PUCRS

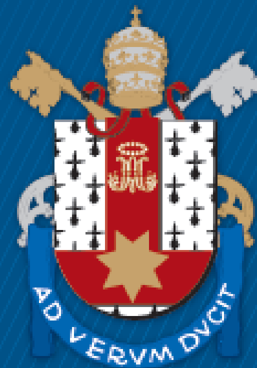
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

TÁRSIO ONOFRIO CARDOSO DA SILVA

FAST CONVOLUTION: ALGORITHMS TO MINIMIZE MULTIPLICATIONS AND IMPROVE PERFORMANCE

Porto Alegre
2026

PÓS-GRADUAÇÃO - *STRICTO SENSU*



Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

FAST CONVOLUTION: ALGORITHMS TO MINIMIZE MULTIPLICATIONS AND IMPROVE PERFORMANCE

TÁRSIO ONOFRIO CARDOSO DA SILVA

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2026

Ficha Catalográfica

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

TÁRSIO ONOFRIO CARDOSO DA SILVA

FAST CONVOLUTION: ALGORITHMS TO MINIMIZE MULTIPLICATIONS AND IMPROVE PERFORMANCE

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March xx, 2026.

COMMITTEE MEMBERS:

Prof. Dr. Mateus Grellert (PGMICRO/UFRGS)

Prof. Dr. César Augusto Missio Marcon (PPGCC/PUCRS)

Prof. Dr. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

AGRADECIMENTOS

À minha família, em especial à minha esposa Adriana e às minhas filhas Isabela e Aurora, bem como aos demais familiares, pelo apoio constante, pela compreensão e pelo incentivo ao longo destes dois anos.

Ao meu orientador, Prof. Dr. Fernando Gehm Moraes, pela orientação técnica, pelas discussões ao longo deste trabalho e pela confiança depositada durante todo o desenvolvimento desta pesquisa.

Ao Prof. Dr. Francisco Kern e à Ana Daniela Dalmolin, da Coordenadoria Rede Communitas, pela dedicação e pelo empenho na viabilização da bolsa de estudos.

Aos colegas do GAPH, que, em diferentes etapas, contribuíram com revisões, discussões e suporte técnico.

Ao Programa de Pós-Graduação em Ciência da Computação da PUCRS, pela estrutura acadêmica e pelo ambiente de pesquisa, que viabilizaram a realização deste trabalho.

A Deus, porque dele, por ele e para ele são todas as coisas.

APOIO FINANCEIRO

Este Curso de Mestrado foi financiado pelo fundo do Carrefour para o Carrefour Programa de Bolsas, por meio do Edital Chamamento Público para ações afirmativas de concessão de bolsas de estudo em instituições de ensino públicas e privadas nos programas de graduação e pós-graduação 2023.

CONVOLUÇÃO RÁPIDA: ALGORITMOS PARA MINIMIZAR MULTIPLICAÇÕES E OTIMIZAR DESEMPENHO

RESUMO

As crescentes demandas computacionais de cargas de trabalho de Aprendizado de Máquina, em especial de Redes Neurais Convolucionais (CNNs), motivam técnicas de aceleração em hardware capazes de reduzir o tempo de execução e o consumo de energia sob restrições de área. As camadas de convolução permanecem como um custo dominante, e a maior parte das implementações de convolução rápida voltadas a hardware concentra-se em um subconjunto reduzido de métodos, principalmente Minimum Filter e Toom-Cook, enquanto técnicas alternativas, como Inspection Factorization (IF) e o método de polinômios modulares de Winograd, não são exploradas em hardware. Além disso, para convolução multidimensional, a maioria dos trabalhos recorre à abordagem *nested*, havendo evidências limitadas sobre o comportamento de composições alternativas em um acelerador completo. Esta dissertação revisita algoritmos de convolução rápida que minimizam o número de multiplicações e os avalia para além da contagem de multiplicações, enfatizando as operações de transformação e a sobrecarga de controle. Nossa proposta apresenta matrizes com entradas em $\{-1, 0, 1\}$. Essa abordagem substitui multiplicações por somas e subtrações e mudanças de sinal, aumentando a eficiência computacional e mantendo compatibilidade com um mapeamento em RTL. Este trabalho implementa um acelerador de hardware em RTL parametrizável para comparar a implementação *ingênua* (baseadas em matriz de MACs) a métodos de convolução rápida, usando uma implementação de controle unificada e um fluxo de síntese ASIC. Conduziu-se uma avaliação de Potência-Desempenho-Área (PPA) para analisar o tempo de execução (em ciclos), a potência, a energia e a área de silício tanto no nível do núcleo de convolução quanto no nível de sistema, evidenciando comportamento de saturação de desempenho ao aumentar-se os recursos aritméticos, e os custos sistêmicos associados a armazenamento e movimentação de dados. Os resultados mostram que o método IF^9, nossa proposta, constitui um ponto intermediário prático quando o objetivo é um projeto que equilibra desempenho e energia ao longo de diferentes pontos de operação. Esse método oferece flexibilidade no número de multiplicadores, permitindo um ajuste fino às restrições de área, de frequência e de largura de banda de acesso à memória. No nível do núcleo, apresenta compromissos competitivos sem penalidade de área e, no nível de sistema, é vantajoso sob restrições de largura de banda, melhorando o desempenho e a energia em regimes limitados pelo barramento com a memória. Este trabalho contribui para a aceleração em hardware ao implementar e avaliar um acelerador parametrizável em RTL para convolução rápida, seguindo uma metodologia PPA unificada.

Palavras-Chave: Convolução rápida, Winograd, RISC-V, Aceleração de Hardware, PPA.

FAST CONVOLUTION: ALGORITHMS TO MINIMIZE MULTIPLICATIONS AND IMPROVE PERFORMANCE

ABSTRACT

The growing computational demands of machine learning workloads, particularly Convolutional Neural Networks (CNNs), motivate hardware acceleration techniques that reduce execution time and energy consumption while meeting area constraints. Convolution layers remain a dominant cost, and most hardware-oriented fast convolution implementations focus on a small subset of methods, primarily Minimum Filter and Toom–Cook. In contrast, alternative techniques, such as Inspection Factorization (IF) and Winograd's modular polynomial method, have not been explored in hardware. Moreover, for multidimensional convolution, most works adopt the nested approach, and there is limited evidence regarding the behavior of alternative compositions in a full accelerator. This dissertation revisits fast convolution algorithms that minimize multiplications and evaluates them beyond multiplication counts, emphasizing transform operations and control overhead. Our proposal introduces matrices with entries in $\{-1, 0, 1\}$. This approach replaces multiplications with additions/subtractions and sign changes, improving computational efficiency while remaining compatible with RTL mapping. This work implements a parameterizable RTL hardware accelerator to compare a *naïve* implementation (based on a MAC array) against fast convolution methods, using a unified control implementation and an ASIC synthesis flow. We execute a Power–Performance–Area (PPA) evaluation to analyze execution time (in cycles), power, energy, and silicon area at both the convolution-core level and the system level, revealing performance saturation as arithmetic resources increase and the system-level costs associated with data storage and movement. The results show that the IF^9 method, our proposal, constitutes a practical intermediate point when the goal is a design that balances performance and energy across different operating points. This method provides flexibility in the number of multipliers, enabling fine-grained tuning to area, frequency, and memory-access bandwidth constraints. At the core level, it achieves competitive trade-offs without area penalties. At the system level, it is advantageous under bandwidth constraints, improving performance and energy in memory-bus-limited regimes. This work contributes to hardware acceleration by implementing and evaluating a parameterizable RTL accelerator for fast convolution under a unified PPA methodology.

Keywords: Fast Convolution, Winograd, RISC-V, hardware acceleration, PPA.

LIST OF FIGURES

2.1 Taxonomy of fast convolution algorithm families considered in this work with emphasis on reference algorithms. 24
2.2 General form of convolution algorithm (adapted from McClellan and Rader [1979]), where d represents the input feature map, g the filter weights, and s the output feature map. 26
2.3 Comparison between convolution using Fourier Transform and Winograd methods, adapted from Blahut [2010]. 27
2.4 Toom-Cook matrices $\{M_3, N_2, L_2, K_3\}$ for 1D convolution. 27
2.5 Toom-Cook matrices $\{M_4, N_2, L_3, K_4\}$ for 1D convolution. 27
2.6 Toom-Cook matrices $\{M_5, N_3, L_3, K_5\}$ for 1D convolution. 28
2.7 General form of tap-filter algorithm (adapted from McClellan and Rader [1979]), where d is the input feature map, g the filter weights, and s the output feature map at the filter output. 30
2.8 Multiplications required for 2D-naïve (Eq. A.14) and 2D Toom-Cook Filter (Eq. 2.6) convolutions for $L_{3\times3=9}$. 35
3.1 Generic architecture and the modules required to build the naïve convolutional accelerators (adapted from Juracy [2022]). 40
3.2 Summary histograms for the three main metadata columns in the systematic review B.1: fast-convolution generator, binding strategy, and language paradigm - HDL or HLS (High-Level Synthesis). 41
3.3 Distribution of reviewed articles across Winograd output tile $(m \times m)$ for 3×3 kernel. 42
4.1 Taxonomy of fast convolution algorithm families considered in this work with emphasis on proposed algorithms. 49
4.2 Comparison between Toom–Cook and Winograd polynomial in 2D Nested for output blocks from 2×2 up to 5×5. 56
4.3 Comparison between the nested and Kronecker 2D Toom-Cook constructions for output blocks from 2×2 up to 5×5. 57
6.1 Top-level organization of the Winograd convolution accelerator. 64
6.2 Approximate timing diagram for the joint evolution of the three FSMs: input, convolution, and output. 65
6.3 Block-level organization of the Winograd convolution core, showing the register bank, Transform, Hadamard and Inverse modules. 65
6.4 Sequence of data transfers between the register bank and the Transform, Hadamard, and Inverse modules for a single Winograd tile. 67

6.5 Internal organization of a representative Transform MatrixC0 stage for the nested IF convolution. 70
6.6 FSM for the fast convolution core. 72
6.7 High-level block diagram of the control module. 73
6.8 Input-side control state machine, showing the phases for loading weights, filling input windows, handing tiles to the convolution core, and coordinating the two hold phases across channels. 75
6.9 Output-side control FSM, showing the states for reading existing tiles, accepting new convolution results, writing accumulated tiles, and stepping across output channels. 77
6.10 Conceptual sliding-window view of the accelerator. 79
6.11 Sequence-level view of data movement across the control unit. 80
6.12 Input-side handshake between the control unit and the convolution core from a simplified behavioral trace. 81
6.13 Output-side handshake timing between the convolution core, the output control block, and the output RAM interface, highlighting when completed tiles are committed to memory. 82
6.14 Approximate timing diagram for convolution core, input and output memory of IF with six multipliers 82
7.1 Compute-only total cycles for for $F_{\text{map}} = 32$ and $C_{\text{in}} = C_{\text{out}} = 3$, compared against the Naïve baseline. 86
7.2 Simulated performance for the evaluated algorithms (y-axis), and number of MACs (x-axis). The reference, Naïve, requires 27000 cycles to execute. 87
7.3 Total ell area per configuration (μm^2) as a function of the multiplier count. . . 88
7.4 Power consumption (mW) as a function of the multiplier count. 89
7.5 Energy evaluation as a function of the multiplier count. 90
7.6 Convolution-core power–cycle scatter plot. 90
7.7 Cell area (x-axis) versus power (y-axis) for the TCn^9 and TCk^9 configurations. 92
7.8 Analytic performance (y-axis), varying the MAC number (x-axis), for different bus widths. 95
7.9 Convolution with row- and column-major order memory organizations. 96
7.10 Simulated system-level runtime versus multiplier number. 97
7.11 System-level cell area versus multiplier count. 98
7.12 System-level power versus multiplier count. 99
7.13 System-Level Energy per operation in nJ as a function of the multiplier count. 99
7.14 System-level power versus runtime. 100

A.1 Graphical symbolic convolution. 118

A.2 Graphical numeric convolution. 118

A.3 Matrix B decomposed into powers of two constants. 127

D.1 Steady-state cycle models for the IF-based fast-convolution accelerator ($M = 5$, $N = 3$, $K = 6$) to produce an $F_{\text{map}} \times F_{\text{map}}$ output feature map, as a function of the output side F_{map}, for MAC counts $M_{\text{AC}} \in \{1, 2, 3, 4, 6, 9, 18, 36\}$, compared with the naïve-convolution baseline. 151

D.2 Steady-state cycle models for a Winograd minimal-filter accelerator ($M = 4$, $N = 2$, $K = 4$) to produce an $F_{\text{map}} \times F_{\text{map}}$ output feature map, as a function of the output side F_{map}, for MAC counts $M_{\text{AC}} \in \{1, 2, 4, 8, 16\}$, compared with the naïve-convolution baseline. 152

D.3 Steady-state cycle models for a Winograd convolution with $M = 6$, $N = 4$, filter length $L = 3$, and transformed-kernel size $K = 9$ to produce an $F_{\text{map}} \times F_{\text{map}}$ output feature map, as a function of the output side F_{map}, for MAC counts $M_{\text{AC}} \in \{1, 3, 9, 27, 81\}$, compared with the naïve-convolution baseline. 152

D.4 Steady-state cycle models for a Toom–Cook-3 accelerator ($M = 5$, $N = 3$, $K = 5$) to produce an $F_{\text{map}} \times F_{\text{map}}$ output feature map, as a function of the output side F_{map}, for MAC counts $M_{\text{AC}} \in \{1, 5, 25\}$, compared with the naïve-convolution baseline. 153

LIST OF TABLES

5.1 Number of multiplications and matrices' size. 58

5.2 Instruction Profiling for the Convolution Methods. The profiling of the Fast Convolution methods is expressed as a percentage w.r.t the N^9 method. 60

7.1 Evaluated algorithms with the sizes of the input M and the output N. 84

7.2 Evaluated algorithms, and the number of configurations of each one, defined by the number of MACs. 85

7.3 Analytical compute-only ratio summary for the evaluated fast-convolution algorithms. Ratio corresponds to the performance improvement w.r.t. the Naïve algorithm. 86

7.4 Simulated compute-only ratio summary for the evaluated fast-convolution families. 87

7.5 Evaluation of convolution algorithms. Normalized ratios of performance (simulation), cell area, power, and energy. All values are reported relative to the Naïve baseline under the same workload and clock constraint. 91

7.6 Word-size settings for the bus-width study, where M is the input tile size and N is the output tile size. Bits corresponds to the bus width, which is the $20 \times w_{size}$, where 20 is the reference word size. 94

7.7 System-level cycle ratios (normalized to the Naïve baseline) comparing each algorithm across multiplier counts and word sizes. 97

7.8 System-level normalized ratios for performance (simulation), cell area, power, and energy, with emphasis on comparing algorithms across different w_{size} configurations. All values are reported relative to the Naïve baseline under the same workload and clock constraint. 100

B.1 Winograd algorithms and implementation details for each article in the systematic review. 129

B.2 Winograd output size $m \times m$ for 3×3 kernel implemented by each article. A mark "X" indicates that the corresponding configuration is explicitly reported in the paper. 130

