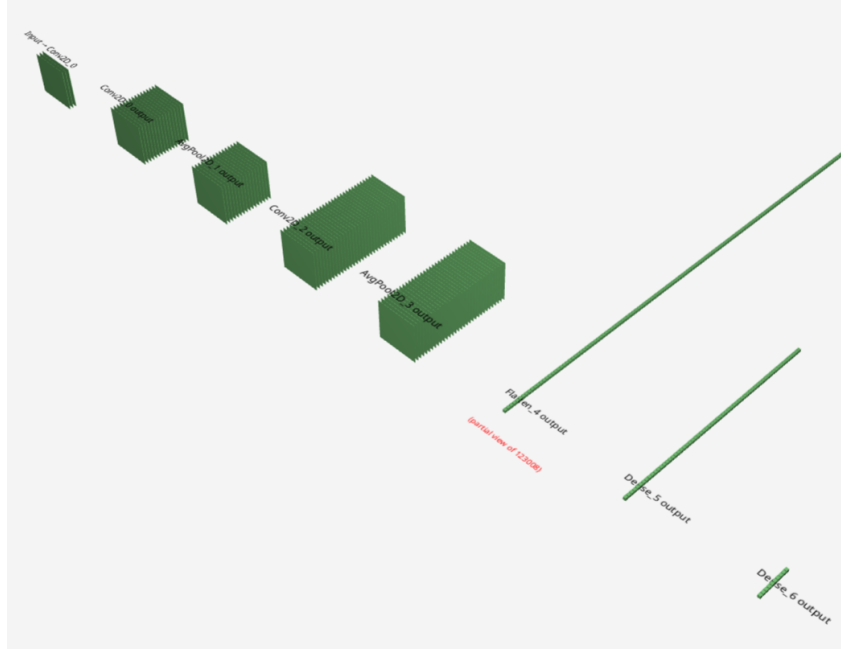


Figure 5.2: 3D representation of the prototype CNN model’s feature map layers generated by MIND

5.3 Evaluation Metrics

In this section, we formalize the equations for the performance and structural metrics used to compare each regrouping policy; these will guide our analysis and enable a more rigorous evaluation. Table 5.1 summarizes the notation used throughout the experiments in this chapter.

Symbol	Name	Units	Definition / How obtained
WCRT	Worst-Case Response Time	cycles	End-to-end latency of the CNN model which is the DAG schedule returned by MIA.
W	Total of the WCET	cycles	Total compute work in isolation over all YAML nodes.
M	Total of the WCMA	cycles	Local core and SMEM-bank load/store traffic exchange cost only (no transfers between clusters).
X	Transfer cost	cycles	Inter/intra cluster data transfer cycles derived from edges/arcs in the DAG (core communication).
Intf	Total of interference	cycles	Total interference in the WCRT calculation is produced by M and X , and it is a value added to W .
N	Number of cores	–	Total hardware cores on the target architecture; here Bostan and Coolidge respectively $N \in \{256, 80\}$

Table 5.1: Summary of notations used in this chapter

5.3.1 Runtime Cost Performance

Taking the double MMM policy as the reference, we compute the runtime impact and performance of our DAG topology relative to each policy as follows:

- **WCRT:** In parallel scheduling, the actual makespan, *i.e.*, time between the start of the first task and the completion of the last task, including dependencies, queueing on shared resources, and any transfer delays. We should note that the smaller is better and we use it for comparison purposes between our policies.
- **Speedup Factor:** We use the double MMM policy, *i.e.*, MMM_MMM, in both CPL and MLP, as our baseline, since it is the longest and most time-consuming policy, making it an appropriate reference for evaluating execution speed-up on the Kalray architectures:

$$\text{Speedup} = \frac{\text{WCRT}_{\text{MMM_MMM}}}{\text{WCRT}_{\text{policy}}} \quad (5.1)$$

The result for whichever chosen policy, *e.g.*, 1.65, which means that this chosen policy is $\times 1.65$ faster than double MMM. We compute this ratio only within the same architecture and precision.

- **Compute Usage / Memory Usage / Transfer Usage:** According to the generated YAML file from MIND, we let:

$$W = \sum \text{procDemand}$$

$$M = \sum \text{memDemand}$$

$$X = \sum \text{accessTo}$$

Then we calculate respectively the percentages of Compute Usage “UComm”, Memory Usage “UMem” and Transfer Usage “UXfer”:

$$\text{UComm}\% = \frac{W}{W + M + X} \times 100 \quad (5.2)$$

$$\text{UMem}\% = \frac{M}{W + M + X} \times 100 \quad (5.3)$$

$$\text{UXfer}\% = \frac{X}{W + M + X} \times 100 \quad (5.4)$$

We should note that for policies that raise partition overlap, they may increase UXfer% even if W decreases which may generate an overhead.

- **Interference Rate:** In MIA there are two modes for computing WCRT:
 - i The “NoInter” mode, which evaluates the execution without interference and corresponds to the pure WCET of the critical path under parallel execution.
 - ii The “Bostan” or “Coolidge_0” mode, which accounts for interference on the target architecture.

Thus, the WCRT including interference can be expressed as:

$$\text{WCRT}_{\text{Intf}} = \text{WCET}_{\text{critical path}} + \text{Total Interference} \quad (5.5)$$

For simplicity, we denote:

$$\text{WCRT} = \text{WCET} + \text{Intf} \quad (5.6)$$

We introduce two ratios to compare the relative contributions of computation and interference:

$$\text{WCET}\% = \frac{\text{WCET}}{\text{WCRT}} \times 100 \quad (5.7)$$

$$\text{Intf}\% = \frac{\text{WCRT} - \text{WCET}}{\text{WCRT}} \times 100 = \frac{\text{Intf}}{\text{WCRT}} \times 100 \quad (5.8)$$

Ideally, the computation cost should dominate over the interference. However, when interference exceeds the computation share, it introduces significant overhead.

5.3.2 Structural Expansion

We denote the double MMM policy as our reference. The structural expansion of our DAG topology with respect to each policy is then calculated as follows:

- **Node Growth Factor:** Let $\mathcal{C}$ denote the set of DAG nodes as:

$$\mathcal{C}_{\text{nodes}} = \frac{|\mathcal{C}_{\text{policy}}|}{|\mathcal{C}_{\text{MMM_MMM}}|} \quad (5.9)$$

This captures how much the policy increases the number of computational tasks, *i.e.*, $\mathcal{C}_{\text{nodes}} = 1$ means no change; > 1 means expansion.

- **Edge Growth Factor:** Let $\mathcal{L}$ denote the set of DAG edges (dependencies/communications). We define:

$$\mathcal{L}_{\text{edges}} = \frac{|\mathcal{L}_{\text{policy}}|}{|\mathcal{L}_{\text{MMM_MMM}}|} \quad (5.10)$$

This captures how much the policy increases dependency/communication paths, *i.e.*, $\mathcal{L}_{\text{edges}} = 1$ means no change; > 1 means expansion.

5.4 Testing with MMM_MMM

The baseline configuration `MMM_MMM` serves as our primary reference point against which all other policies are evaluated. Table 5.2 summarizes the headline performance metrics obtained under this policy.

WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
59948212	$\times 1$	100	0	64.45	35.18	0.37

Table 5.2: Headline performance metrics for `MMM_MMM`

As expected, the **Speedup** is normalized to $\times 1$, since **MMM_MMM** defines the baseline itself. The **WCRT** reaches 59.95×10^6 cycles (119.89 ms), which represents the end-to-end latency including all task scheduling and dependencies. Importantly, **MMM_MMM** does not introduce partition overlap or extra communication because it is a single-core modeling, the **interference share** is zero ($\text{Intf}\% = 0$) and the full response time is attributable to computation alone ($\text{WCET}\% = 100$). Regarding the usage ratios, the majority of the workload is compute-bound with $\text{UComm} = 64.45\%$, while memory traffic accounts for $\text{UMem} = 35.18\%$, and transfers remain negligible at 0.37% . This shows that the **MMM** baseline does not stress the inter-cluster communication fabric, which is expected since it executes whole layers as monolithic tasks. This justifies our choice using **MMM_MMM** as a “worst-case” baseline in terms of parallel utilization.

Table 5.3 provides the structural metrics of the **MMM_MMM** DAG.

$\mathcal{C}_{\text{MMM_MMM}}$	$\mathcal{L}_{\text{MMM_MMM}}$
18	17

Table 5.3: DAG node and edge of the **MMM_MMM**

The model generates only 18 computational nodes and 17 dependency edges. This structure minimizes communication overhead, and it also severely restricts the amount of task-level parallelism that we can exploit. This compactness is precisely what motivated us to introduce more fine-grained regrouping policies, whose effect will be assessed in the following sections.