LIST OF ACRONYMS

CNN – Convolutional Neural Network

DNN – Deep Neural Network

FFT – Fast Fourier Transform

DFT – Discrete Fourier Transform

DSP – Digital Signal Processor

FPGA – Field-Programmable Gate Array

ASIC – Application-Specific Integrated Circuit

SOC – System-on-Chip

RTL – Register-Transfer Level

PPA – Power, Performance, and Area

MAC – Multiply–Accumulate

PE – Processing Element

IFMAP – Input Feature Map

OFMAP – Output Feature Map

IOT – Internet of Things

GEMM – General Matrix–Matrix Multiplication

RS5 – RS5 RISC-V processor

RISC-V – Reduced Instruction Set Computer – Five

GAN – Generative Adversarial Network

BRAM – Block RAM

LUT – Look-Up Table

AXI – Advanced eXtensible Interface

LIST OF SYMBOLS

N – Length of the input data sequence $d = (d_0, \ldots, d_{N-1})$ 12
L – Length of the filter sequence $g = (g_0, \ldots, g_{L-1})$ 12
M – Length of the output sequence s, $M = N + L - 1$ for linear convolution 12
K – Number of multiplications required by a convolution algorithm for given N and L 12
W – Transform length used in FFT-based methods and DFT matrices 12
A – Input transform matrix applied to the data sequence 12
B – Kernel transform matrix applied to the filter coefficients 12
C – Output transform matrix applied after the Hadamard product 12
Q – Scaling/quotient matrix used in the kernel transform 12
G – Transformed kernel, $G = QBg$ 12
D – Transformed data, $D = Ad$ 12
S – Hadamard product in the transform domain, $S = G \odot D$ 12
n_t – Linear size of the input tile processed by the Winograd core 12
m_t – Linear size of the output tile produced by the Winograd core 12
H – Number of scalar products (Hadamard multiplications) required to process one transformed tile 12
P – Number of parallel multipliers instantiated in the Hadamard engine 12

CONTENTS

1 INTRODUCTION **17**

1.1 MOTIVATION 18

1.2 OBJECTIVES 19

1.3 METHODOLOGY 19

1.4 CONTRIBUTIONS 20

1.5 PUBLICATIONS DURING THE MSC PERIOD 21

1.6 DOCUMENT ORGANIZATION 21

2 REFERENCE FAST CONVOLUTION ALGORITHMS **23**

2.1 FAST CONVOLUTION 23

2.2 TOOM-COOK 26

2.3 FILTER FORM 29

2.4 NESTED 31

3 RELATED WORK **36**

3.1 GENERIC HARDWARE ARCHITECTURES FOR CONVOLUTION 36

3.2 ACCELERATORS FOR NAÏVE CONVOLUTION 37

3.3 WINOGRAD-BASED FAST CONVOLUTION ACCELERATORS 40

3.3.1 WINOGRAD IMPLEMENTATION ON HARDWARE 40

3.3.2 CNN-LEVEL WINOGRAD ACCELERATORS ON FPGA/ASIC 43

3.3.3 APPLICATION-SPECIFIC AND MULTIDIMENSIONAL DESIGNS 44

3.3.4 TEMPLATE-BASED AND FRAMEWORK-DRIVEN CONVOLUTION ACCELERATORS 44

3.3.5 WINOGRAD AND KRONECKER PRODUCT 45

3.3.6 OTHER CLASSIC FAST CONVOLUTION METHODS 46

3.3.7 SIMILAR WORKS 47

4 PROPOSED FAST CONVOLUTION ALGORITHMS **49**

4.1 MULTIDIMENSIONAL KRONECKER 49

4.2 INSPECTION FACTORIZATION 52

4.3 WINOGRAD POLYNOMIAL MODULAR CONVOLUTION 54

4.4 COMPARISON BETWEEN REFERENCE AND PROPOSED ALGORITHMS 55

4.5 CONCLUSION 57

5 SOFTWARE IMPLEMENTATION . . . 58
5.1 SOFTWARE IMPLEMENTATION AND TOOLBOX . . . 58
5.2 EXPERIMENTAL RESULTS ON RS5 . . . 60

6 HARDWARE IMPLEMENTATION OF THE WINOGRAD CONVOLUTION ACCELERATOR . . . 62
6.1 DESIGN GOALS AND CONSTRAINTS . . . 62
6.2 TOP-LEVEL ARCHITECTURE . . . 63
6.2.1 EXTERNAL INTERFACE . . . 63
6.2.2 INTERNAL ORGANIZATION . . . 64
6.3 CONVOLUTION CORE . . . 65
6.3.1 WINOGRAD DATAPATH IMPLEMENTATION . . . 66
6.3.2 TRANSFORM AND INVERSE MATRIX GENERATORS . . . 67
6.3.3 HADAMARD PRODUCTS AND MULTIPLIER SHARING . . . 69
6.3.4 STATE MACHINE OF THE CONVOLUTION CORE . . . 71
6.4 CONTROL UNIT AND SCHEDULING . . . 73
6.4.1 FUNCTIONAL ROLE OF THE CONTROL MODULE . . . 74
6.4.2 INPUT-SIDE FSM . . . 74
6.4.3 OUTPUT-SIDE FSM . . . 76
6.4.4 INTERACTION WITH DATAPATH AND MEMORY . . . 78
6.4.5 HANDSHAKE PROTOCOL AND TIMING . . . 81
6.5 CONCLUSION . . . 83

7 RESULTS . . . 84
7.1 CONVOLUTIONAL CORES EVALUATION . . . 84
7.1.1 ANALYTICAL PERFORMANCE EVALUATION . . . 85
7.1.2 SIMULATED PERFORMANCE EVALUATION . . . 87
7.1.3 AREA . . . 88
7.1.4 POWER . . . 89
7.1.5 ENERGY . . . 89
7.1.6 KRONECKER BINDING AND CSA IMPLEMENTATIONS . . . 91
7.2 SYSTEM LEVEL: CONTROL AND CONVOLUTION . . . 92
7.2.1 ANALYTICAL PERFORMANCE EVALUATION . . . 92
7.2.2 SIMULATED PERFORMANCE EVALUATION . . . 95
7.2.3 AREA . . . 96

7.2.4 POWER . . . 98
7.2.5 ENERGY . . . 98
7.3 FINAL REMARKS . . . 101

8 CONCLUSIONS AND FUTURE WORK . . . 103
References . . . 107

APPENDIX A – Foundations of Discrete Convolution and Fast Multiplication . **115**
A.1 MULTIPLICATION . . . 115
A.1.1 NAIVE MULTIPLICATION . . . 115
A.1.2 KARASTUBA MULTIPLICATION . . . 116
A.2 CONVOLUTION . . . 118
A.2.1 LINEAR CONVOLUTION . . . 118
A.2.2 CIRCULAR CONVOLUTION . . . 120
A.3 VANDERMONDE MATRIX . . . 121
A.4 TOOM–COOK ALGORITHM AND MATRIX CONSTRUCTION . . . 121
A.5 MULTIDIMENSIONAL ALGORITHMS . . . 125
A.6 BACKGROUND: KRONECKER PRODUCT . . . 126
A.7 POWER-OF-TWO FACTORIZATION . . . 126
A.8 AGARWAL–COOLEY AND SPLIT-NESTING . . . 127

APPENDIX B – Systematic Review Tables . . . **129**

APPENDIX C – Proposed Fast Convolution Algorithms . . . **131**
C.1 GENERATING FAST LINEAR CONVOLUTION ALGORITHMS VIA CRT . . . 131
C.2 LINEAR 2×2 CONVOLUTION VIA EVALUATE–HADAMARD–INTERPOLATE (THE A, B, C FORM) . . . 136
C.3 LINEAR 2×3 CONVOLUTION VIA EVALUATE–HADAMARD–INTERPOLATE (THE A, B, C FORM) . . . 141

APPENDIX D – Additional Results . . . **148**
D.1 DERIVATION OF THE FAST-CONVOLUTION ANALYTICAL MODEL . . . 148
D.2 INSTANTIATION FOR THE IF 2D CONFIGURATION . . . 150
D.3 SPECIALIZATIONS: MINIMAL WINOGRAD AND TOOM–COOK-3 . . . 150
D.4 COMPUTE-ONLY UPPER BOUND FOR THE CONVOLUTIONAL CORE . . . 153

1. INTRODUCTION

In recent years, convolutional neural networks (CNNs) have been increasingly adopted in computer vision tasks such as object detection, image classification, and semantic segmentation. Their widespread success stems from the ability to automatically learn salient image features, outperforming traditional approaches on numerous benchmarks [Krizhevsky et al., 2017; Goodfellow et al., 2016]. At the same time, the increased depth of these architectures comes with higher computational demands, driven primarily by the extensive multiply-accumulate (MAC) operations in convolutional layers. As a result, accelerating the execution time of these essential layers is critical to improving the overall performance of CNN-based systems [Juracy, 2022].

Various hardware accelerators are employed to manage the high computational cost. Traditionally, GPUs stand out for their large number of cores specialized in floating-point operations and their relative ease of programming. However, they suffer from high power consumption and offer less flexibility for specific pipeline modifications [Dally et al., 2020]. On the other hand, FPGAs offer reconfigurability, allowing deeper exploitation of data parallelism and the implementation of highly optimized pipelines. They also provide good energy efficiency, although their learning curve is steeper due to the need to design hardware (using HDL or high-level synthesis tools) and manage resources such as DSPs, LUTs, and BRAMs [Shawahna et al., 2019]. In parallel, ASICs (Application-Specific Integrated Circuits) offer the highest performance and lowest energy consumption in well-defined scenarios, as they can be designed to meet the application's requirements. However, the lack of flexibility, high development cost, and long design time limit their use in situations where networks or algorithms will not change significantly [Moolchandani et al., 2021].

A fundamental part of optimizing CNNs lies in how convolution is performed. Conventional methods map convolution to matrix multiplications or execute it directly in the spatial domain. Although efficient, these approaches do not reduce the number of multiplications. On the other hand, fast convolution algorithms, such as Winograd and FFT [Winograd, 1987; Blahut, 2010; Tolimieri et al., 2013; Parhi, 1999], have been increasingly investigated for significantly reducing the computational load, especially for small filters (3×3 or 5×5) found in popular networks like VGG, ResNet, GoogLeNet, and YOLO. However, implementing on FPGAs offers challenges, requiring the mapping of hardware resources (DSPs, LUTs, BRAMs) and an effective balance between data throughput and computational capacity to avoid memory bottlenecks. Exploring parameters such as the number of processing elements, tile sizes, and pipeline depth can affect the design efficiency [Liang et al., 2020; Kala et al., 2019a; Xiao et al., 2017].

These factors highlight the importance of investigating techniques and architectures that combine fast convolution algorithms with small filters, leveraging optimized hardware implementations such as FPGAs or ASICs. This approach can enhance the speed and feasibility

of CNNs in embedded systems and high-computation-demand applications while considering energy consumption constraints and the ongoing evolution of network models.

1.1 Motivation

The main motivation is the growing use of CNNs in embedded and IoT systems, where convolution layers dominate time and energy and therefore need specialized hardware under tight constraints. This connects to the need for RTL accelerator designs that *meet performance requirements* while operating near memory bandwidth limits, where data movement, buffering, and control can be as important as the datapath. This motivates a parameterizable architecture and a systematic PPA evaluation across naïve and fast configurations.

Convolution remains a practical bottleneck in many workloads, especially in vision models. Even as model design changes, convolution still consumes a large share of execution time and energy because it is applied repeatedly across large feature maps and channel dimensions. This makes convolution a strong target for algorithmic acceleration: any systematic reduction in compute or cycles at this kernel level tends to produce measurable end-to-end gains.

Fast-convolution methods are especially relevant for small kernels, where Winograd-style constructions and related families can reduce multiply complexity while keeping a regular computation structure. Small kernels such as 3×3 remain common building blocks in practical networks. This prevalence motivates a focused study of fast convolution for small spatial filters, including how algorithmic choices interact with architectural parameters such as parallelism, buffering, and ordering to determine the accelerator's efficiency.

In edge computing and IoT scenarios, the limiting factors often shift from raw arithmetic throughput to energy per inference and memory-system constraints. The cost of moving data through buffers, interconnects, and memories can exceed the cost of arithmetic. Therefore, improvements must be evaluated in terms of system-level energy and runtime rather than just operation counts. Fast-convolution techniques are motivated in this context because they reduce the active time of the compute datapath and/or reduce switching activity by decreasing the amount of multiply-accumulate work per output.

Another motivation is the lack of work on alternative techniques for fast, multidimensional convolution. In hardware-oriented CNN acceleration, most implementations remain limited to the minimum filter and the Toom-Cook method, typically for a small set of fixed 2D kernel sizes. Although classic works point to broader algorithmic options, including modular polynomials, split nesting, and prime-length convolutions, these techniques are not explored in the context of CNN accelerators. In particular, the Winograd modular polynomial method has not been studied as a practical design space for CNN acceleration. For multidimensional convolution, the literature relies mainly on the nested approach to compose 1D algorithms.

This gap motivates revisiting these methods and evaluating them under a unified hardware template so that their benefits and overheads can be assessed beyond multiplication counts.

1.2 Objectives

The **primary objective** of this dissertation is to **implement and evaluate fast, multidimensional convolution in hardware under a unified RTL baseline design**, quantifying when arithmetic savings translate into system-level improvements in performance, power, energy, and area.

The **specific objectives** are:

1. **Expand the fast-convolution design space beyond Minimum Filter and Toom-Cook**, including alternative Winograd-style formulations such as the **modular polynomial method and Inspection Factorization**, and relate these choices to practical CNN convolution layers.
2. **Investigate multidimensional fast convolution beyond the nested approach**, considering alternatives such as the **Kronecker product**, and analyze how these compositions affect area and energy consumption.
3. **Develop a parameterizable RTL implementation** that supports a naïve baseline and multiple fast-convolution algorithms, enabling controlled comparisons across algorithm families, sizes, and architectural parameters (e.g., number of MAC units and internal bandwidth).
4. **Perform a systematic PPA evaluation at core and system level**, identifying **saturation** regimes and quantifying costs beyond multiplication count, including transform complexity, resgister banks and **control overhead**, to derive comparative conclusions across configurations.

1.3 Methodology

This dissertation follows a unified workflow to evaluate fast and multidimensional convolution in software and hardware. First, we formalize the fast-convolution algorithms using a common matrix form and notation so that each method can be expressed as a sequence of transforms, elementwise products, and an inverse step. This formulation supports both nested and Kronecker constructions and enables exploration of alternative generators, such as inspection factorization and modular polynomials, while maintaining a comparable interface at the tile level.

Second, we review the state-of-the-art of CNN accelerators and Winograd-based fast convolution to identify the dominant design choices in practice (generator family, multidimensional binding, and language paradigm) and to define a consistent baseline for comparison.

Based on this baseline, we select a set of reference algorithms and architectural parameters that can be evaluated under the same assumptions.

Third, we implement and validate the selected algorithms in software using a Python toolbox that supports design, numerical validation, and code generation in C and SystemVerilog. The software implementation of fast convolution algorithms was evaluated on the RS5 RISC-V processor, simulated using Verilator. This software stage is used to prototype the transforms, verify correctness against a naïve reference, and extract algorithm parameters required for hardware mapping.

Fourth, we map the algorithms to a parameterizable RTL accelerator. The implementation is organized as a complete system, including a top-level System module, a convolution core built around a matrix-multiplication engine, and a control unit responsible for memory access and data registers. The RTL template supports a naïve MAC-based baseline and multiple fast-convolution configurations by varying the number of MAC units and the internal bandwidth (w_{size}), enabling controlled exploration of the design space under the same architectural organization.

Finally, we perform a systematic Power–Performance–Area (PPA) evaluation at both the convolution-core and system levels. Performance is evaluated analytically and validated through RTL simulation. Area is measured after logical synthesis under a fixed clock constraint, and power is estimated with an activity-annotated flow; energy is computed as the product of average power and execution time. Reporting both core-level and system-level results makes explicit when improvements are compute-bound and when they are dominated by bus throughput, memory latency, buffering, and control overhead, including saturation regimes where increasing MAC yields only small cycle reductions.

1.4 Contributions

The contributions of this Dissertation are:

1. Proposed fast convolution algorithms beyond the standard Toom-Cook and nested baselines, including a multidimensional Kronecker formulation, Inspection Factorization, and a Winograd polynomial modular convolution, with a direct comparison against reference algorithms (Chapter 4).
2. Software implementation and evaluation of the fast convolution algorithms in Python, C, and SystemVerilog, supported by a toolbox for design, validation, and code generation, including experiments on the RS5 RISC-V processor simulated using Verilator (Chapter 5).
3. A parameterizable RTL implementation of a Winograd-based convolution accelerator, including the top-level System module, the convolution core with its matrix-multiplication engine, and a control unit responsible for accessing tiles and managing data registers under memory-bandwidth constraints (Chapter 6).

4. A unified Power–Performance–Area (PPA) evaluation at the convolution-core and system levels, combining analytical performance modeling with RTL validation, and reporting area, power, and energy to expose saturation and the system-level impact of bandwidth, buffering, and control overhead across configurations (Chapter 7).
5. Public research artifacts released as open-source repositories, connecting algorithm design, software evaluation, and hardware implementation:
   - https://github.com/tarsioonofrio/fast-convolution-rtl: Python package used to implement the fast-convolution algorithms described in Chapters 2 and 4, with code-generation support for C (Chapter 5) and SystemVerilog (Chapter 6).
   - https://github.com/tarsioonofrio/FastConv_SystemVerilog: synthesizable SystemVerilog implementation of the accelerator pipeline, including RTL modules, simulation/synthesis automation, and report-generation scripts, and also the stored result files used in Chapter 7.
   - https://github.com/tarsioonofrio/fast-conv-RS5: software implementation of fast convolution for the RS5 RISC-V platform, with execution flows for optimized/non-optimized variants and consolidated reporting, and also the stored result files used in Chapter 5.

1.5 Publications During the MsC Period

The development of this work resulted in the following publication:

> Fast and Energy-Efficient 2D Convolution: Inspection-Based Methods for Hardware Acceleration
> ONOFRIO, Társio; MORAES, Fernando Gehm
> In: SBCCI, 2025
> https://doi.org/10.1109/SBCCI66862.2025.11218694

1.6 Document organization

This Dissertation is organized as follows.

- Chapter 2 introduces the mathematical background used throughout the text, revisiting fast convolution in the Winograd framework and the main constructions used to derive multidimensional algorithms.
- Chapter 3 reviews the state-of-the-art in CNN acceleration and summarizes how fast convolution is commonly employed in practice.
- Chapter 4 presents a concise, algebraic description of the proposed fast convolution algorithms adopted in this Dissertation, focusing on deriving schemes that reduce or eliminate multiplications by general constants.

- Chapter 5 presents the software infrastructure used to design and validate the algorithms, including the evaluation of the fast convolution implementations on the RS5 RISC-V processor.
- Chapter 6 describes the parameterizable RTL accelerator used as a common baseline, including the convolution core and the control organization that enables comparisons across naïve and fast configurations.
- Chapter 7 presents the Power-Performance-Area (PPA) evaluation of the Convolutional Core and the complete system: performance is evaluated analytically and validated through RTL simulation.
- Chapter 8 summarizes the main conclusions and outlines future work.

2. REFERENCE FAST CONVOLUTION ALGORITHMS

This chapter presents a concise, purely mathematical **reference** for the fast convolution algorithms that appear throughout this Dissertation. The focus is deliberately restricted to the Toom–Cook and nested Winograd-type methods used by the works surveyed in Chapter 3. For each family, we provide algebraic formulations, notation, and complexity expressions so that all related algorithms can be described and compared within a single, unified framework without committing to any particular implementation style or hardware platform.

Readers who are not yet familiar with the basic notions of multiplication, fast multiplication, and linear and circular convolution may benefit from consulting Appendix A first, which develops these topics in a linear and pedagogical fashion. That appendix provides the groundwork that makes the fast-convolution material in this chapter easier to follow. For a deeper repertoire in abstract algebra, the mathematical basis of fast convolution theory, one may also refer to Chapters 1 and 2 of Blahut [2010], which motivate the polynomial and algebraic structures underpinning Toom–Cook and related methods.

The mathematical descriptions developed here play multiple roles in the remainder of the document. They supply the vocabulary used in Chapter 3 to classify prior accelerators as Toom–Cook or nested schemes; they define the algorithms that are implemented in software in Chapter 5 and mapped to hardware in Chapter 6; and they provide the reference needed to interpret the performance and energy results reported in Chapter 7. Finally, the operators, transforms, and structural properties introduced here are essential for understanding the new algorithmic proposals developed in Chapter 4.

Notation is summarized in the global list of symbols (see the List of Symbols on page 13).

2.1 Fast convolution

A commonly used approach to enhance the efficiency of convolution calculations is to apply the convolution theorem with the Fast Fourier Transform (FFT). FFT can significantly reduce the computational load to $W \log_2 W$ (W is width when $M = L$), which is particularly advantageous for lengthy convolutions. However, this method has limitations, as it requires complex numbers, which may not be ideal for all real-valued convolutions [Nussbaumer, 1982].

Fast convolution algorithms have been developed to overcome the limitations of FFT and reduce computational demand, especially for short block lengths common in CNNs. These methods minimize the number of costly operations, such as multiplications, even if this increases the number of additions, thereby reducing overall computational cost. This makes them well-suited for software and dedicated hardware implementations [Parhi, 1999]. Notably, Winograd algorithms are highly efficient for small block-length convolutions.

Fast convolution is not a single algorithm but an umbrella term for several families of techniques that accelerate discrete convolution by restructuring it as a sequence of transforms, pointwise multiplications, and inverse transforms. In this Dissertation, we focus on a subset of these families that are relevant for small convolution sizes and hardware-oriented implementations.

Figure 2.1 organizes these methods into three nested sets. The outer box, labeled "Fast convolution algorithms", represents the fast methods considered in this Dissertation. Inside it, the "Winograd Framework" box delimits the polynomial and bilinear-form constructions derived from Winograd's theory. Finally, the shaded region within the Winograd box highlights the specific reference algorithms used throughout this Dissertation: Toom–Cook schemes combined with their nested extensions. This three-level view helps situate the reference algorithms within both the broader fast-convolution landscape and the Winograd family in particular, and serves as a roadmap for the sections that follow.

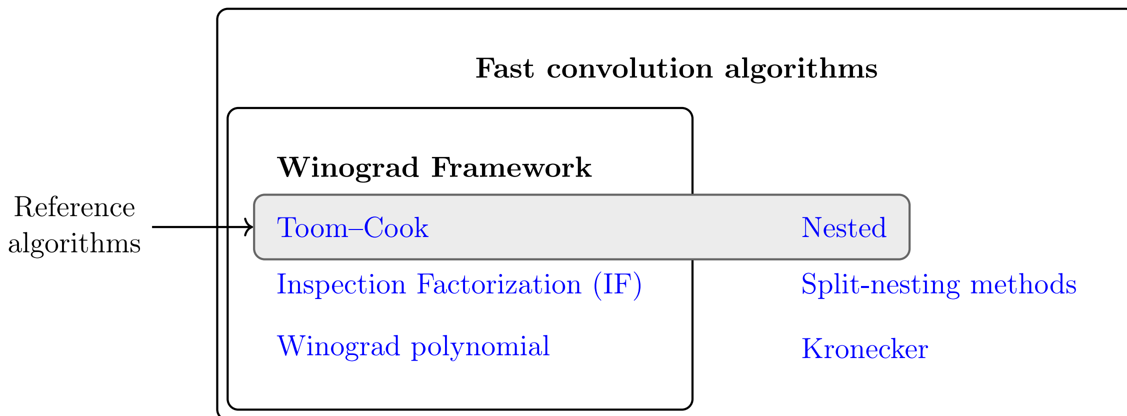


Figure 2.1: Taxonomy of fast convolution algorithm families considered in this work with emphasis on reference algorithms.

Within this umbrella, Fourier transform–based methods use the Discrete Fourier Transform (DFT) or number-theoretic transforms to map convolution into pointwise multiplication in the frequency domain. Classical FFT-based schemes and the number-theoretic methods of Nussbaumer fall into this class. Other approaches, such as the Agarwal–Cooley split-radix constructions and related nested schemes, further decompose the transforms to reduce arithmetic costs or to better exploit structural regularity.

A second major family is the Winograd framework. Rather than starting from the Fourier transform, Winograd formulates convolution as a bilinear form and derives optimal or near-optimal algorithms (in terms of scalar multiplications) from algebraic properties of polynomials and their evaluation. The Winograd framework contains several concrete instances. Toom–Cook methods apply polynomial interpolation and evaluation at carefully chosen points, and other polynomial-based constructions explore different choices of evaluation points and bases. These methods are attractive for small block-length convolutions, which motivate their use in CNN accelerators.

The remainder of this section is written with Chapter 3 in mind. Our objective is to define the main families of fast convolution (FFT-based, Winograd-type, Toom–Cook, inspection factorization, and related nested or Kronecker constructions), to introduce the notation used for their arithmetic complexity, and to establish the terminology that will later appear in the taxonomy of prior work. When Chapter 3 classifies accelerators as "naïve," "minimal Winograd," "Toom–Cook," or "IF-based," it relies directly on the definitions and symbols introduced in the following subsections and in Figure 2.1.

Winograd [1976] developed a theorem that determines the minimum number of multiplications required for performing convolutions, subsequently establishing a comprehensive framework using Karatsuba, Toom, Cook, Agarwal, and FFT works for deriving algorithms that execute convolutions with an optimal number of multiplications and additions [Winograd, 1987]. That framework, a suite of algorithms, optimizes convolution, with the general formulation shown in Equation 2.1 and illustrated in Figure 2.2, where:

- d is the input feature map;
- g is the learned filter (weight) vector;
- s is the output feature map;
- Q is the matrix of quotients;
- A and B are, respectively, the input and kernel transform matrices;
- C is the inverse transform matrix that maps the transformed products back to the spatial domain;
- $\odot$ denotes the Hadamard (elementwise) product.

$$s = C[(QBg) \odot (Ad)] \tag{2.1}$$

Equation 2.2 shows the operations involved in implementing a Winograd fast convolution.

$$G = QBg, \quad D = Ad, \quad S = G \odot D, \quad s = CS \tag{2.2}$$

This matrix representation provides a visual framework for the algorithm, though the computation is not a direct matrix product. Instead, the multiplications involving matrices A and C are performed as sequences of additions. Multiplication $S = G \odot D$ contains all multiplications of the Winograd convolution [Myers, 1990; Tolimieri et al., 1997; McClellan and Rader, 1979].

The matrix $Q \in \mathbb{Q}$ introduces divisions due to the fractions (matrix Q in Figures 2.5 and 2.6 - page 27). It is recommended that this computation be preprocessed offline to reduce the computational complexity associated with $G = QBg$.

The Winograd convolution algorithm generalizes the convolution computation using a transform structure, as shown in Figure 2.3. For example, when using the Fast Fourier

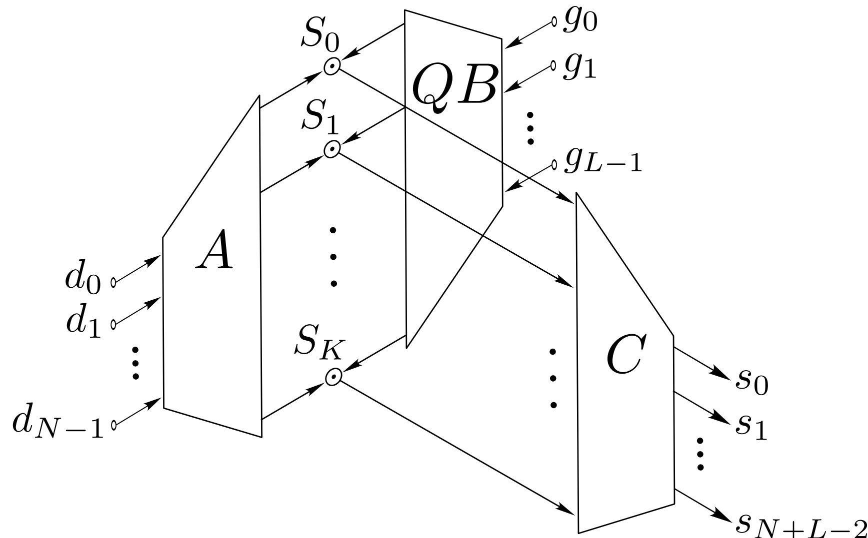


Figure 2.2: General form of convolution algorithm (adapted from McClellan and Rader [1979]), where d represents the input feature map, g the filter weights, and s the output feature map.

Transform (FFT), the matrices A and B correspond to transformations, while C represents the inverse.

2.2 Toom-Cook

One of the key algorithms developed by Winograd for fast convolution is the Toom-Cook algorithm [Blahut, 2010; Winograd, 1987; Myers, 1990; Tolimieri et al., 2013], initially designed for multiplication. This algorithm enables fast convolution for 1D data and guarantees the minimum number of multiplications. The number of multiplications is given by $K = L + N - 1$, a linear growth, different from the quadratic growth of the naïve 1D convolution ($K = W^2$, for L and N having the same width).

The algebraic derivation of Toom–Cook—including the choice of interpolation points β, the use of Vandermonde and inverse Vandermonde matrices, and the separation of numerator and denominator terms via Lagrange interpolation to obtain the C and Q matrices—is developed in detail in Chapter A, Section A.4. In this chapter, we focus on the concrete matrix instances that are used as reference fast-convolution algorithms in the remainder of the Dissertation.