5.5 Testing with **MLWM_MMM**

The **MLWM_MMM** configuration illustrates the effect of macroscopic layer-wise modeling applied at the CPL stage while keeping **MMM** at the MLP stage as it is shown in Table 5.4.

WCRT	Speedup	<i>WCET%</i>	<i>Intf%</i>	<i>UComm%</i>	<i>UMem%</i>	<i>UXfer%</i>
10978053	$\times 5.46$	90.20	9.80	63.72	34.78	1.50

Table 5.4: Headline performance metrics for **MLWM_MMM**

The table shows that the **WCRT** drops in a strong way to 1.09×10^7 cycles (21.95 ms), yielding a **Speedup** of $\times 5.46$ compared to the baseline **MMM_MMM**. This confirms that exposing finer task granularity at the convolutional level enables much better parallel execution. Nevertheless, the cost of this decomposition is visible in the interference share: $\text{WCET}\% = 90.20$ while $\text{Intf}\% = 9.80$, indicating that nearly 10% of the runtime overhead is due to contention on local memory banks and transfers.

The workload distribution remains similar to the baseline in terms of compute and memory usage ($\text{UComm} = 63.72\%$, $\text{UMem} = 34.78\%$), but the **transfer usage** increases to 1.50% , reflecting the additional communication DAG edges resulting from the per-feature-map calculation.

Table 5.5 highlights this structural expansion: the DAG grows from 18 to 156 nodes and from 17 to 743 edges. This represents an $\mathcal{C}_{\text{nodes}}$ factor of 8.66 and an $\mathcal{L}_{\text{edges}}$ factor of 43.70, which means the number of communication dependencies has increased compared to the baseline.

$\mathcal{C}_{\text{MLWM_MMM}}$	$\mathcal{C}_{\text{nodes}}$	$\mathcal{L}_{\text{MLWM_MMM}}$	$\mathcal{L}_{\text{edges}}$
156	8.66	743	43.70

Table 5.5: DAG node and edge of the MLWM_MMM

5.6 Testing with SSP_MMM

The SSP_MMM configuration applies the structured spatial partitioning policy at the CPL stage. As a reminder, the operator $\lambda(h, w)$ denotes the partitioning factor along the height h and width w dimensions of the feature maps. By increasing $\lambda(h, w)$, the feature maps are spatially divided into smaller tiles, which expose more parallelism but also may increase communication overhead. Table 5.6 shows the performance results for different values of $\lambda(h, w)$.

$\lambda(h, w)$	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
$\lambda(1, 2)$	9559556	$\times 6.27$	93.53	6.47	63.86	34.60	1.54
$\lambda(2, 1)$	9559028	$\times 6.27$	93.54	6.46	63.86	34.60	1.54
$\lambda(2, 2)$	8909746	$\times 6.72$	94.57	5.43	63.89	34.44	1.68
$\lambda(4, 4)$	8813184	$\times 6.80$	92.70	7.30	63.83	33.78	2.39
$\lambda(4, 5)$	8906856	$\times 6.73$	91.74	8.26	63.83	33.54	2.63
$\lambda(5, 4)$	8909074	$\times 6.72$	91.65	8.35	63.83	33.54	2.64
$\lambda(5, 5)$	9081398	$\times 6.60$	90.18	9.82	63.83	33.30	2.87
$\lambda(6, 6)$	9305446	$\times 6.44$	88.18	11.82	63.73	32.83	3.44

Table 5.6: Headline performance metrics for SSP_MMM

The smallest partitions, such as $\lambda(1, 2)$ and $\lambda(2, 1)$, already achieve significant gains with a **Speedup** of $\times 6.27$, and an overhead limited to 6.47% interference. Increasing the partitioning to $\lambda(2, 2)$ further reduces the **WCRT** to 8.91×10^6 cycles (17.81 ms), giving a **Speedup** of $\times 6.72$ and the highest $WCET\% = 94.57$, which means that computation dominates strongly over interference.

We observe that the configuration with $\lambda(4, 4)$ delivers the smallest **WCRT** at 8.81×10^6 cycles (17.62 ms) and the maximum **Speedup** of $\times 6.80$. At this point, the balance between finer-grained parallelism and interference remains favorable, although the interference share increases moderately to 7.30%. However, when $\lambda(h, w)$ is increased further, *e.g.*, $\lambda(5, 5)$ or $\lambda(6, 6)$, the **WCRT** starts to rise again, and the interference share reaches 9.82% and 11.82%, respectively. Moreover, the **transfer usage** also increases, reaching a value of $UXfer = 3.44\%$. This shows that excessive partitioning introduces diminishing

returns: while the DAG exposes more concurrency, the overhead of transfers surpasses the compute benefits.

This observation is reinforced by Table 5.7, which shows the structural expansion.

$\lambda(h, w)$	$\mathcal{C}_{\text{SSP_MMM}}$	$\mathcal{C}_{\text{nodes}}$	$\mathcal{L}_{\text{SSP_MMM}}$	$\mathcal{L}_{\text{edges}}$
$\lambda(1, 2)$	300	16.66	1,479	87.00
$\lambda(2, 1)$	300	16.66	1,479	87.00
$\lambda(2, 2)$	588	32.66	2,951	173.58
$\lambda(4, 4)$	2,316	128.66	11,783	693.11
$\lambda(4, 5)$	2,892	160.66	14,727	866.29
$\lambda(5, 4)$	2,892	160.66	14,727	866.29
$\lambda(5, 5)$	3,612	200.66	18,407	1082.76
$\lambda(6, 6)$	5,196	288.66	26,503	1559.00

Table 5.7: DAG node and edge growth of SSP_MMM

For small partitions $\lambda(1, 2)$ and $\lambda(2, 1)$, the DAG grows to 300 nodes and 1,479 edges, a moderate expansion compared to the baseline. At $\lambda(2, 2)$, the graph nearly doubles again to 588 nodes and 2,951 edges. The $\lambda(4, 4)$ configuration produces 2,316 nodes and 11,783 edges, corresponding to both factors of $\mathcal{C}_{\text{nodes}} = 128.66$ and $\mathcal{L}_{\text{edges}} = 693.11$. While this structural growth is considerable, it remains beneficial in terms of WCRT. However, larger partitions such as $\lambda(5, 5)$ or $\lambda(6, 6)$ result in explosive growth up to 5,196 nodes and 26,503 edges, which directly correlates with higher interference. These results confirm that there are benefits from SSP policy, but only up to an optimal point. The $\lambda(4, 4)$ configuration is the sweet spot, achieving the best WCRT and speedup, whereas higher partitioning leads to overheads that degrade performance.

5.7 Testing with RCCT_MMM

The RCCT_MMM configuration applies the channel-wise regrouping in convolutional layers at the CPL zone. This policy divides the input channels of a selected convolutional layer into groups. We recall that the partition operator for this policy is $\zeta(c)$. A critical design choice that should be taken into consideration is the convolutional layer to which RCCT is applied: it is preferable to select a layer with a minimal number of filters in order to avoid excessive redundancy in the computations. In this example, the CNN network contains two convolutional layers, with 16 and 32 filters respectively. We apply the RCCT policy only on the first layer with 16 filters. Table 5.8 presents the performance results for different partition values of $\zeta(c)$.

$\zeta(c)$	WCRT	Speedup	<i>WCET</i> %	Intf%	UComm%	UMem%	UXfer%
$\zeta(2)$	10391413	$\times 5.76$	89.65	10.35	55.09	43.01	1.90
$\zeta(4)$	10365781	$\times 5.78$	89.60	10.40	54.98	43.04	1.98
$\zeta(5)$	10371793	$\times 5.77$	89.58	10.42	54.85	43.16	1.99
$\zeta(8)$	10393393	$\times 5.76$	89.52	10.48	54.49	43.49	2.02

Table 5.8: Headline performance metrics for RCCT_MMM

The **WCRT** values are stable across all configurations, around 1.04×10^7 cycles, with a **Speedup** of approximately $\times 5.76$ - $\times 5.78$ even though the operator $\zeta(c)$ is increasing. The interference share is non-negligible, around 10.35 - 10.48%, with *WCET*% close to 90%. Compared to previous policies, the distribution of workload shows a different profile: compute usage decreases to nearly 55%, while memory usage rises above 43%, indicating that RCCT amplifies local memory accesses due to redundant calculations caused by channel tiling. The transfer usage UXfer remains modest, below 2.1%.

Among the different tiling factors tested, the best trade-off is observed for $\zeta(4)$. This configuration achieves the lowest WCRT at 1.036×10^7 cycles (20.73 ms) with a speedup of $\times 5.78$, while keeping the interference rate at 10.40%, very similar to the other tiling factors.

The structural impact of RCCT is summarized in Table 5.9.