We show in Figure 2.4 the basic fast convolution algorithm, the Karatsuba or Toom-Cook configuration $\{M_3, N_2, L_2, K_3\}$, where M is the input length, L is the filter length, N is the number of output samples, and K is the number of scalar multiplications:

We use two sets of matrices generated by the Toom-Cook algorithm, with $L = 3$. Figure 2.5 presents a set of matrices for parameters $\{M_4, N_2, L_3, K_4\}$, corresponding to 4 multi-

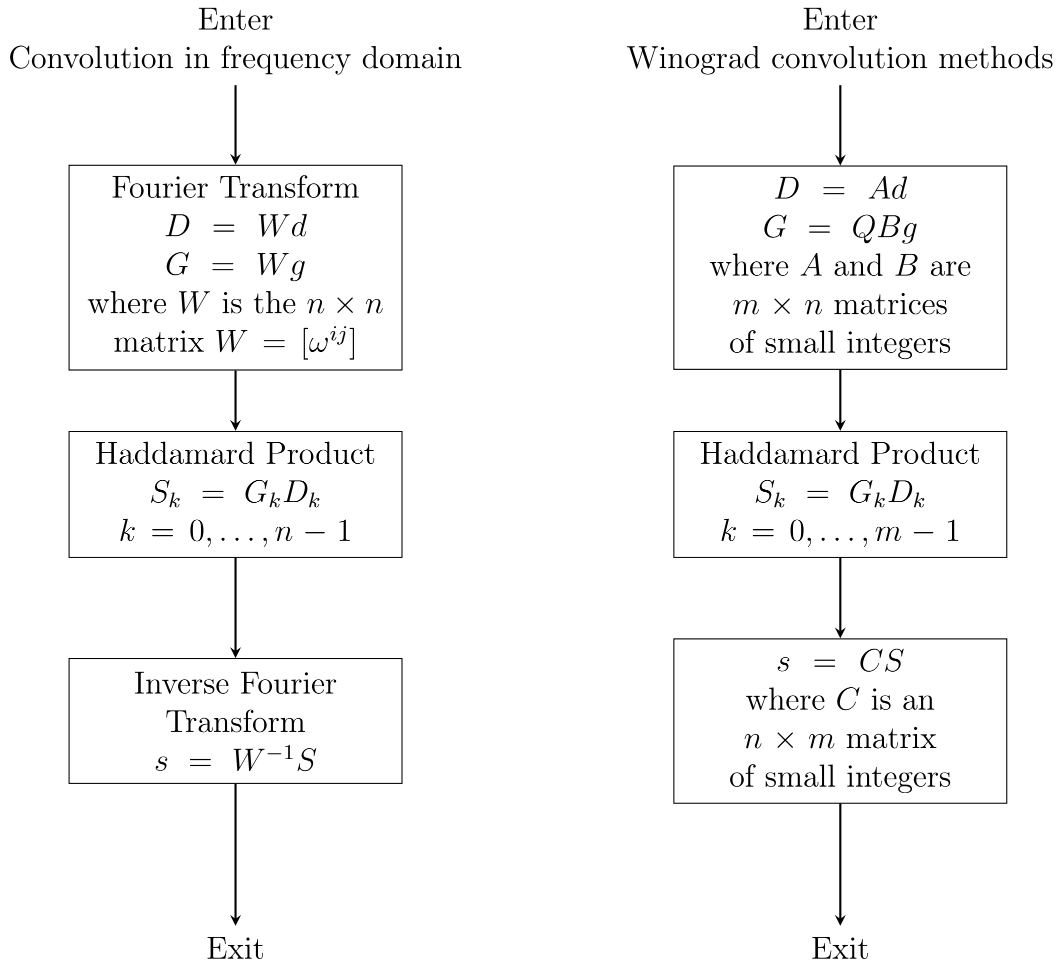


Figure 2.3: Comparison between convolution using Fourier Transform and Winograd methods, adapted from Blahut [2010].

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Figure 2.4: Toom-Cook matrices $\{M_3, N_2, L_2, K_3\}$ for 1D convolution.

plications and two output values. Figure 2.6 shows another set of matrices for $\{M_5, N_3, L_3, K_5\}$, corresponding to 5 multiplications and three output values.

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} -1 \\ \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix}.$$

Figure 2.5: Toom-Cook matrices $\{M_4, N_2, L_3, K_4\}$ for 1D convolution.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ -1 & -2 & 2 & -1 & 2 \\ -2 & -1 & -3 & 0 & -1 \\ 1 & 1 & 1 & 1 & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{6} \\ \frac{1}{6} \\ 1 \end{bmatrix}.$$

Figure 2.6: Toom-Cook matrices $\{M_5, N_3, L_3, K_5\}$ for 1D convolution.

We develop a complete example for the matrices of Figure 2.5 where we replace the variables d and g of Equation 2.1 with the matrices:

$$d = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad g = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

we have the full form (Equation 2.1): $s = C[(QBg) \odot (Ad)]$): in this example d represents the input feature map and g the corresponding filter weights.

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \left\{ \left[\begin{bmatrix} -1 \\ \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix} \odot \left(\begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \right) \right] \odot \left(\begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) \right\}.$$

First, transform the weights g in G using $G = Q \odot (Bg)$; these operations can be executed offline and correspond to the weight transformation step of the fast convolution:

$$\begin{bmatrix} G_0 \\ G_1 \\ G_2 \\ G_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 \\ \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 2 \\ 6 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 \\ \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix} \odot \left(\begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \right).$$

Now transform the input data d in D making $D = Ad$:

$$\begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

Calculate, $S = G \odot D$, the Haddamard Product:

$$\begin{bmatrix} -1 \\ -1 \\ 9 \\ 6 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 3 \\ 3 \end{bmatrix} \odot \begin{bmatrix} 1 \\ -1 \\ 3 \\ 2 \end{bmatrix}.$$

And finally, we calculate the inverse $s = cS$

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 7 \\ 6 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ 9 \\ 6 \end{bmatrix}.$$

In the convolution setting, this vector s corresponds to the output feature map produced by the fast algorithm.

Note that the s result is the same as the one obtained in Figure A.2 (page 118).

2.3 Filter Form

Efficient, fast convolutions are typically designed for small data sequences. However, CNNs use images of very long data sequences, which are not optimal for execution via fast convolutions. To be accommodated effectively in fast convolutions, images must be converted into shorter data sequences. Generating a fast convolution for an entire image proves impractical, primarily due to the significant amount of memory required to store both the input data and the intermediate results from preceding operations before yielding the final output.

Filter techniques have been developed to reduce long convolutions into sequences of shorter convolutions. That necessitates segmenting the image into smaller blocks, which the filter can process as they become available. As soon as a block of an image is processed, its output is concatenated with the previous output block, ultimately forming the complete final output.

Blahut [2010] express how to process smaller data blocks using linear algebra. Let's start with the fast convolution form $s = CGAd$, where G is a diagonal matrix.

To convert the fast convolution form to a filter form, using the overlap-save method, apply:

$$s = A^T G C^T d. \tag{2.3}$$

Compare Figure 2.2 with Figure 2.7, noting that the positions of the signals d and s, as well as the matrices A and C, are interchanged. In both cases d denotes the input feature

map, g the filter weights, and s the output feature map; the diagrams differ only in whether the structure is presented in convolution form or in filter form.

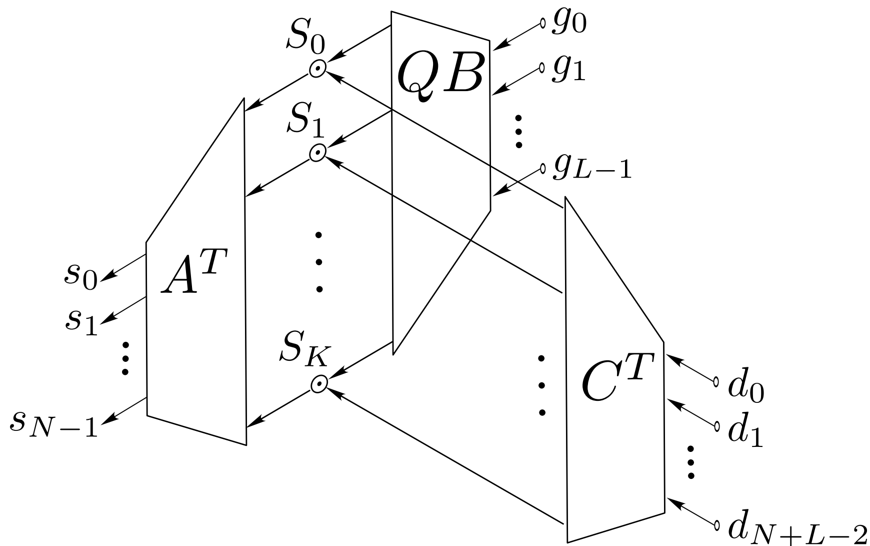


Figure 2.7: General form of tap-filter algorithm (adapted from McClellan and Rader [1979]), where d is the input feature map, g the filter weights, and s the output feature map at the filter output.

This can be used to construct a short filter section. Using a Toom Cook linear convolution from Figure 2.5:

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} g_0 & & & \\ & \frac{1}{2}(g_0+g_1) & & \\ & & \frac{1}{2}(g_1-g_1) & \\ & & & g_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \end{bmatrix}.$$

Then, by Equation 2.3, we replace A by C^t and C by A^t, and invert the index order of s and d:

$$\begin{bmatrix} s_2 \\ s_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} g_0 & & & \\ & \frac{1}{2}(g_0+g_1) & & \\ & & \frac{1}{2}(g_1-g_1) & \\ & & & g_2 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \end{bmatrix}. \tag{2.4}$$

that is a $\{M_2, N_4, L_3, K_4\}$ where M is the output of filter, N is the input data sequence. In filter form we change the input by the output size.

The algorithm can be used to compute an unending stream of filter outputs, two outputs at a time, by writing

$$\begin{bmatrix} s_{r+2} \\ s_{r+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} g_0 & & & \\ & \frac{1}{2}(g_0+g_1) & & \\ & & \frac{1}{2}(g_1-g_1) & \\ & & & g_2 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_{r+3} \\ d_{r+2} \\ d_{r+1} \\ d_r \end{bmatrix}.$$

The offset r, or sliding window, for each iteration or batch of filtered data, should consistently equal the sample size N.

2.4 Nested

Multidimensional fast convolution can be obtained by applying a one-dimensional fast convolution algorithm successively along each dimension of the data. Rather than deriving a separate algorithm for each dimension, the same 1D scheme is "nested" inside itself. Let

$$s = A^T[(QBg) \odot (C^T d)] \tag{2.5}$$

denote a generic 1D Winograd-type convolution in filter form (cf. Equation 2.3). In the nested approach, this algorithm is first applied along one dimension of the 2D data (for example, along the rows) and then applied again along the other dimension (the columns), with appropriate permutations of indices between the two stages [Winograd, 1978].

Operationally, this means that the matrices C, Q, B and A associated with the 1D algorithm act twice on the same data: once when the algorithm is applied to each row, and once when it is applied to each column of the intermediate results. By reordering the indices and grouping the resulting operations, this double application can be expressed compactly as a two-dimensional transform, without introducing a second, independent set of matrices.

[Lavin and Gray, 2016] presents in Equation 2.6 the 2D nested fast convolution in a matrix-multiplication form suitable for filter interpretation. First, each row of array d is multiplied by matrix C resulting in matrix δ, and each row of array g by matrix QB resulting in matrix γ. Next, δ is multiplied by the columns of matrix C and γ is multiplied by the columns of matrix QB. These arrays are then combined using the Hadamard product that produces S. Finally, a reduction process multiplies S by each row of A and by each column of A, producing an output array s of size N^2 [Blahut, 2010].

$$s = A^T\left\{\left[(QB)g(QB)^T\right] \odot (C^T dC)\right\}A. \tag{2.6}$$

In the next example, we use two Toom-Cook of $\{M_4, N_2, L_3, K_4\}$ (presented in Equation 2.4), binding both using the Nested method and building a $\{M_{4\times4}, N_{2\times2}, L_{3\times3}, K_{4\times4}\}$ [1]:

The initial data sequence is given as:

$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \\ d_4 & d_5 & d_6 & d_7 \\ d_8 & d_9 & d_{10} & d_{11} \\ d_{12} & d_{13} & d_{14} & d_{15} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}. \tag{2.7}$$

The kernel can be expressed as:

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \\ g_3 & g_4 & g_5 \\ g_6 & g_7 & g_8 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}. \tag{2.8}$$

Subsequently, we perform a transformation of the kernel g into the matrix G

$$G = (QB)g(QB)^t.$$

Initially, we multiply the kernel g by the transformation matrix along the rows, resulting in γ:

$$\begin{aligned} \begin{bmatrix} \gamma_0 & \gamma_1 & \gamma_2 & \gamma_3 \\ \gamma_4 & \gamma_5 & \gamma_6 & \gamma_7 \\ \gamma_8 & \gamma_9 & \gamma_{10} & \gamma_{11} \end{bmatrix} &= \begin{bmatrix} g_0 & g_1 & g_2 \\ g_3 & g_4 & g_5 \\ g_6 & g_7 & g_8 \end{bmatrix} \left(\begin{bmatrix} -1 \\ \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix} \odot \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \right)^t \\ &= \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \begin{bmatrix} -1 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & \frac{3}{2} & \frac{1}{2} & 2 \\ -3 & 6 & 2 & 5 \\ -6 & \frac{21}{2} & \frac{7}{2} & 8 \end{bmatrix}. \end{aligned} \tag{2.9}$$

[1] A supplementary video, **"Nested Algorithm for IF Fast Convolution"**, gives a high-level, stepwise visualization of the nested pipeline described above and is available at https://youtu.be/J5lgvCpZV64. Another short video, **"2D Fast Convolution Sliding-Window Mathematics"**, concisely illustrates a two-dimensional fast convolution implemented with a sliding window: https://youtu.be/Toe6KPWF5RQ.