$\zeta(c)$	$\mathcal{C}_{\text{RCCT_MMM}}$	$\mathcal{C}_{\text{nodes}}$	$\mathcal{L}_{\text{RCCT_MMM}}$	$\mathcal{L}_{\text{edges}}$
$\zeta(2)$	188	10.44	791	46.52
$\zeta(4)$	220	12.22	855	50.29
$\zeta(5)$	236	13.11	887	52.17
$\zeta(8)$	284	15.77	983	57.82

Table 5.9: DAG node and edge growth of RCCT_MMM

As $\zeta(c)$ increases, the number of DAG nodes grows from 188 for $\zeta(2)$ up to 284 for $\zeta(8)$, with the corresponding number of edges ranging from 791 to 983. This expansion is moderate compared to SSP, with node growth factors in RCCT between 10.44 and 15.77 and edge growth factors between 46.52 and 57.82. The moderate growth explains why WCRT remains relatively stable: the additional parallelism is counterbalanced by higher memory contention.

The RCCT policy demonstrates that channel tiling is beneficial for exposing parallelism. Applying RCCT on layers with fewer filters reduces redundancy while still achieving meaningful speedup.

5.8 Testing Previous CPLs Using DF in the MLP

The DF policy is applied at the MLP stage in order to reduce the number of neurons in dense layers. Recall that $\iota(g)$ is the operator of the DF policy, which reduces a chosen

dense layer to only g neurons by regrouping them. It is preferable to choose a dense layer with a large number of neurons so that the fusion produces meaningful compression. In our network, we have two dense layers with 64 and 10 neurons respectively, and the policy is applied only to the first dense layer of 64 neurons. Table 5.10 shows the headline performance metrics when using a moderate regrouping factor $\iota(2)$.

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM	56011956	$\times 1.07$	100	0	64.41	35.16	0.44
MLWM	7041797	$\times 8.51$	84.72	15.28	63.68	34.76	1.56
SSP $\lambda(4, 4)$	616127	$\times 97.29$	46.97	53.03	63.79	33.76	2.45
SSP $\lambda(4, 5)$	883943	$\times 67.81$	32.95	67.05	63.79	33.52	2.69

Table 5.10: Headline performance metrics per policy after applying DF policy with $\iota(2)$

Compared to their original CPL counterparts, all policies benefit from speedup: the MMM baseline improves slightly $\times 1.07$, while MLWM achieves $\times 8.51$. The best improvements are seen for SSP with $\lambda(4, 4)$ and $\lambda(4, 5)$, where the speedups reach $\times 97.29$ and $\times 67.81$ respectively. However, these extreme values come at the cost of very high interference shares: more than 53% for $\lambda(4, 4)$ and 67% for $\lambda(4, 5)$.

The structural growth shown in Table 5.11 confirms this behavior.

Policies	$\mathcal{C}_{\text{CPL_DF}}$	$\mathcal{C}_{\text{nodes}}$	$\mathcal{L}_{\text{CPL_DF}}$	$\mathcal{L}_{\text{edges}}$
MMM	20	1.11	20	1.17
MLWM	158	8.77	746	43.88
SSP $\lambda(4, 4)$	2318	128.77	11,786	693.29
SSP $\lambda(4, 5)$	2894	160.77	14,730	866.47

Table 5.11: DAG node and edge growth per policy after applying DF policy with $\iota(2)$

The MMM barely changes, with only 20 nodes and 20 edges. MLWM grows moderately to 158 nodes and 746 edges. But SSP configurations explode in size, reaching over 2,300 nodes and 11,700 edges for $\lambda(4, 4)$, and close to 2,900 nodes and 14,700 edges for $\lambda(4, 5)$. Such expansion introduces considerable overhead and explains the strong increase in interference.

When the fusion factor is increased to $\iota(16)$, the impact is even more pronounced as it is shown in Table 5.12.

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM	52569076	$\times 1.14$	100	0	63.81	34.83	1.36
MLWM	3598521	$\times 16.65$	70.08	29.92	63.10	34.44	2.46
SSP $\lambda(4, 4)$	541230	$\times 110.76$	53.47	46.53	63.30	33.50	3.20
SSP $\lambda(4, 5)$	473928	$\times 126.49$	25.73	74.27	63.32	33.27	3.42

Table 5.12: Headline performance metrics per policy after applying DF policy with $\iota(16)$

The MMM baseline now reaches $\times 1.14$ speedup, while MLWM improves to $\times 16.65$. The SSP configurations reach extreme values, with $\times 110.76$ for $\lambda(4, 4)$ and $\times 126.49$ for $\lambda(4, 5)$. However, this acceleration is deceptive: interference dominates in these cases up to 46.53% and 74.27%, respectively.

The structural analysis in Table 5.13 illustrates this again.

Policies	$\mathcal{C}_{\text{CPL_DF}}$	$\mathcal{C}_{\text{nodes}}$	$\mathcal{L}_{\text{CPL_DF}}$	$\mathcal{L}_{\text{edges}}$
MMM	48	2.66	76	4.47
MLWM	186	10.33	802	47.17
SSP $\lambda(4, 4)$	2346	130.33	11,842	696.58
SSP $\lambda(4, 5)$	2922	162.33	14,786	869.76

Table 5.13: DAG node and edge growth per policy after applying DF policy with $\iota(16)$

The MMM and MLWM show limited additional growth, but SSP scales explosively: more than 2,300 nodes and 11,800 edges for $\lambda(4, 4)$, and nearly 3,000 nodes and 14,800 edges for $\lambda(4, 5)$. This exponential growth results in significant overhead.

Overall, applying DF to dense layers with many neurons significantly boosts parallelism but also introduces severe overheads when combined with aggressive CPL policies like SSP. The results suggest that DF is most effective when applied moderately, *e.g.*, $\iota(2)$ on MLWM, where it generates high speedup without collapsing efficiency. Over-aggressive fusion factors, *e.g.*, $\iota(16)$, especially when combined with fine partitioning, *e.g.*, SSP, can cause huge interference and structural blow-up that degrade practical performance despite theoretical gains.

It is important to note that continuously increasing the DF operator $\iota(g)$ conceptually drives the system toward the ideal/perfect case where all dense neurons are fully disjoint, corresponding to the MDM method. In this theoretical scenario, redundancy in neuron processing is maximized, and the DAG expands to its most inflated form for dense layers. As a result, further increasing $\iota(g)$ systematically reduces the WCRT as each neuron is computed individually. However, while this trend indicates convergence toward an ideal state, it must be balanced against the structural expansion and interference overhead observed in practice when $\iota(g)$ grows excessively.

5.9 Testing with Input Tensor Variation Using MMM in the MLP

This experiment evaluates the effect of varying the input tensor size (input image size) while keeping the MLP stage fixed to **MMM**. Two configurations are compared: an input tensor of $84 \times 84 \times 3$ shown in Table 5.14 and a larger tensor of $104 \times 104 \times 3$ shown in Table 5.15. Since the structural form of the DAG does not change, we omit explicit DAG expansion tables here and focus on the runtime metrics.

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM	104048052	$\times 1$	100	0	64.39	35.23	0.37
MLWM	19102173	$\times 5.44$	90.47	9.53	63.69	34.85	1.46
SSP $\lambda(4, 4)$	15096143	$\times 6.89$	94.45	5.55	63.90	34.13	1.97
SSP $\lambda(4, 5)$	15222196	$\times 6.83$	93.64	6.36	63.93	33.95	2.12
RCCT $\zeta(4)$	18042601	$\times 5.76$	89.87	10.13	54.95	43.12	1.93
RCCT $\zeta(5)$	18053053	$\times 5.76$	89.85	10.15	54.83	43.23	1.94

Table 5.14: Headline performance metrics per policy for an input tensor of size $84 \times 84 \times 3$

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM	160231092	$\times 1$	100	0	64.36	35.27	0.37
MLWM	29473753	$\times 5.43$	90.60	9.40	63.66	34.89	1.45
SSP $\lambda(4, 4)$	23350813	$\times 6.86$	94.36	5.64	63.90	34.33	1.77
SSP $\lambda(4, 5)$	23286910	$\times 6.88$	94.56	5.44	63.95	34.19	1.87
SSP $\lambda(5, 4)$	23281778	$\times 6.88$	94.54	5.46	63.95	34.19	1.87
SSP $\lambda(5, 5)$	23478468	$\times 6.82$	93.93	6.07	63.99	34.05	1.97
RCCT $\zeta(4)$	27843821	$\times 5.75$	90.00	10.00	54.93	43.16	1.91
RCCT $\zeta(5)$	27859913	$\times 5.75$	89.99	10.01	54.81	43.28	1.92

Table 5.15: Headline performance metrics per policy for an input tensor of size $104 \times 104 \times 3$

The results show that increasing the input size systematically increases the **WCRT** across all policies. For instance, in the MMM baseline, WCRT rises from 1.04×10^8 cycles (208.09 ms) to 1.60×10^8 cycles (320.46 ms), while MLWM increases from 1.91×10^7 (38.20 ms) to 2.95×10^7 cycles (58.94 ms). Similarly, SSP configurations also scale upward, with $\lambda(4, 4)$ moving from 1.51×10^7 (30.19 ms) to 2.33×10^7 cycles (46.70 ms). RCCT policies show a comparable trend, with $\zeta(4)$ rising from 1.80×10^7 (36.08 ms) to 2.78×10^7 cycles (55.68 ms).