Finally, we multiply γ by the transformation matrix along the columns to obtain G:

$$\begin{aligned}
\begin{bmatrix} G_0 & G_1 & G_2 & G_3 \\ G_4 & G_5 & G_6 & G_7 \\ G_8 & G_9 & G_{10} & G_{11} \\ G_{12} & G_{13} & G_{14} & G_{15} \end{bmatrix} &= \left(\begin{bmatrix} -1 \\ \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix} \odot \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} \gamma_0 & \gamma_1 & \gamma_2 & \gamma_3 \\ \gamma_4 & \gamma_5 & \gamma_6 & \gamma_7 \\ \gamma_8 & \gamma_9 & \gamma_{10} & \gamma_{11} \end{bmatrix} \\
&= \begin{bmatrix} -1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & \frac{3}{2} & \frac{1}{2} & 2 \\ -3 & 6 & 2 & 5 \\ -6 & \frac{21}{2} & \frac{7}{2} & 8 \end{bmatrix} \\
&= \begin{bmatrix} 0 & -\frac{3}{2} & -\frac{1}{2} & -2 \\ -\frac{9}{2} & 9 & 3 & \frac{15}{2} \\ -\frac{3}{2} & 3 & 1 & \frac{5}{2} \\ -6 & \frac{21}{2} & \frac{7}{2} & 8 \end{bmatrix}.
\end{aligned} \tag{2.10}$$

At this stage, we proceed to transform the sequence data d into matrix D

$$D = C^t dC.$$

We initially multiply the kernel d by the transformation matrix along the rows, resulting δ:

$$\begin{bmatrix} \delta_0 & \delta_1 & \delta_2 & \delta_3 \\ \delta_4 & \delta_5 & \delta_6 & \delta_7 \\ \delta_8 & \delta_9 & \delta_{10} & \delta_{11} \\ \delta_{12} & \delta_{13} & \delta_{14} & \delta_{15} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 2 \\ 2 & 11 & 1 & 2 \\ 2 & 19 & 1 & 2 \\ 2 & 27 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Subsequently, we determine δ by multiplying with the transformation matrix in the column direction, which results in D:

$$\begin{bmatrix} D_0 & D_1 & D_2 & D_3 \\ D_4 & D_5 & D_6 & D_7 \\ D_8 & D_9 & D_{10} & D_{11} \\ D_{12} & D_{13} & D_{14} & D_{15} \end{bmatrix} = \begin{bmatrix} 0 & 16 & 0 & 0 \\ 4 & 30 & 2 & 4 \\ 0 & 8 & 0 & 0 \\ 0 & 16 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 & 1 & 2 \\ 2 & 11 & 1 & 2 \\ 2 & 19 & 1 & 2 \\ 2 & 27 & 1 & 2 \end{bmatrix}.$$

A Hadamard product $S = G \odot D$ is then computed, which is the pointwise multiplication of the transformed kernel and the transformed data sequence

$$\begin{bmatrix} 0 & -24 & 0 & 0 \\ -18 & 270 & 6 & 30 \\ 0 & 24 & 0 & 0 \\ 0 & 168 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\frac{3}{2} & -\frac{1}{2} & -2 \\ -\frac{9}{2} & 9 & 3 & \frac{15}{2} \\ -\frac{3}{2} & 3 & 1 & \frac{5}{2} \\ -6 & \frac{21}{2} & \frac{7}{2} & 8 \end{bmatrix} \odot \begin{bmatrix} 0 & 16 & 0 & 0 \\ 4 & 30 & 2 & 4 \\ 0 & 8 & 0 & 0 \\ 0 & 16 & 0 & 0 \end{bmatrix}.$$

Finally, we perform an inverse transformation of the Hadamard Product $s = A^tSA$

We first multiply the kernel S by the inverse transformation matrix along the rows, yielding σ:

$$\begin{bmatrix} \sigma_0 & \sigma_1 \\ \sigma_2 & \sigma_3 \\ \sigma_4 & \sigma_5 \\ \sigma_6 & \sigma_7 \end{bmatrix} = \begin{bmatrix} -24 & -24 \\ 258 & 294 \\ 24 & 24 \\ 168 & 168 \end{bmatrix} = \begin{bmatrix} 0 & -24 & 0 & 0 \\ -18 & 270 & 6 & 30 \\ 0 & 24 & 0 & 0 \\ 0 & 168 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}.$$

Ultimately, we determine σ and then multiply by the inverse transformation matrix in the column direction to obtain s

$$\begin{bmatrix} s_0 & s_1 \\ s_2 & s_3 \end{bmatrix} = \begin{bmatrix} 258 & 294 \\ 402 & 438 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} -24 & -24 \\ 258 & 294 \\ 24 & 24 \\ 168 & 168 \end{bmatrix}.$$

Figure 2.8 compares, for a fixed 3×3 kernel ($L_{3\times3} = 9$), how the total number of multiplications grows with the output tile size when we use the naïve 2D convolution (Equation (A.14)) or the 2D Toom–Cook filter (Equation (2.6)). The blue curve corresponds to the direct method, where each $N \times N$ output block requires $K_{\text{naïve}} = 9N$ scalar multiplications, reflecting the fact that every output pixel needs 3×3 MACs. The red curve shows the Toom–Cook nested scheme, where the effective number of multiplications is $K_{\text{TC}} = (\sqrt{N}+2)^2$, because a $N \times N$ block is produced by binding two 1D Toom–Cook transforms of length $\sqrt{N}$ with two extra rows/columns of transform overhead. For small tiles, such as 2×2 and 3×3, both methods are close, but as N grows the linear behaviour $K_{\text{naïve}} \propto N$ quickly dominates the quadratic term in $(\sqrt{N} + 2)^2$. This illustrates that Toom–Cook can substantially reduce the number of multiplications per output tile compared with the naïve implementation, even though the transform stages introduce additional additions that must later be balanced at the architectural level.

Conclusion

The mathematical tools presented in this chapter provide a common foundation for the remainder of the Dissertation. The convolution models, fast convolution algorithms, and multidimensional constructions introduced here are used in Chapter 3 to structure the survey

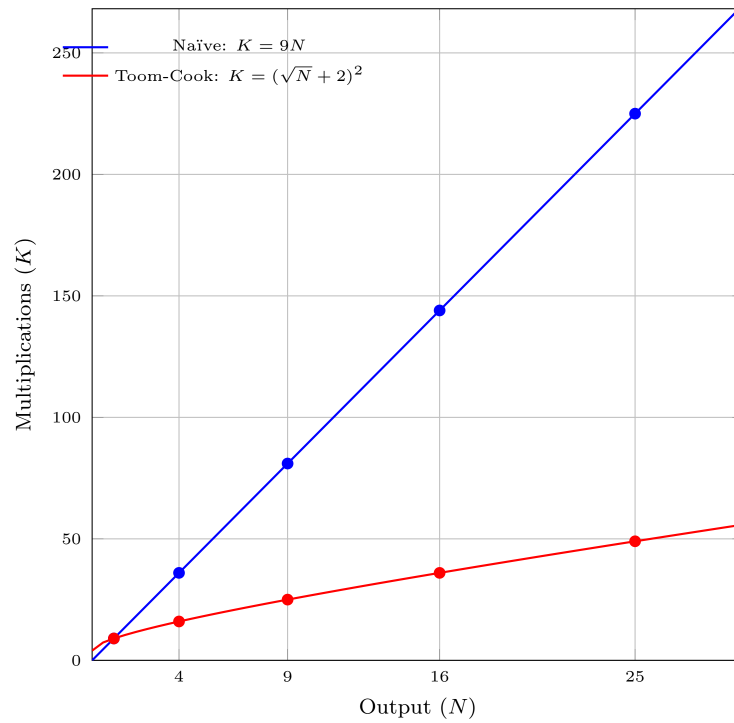


Figure 2.8: Multiplications required for 2D-naïve (Eq. A.14) and 2D Toom-Cook Filter (Eq. 2.6) convolutions for $L_{3\times3=9}$.

of related work, in particular, to define the families and complexity trends that organize the comparison between naïve and fast methods, in Chapter 4 to derive the specific fast convolution methods explored in this Dissertation, and in Chapters 5 to 7 to analyze the behavior and efficiency of the proposed software and hardware implementations.

3. RELATED WORK

Building on the mathematical background introduced in Chapter 2, the related work presented in this chapter focuses on how convolution is **implemented in hardware**, rather than on the derivation of fast algorithms themselves. Here, we concentrate on architectural choices, dataflows, and memory organizations used to accelerate convolution layers on FPGAs and ASICs, and on how different works exploit the structure of fast convolution algorithms in practical accelerator designs. Acronyms such as CNN, FPGA, ASIC, and RTL follow the notation summarized in the List of Acronyms on page 12.

This chapter is organized as follows.

- Section 3.1 reviews generic CNN accelerator architectures based on direct convolution and **im2col**+GEMM mappings (General Matriz Multiplication).
- Section 3.2 surveys accelerators for naïve convolution and the associated design-space exploration frameworks.
- Section 3.3 reviews the literature related to Winograd-based fast convolution accelerators.
- Section 3.3.7 summarizes the main gaps and motivates the unified algorithm-hardware framework developed in the following chapters.

3.1 Generic hardware architectures for convolution

Early CNN accelerators typically implement convolution either as direct sliding-window computation or by transforming convolution into matrix multiplication via the **im2col**+GEMM approach. In both cases, the hardware organization is often based on arrays of multiply–accumulate (MAC) units, together with line buffers or on-chip SRAM blocks to exploit spatial and temporal reuse of feature maps and weights, as seen in several FPGA and ASIC designs for CNN inference [Silvano et al., 2025; Reuther et al., 2020; Chen et al., 2020]. A typical pattern is to adopt an output-stationary or weight-stationary dataflow, depending on whether partial sums or filter coefficients are kept in local registers.

FFT-based accelerators follow a different strategy: feature maps and filters are transformed to the frequency domain using 1D or 2D FFTs, element-wise multiplications are performed, and then inverse FFTs reconstruct the results. Hardware implementations of FFT engines organize FFT engines as streaming pipelines or small systolic arrays. They must carefully schedule memory accesses to support the large intermediate tensors in the frequency domain. While FFT-based designs excel for large kernels, their transform overhead and complex-number arithmetic make them less attractive for the small 3×3 filters that dominate modern CNNs. Comparative studies and design-space exploration works [Lavin and Gray, 2016; Liang et al.,

2020; Kala et al., 2018; Zhao et al., 2019] indicate that, for typical vision workloads on FPGA, FFT is usually beneficial only for larger kernels.

In contrast, Winograd-type methods tend to offer a better trade-off for small kernels. These results reinforce the idea that Winograd-type algorithms better serve small-kernel 2D convolution than FFT on current FPGA platforms. These generic convolution accelerators serve as a baseline for comparing later designs, including those that employ fast convolution algorithms such as Winograd.

In the rest of this chapter, we focus on how specialized accelerators depart from these generic patterns to support naïve and fast convolution algorithms.

From the perspective of this Dissertation, these generic architectures and dataflow taxonomies provide the hardware context into which our fast convolution algorithms must fit. In Chapter 4, we explicitly connect the algebraic structure of Winograd-type methods and their multidimensional extensions to these architectural dimensions so that algorithmic choices can be evaluated using the same metrics and vocabulary adopted in this section.

3.2 Accelerators for naïve convolution

Although based on a conceptually simple convolution algorithm, a naïve convolution accelerator serves as the baseline for evaluating acceleration techniques. Understanding these accelerators enables the establishment of reference parameters and quantifies the performance gains achieved relative to faster convolution techniques. This comparison provides a clear, objective demonstration of the performance, energy consumption, and area improvements offered by each method.

While not focused on minimizing multiplications, these accelerators emphasize memory access strategies, such as data management and data reuse during processing. This approach aims to optimize data transfer between memory modules and the processing core through mechanisms like buffering and prefetching, thereby minimizing the impact of access latencies and redundant operations. When applied to fast convolution accelerators, these strategies can enhance performance, energy efficiency, and area utilization.

Analyzing these accelerators highlights the trade-offs involved in hardware design for convolution operations. Through the study of traditional methods, it is possible to define how the relationships between the number of multiplications and additions impact designs, justifying the trade-offs proposed by fast algorithms without compromising the system's overall efficiency.

A key reference for this class of accelerators is the work by Juracy et al., who systematically study convolutional hardware accelerators that implement naïve convolution and propose a framework to model, evaluate, and explore their design space [Juracy et al., 2021b,a, 2022; Juracy, 2022; Juracy et al., 2023]. Together, these works provide (*i*) high-level analytical and simulation-based models to estimate performance, power, and area (PPA) of convolution

accelerators with a RTL implementation; (*ii*) a taxonomy to classify academic and industrial designs according to their architectural choices; and (*iii*) a framework for fast architecture exploration of CNN accelerators. The original experiments and models target convolution layers with stride 2 as the dominant configuration; a subsequent study by Silva [2023] extends this exploration to stride 1 at the RTL level, enabling a finer-grained analysis of more common CNN layers. In this Dissertation, we adopt Juracy's analytical framework and use the RTL environment of Silva [2023] as the main reference baseline for quantitative comparisons with the proposed accelerators.

Based on a survey of academic and industrial designs, Juracy proposes a taxonomy for convolution accelerators that exposes the main architectural decisions relevant to CNN workloads [Juracy, 2022; Juracy et al., 2023]. For academic accelerators implementing naïve convolution, the taxonomy classifies designs along the following dimensions:

- **Array style**: whether the compute array is a vector (1D) or a matrix (2D), and whether it uses a systolic method to feed data to the arithmetic blocks.
- **Convolution technique**: how multiplication is executed, distinguishing naïve convolution from fast algorithms (e.g., Winograd or FFT).
- **Accelerator goal**: the primary design objective, such as maximum performance, minimal area, low power consumption, or a balanced trade-off among these metrics.
- **Dataflow type**: how values are loaded from memory into internal buffers and propagated through the compute array.
- **Word size**: the bit-width of the data being processed, which affects both numerical precision and resource usage.
- **Data format**: whether data is represented in fixed-point or floating-point format and how this choice interacts with the target technology and performance goals.

In addition to these architectural dimensions, Juracy's evaluation also refines the classification of dataflows into a set of categories [Juracy et al., 2022]. The most commonly cited are:

- **No Local Reuse (NLR)**: an approach in which data is stored only in external memory; the accelerator maintains no on-chip buffers, and all data is read from memory in every iteration.
- **Weight Stationary (WS)**: a strategy in which the weights are kept stationary in local registers or buffers, while the input feature map (IFMAP) is read multiple times and partial sums are accumulated externally or in smaller buffers.
- **Output Stationary (OS)**: a scheme in which partial sums for the output feature map (OFMAP) remain stationary in registers or small buffers within the processing elements. At the same time, IFMAP and weights are streamed through the array.
- **Row Stationary (RS)**: a dataflow that reuses IFMAP rows and weights simultaneously, often storing rows of IFMAP and corresponding weights into registers to reduce both memory traffic and energy consumption.

- **Fine Grain (FG)**: an approach in which IFMAP, weights, and partial sums are partitioned and stored across multiple levels of internal buffers, so that only a reduced subset of values is fetched from external memory each time.

Beyond the taxonomy, the same research introduces a high-level modeling framework and a set of tools to estimate hardware metrics of convolution accelerators with an RTL implementation [Juracy et al., 2021b,a, 2022]. Starting from a CNN description, the framework maps convolution layers onto a parameterized accelerator template and uses analytical models and a system-level simulator to predict performance, power, and area. These estimates feed back into an architecture exploration loop [Juracy, 2022], enabling designers to evaluate different array sizes, dataflows, word sizes, and memory organizations for naïve-convolution accelerators before committing to low-level design. This modeling and PPA-estimation capability is significant for this Dissertation, as it illustrates how quantitative evaluation of architectural choices can be integrated into an algorithm–hardware co-design flow.

Establishing the general architecture of these accelerators, Juracy proposes a generic template that captures the main modules required to build a CNN accelerator based on naïve convolution [Juracy, 2022]. As illustrated in Figure 3.1, the architecture is organized around a compute array (2D MAC systolic array), surrounded by four main structures: an input feature map (IFMAP) buffer, a weight buffer, an output feature map (OFMAP) buffer, and control logic that orchestrates data movement and computation. The IFMAP buffer stores incoming data according to the chosen dataflow, often using line buffers or tiling schemes to maximize spatial and temporal reuse. The weight buffer keeps filter coefficients close to the compute array, which is essential for weight-stationary dataflows. The OFMAP buffer collects partial and final results, allowing output-stationary dataflows to hold partial sums locally while new input tiles stream through the array. Finally, the controller implements the address generation, scheduling, and handshake logic that manages memory accesses and the sequencing of convolution operations.