Interestingly, the **relative speedup factors** remain remarkably stable across input sizes. For the 84×84 case, SSP with $\lambda(4, 4)$ achieves $\times 6.89$, while for the 104×104 case

it achieves $\times 6.86$. Similar stability is observed in MLWM ($\times 5.44$ vs. $\times 5.43$) and RCCT ($\times 5.76$ vs. $\times 5.75$). This indicates that the policies scale proportionally with input size: larger inputs generate more computational work, but the partitioning policies preserve their relative advantage over the MMM baseline.

The interference behavior is relatively stable: MLWM maintains around 9 - 10% interference, SSP stays between 5% - 6%, and RCCT close to 10%. Usage ratios (UComm, UMem, UXfer) likewise remain stable, demonstrating that scaling the input size increases the absolute workload but does not fundamentally alter the balance between compute, memory, and transfers.

This analysis confirms that input tensor size primarily affects the absolute runtime cost WCRT, but does not change the relative performance ranking of the different CPL policies. SSP remains the best-performing approach, MLWM provides moderate improvements, and RCCT shows a clear memory-dominated profile.

5.10 Testing with Convolution Hyper-Parameters Variation Using MMM in the MLP

In this section, we evaluate how variations in convolutional hyper-parameters affect the performance of all CPL policies within the fixed MMM in MLP zone. Two key parameters are considered in convolution layers: the number of filters and the kernel size.

Filters determine the depth of the output feature maps, directly influencing the computational load, while kernel size controls the spatial extent of each convolution, thereby affecting both computation and memory traffic. The following subsections analyze these two factors separately to highlight their respective impact on WCRT, interference, and overhead.

5.10.1 Filters Variation

This experiment studies the impact of increasing the number of convolution filters on the runtime metrics of the different CPL policies. Two filter configurations are considered: (19, 35) filters in Conv1 and Conv2 (Tables 5.16–5.18), and (27, 37) filters in Conv1 and Conv2 (Tables 5.17–5.19).

We begin by analyzing the performance metrics presented in Tables 5.16 and 5.17.

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM	70505770	$\times 1$	100	0	63.73	35.92	0.35
MLWM	12145392	$\times 5.80$	88.49	11.51	62.94	35.47	1.58
SSP $\lambda(4, 4)$	9902512	$\times 7.11$	90.57	9.43	62.92	34.56	2.53
SSP $\lambda(4, 5)$	10045455	$\times 7.01$	89.25	10.75	62.91	34.34	2.76
RCCT $\zeta(4)$	11432240	$\times 6.16$	87.71	12.29	53.05	44.82	2.13
RCCT $\zeta(5)$	11441420	$\times 6.16$	87.67	12.33	52.95	44.91	2.14

Table 5.16: Headline performance metrics per policy for convolutional layers with 19 filters in Conv1 and 35 filters in Conv2

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM	77901352	$\times 1$	100	0	63.29	36.37	0.34
MLWM	12943468	$\times 6.01$	87.39	12.61	62.47	35.90	1.63
SSP $\lambda(4, 4)$	10684392	$\times 7.29$	88.83	11.17	62.38	35.03	2.59
SSP $\lambda(4, 5)$	10714881	$\times 7.27$	88.86	11.14	62.38	34.84	2.78
RCCT $\zeta(4)$	12162265	$\times 6.40$	86.50	13.50	51.85	45.94	2.21
RCCT $\zeta(5)$	12168937	$\times 6.40$	86.48	13.52	51.77	46.01	2.22

Table 5.17: Headline performance metrics per policy for convolutional layers with 27 filters in Conv1 and 37 filters in Conv2

The results show that increasing the number of filters systematically increases the **WCRT** across all policies. For example, in the MMM baseline, WCRT rises from 7.05×10^7 (141.01 ms) to 7.79×10^7 cycles (155.80 ms), while MLWM grows from 1.21×10^7 (24.29 ms) to 1.29×10^7 (25.88 ms). SSP also increases, with $\lambda(4, 4)$ moving from 9.90×10^6 (19.80 ms) to 1.07×10^7 (21.36 ms), and RCCT with $\zeta(4)$ increasing from 1.14×10^7 (22.86 ms) to 1.22×10^7 (24.32 ms). Despite this increase in absolute cost, the **relative speedups** remain stable, with SSP consistently outperforming the other policies ($\times 7.11$ vs. $\times 7.29$ for $\lambda(4, 4)$).

In terms of interference, MLWM and RCCT show slightly higher shares as filters increase ($\sim 11 - 13\%$), while SSP remains in the 9 - 11% range, showing that spatial partitioning continues to balance computation against interference. The usage ratios highlight the same pattern as before: MLWM and SSP are compute-dominated, while RCCT shifts workload toward memory, with UMem exceeding 45% in the larger filter configuration.

The structural expansion reported in Table 5.18 and Table 5.19 confirms the growth trend analyzed previously.

Policies	$\mathcal{C}_{\text{CPL_MMM}}$	$\mathcal{C}_{\text{nodes}}$	$\mathcal{L}_{\text{CPL_MMM}}$	$\mathcal{L}_{\text{edges}}$
MMM	18	1	17	1
MLWM	174	9.66	923	54.29
SSP $\lambda(4, 4)$	2604	144.66	14,663	862.52
SSP $\lambda(4, 5)$	3252	180.66	18,327	1078.05
RCCT $\zeta(4)$	250	13.88	1,056	62.11
RCCT $\zeta(5)$	269	14.94	1,094	64.35

Table 5.18: DAG node and edge growth per policy for convolutional layers with 19 filters in Conv1 and 35 filters in Conv2

Policies	$\mathcal{C}_{\text{CPL_MMM}}$	$\mathcal{C}_{\text{nodes}}$	$\mathcal{L}_{\text{CPL_MMM}}$	$\mathcal{L}_{\text{edges}}$
MMM	18	1	17	1
MLWM	186	10.33	1,053	61.94
SSP $\lambda(4, 4)$	2796	155.33	16,743	984.88
SSP $\lambda(4, 5)$	3492	194.00	20,927	1231.00
RCCT $\zeta(4)$	270	15.00	1,200	70.58
RCCT $\zeta(5)$	291	16.16	1,242	73.05

Table 5.19: DAG node and edge growth per policy for convolutional layers with 27 filters in Conv1 and 37 filters in Conv2

The MLWM policy shows a modest increase when filters grow: from 174 to 186 nodes and from 923 to 1,053 edges. This corresponds to a node growth factor $\mathcal{C}_{\text{nodes}}$ rising slightly from 9.66 to 10.33, and an edge growth factor $\mathcal{L}_{\text{edges}}$ from 54.29 to 61.94.

In contrast, in SSP policy, for $\lambda(4, 4)$, nodes increase from 2,604 to 2,796 and edges from 14,663 to 16,743, leading to growth factors of $\mathcal{C}_{\text{nodes}} = 144.66 \rightarrow 155.33$ and $\mathcal{L}_{\text{edges}} = 862.52 \rightarrow 984.88$. The effect is even more pronounced for $\lambda(4, 5)$, where nodes expand from 3,252 to 3,492 and edges from 18,327 to 20,927, giving factors of $180.66 \rightarrow 194.00$ and $1078.05 \rightarrow 1231.00$. This sharp increase highlights how spatial partitioning strongly amplifies DAG complexity under higher filter counts.

RCCT remains much more moderate in its expansion. The number of nodes grows from 250 to 291 and edges from 1,056 to 1,242, which translates into factors of $\mathcal{C}_{\text{nodes}} = 13.88 \rightarrow 16.16$ and $\mathcal{L}_{\text{edges}} = 62.11 \rightarrow 73.05$.

Overall, the comparison shows that filter scaling has a moderate impact on MLWM and RCCT, but results in explosive structural growth for SSP, with higher node and edge factors between the two filter configurations. We have observed that filter variation scales the runtime cost proportionally for all CPL policies. SSP continues to deliver the best speedup, though at the price of strong structural expansion, while RCCT exhibits higher memory demand and MLWM provides a balanced middle ground.

5.10.2 Kernels Variation

This experiment analyzes the effect of changing the convolution kernel size. Since kernel size does not affect the graph topology, the DAG representation remains unchanged, and only runtime metrics are reported. Two cases are considered: kernels of size 6×6 in Table 5.20 and 9×9 in Table 5.21.