From the perspective of this Dissertation, Juracy's taxonomy, modeling framework, and generic architecture provide a baseline for dense, naïve-convolution accelerators. In the following sections, we extend this perspective to fast-convolution algorithms of the Winograd family, which introduce additional degrees of freedom at the algorithmic level but should still be understood and evaluated in terms of the exact architectural dimensions of array style, dataflow, word size, and buffering organization.

These naïve convolution accelerators and dataflow classifications provide the architectural vocabulary used in this Dissertation. In later chapters, we reuse the same concepts (e.g., output-stationary, weight-stationary, row-stationary) to describe how Winograd-based and multidimensional fast convolution algorithms are mapped to hardware.

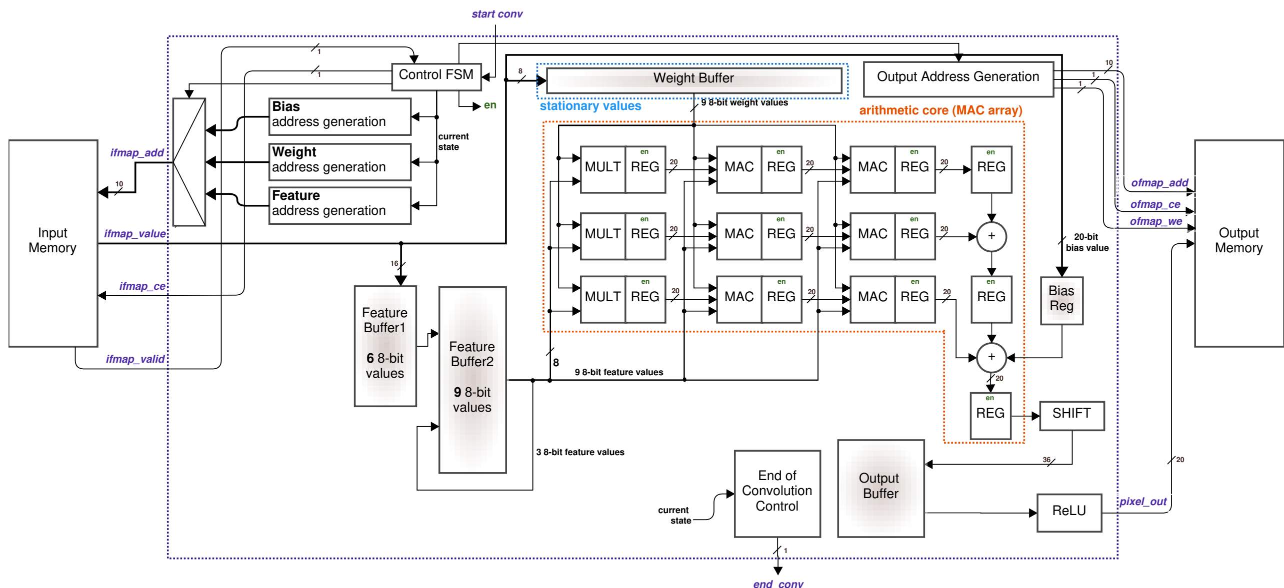


Figure 3.1: Generic architecture and the modules required to build the naïve convolutional accelerators (adapted from Juracy [2022]).

3.3 Winograd-based fast convolution accelerators

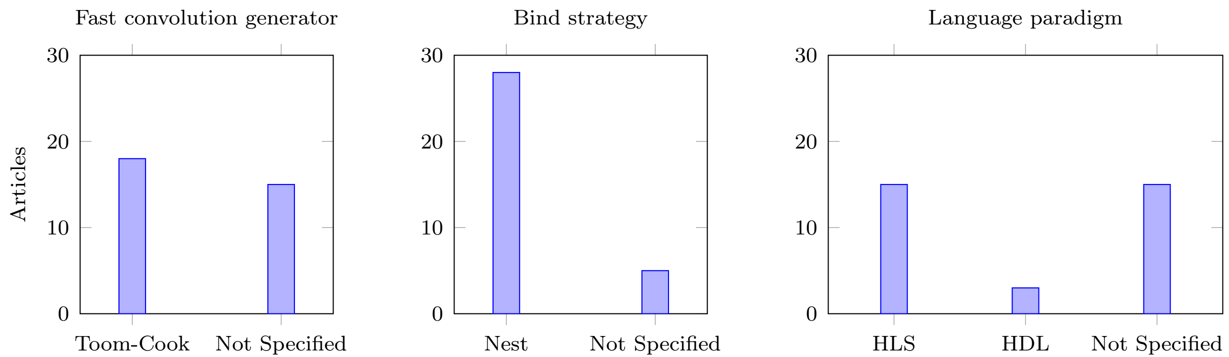
This section reviews the literature on fast convolution accelerators. This review identifies gaps and opportunities in developing this type of accelerator.

3.3.1 Winograd Implementation on Hardware

This review analyzes the application of one- and two-dimensional fast convolution methods in hardware accelerators. This analysis does not address other characteristics, such as memory access or state machine architectures. The search was conducted in Scopus[1]. A total of 135 articles were considered, of which 30 were selected for analysis.

The field of fast convolution is quite extensive, which makes it challenging to conduct a systematic review that relies solely on the term *Winograd*. Given that the objective of this work is to accelerate convolution in hardware, we narrowed the research scope to focus exclusively on the terms *Winograd* and *FPGA*. We use *Winograd* because it is the principal method in fast convolution. *FPGAs* are devices frequently employed for hardware acceleration, offering lower energy consumption than GPUs and a shorter development time than ASICs. To identify methods for fast convolution, we investigated how these methods are presented, either as matrices or via generation methods such as Toom-Cook or Winograd modular polynomials. To identify the multidimensional methods in the papers, we sought the Lavin equation and schematics or flowcharts similar to those presented by Blahut. A detailed, per-paper summary of the reviewed works is presented in Table B.1 in Chapter B.

[1] Search string: `winograd AND fpga`



> Each plot reports how many articles explicitly fall into each category, including a "Not specified" bar that counts papers where the corresponding field is not specified in the paper.

Figure 3.2: Summary histograms for the three main metadata columns in the systematic review B.1: fast-convolution generator, binding strategy, and language paradigm - HDL or HLS (High-Level Synthesis).

The three graphs in Figure 3.2 show that the current literature on Winograd-based accelerators is concentrated on Toom–Cook generators, nested bindings, and HLS (High-Level Synthesis). In each subplot, the dominant bar indicates that most papers make the same design choice, while the "Not specified" bars reveal that a sizeable subset of works does not document how the fast algorithm is generated, how it is bound to convolution layers, or which hardware language paradigms is used. This combination of convergence and missing detail motivates the more explicit, parameterized treatment of generators, bindings, and implementation flows developed in the remainder of this Dissertation.

Most fast convolution techniques utilize either the minimum filter or the Toom-Cook method. These methods are widely adopted due to their simplicity of implementation. The matrices associated with the minimum filter are readily available in the literature. The Toom-Cook method is also favorable for implementation, as it is closely related to Lagrange interpolation, a well-known mathematical procedure.

Only one multi-dimensional convolution method was identified: the Nested method. Generally, the 2D convolution approach is presented using the Lavin equation, expressed in papers as a single line or broken down into several equations. This method is employed because Lavin simplifies the understanding of the process by describing it as a matrix multiplication. No other methods were found beyond this.

This systematic review revealed that no article uses Winograd's modular polynomial method. Additionally, no works proposing alternative approaches to the Nested method were identified.

Figure 3.3 presents the distribution of articles across various Winograd output tile sizes $(m \times m)$ with 3×3 kernel size $F(m \times m, 3 \times 3)$. The data reveal that the 2×2 category has the highest representation, followed by 4×4. In contrast, categories such as 3×3 and 8×8 have fewer than four articles each. Complementing this high-level histogram, Table B.2

in Chapter B lists, for each individual article, which output sizes $F(m \times m, 3 \times 3)$ are explicitly implemented.

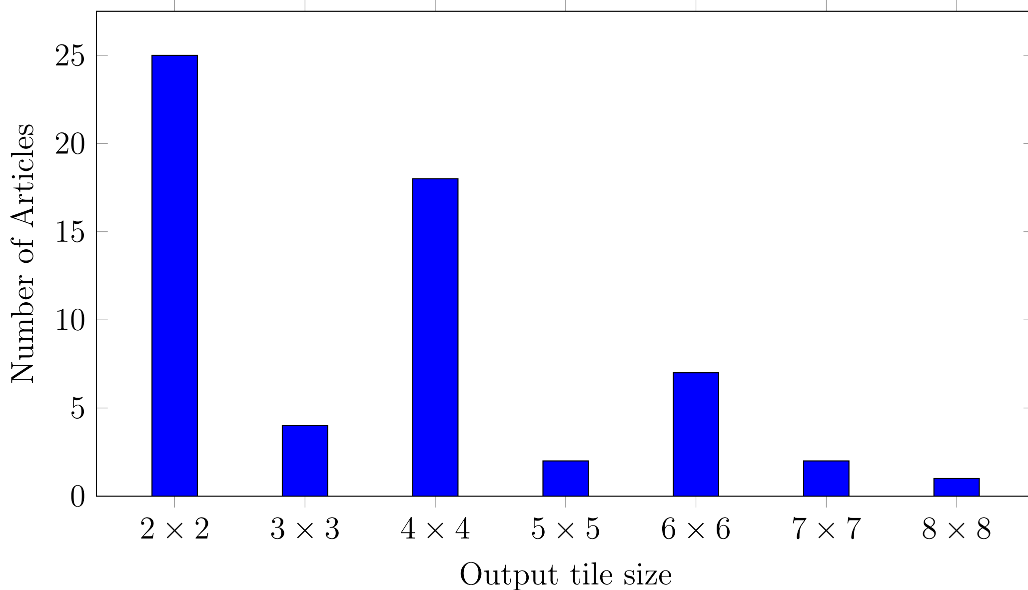


Figure 3.3: Distribution of reviewed articles across Winograd output tile $(m \times m)$ for 3×3 kernel.

The dominance of the minimal Winograd filter 2×2 in our systematic review can be explained from both algorithmic and hardware perspectives. When fast convolutions are generated via Toom–Cook, increasing the output tile size beyond $m = 2$ rapidly increases the number and the magnitude of the coefficients in the transform matrices A and C. In practice, larger Toom–Cook schemes produce many entries that are neither 0 nor 1, which significantly increases the number of additions and constant multiplications required to implement the input and output transforms in hardware. Even when constant multipliers can be implemented as shift-and-add networks, the associated area, routing, and timing costs grow quickly and may offset the savings in arithmetic operations obtained in the convolution core itself. In contrast, the Toom–Cook–2 case yields transform matrices A and C with only 0 and 1 entries, allowing the transforms to be implemented with wiring and additions, with minimal impact on area and the critical path. As a result, the combination of a 3×3 kernel with a 2×2 output tile becomes a practical sweet spot for many designers.

It is important to note that there exist families of fast convolution algorithms that enforce elementary coefficient sets in the transform matrices, such as polynomial-modular Winograd schemes that produce only 0 and 1 entries in A and C. From a hardware perspective, such algorithms are desirable, as they promise minimal transform cost and excellent suitability for fixed-point implementations. However, our systematic review did not find any published accelerator that employs these polynomial-modular Winograd variants. This gap reinforces one of the motivations of this Dissertation: to revisit the broader space of $F(m \times m, r \times r)$ algorithms with hardware cost in mind, and to investigate whether alternative formulations with simple

transform coefficients can lead to more efficient 2D and 3D convolution accelerators than the current de facto choice $F(2 \times 2, 3 \times 3)$.

In summary, most Winograd-based accelerators implement three main stages (input transform, element-wise products, and output transform) as deeply pipelined datapaths, sometimes using systolic-style PE arrays, and must balance transform overhead, buffer sizes, and DSP utilization.

3.3.2 CNN-level Winograd accelerators on FPGA/ASIC

Beyond implementing isolated Winograd kernels, several works integrate Winograd-based convolution into full CNN accelerators. Early FPGA studies evaluate and compare fast convolution algorithms, including Winograd and FFT, under realistic CNN workloads, reporting performance and resource trade-offs for different tile sizes and kernel shapes [Lu et al., 2017; Liang et al., 2020; Kala et al., 2018; Zhao et al., 2019]. These results show that Winograd is desirable for 3×3 convolutions, which are pervasive in modern networks, provided that the additional additions and buffering are carefully managed in hardware.

A first group of accelerators embeds Winograd as a fixed-function block inside a more conventional convolution engine. For example, Yu et al. [2020]; Zhuge et al. [2018]; Ahmad and Pasha [2020] implement Winograd transforms and element-wise products as dedicated datapaths, while still relying on GEMM-like structures or other convolution modes for layers that are not Winograd-friendly. In these designs, input feature maps and filters are transformed by small specialized units, and the transformed tiles are then processed by MAC arrays that can also execute direct or matrix-multiplication-based convolution.

A second group of works explicitly targets Winograd as the primary computation mode for most convolution layers in a CNN. Architectures such as WRA [Yang et al., 2019] or kernel-sharing Winograd systolic arrays [Liu et al., 2021; Wang et al., 2020] adopt systolic-style PE arrays operating in the Winograd domain, where each PE performs element-wise multiplications and partial sums for a subset of tiles. The input and output transforms are deeply pipelined, and the architecture exploits high data reuse by keeping tiles in local registers or small SRAM blocks while streaming different channels and filters through the array.

Finally, unified Winograd–GEMM architectures [Kala et al., 2019b,a] demonstrate that the same PE array can operate either as a Winograd engine or as a standard matrix-multiplication core. In these designs, most of the Winograd-specific logic is confined to the input and output transform units and to the data packing in the PE array. At the same time, the underlying MAC structure remains unchanged. This approach is desirable for CNNs that contain a mix of Winograd-friendly layers (e.g., 3×3 convolutions) and layers that are better served by direct or GEMM-based implementations.

3.3.3 Application-specific and multidimensional designs

Several Winograd-based accelerators target specific application domains, such as real-time object detection, face recognition, and generative models. For object detection, FPGA accelerators tuned to YOLO-like networks use Winograd-based convolutions to reduce DSP usage and improve throughput under strict latency and power constraints [Bao et al., 2020; Lv et al., 2020; Nugroho et al., 2021]. These designs adopt narrow but deeply pipelined PE arrays, and dimensioned line buffers and on-chip buffers to sustain streaming inputs and minimize off-chip memory traffic. Face-recognition-oriented accelerators, such as Zhuge et al. [2018], combine Winograd and other convolution variants to build end-to-end pipelines that integrate convolution, fully connected layers, and activation functions.