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM	120124084	$\times 1$	100	0	56.30	43.54	0.16
MLWM	13024437	$\times 9.22$	91.74	8.26	56.02	43.32	0.66
SSP $\lambda(4, 4)$	9204554	$\times 13.05$	94.85	5.15	56.15	43.01	0.84
SSP $\lambda(4, 5)$	9330418	$\times 12.87$	94.01	5.99	56.16	42.94	0.90
RCCT $\zeta(4)$	10529955	$\times 11.40$	89.76	10.24	34.78	64.19	1.04
RCCT $\zeta(5)$	10535967	$\times 11.40$	89.75	10.25	35.24	63.73	1.03

Table 5.20: Headline performance metrics per policy for kernels of size 6×6

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM	220417204	$\times 1$	100	0	53.24	46.67	0.08
MLWM	16435077	$\times 13.41$	93.46	6.54	53.11	46.55	0.34
SSP $\lambda(4, 4)$	10339353	$\times 21.31$	95.68	4.32	53.17	46.47	0.36
SSP $\lambda(4, 5)$	10520900	$\times 20.95$	96.04	3.96	53.18	46.45	0.37
RCCT $\zeta(4)$	10803577	$\times 20.40$	90.02	9.98	24.90	74.52	0.58
RCCT $\zeta(5)$	10809589	$\times 20.39$	90.01	9.99	25.87	73.56	0.57

Table 5.21: Headline performance metrics per policy for kernels of size 9×9

The results clearly show that increasing kernel size substantially increases the **WCRT** for the MMM baseline, rising from 1.20×10^8 cycles (240.24 ms) for 6×6 kernels to 2.20×10^8 cycles (440.83 ms) for 9×9 kernels. This is expected, since larger kernels increase the number of MAC operations per output element if the bias term is enabled (as in our case). All other policies scale accordingly, but the relative **speedup factors** remain high and even improve with larger kernels. For instance, MLWM improves from $\times 9.22$ to $\times 13.41$, SSP with $\lambda(4, 4)$ jumps from $\times 13.05$ to $\times 21.31$, and RCCT with $\zeta(4)$ nearly doubles from $\times 11.40$ to $\times 20.40$. This shows that partitioning policies benefit more from increased kernel size, since the additional compute work allows better exploitation of parallelism.

The interference shares remain low for SSP (3 - 6%) and moderate for MLWM (6 - 8%), while RCCT continues to exhibit higher interference ($\sim 10\%$). The workload distribution shifts slightly: MMM and SSP remain compute-dominated, while RCCT is strongly memory-dominated, with UMem rising to more than 74% in the 9×9 case.

We conclude that larger kernels increase the absolute runtime cost but also increase the relative advantage of fine-grained policies such as SSP. The SSP $\lambda(4, 4)$ and $\lambda(4, 5)$ configurations deliver the highest speedups with stable interference levels, making them the most effective choices under kernel scaling. RCCT maintains strong speedup as well but remains constrained by memory overhead, while MLWM provides balanced improvements.

5.11 Projection on Coolidge Architecture

The projection on the Coolidge architecture allows us to compare the impact of hardware differences with respect to the Bostan architecture. Having finished the analysis on Bostan, we now extend our study to Coolidge in order to observe the architectural effect of a more optimized architecture, wider buses, and the presence of the TCA co-processor. In particular, we separate our analysis into two modes: (i) FP32 computations where the TCA is excluded and computation proceeds on the general cores, and (ii) FP16 computations where the TCA is activated. The DAG structure does not change between Bostan and Coolidge, so we will focus only on runtime metrics and relative speedups by comparing WCRT.

5.11.1 Using FP32 Data (TCA Exclusion)

In the FP32 data precision case, where the TCA is excluded, in Table 5.22, the MMM_MMM baseline achieves a WCRT of 4.39×10^7 cycles (48.82 ms), setting the reference point. The MLWM and RCCT policies both improve runtime significantly relative to MMM, reaching speedups of $\times 3.14$ and $\times 3.09$ respectively. SSP, however, performs worse on Coolidge compared to its strong results on Bostan, reaching only $\times 2.22$ speedup, with interference dominating at 57.09%. The DF $\iota(2)$ case, shows the lowest WCRT at 5.72×10^6 cycles (6.35 ms) and an impressive $\times 7.68$ local speedup, but it comes with a prohibitive 94.77% interference.

The usage ratios show that computation dominates across all policies. For MMM, the compute usage UComm reaches 72.66%, with memory at 27.05% and transfers negligible at 0.29%. The same trend is observed in MLWM and RCCT, where UComm remains around 72% and UMem around 27%. SSP introduces slightly higher transfer usage (2.07%), reflecting its finer partitioning and additional communication edges, while DF with $\iota(2)$ further raises transfer usage to 2.17%. Although these values remain small, they reveal that as policies become more fine-grained, transfers begin to degrade the compute/memory balance. In general, workloads are largely compute-bound with a relatively stable distribution across policies.

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM_MMM	43939995	$\times 1$	100	0	72.66	27.05	0.29
MLWM_MMM	13989555	$\times 3.14$	65.97	34.03	72.01	26.82	1.17
SSP $\lambda(4, 4)$ _MMM	19761660	$\times 2.22$	42.91	57.09	71.88	26.05	2.07
RCCT $\zeta(4)$ _MMM	14177770	$\times 3.09$	64.62	35.38	71.70	27.05	1.26
SSP $\lambda(4, 4)$ _DF $\iota(2)$	5720324	$\times 7.68$	5.23	94.77	73.87	23.96	2.17

Table 5.22: Headline performance metrics per policy for Coolidge architecture

Comparing across architectures in Table 5.23, Coolidge in FP32 improves the baseline MMM by $\times 1.36$ relative to Bostan, but MLWM, RCCT, and SSP are all slower than on Bostan, with relative speedups below $\times 1$. This shows that Coolidge without the TCA struggles to exploit fine-grained policies: while absolute runtimes decrease, the interference overhead is significantly higher, leading to worse relative performance than on Bostan. The limited number of cores in Coolidge compared to Bostan may contribute to this effect.

Moreover, focusing only on WCET values without considering clock frequency differences, since Coolidge operates at a higher clock frequency than Bostan. In this case, however, when measured purely in cycles, Coolidge performs worse than Bostan.

Bostan/Coolidge	Speedup
MMM_MMM	$\times 1.36$
MLWM_MMM	$\times 0.78$
SSP $\lambda(4,4)$ _MMM	$\times 0.44$
RCCT $\zeta(4)$ _MMM	$\times 0.73$
SSP $\lambda(4,4)$ _DF $_{\ell}(2)$	$\times 0.10$

Table 5.23: Speedup metric per policy comparing between Coolidge and Bostan architectures

5.11.2 Using FP16 Data (TCA Inclusion)

For FP16 data precision with the TCA included, in Table 5.24, the benefit of hardware acceleration becomes clear. The MMM_MMM baseline drops to 1.77×10^7 cycles (19.71 ms), giving a $\times 3.37$ speedup relative to Bostan (in Table 5.25) and a $\times 2.47$ speedup compared to Coolidge on FP32 (in Table 5.26). All other policies benefit even more: MLWM achieves a $\times 4.75$ speedup locally and is $\times 2.93$ faster than on Bostan, while RCCT shows similar gains with $\times 4.58$ local speedup and $\times 2.67$ improvement over Bostan. SSP becomes highly accelerated under the TCA, reaching $\times 4.73$ local speedup and $\times 5.27$ relative to Coolidge on FP32, though at the cost of extreme interference (79.75%). The DF case achieves the smallest WCRT overall at 2.66×10^6 cycles (2.95 ms) and $\times 6.66$ local speedup, but with almost all execution time lost to interference (95.85%).

Meanwhile, the balance shifts significantly toward computation. The MMM baseline reaches UComm = 82.43%, while UMem drops to 17.17% and transfers remain marginal at 0.40%. This compute dominance becomes even more pronounced in MLWM and RCCT, where UComm exceeds 91% and memory usage falls below 7%. SSP and DF show the highest transfer costs, with UXfer around 4.6 - 4.7%, reflecting the stress of fine-grained partitioning under accelerated compute. These values indicate that the TCA shifts the workload balance strongly toward computation, while memory and transfers become relatively less significant, though interference grows substantially due to the faster execution pace.