Winograd-style fast convolution has also been applied to transposed convolutions in GANs and super-resolution networks. In this context, the main challenge is to handle upsampling patterns and overlapping output tiles. For instance, Di et al. [2020] proposes an FPGA architecture for Winograd-transformed transposed convolution in GANs, introducing additional control logic and buffering schemes to merge overlapping tiles on the fly while reusing the same PE array as for standard convolution.

Finally, a smaller set of designs considers multidimensional convolutions, especially 3D CNNs for video or volumetric data. Flexible accelerators [Deng et al., 2021] and template-based architectures for 2D/3D CNNs [Shen et al., 2018, 2020] employ hierarchical tiling across depth and spatial dimensions, as well as multi-level buffering, to cope with the increased data volume and on-chip memory requirements. Even when the underlying fast algorithm is conceptually a multidimensional extension of 2D Winograd, mapping it efficiently to hardware requires careful decisions about how tiles are scheduled, how feature maps are buffered, and how intermediate transformed data is reused across channels and depth slices. Compared with the large number of 2D accelerators focused on 3×3 kernels, these 3D and transposed-convolution designs remain relatively rare, suggesting an opportunity for further exploration.

3.3.4 Template-based and framework-driven convolution accelerators

Beyond individual accelerator instances, several works propose template-based architectures or full frameworks that automatically generate convolution accelerators from a CNN description. Examples include generic FPGA frameworks for CNNs [DiCecco et al., 2016; Xiao et al., 2017] and more recent hybrid DNN frameworks such as HybridDNN [Ye et al., 2020], which aim to select and configure accelerator templates under performance and resource constraints. These frameworks typically start from a parameterized hardware template—for example, a systolic array with configurable dimensions, a set of line buffers and on-chip SRAM

blocks, and a configurable number of MAC units—and then perform design space exploration to select the best tiling, unrolling, and dataflow options for a given device.

Some frameworks support fast convolution algorithms such as Winograd or FFT, either directly or through specialized accelerator blocks [Liang et al., 2020; Zhao et al., 2019]. In this case, the hardware template typically includes optional transform blocks and supports multiple dataflows, enabling the same PE array to execute direct, GEMM-based, or Winograd-based convolution. The framework then decides, on a per-layer basis, whether it is beneficial to incur the transform overhead in exchange for fewer multiplications. From a hardware perspective, this requires flexible control logic, configurable memory-addressing units, and, in some cases, reconfigurable interconnects to route data through different computation pipelines.

However, even these frameworks tend to treat the underlying fast algorithms as a small set of fixed building blocks (e.g., a few pre-defined Winograd tiles) rather than as a continuous design space. As a consequence, their hardware templates are tuned for a handful of convolution shapes (typically 2D, small kernels, unit stride), and support for 3D, transposed, or depthwise convolution is either limited or delegated to separate, less optimized accelerators. This suggests that there is still room for frameworks that co-design algorithmic and architectural parameters from the beginning, rather than simply selecting from a small set of pre-defined algorithm/hardware configurations.

3.3.5 Winograd and Kronecker Product

This review also investigated the application of the Winograd convolution with the Kronecker product technique to facilitate multidimensional convolution. In this analysis, we seek to understand how the Kronecker technique has been implemented and whether it has been implemented in an FPGA. The search yielded nine articles published after 2015, of which only two specifically address the topic of interest.

The search was conducted in the Scopus[2] and IEEExplore[3] databases. The search in Scopus yielded five papers, of which only one concerns fast convolution Cheng and Parhi [2020]. Similarly, the search in IEEExplore returned five articles; however, only two discuss fast convolution: once again, Cheng and Parhi [2020] and Wang et al. [2018].

Cheng and Parhi [2020] employs Toom-Cook (3, 3) (Figure 2.6) and manual inspection factorization (Section 4.2) as examples for 1D fast convolution. The work uses a set of operations that incorporates the Kronecker product and the identity matrix to avoid the need for two matrix multiplications, as in the nested method, thereby performing only a single matrix multiplication. However, the work remains at the algorithmic level and does not report any software or hardware implementation.

[2] Search string: `(TITLE-ABS-KEY (winograd) AND TITLE-ABS-KEY (kronecker)) AND PUBYEAR > 2015`.

[3] Search string: `("All Metadata":winograd) AND ("Full Text & Metadata":kronecker)` for articles published after 2015.

Wang et al. [2018] is the only work that employs the Kronecker product to bind fast convolution implementations executed on an FPGA. This paper proposes a Reconfigurable Convolution Core (RCC) that employs fast convolution algorithms to implement five mainstream convolution kernel sizes efficiently. The architecture utilizes the Toom-Cook (2, 2) (Figure 2.5) and manual inspection factorization (Section 4.2). The method incorporates the Kronecker product alongside an identity matrix to facilitate the binding of the two fast convolutions.

In the context of this Dissertation, these works confirm that Kronecker-based formulations are promising for structuring multidimensional fast convolution, but they also highlight two gaps: the lack of a general design-space view that goes beyond a few fixed kernel sizes and the absence of an integrated framework that connects Kronecker-based algorithms to systematic hardware design and evaluation. These gaps motivate the multidimensional Kronecker method and its hardware realization discussed in Chapters 4 and 6.

3.3.6 Other classic fast convolution methods

In the previous section, we reviewed papers implementing fast convolution in hardware, highlighting that the adopted techniques are limited to the minimum filter, the Toom–Cook method, and the nested approach. Additionally, only two papers use the Kronecker product for multidimensional convolution. Given the scarcity of works, our current objective is to deepen the investigation by seeking citations from other authors, excluding Winograd, who have contributed to fast convolution. We reference works by Agarwal and Burrus, Nussbaumer, Selesnick and Burrus, and Tolimieri and Breitzman. We continue our search by considering publications from 2016 onward, the year Lavin's work was published, focusing on studies aimed at accelerating CNNs in hardware. Works prior to this period were addressed in the concepts chapter.

Agarwal and Cooley [1977] (Section A.8) developed multidimensional convolution techniques, with seven citations since 2015, of which only three [Jiang et al., 2023; Pitas, 2021; Nussbaumer, 1982] address the topic of multidimensional fast convolution.

Tolimieri et al. [2013] (Section 4.3) discusses the transformation of circular convolution into linear convolution. Among its post-2015 citations, only Alam et al. [2022] explicitly addresses the topic of fast convolution.

Selesnick and Burrus, who developed the technique of CRT-Based Cyclic Convolution Algorithms for Prime Powers, do not have citations related to Convolutional Neural Networks (CNN) from the articles Selesnick and Burrus [1993, 1994]; Burrus and Selesnick [1995]; Selesnick and Burrus [1996].

Nussbaumer (Section A.8) develops several techniques for fast convolution, including the split-nesting method. In his book [Nussbaumer, 1982], searches focused on the term "split nesting" reveal only a small number of recent citations directly related to our topic. In partic-

ular, the Scopus and IEEExplore databases identify Pitas [2021] as the only recent work that connects split nesting to multidimensional fast convolution in a way that is relevant to this Dissertation.

Breitzman and Johnson [2004] has six citations, only two of which are relevant to our work: Alam et al. [2022]; Ju and Solomonik [2019].

All these works remain at the algorithmic or software level, and none of them provides a complete algorithm–hardware co-design framework or a physically implemented accelerator.

3.3.7 Similar works

These works are similar to our proposal in that they seek general formulations or optimizations for multidimensional convolutions. However, they stop short of providing a hardware-oriented framework or concrete accelerator implementations.

Alam et al. [2022] employs the Toom-Cook algorithm and utilizes a nested approach for binding; however, it does not include any implementation in hardware. Similarly, Jiang et al. [2023] also adopts a nested approach for binding and lacks a hardware implementation.

Pitas [2021] presents optimal n-D cyclic convolution algorithms with minimal multiplicative complexity and requiring fewer multiplications than other algorithms. The example provided involves a 2D cyclic convolution $(3, 3) \times (3, 3)$ with 13 multiplications, where matrices A and B consist solely of 0s and 1s. However, matrix C requires division by a constant multiple of 3, resulting in 117 multiplications involving constants that are multiples of 3. The author applies the split-nesting technique but does not provide a hardware implementation.

Ju and Solomonik [2019] do not directly use the Winograd polynomial method; instead, they employ a variation approach incorporating Toom-Cook to generate the matrix of irreducible polynomials. As a result, the matrices consist of more than just 0s and 1s, requiring additional multiplications by constants. They also explore inspection-based factorization. They use the nested method and present the same Kronecker method we have developed for binding. However, their testing is limited to software implementations and does not extend to hardware applications.

In our research, using resources beyond Scopus and IEEE Explore, we identified three articles that explore alternative techniques for fast convolution. In the works of Cariow and Paplinski [2020], Cariow and Paplinski [2021], and Cariow et al. [2023], various sizes of fast convolution matrices are presented, which contain only 0s and 1s. Cariow and Paplinski [2020] introduces rapid linear convolution for sizes $M = N = (2, 3, 4, 5, 6, 7, 8)$. The author develops the matrices for fast convolution by decomposing the linear convolution matrices; however, no hardware testing is reported. Cariow and Paplinski [2021] presents rapid circular linear convolution for sizes $M = N = (2, 3, 4, 5, 6, 7, 8, 9)$. Cariow et al. [2023] discusses rapid linear convolution derived from the factorization of Hansel matrices for sizes $M = N = (3, 5, 7, 9)$.

A comparison is made between the usage of Look-Up Tables (LUTs) and multipliers for both naïve and fast convolution in several FPGA implementations.

Taken together, these works show that there is rich algorithmic diversity beyond the minimum filter and Toom–Cook methods, including modular polynomials, split nesting, prime-length convolutions, and matrix structures tailored to hardware. However, they are either evaluated only in software or explored in isolation from concrete accelerator templates.

Conclusion

From a hardware perspective, the literature reviewed in this chapter shows a clear evolution from fixed-function, layer-specific Winograd blocks towards more flexible PE arrays and template-based accelerators that can support different convolution types and even multiple algorithms (direct, GEMM-based, and Winograd) [Liang et al., 2020; Kala et al., 2018, 2019b; Liu et al., 2021; Ye et al., 2020]. Nevertheless, most existing designs still focus on a limited set of 2D layer shapes with small kernels and unit stride, and treat more complex cases such as 3D convolution, strided and transposed convolution, or depthwise separable layers as exceptional cases, often falling back to less optimized datapaths [Deng et al., 2021; Shen et al., 2018, 2020]. Furthermore, current templates and frameworks typically expose architectural parameters (array size, tiling factors, buffer sizes). However, they rarely make the underlying fast convolution algorithms' structure an explicit part of the hardware design space.

A few recent studies have explored alternative techniques for fast, multidimensional convolution. Research beyond the minimum filter and Toom-Cook methods remains limited for fast convolution, and the modular polynomial method has not been explored. Regarding multidimensional convolution, only one alternative to the nested approach has been identified, split nesting, which has not yet been implemented in hardware.

These gaps motivate the architectural direction taken in this Dissertation. Rather than hard-coding a small set of Winograd configurations into the hardware, the goal is to develop parameterizable accelerator structures whose datapaths, buffering schemes, and control logic can be systematically derived from the properties of a family of fast convolution algorithms, including multidimensional and non-standard layers. The mathematical tools of Chapter 2 and the algorithmic insights of Breitzman, Nussbaumer, and others provide the foundation for this unified algorithm–hardware framework. In Chapter 4 we formalize the specific fast convolution methods adopted in this Dissertation, and in the subsequent software and hardware chapters, we show how they can be implemented, evaluated, and compared within a design-space exploration flow.

4. PROPOSED FAST CONVOLUTION ALGORITHMS

This chapter presents a concise, algebraic description of the **proposed** fast convolution algorithms that form the core of this Dissertation. Building on the reference formulations of Chapter 2 and the gaps identified on Chapter 3, we focus on two complementary goals: (i) deriving fast convolution schemes that reduce or eliminate multiplications by general constants, favoring matrices with entries in $\{-1, 0, 1\}$ whenever possible, and (ii) extending these schemes to multidimensional convolution in a way that is compatible with the hardware templates and dataflows reviewed in the previous chapter. The methods described here will later be instantiated in software and hardware in Chapters 5 and 6 and evaluated experimentally in Chapter 7.

Figure 4.1 mirrors the same three-level organization. The outermost box again denotes the set of fast convolution algorithms; inside it, the "Winograd Framework" box captures the Winograd-type methods defined in Chapter 2. The shaded subset within that box marks the algorithms that are **proposed** and developed in detail in this chapter, namely the inspection-factorization (IF) schemes and their polynomial refinements. The remaining families (FFT-based, Nussbaumer, Agarwal–Cooley, classical Toom–Cook, Multidimensional Kronecker and nested schemes) are presented in context, clarifying how the proposed methods relate to both the reference repertoire and the broader fast-convolution landscape.

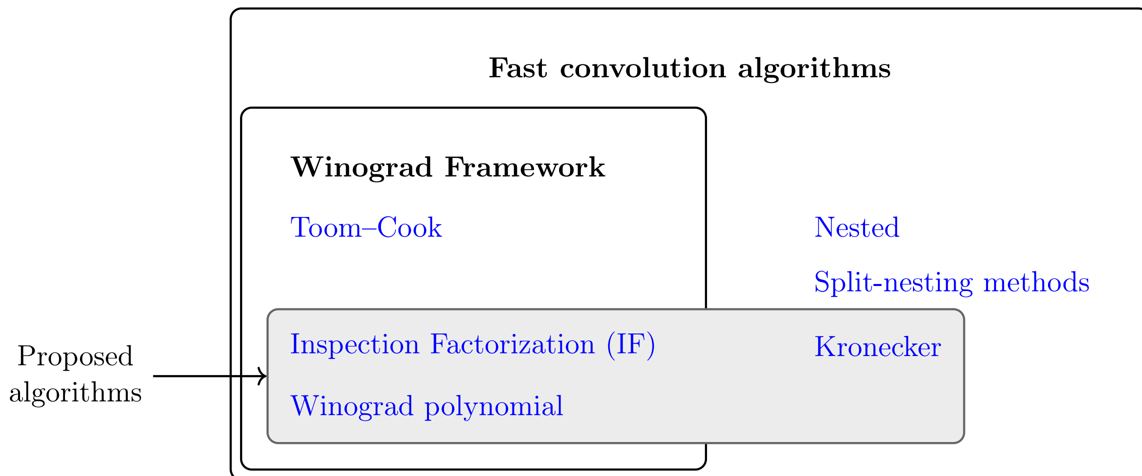


Figure 4.1: Taxonomy of fast convolution algorithm families considered in this work with emphasis on proposed algorithms.

4.1 Multidimensional Kronecker

We propose a Kronecker product method to bind multiple fast convolution matrices together and build a multidimensional fast convolution. This algorithm performs convolution in parallel, but the nested method (2.4) executes in two stages. This method is based on

existing studies on multidimensional algorithms (transforms and fast convolution). Equation 4.1 synthesizes our method.

$$s = (A_1 \otimes A_2)^t [G \odot (C_1 \otimes C_2)^t \mathbf{vec}(d)] \tag{4.1}$$

The Kronecker product, denoted by $\otimes$, combines two matrices into a larger composite matrix while preserving structural properties. Our approach uses it to merge matrices from the first and second dimensions of the fast convolution algorithm, resulting in matrices $A = A_1 \otimes A_2$ and $C = C_1 \otimes C_2$.