Policies	WCRT	Speedup	WCET%	Intf%	UComm%	UMem%	UXfer%
MMM_MMM	17745490	$\times 1$	100	0	82.43	17.17	0.40
MLWM_MMM	3735875	$\times 4.75$	31.60	68.40	91.80	6.23	1.97
SSP $\lambda(4,4)$ _MMM	3749103	$\times 4.73$	20.25	79.75	90.12	5.24	4.64
RCCT $\zeta(4)$ _MMM	3872057	$\times 4.58$	30.54	69.46	91.53	6.35	2.12
SSP $\lambda(4,4)$ _DF $_{\ell}(2)$	2660811	$\times 6.66$	4.15	95.85	90.06	5.24	4.71

Table 5.24: Headline performance metrics per policy for Coolidge architecture with the TCA co-processor

A comparison of Tables 5.22 and 5.24 highlights the architectural shift induced by

hardware acceleration. Without TCA, compute usage stabilizes near 72% for most policies, leaving around 27% for memory, while transfers rarely exceed 2%. With TCA enabled, UComm jumps to 82 - 92%, confirming that the co-processor massively boosts compute throughput. Memory usage correspondingly drops to below 7% in MLWM and RCCT, and transfer usage rises modestly to 2 - 5% in the fine-grained SSP and DF cases. This shows that Coolidge with TCA reorients the balance sharply toward compute-bound execution, but at the same time, it amplifies the visibility of communication overheads in more partitioned policies.

Additionally, Tables 5.25 and 5.26 highlight a consistent pattern: Coolidge with TCA clearly outperforms both Bostan and Coolidge without TCA across all policies,

Bostan/Coolidge&TCA	Speedup
MMM_MMM	$\times 3.37$
MLWM_MMM	$\times 2.93$
SSP $\lambda(4, 4)$ _MMM	$\times 2.35$
RCCT $\zeta(4)$ _MMM	$\times 2.67$
SSP $\lambda(4, 4)$ _DF $\iota(2)$	$\times 0.23$

Table 5.25: Speedup metric per policy comparing between Coolidge & TCA and Bostan architectures

Coolidge/Coolidge&TCA	Speedup
MMM_MMM	$\times 2.47$
MLWM_MMM	$\times 3.74$
SSP $\lambda(4, 4)$ _MMM	$\times 5.27$
RCCT $\zeta(4)$ _MMM	$\times 3.66$
SSP $\lambda(4, 4)$ _DF $\iota(2)$	$\times 2.14$

Table 5.26: Speedup metric per policy comparing between Coolidge & TCA and Coolidge architectures

This whole comparison confirms that the TCA unlocks considerable speedups, but the higher interference observed indicates that memory and transfers become the new bottlenecks under accelerated compute.

5.12 Evaluation and Discussion

In this section, we synthesize the results of all experiments and confront them with our evaluation framework. The objective is not only to present numerical outcomes, but also to judge each regrouping policy against a set of clearly defined criteria.

To compare our regrouping policies rigorously, we adopt those following success criteria ordered by priority:

1. **C1; Low WCRT:** We prioritize reduced end-to-end latency in cycles.

2. **C2; Reduced Interference and Memory Load:** We prioritize configurations that minimize both Intf% and UMem% as much as possible. Otherwise, the configuration is considered risky
3. **C3; Controlled Structural Growth:** We recommend a smaller $\mathcal{C}_{\text{nodes}}$ and $\mathcal{L}_{\text{edges}}$ to keep memory and communication overheads in check.

We report per-policy outcomes against these criteria and recommend the most balanced choices per architecture/precision.

5.12.1 Analysing in Bostan

On Bostan, SSP with $\lambda(4, 4)$ achieves the lowest WCRT and a low percentage of interference (Table 5.6), thus satisfying **C1** and **C2**. However, it violates **C3**, since it reaches a massive DAG growth (nodes: $\mathcal{C}_{\text{SSP_MMM}} = 2316$, factor: $\mathcal{C}_{\text{nodes}} = 128.66$; edges: $\mathcal{L}_{\text{SSP_MMM}} = 11,783$, factor: $\mathcal{L}_{\text{edges}} = 693.11$). In contrast, MLWM achieves a slightly higher WCRT (+24% above SSP) but provides a much safer balance: interference is kept under 10% and structural expansion remains small ($\mathcal{C}_{\text{nodes}} = 156$), thereby satisfying **C2** and **C3**. RCCT achieves WCRT close to MLWM, but it becomes memory-dominated, with UMem% significantly higher, making it less preferable under **C2**. DF on top of SSP drives WCRT to extreme lows (satisfying **C1**) but fails both **C2** (interference saturation) and **C3** (structural increase). Accordingly:

- **Lowest WCRT (C1):** SSP $\lambda(4, 4)$ + DF
- **Best Balance (C2 & C3 with acceptable C1):** MLWM

Furthermore, sensitivity analyses of varying the CNN network parameters and inputs highlight the following:

Input size. Increasing input resolution raises WCRT almost proportionally across all policies (**C1** impacted), but their relative speedup ordering remains unchanged. Interference and usage ratios remain stable, so **C2** and **C3** are not strongly affected.

Filters. Adding more filters increases WCRT for all policies (**C1** impacted), with SSP showing bigger structural growth in both $\mathcal{C}_{\text{nodes}}$ and $\mathcal{L}_{\text{edges}}$ factors, thereby violating **C3**. In contrast, MLWM and RCCT scale more modestly and maintain their balance under **C2–C3**, while preserving similar speedups.

Kernel size. Larger kernels increase WCRT due to higher MAC counts (**C1** penalty), but fine-grained policies benefit in relative speedup. SSP keeps interference low (satisfying **C2**) despite structural costs, while RCCT remains memory-dominated, with high UMem% violating **C2**.

5.12.2 Analysing in Coolidge & Coolidge+TCA

On Coolidge with FP32 (no TCA), the MMM baseline improves compared to Bostan, but fine-grained policies (SSP, DF) occur very high interference, violating **C2**. MLWM remains the most balanced: it provides meaningful speedup while keeping interference and structural growth moderate, thus meeting **C2** and **C3**.

With FP16+TCA enabled, absolute runtimes drop substantially for all policies, satisfying **C1**. However, the acceleration amplifies interference in SSP and DF, breaking **C2**. Once again, MLWM delivers the best compromise: it keeps interference lower than SSP,

while its DAG growth is modest, therefore fulfilling both **C2** and **C3**. **DF+SSP** produces the smallest WCRT numerically but fails efficiency due to excessive interference and structural expenses (**C2** and **C3** violated). Accordingly:

- **Lowest WCRT (C1):** SSP $\lambda(4, 4)$ + DF
- **Best Balance (C2 & C3 with acceptable C1):** MLWM

5.12.3 Policy Recommendation

Table 5.27 summarizes how each policy behaves against the success criteria. We define the indicators as follows: $\checkmark$ denotes compliance, $\sim$ an acceptable but not optimal result, and $\times$ a violation.

Policies	C1	C2	C3
MMM	$\times$	$\checkmark$	$\checkmark$
MLWM	$\sim$	$\checkmark$	$\checkmark$
SSP $\lambda(4, 4)$	$\checkmark$	$\checkmark$	$\times$
RCCT $\zeta(4)$	$\sim$	$\times$	$\checkmark$
SSP $\lambda(4, 4)$ + DF $\iota(2)$	$\checkmark$	$\times$	$\times$

Table 5.27: Policy compliance across success criteria

Interpretation. MMM is safe but slow. MLWM achieves a balanced profile, meeting C2 and C3 with near-acceptable C1. SSP $\lambda(4, 4)$ minimizes WCRT but grows the DAG excessively. RCCT tiling adds memory pressure, failing C2. SSP+DF generates the smallest WCRT but collapses under interference and growth.

Globally, MLWM is the most robust choice, while SSP is preferable only when minimizing WCRT is the overriding requirement, even at the expense of other factors.

5.12.4 Additional Practical Considerations

Beyond the evaluation itself, it is important to note some practical aspects of our project toolchain.

Our **MIND** software converts CNN to DAGs in `.yaml` file format and produces reports within seconds, even for large CNNs, and supports a wide range of layer types and parameterizations.

In contrast, **MIA** is a single-threaded program: for a small model (like our CNN example) it can take up to ~ 40 minutes to execute under highly partitioned SSP, and for large models, *e.g.*, *VGG16*, at least ~ 5 days. Therefore, we used a small CNN for a reason to keep the analysis time practical.

5.13 Conclusion

This chapter established a methodology to evaluate regrouping policies for CNN inference on Kalray many-core architectures. We formalized performance/structural metrics,

applied them to a prototype CNN, and conducted sensitivity studies (input size, filter count, kernel size) alongside a cross-architecture projection from Bostan to Coolidge with and without TCA.

The results reveal a consistent tension between raw speed and operational compromises. On Bostan, **SSP** $\lambda(4, 4)$ reaches the smallest WCRT, but **MLWM** provides the best overall trade-off by maintaining low interference and limited structural growth. On Coolidge, FP16+TCA dramatically reduces absolute WCRT for all policies; however, interference rises sharply for fine-grained DAGs. In this regime, **MLWM** again emerges as the most balanced option, whereas aggressive **SSP+DF** reduces WCRT further but violates the interference and structural expenses criteria.

We recommend **MLWM as the default policy** for robust deployment across architectures and precisions, while **SSP+DF** can be selectively used when absolute WCRT minimization is mandatory and the system can tolerate higher interference and analysis cost.

General Conclusion

Conclusion

Across design, modeling, and toolchain integration, our project delivers an end-to-end methodology, from CNN parsing to architecture-aware timing analysis, together with software deliverable that makes the process reproducible and extensible.

With respect to the objectives outlined in the General Introduction chapter of this report, Chapter , this research successfully accomplished all its intended goals. In particular, we solved three major challenges:

1. **Conversion of CNNs to DAGs:** We developed a methodology to systematically transform convolutional neural networks into directed acyclic graphs using our MIND software, and we proposed multiple regrouping policies to represent different partitioning strategies (MLWM, SSP, RCCT, and DF).
2. **Integration of real-time constraints:** We defined and formalized real-time system constraints, *e.g.*, WCET in isolation, WCMA, etc., for each policy and integrated them into the DAG representation.
3. **Unified YAML generation:** We introduce an automatic generation, within MIND, of a unified YAML specification capturing the DAG structure together with all associated RTS constraints. This artifact serves as input to MIA for WCRT calculation and for evaluation under criteria we formalized.

Taken together, these steps demonstrate a systematic, automated bridge between CNN modeling and DAG-based real-time analysis. All evaluations were conducted on Kalray many-core architectures, Bostan and Coolidge, with and without TCA.

Despite these achievements, the project also has notable limitations. First, the workflow relies heavily on the MIA tool, which itself has constraints. In particular, the current version of MIA executes in a single-threaded manner and is not optimized for handling very large YAML files, leading to long analysis times for huge CNN models, *e.g.*, up to 5–6 days for complex models. As a result, the present version of MIA tool is more suited to lightweight models. Moreover, several functionalities planned for MIA, which are essential for more detailed analyses, are not yet implemented. Once MIA is updated, our methodology can easily be extended accordingly.

Second, the WCET and WCMA modeling in this work relies primarily on theoretical estimations, constrained by the limited public documentation of the Kalray architectures. In particular, many details of the Kalray MPPA architectures remain confidential. Consequently, the accuracy of our results is bounded by the assumptions we were able to make. To refine the numerical precision of our models, real hardware experimentation on Kalray platforms, or access to detailed simulation tools and technical documentation, will be indispensable.

Third, we observed that the availability of references for this research is limited, as the topic represents a relatively new subject and a novel perspective. At the beginning of this project, this lack of literature posed a significant challenge in establishing a comprehensive state of the art. Consequently, the limited number of references included here primarily serves to provide essential contextual support. Most of the work presented in our project report is therefore the result of our original reflection and development, rather than adaptation from existing sources, which are not yet available.

Perspectives & Future Work

Throughout this report, we have mentioned several possible directions for future work within the context of individual chapters. In this section, we recapitulate the most relevant of these, while also adding some additional perspectives that were not previously cited. The potential future directions include:

- Varying CNN network parameters.
- Varying regrouping-policy parameters to precisely characterize their effects.
- Precise modeling of real-time system constraints on Kalray architectures.
- Exploring other analysis modes in MIA.
- Extending MIND to support additional modern CNN layers.

References

- [1] X. Zhao, L. Wang, Y. Zhang, X. Han, M. Deveci, and M. Parmar, “A review of convolutional neural networks in computer vision,” *Artificial Intelligence Review*, vol. 57, no. 4, p. 99, 2024.
- [2] I. D. A. Silva, “Implantation certifiable et efficace de réseaux de neurones sur des systèmes embarqués temps-réel critiques,” Ph.D. dissertation, Toulouse, ISAE, 2024.
- [3] S. Lounnaci, D. Bendouda, P.-E. Hladik, and C. Maiza, “Analysis of convolutional neural networks on real-time multi-core embedded platforms,” in *Proceedings of the 1st International Conference on Artificial Intelligence and Cyber Physical Systems (AICPS)*. IEEE, 2025.
- [4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [5] Kalray, “Kalray - Leading Processing Acceleration Solutions,” <https://www.kalrayinc.com/>, 2025.
- [6] B. Dupont de Dinechin, “Kalray mppa: Engineering a manycore processor platform for mission-critical applications,” Keynote slides at MCSoc 2016, Kalray SA, 2016, cTO, Kalray.
- [7] —, “Securing the kalray mppa manycore architecture,” Presentation slides, Kalray SA, Dec. 2018, cTO, Kalray.
- [8] K. Rihani, “Many-core timing analysis of real-time systems,” Ph.D. dissertation, Université Grenoble Alpes, 2017.
- [9] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, “Response time analysis of synchronous data flow programs on a many-core processor,” in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, 2016, pp. 67–76.
- [10] M. Schuh, “Implantation sûre d’applications temps-réel critiques sur plateforme pluri-coeur,” Ph.D. dissertation, Université Grenoble Alpes [2020-....], 2022.
- [11] M. D. de Dinechin, M. Schuh, M. Moy, and C. Maiza, “Scaling up the memory interference analysis for hard real-time many-core systems,” in *DATE 2020-Design, Automation and Test in Europe Conference*, 2020, pp. 1–4.
- [12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 2002.

- [13] G. G. Chrysos, Y. Wu, R. Pascanu, P. Torr, and V. Cevher, “Hadamard product in deep learning: Introduction, advances and challenges,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2025.

Appendices

Appendix A

Details of WCMA Calculation

To clarify the origin of the load and store data, we provide an example of the per-element load and store calculations in the convolution layer:

$$WCMA_{\text{per output}}^{\text{Conv}} = \text{input reads} + \text{weight reads} + \text{output write} = 2k_H \cdot k_W \cdot D_l + 1_{\text{store}}$$

Where :

- input reads: $k_H \cdot k_W \cdot D_l$
- weight reads: $k_H \cdot k_W \cdot D_l$
- output write: 1_{store}

A.1 Bostan WCMA Calculation

A.1.1 Convolution Operation

Without Bias term:

$$(WCMA_{Bostan}^{\text{Conv}})_{FP32/INT32} = \left\lceil 2k_H k_W D_l \cdot \frac{32}{64} \right\rceil + \left\lceil 1_{\text{store}} \cdot \frac{32}{64} \right\rceil = k_H k_W D_l + 1$$

$$(WCMA_{Bostan}^{\text{Conv}})_{FP16/INT16} = \left\lceil 2k_H k_W D_l \cdot \frac{16}{64} \right\rceil + 1 = \left\lceil \frac{k_H k_W D_l}{2} \right\rceil + 1$$

$$(WCMA_{Bostan}^{\text{Conv}})_{INT8} = \left\lceil 2k_H k_W D_l \cdot \frac{8}{64} \right\rceil + 1 = \left\lceil \frac{k_H k_W D_l}{4} \right\rceil + 1$$

With Bias term:

$$\begin{aligned} (WCMA_{Bostan}^{\text{Conv}+\text{bias}})_{FP32/INT32} &= \left\lceil (2k_H k_W D_l + 1_{\text{bias}}) \cdot \frac{32}{64} \right\rceil + \left\lceil 1_{\text{store}} \cdot \frac{32}{64} \right\rceil \\ &= \left\lceil k_H k_W D_l + \frac{1}{2} \right\rceil + 1 = k_H k_W D_l + 2 \end{aligned}$$

$$(WCMA_{Bostan}^{\text{Conv}+\text{bias}})_{FP16/INT16} = \left\lceil (2k_H k_W D_l + 1_{\text{bias}}) \cdot \frac{16}{64} \right\rceil + 1$$

$$= \left\lceil \frac{k_H k_W D_l}{2} + \frac{1}{4} \right\rceil + 1 = \left\lceil \frac{k_H k_W D_l}{2} \right\rceil + 2$$

$$(WCMA_{Bostan}^{Conv+bias})_{INT8} = \left\lceil (2k_H k_W D_l + 1_{bias}) \cdot \frac{8}{64} \right\rceil + 1 = \left\lceil \frac{k_H k_W D_l}{4} + \frac{1}{8} \right\rceil + 1 = \left\lceil \frac{k_H k_W D_l}{4} \right\rceil + 2$$

A.1.2 Pooling Operation

$$(WCMA_{Bostan}^{Pool})_{FP32/INT32} = \left\lceil mn \cdot \frac{32}{64} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{64} \right\rceil = \left\lceil \frac{mn}{2} \right\rceil + 1$$

$$(WCMA_{Bostan}^{Pool})_{FP16/INT16} = \left\lceil mn \cdot \frac{16}{64} \right\rceil + 1 = \left\lceil \frac{mn}{4} \right\rceil + 1$$

$$(WCMA_{Bostan}^{Pool})_{INT8} = \left\lceil mn \cdot \frac{8}{64} \right\rceil + 1 = \left\lceil \frac{mn}{8} \right\rceil + 1$$

A.1.3 Dense Operation

Without Bias term:

$$(WCMA_{Bostan}^{Dense})_{FP32/INT32} = \left\lceil 2N_l \cdot \frac{32}{64} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{64} \right\rceil = N_l + 1$$

$$(WCMA_{Bostan}^{Dense})_{FP16/INT16} = \left\lceil 2N_l \cdot \frac{16}{64} \right\rceil + 1 = \left\lceil \frac{N_l}{2} \right\rceil + 1$$

$$(WCMA_{Bostan}^{Dense})_{INT8} = \left\lceil 2N_l \cdot \frac{8}{64} \right\rceil + 1 = \left\lceil \frac{N_l}{4} \right\rceil + 1$$

With Bias term:

$$(WCMA_{Bostan}^{Dense+bias})_{FP32/INT32} = \left\lceil (2N_l + 1_{bias}) \cdot \frac{32}{64} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{64} \right\rceil = \left\lceil N_l + \frac{1}{2} \right\rceil + 1 = N_l + 2$$

$$(WCMA_{Bostan}^{Dense+bias})_{FP16/INT16} = \left\lceil (2N_l + 1_{bias}) \cdot \frac{16}{64} \right\rceil + 1 = \left\lceil \frac{N_l}{2} + \frac{1}{4} \right\rceil + 1 = \left\lceil \frac{N_l}{2} \right\rceil + 2$$

$$(WCMA_{Bostan}^{Dense+bias})_{INT8} = \left\lceil (2N_l + 1_{bias}) \cdot \frac{8}{64} \right\rceil + 1 = \left\lceil \frac{N_l}{4} + \frac{1}{8} \right\rceil + 1 = \left\lceil \frac{N_l}{4} \right\rceil + 2$$

A.1.4 Softmax Operation

$$(WCMA_{Bostan}^{Softmax})_{FP32/INT32} = \left\lceil V \cdot \frac{32}{64} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{64} \right\rceil = \left\lceil \frac{V}{2} \right\rceil + 1$$

$$(WCMA_{Bostan}^{Softmax})_{FP16/INT16} = \left\lceil V \cdot \frac{16}{64} \right\rceil + 1 = \left\lceil \frac{V}{4} \right\rceil + 1$$

$$(WCMA_{Bostan}^{Softmax})_{INT8} = \left\lceil V \cdot \frac{8}{64} \right\rceil + 1 = \left\lceil \frac{V}{8} \right\rceil + 1$$

A.2 Coolidge WCMA Calculation

A.2.1 Convolution Operation

Without Bias term:

$$(WCMA_{Coolidge}^{Conv})_{FP32/INT32} = \left\lceil 2k_H k_W D_l \cdot \frac{32}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{128} \right\rceil = \left\lceil \frac{k_H k_W D_l}{2} \right\rceil + 1$$

$$(WCMA_{Coolidge}^{Conv})_{FP16/INT16} = \left\lceil 2k_H k_W D_l \cdot \frac{16}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{16}{128} \right\rceil = \left\lceil \frac{k_H k_W D_l}{4} \right\rceil + 1$$

$$(WCMA_{Coolidge}^{Conv})_{INT8} = \left\lceil 2k_H k_W D_l \cdot \frac{8}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{8}{128} \right\rceil = \left\lceil \frac{k_H k_W D_l}{8} \right\rceil + 1$$

With Bias term:

$$\begin{aligned} (WCMA_{Coolidge}^{Conv+bias})_{FP32/INT32} &= \left\lceil (2k_H k_W D_l + 1_{bias}) \cdot \frac{32}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{128} \right\rceil \\ &= \left\lceil \frac{k_H k_W D_l}{2} + \frac{1}{4} \right\rceil + 1 = \left\lceil \frac{k_H k_W D_l}{2} \right\rceil + 2 \end{aligned}$$

$$\begin{aligned} (WCMA_{Coolidge}^{Conv+bias})_{FP16/INT16} &= \left\lceil (2k_H k_W D_l + 1_{bias}) \cdot \frac{16}{128} \cdot \left(\frac{1}{128}\right)_{TCA} \right\rceil + \left\lceil 1_{store} \cdot \frac{16}{128} \cdot \left(\frac{1}{128}\right)_{TCA} \right\rceil \\ &= \left\lceil \frac{k_H k_W D_l}{2^9} \right\rceil + 2 \end{aligned}$$

$$\begin{aligned} (WCMA_{Coolidge}^{Conv+bias})_{INT8} &= \left\lceil (2k_H k_W D_l + 1_{bias}) \cdot \frac{8}{128} \cdot \left(\frac{1}{128}\right)_{TCA} \right\rceil + \left\lceil 1_{store} \cdot \frac{8}{128} \cdot \left(\frac{1}{128}\right)_{TCA} \right\rceil \\ &= \left\lceil \frac{k_H k_W D_l}{2^{10}} \right\rceil + 2 \end{aligned}$$

A.2.2 Pooling Operation

$$(WCMA_{Coolidge}^{Pool})_{FP32/INT32} = \left\lceil mn \cdot \frac{32}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{128} \right\rceil = \left\lceil \frac{mn}{4} \right\rceil + 1$$

$$(WCMA_{Coolidge}^{Pool})_{FP16/INT16} = \left\lceil mn \cdot \frac{16}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{16}{128} \right\rceil = \left\lceil \frac{mn}{8} \right\rceil + 1$$

$$(WCMA_{Coolidge}^{Pool})_{INT8} = \left\lceil mn \cdot \frac{8}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{8}{128} \right\rceil = \left\lceil \frac{mn}{16} \right\rceil + 1$$

A.2.3 Dense Operation

Without Bias term:

$$\begin{aligned}
 (WCMA_{Coolidge}^{Dense})_{FP32/INT32} &= \left\lceil 2N_l \cdot \frac{32}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{128} \right\rceil = \left\lceil \frac{N_l}{2} \right\rceil + 1 \\
 (WCMA_{Coolidge}^{Dense})_{FP16/INT16} &= \left\lceil 2N_l \cdot \frac{16}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{16}{128} \right\rceil = \left\lceil \frac{N_l}{4} \right\rceil + 1 \\
 (WCMA_{Coolidge}^{Dense})_{INT8} &= \left\lceil 2N_l \cdot \frac{8}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{8}{128} \right\rceil = \left\lceil \frac{N_l}{8} \right\rceil + 1
 \end{aligned}$$

With Bias term:

$$\begin{aligned}
 (WCMA_{Coolidge}^{Dense+bias})_{FP32/INT32} &= \left\lceil (2N_l + 1_{bias}) \cdot \frac{32}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{128} \right\rceil = \left\lceil \frac{N_l}{2} + \frac{1}{4} \right\rceil + 1 \\
 &= \left\lceil \frac{N_l}{2} \right\rceil + 2 \\
 (WCMA_{Coolidge}^{Dense+bias})_{FP16/INT16} &= \left\lceil (2N_l + 1_{bias}) \cdot \frac{16}{128} \cdot \left(\frac{1}{128}\right)_{TCA} \right\rceil + \left\lceil 1_{store} \cdot \frac{16}{128} \cdot \left(\frac{1}{128}\right)_{TCA} \right\rceil \\
 &= \left\lceil \frac{N_l}{2^9} \right\rceil + 2 \\
 (WCMA_{Coolidge}^{Dense+bias})_{INT8} &= \left\lceil (2N_l + 1_{bias}) \cdot \frac{8}{128} \cdot \left(\frac{1}{128}\right)_{TCA} \right\rceil + \left\lceil 1_{store} \cdot \frac{8}{128} \cdot \left(\frac{1}{128}\right)_{TCA} \right\rceil \\
 &= \left\lceil \frac{N_l}{2^{10}} \right\rceil + 2
 \end{aligned}$$

A.2.4 Softmax Operation

$$\begin{aligned}
 (WCMA_{Coolidge}^{Softmax})_{FP32/INT32} &= \left\lceil V \cdot \frac{32}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{32}{128} \right\rceil = \left\lceil \frac{V}{4} \right\rceil + 1 \\
 (WCMA_{Coolidge}^{Softmax})_{FP16/INT16} &= \left\lceil V \cdot \frac{16}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{16}{128} \right\rceil = \left\lceil \frac{V}{8} \right\rceil + 1 \\
 (WCMA_{Coolidge}^{Softmax})_{INT8} &= \left\lceil V \cdot \frac{8}{128} \right\rceil + \left\lceil 1_{store} \cdot \frac{8}{128} \right\rceil = \left\lceil \frac{V}{16} \right\rceil + 1
 \end{aligned}$$