The input data is derived from the nested example presented in Section 2.4. The data sequence is specified in Equation (2.7), and the kernel can be found in Equation (2.8). We employ the same transformation method in that example, as indicated in Equations (2.9) and (2.10). That is, we similarly use two Toom-Cook of $\{M_4, N_2, L_3, K_4\}$ presented in Equation (2.4), binding both using the Kronecker Product and building a $\{M_{4\times 4}, N_{2\times 2}, L_{3\times 3}, K_{4\times 4}\}$. Initially, we construct the multidimensional transformation matrix, denoted as $C = (C_1 \otimes C_2)^t$:

$$C = \left(\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^t$$

$$= \begin{bmatrix}
1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1
\end{bmatrix}$$

Subsequently, we compute the multidimensional inverse transformation matrix $A = (A_1 \otimes A_2)^t$:

$$A = \left(\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \right)^t$$

$$= \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 & 0 & 1 & -1 & 1 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & -1 & 1 & -1 & 0 & 1 & -1 & 1 \end{bmatrix}$$

Next, we transform the input data sequence using $D = Cd$:

$$\begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ D_4 \\ D_5 \\ D_6 \\ D_7 \\ D_8 \\ D_9 \\ D_{10} \\ D_{11} \\ D_{12} \\ D_{13} \\ D_{14} \\ D_{15} \end{bmatrix} = \begin{bmatrix} 0 \\ 16 \\ 0 \\ 0 \\ 4 \\ 30 \\ 2 \\ 4 \\ 0 \\ 8 \\ 0 \\ 0 \\ 0 \\ 16 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{bmatrix}$$

We then calculate the Hadamard Product $S = G \odot D$

$$\begin{bmatrix} S_0 & S_1 & S_2 & S_3 \\ S_4 & S_5 & S_6 & S_7 \\ S_8 & S_9 & S_{10} & S_{11} \end{bmatrix} = \begin{bmatrix} 0 & -24 & 0 & 0 \\ -18 & 270 & 6 & 30 \\ 0 & 24 & 0 & 0 \\ 0 & 168 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\frac{3}{2} & -\frac{1}{2} & -2 \\ -\frac{9}{2} & 9 & 3 & \frac{15}{2} \\ -\frac{3}{2} & 3 & 1 & \frac{5}{2} \\ -6 & \frac{21}{2} & \frac{7}{2} & 8 \end{bmatrix} \odot \begin{bmatrix} 0 & 16 & 0 & 0 \\ 4 & 30 & 2 & 4 \\ 0 & 8 & 0 & 0 \\ 0 & 16 & 0 & 0 \end{bmatrix}$$

Lastly, we compute the inverse transformation, denoted as $s = AS$:

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 258 \\ 294 \\ 402 \\ 438 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 & 0 & 1 & -1 & 1 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 & -1 & 1 & -1 & 0 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ -24 \\ 0 \\ 0 \\ -18 \\ 270 \\ 6 \\ 30 \\ 0 \\ 24 \\ 0 \\ 0 \\ 0 \\ 168 \\ 0 \\ 0 \end{bmatrix}$$

Summarizing Equation (4.1), our proposed algorithm uses the same structure presented in Equation (2.1) and Figure 2.3, but the matrices are generated for multidimensional convolution. We use three pre-computed matrices: Q, A, C, executing the fast convolution in 3 steps: (1) $D = Cd$, inputs multiplied by constants; (2) $S = G \odot D$, D multiplied by the preprocessed weights; (3) $s = AS$, S multiplied by constants. Matrices A and C are generated using Kronecker Product that binds two fast convolution matrices together and can execute a multidimensional convolution.

4.2 Inspection factorization

To reduce the number of multiplications in the Toom-Cook algorithms compared to the naïve method, it is essential to increase N, as shown in Figure 2.8. However, with $N > 4$ in 2D or $N > 2$ in 1D convolution, matrices A and C contain values other than 1, -1, and 0, as shown in Figure 2.6. This behavior contrasts with the case when $N \leq 4$ in 2D or $N \leq 2$ in 1D convolution, as depicted in Equation 2.5.

Although fast convolution methods significantly reduce the number of multiplications present in the Hadamard product ($\odot$) (Equation 2.1), for larger values of N, the matrix operations involving A and C, particularly when constants that are not powers of two are involved, introduce trade-offs. These trade-offs must be carefully managed, as they diminish the initial reduction in multiplications offered by fast convolution techniques.

Winograd's techniques for generating fast convolutions do not encompass all conceivable algorithms. Inspection-based factorization can derive some algorithms [Parhi, 1999]. Blahut [2010] demonstrates the following method for performing the convolution or polynomial multiplication of two second-degree polynomials, which results in an output of fourth-degree, or equivalently, two input vectors of size 3 yielding an output vector of size 5. This method was first presented by Winograd [1987]:

$$
\begin{aligned}
s(x) &= (g_0 + g_1 x + g_2 x^2)(d_0 + d_1 x + d_2 x^2) \\
&= (g_0 d_0)x^0 + (g_0 d_1 + g_1 d_0)x^1 + (g_0 d_2 + g_1 d_1 + g_2 d_0)x^2 + (g_1 d_2 + g_2 d_1)x^3 + (g_2 d_2)x^4
\end{aligned}
$$

This convolution requires 9 multiplications and 4 additions; we can reorganize it as follows to clarify what will be done in the next steps:

$$
\begin{aligned}
s_0 &= g_0 d_0 \\
s_1 &= g_0 d_1 + g_1 d_0 \\
s_2 &= g_0 d_2 + g_1 d_1 + g_2 d_0 \\
s_3 &= g_1 d_2 + g_2 d_1 \\
s_4 &= g_2 d_2
\end{aligned}
$$

Can be factored into the following identities:

$$
\begin{aligned}
g_0 d_0 &= g_0 d_0, \\
g_0 d_1 + g_1 d_0 &= (g_0 + g_1)(d_0 + d_1) - g_0 d_0 - g_1 d_1, \\
g_0 d_2 + g_1 d_1 + g_2 d_0 &= (g_0 + g_2)(d_0 + d_2) - g_0 d_0 + g_1 d_1 - g_2 d_2, \\
g_1 d_2 + g_2 d_1 &= (g_1 + g_2)(d_1 + d_2) - g_1 d_1 - g_2 d_2, \\
g_2 d_2 &= g_2 d_2.
\end{aligned}
$$

With this inspection-based factorization, we can perform the convolution using 6 multiplications (4 less) and 10 additions (6 more). In matrix form, this algorithm is represented:

$$
\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 0 & 0 \\ 0 & -1 & 1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \left[\begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ g_0 + g_1 \\ g_0 + g_2 \\ g_1 + g_2 \end{bmatrix} \odot \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} \right) \right] \tag{4.2}
$$

Equivalently, this corresponds to one input vector of size 5 and another of size 3, yielding an output vector of size 3, as illustrated in Equation 4.3. The (IF) method was parameterized by $\{N, L, K\} = \{3, 3, 6\}$, achieves the exact input-output mapping of the Toom–Cook 3 while eliminating the constant-multiplication overhead, as shown in Equation 2.6. In IF, the transform matrices A and B are binary, containing only zeros and ones, thus avoiding multiplications by constants. Moreover, the matrix Q consists entirely of ones, eliminating rounding errors in the weight transform and facilitating straightforward online weight updates.

$$A^T = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad C^T = \begin{bmatrix} 1 & -1 & -1 & 0 & 0 \\ 0 & -1 & 1 & -1 & 0 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad Q = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{4.3}$$

4.3 Winograd Polynomial Modular Convolution

Winograd [1976, 1987] proposed an algorithm for cyclic convolution that employs the Chinese Remainder Theorem applied to irreducible rational factors $x^n - 1$, referred to as cyclotomic polynomials [Breitzman and Johnson, 2004]. The Winograd algorithm can be interpreted as a factorization technique for specific matrices that minimizes multiplications, with constant multiplications generally resulting in small integers (-1, 0, and 1) [Blahut, 2010].

The Toom-Cook algorithm also uses the CRT but relies on integer reduction, leading to an accelerated increase in the number of additions as the convolution size increases. In contrast, Winograd's more generalized application of the CRT results in algorithms that handle small convolutions differently, maintaining the number of additions at levels lower than those observed in the Toom-Cook approach. However, there is a moderate increase in the number of multiplications [Nussbaumer, 1982; Tolimieri et al., 2013].

As a concrete example, consider a 1D linear convolution with four outputs and a 3-tap kernel, as in Chapter 2. Using Tolimieri's construction with modulus factors x, $x^2 - 1$ and $x^2 + 1$, we obtain a Winograd-type scheme in the bilinear form

$$s = C\Big[(Q\,Bg) \odot (Ad)\Big], \tag{4.4}$$

where $d = [d_0, d_1, d_2, d_3]^\mathsf{T}$ is the input block, $g = [g_0, g_1, g_2]^\mathsf{T}$ is the filter, A and B are, respectively, the data and kernel transforms, Q is a diagonal scaling that absorbs fractional factors, and C is the integer recombination matrix.

In this case, the transforms can be written explicitly as a Winograd-style triple (A, B, C) plus a diagonal scaling Q acting on the kernel side:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 1 & 1 & -1 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 1 & 1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & -1 & -1 & 1 & -1 & -1 \\ 0 & 1 & 0 & 1 & -1 & 0 & 1 & 0 \\ 0 & -1 & 1 & -1 & 1 & -1 & 1 & 0 \\ -1 & 1 & 0 & 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad Q = \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix} \tag{4.5}$$

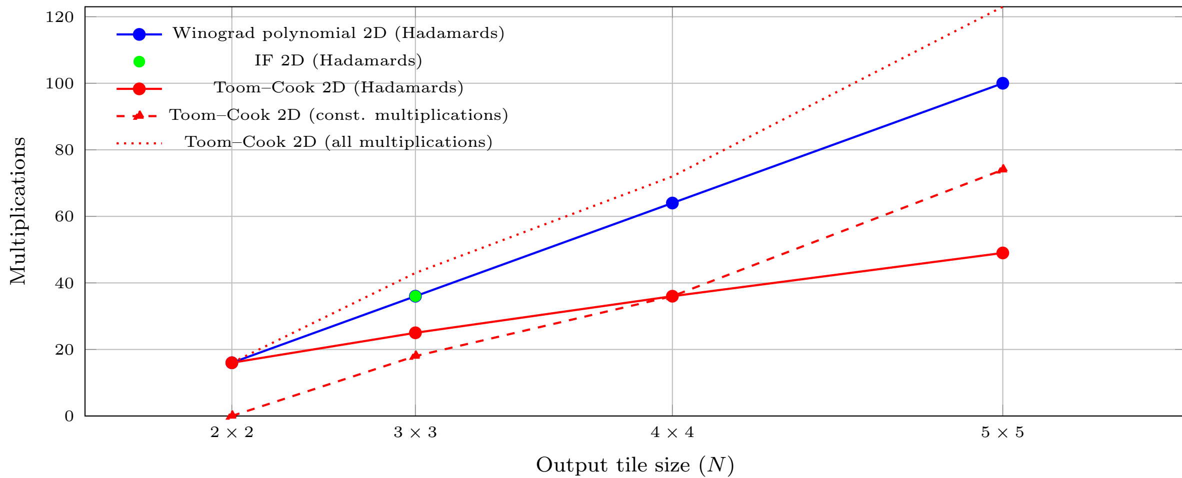
Where all fractional factors $\frac{1}{2}$ are confined to the diagonal scaling Q and the recombination matrix C contains only entries in $\{-1, 0, 1\}$. This mirrors the organization used for the Toom–Cook schemes in Chapter 2: the algebraic derivation of the matrices is relegated to the appendix, while the main chapter works directly with the concrete A, B, C, Q quadruple as a reference Winograd polynomial modular algorithm.

The derivation of Winograd's polynomial modular convolution—including the cyclotomic factorization of $x^n - 1$, the use of the Chinese Remainder Theorem over polynomial rings, and the Tolimieri-style linear-from-cyclic correction that turns a fast cyclic scheme into a fast linear convolution—is presented in full detail in Chapter C. Classical constructions show how a rapid cyclic convolution obtained in this way can be lifted to a rapid linear convolution, either via polynomial manipulations [Tolimieri et al., 2013] or through equivalent matrix-based formulations [Parhi, 1999]. In this chapter we restrict ourselves to the resulting A, B, C structures and to their role as building blocks for the proposed multidimensional and inspection-based methods, while the appendix provides the step-by-step algebra and small worked examples that justify these matrices.

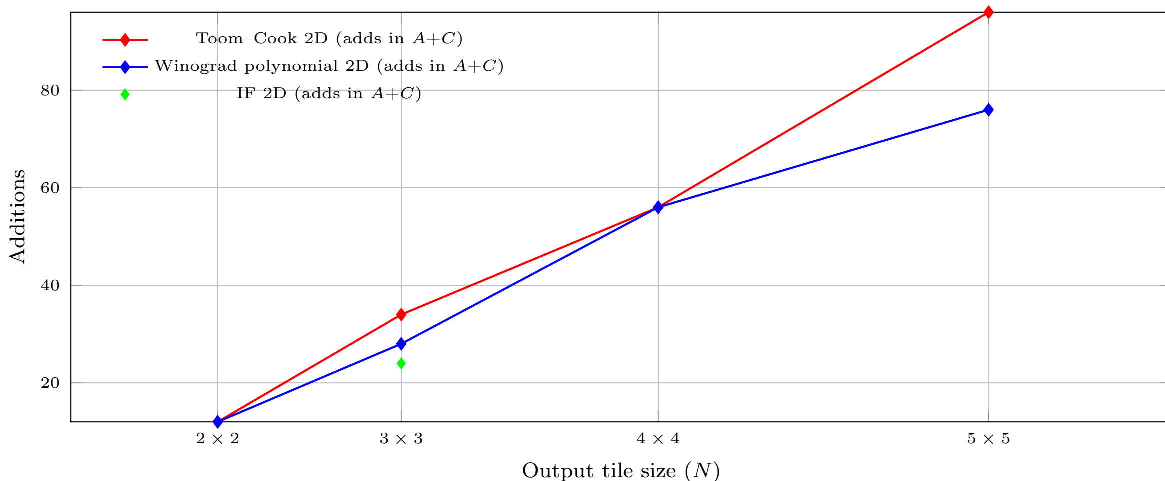
From a broader algorithm-design perspective, Breitzman [Breitzman and Johnson, 2004] develops a uniform algebraic framework for **automatically** deriving and implementing fast convolution algorithms. Starting from a small library of fast linear and cyclic convolution kernels and using tensor products together with CRT-based factorizations, the framework explores a large design space of Winograd-type and related schemes and searches for combinations with low arithmetic cost. In particular, we leverage these Breitzman-style constructive techniques in the Winograd polynomial modular setting to synthesize fast convolution instances whose transform matrices contain only coefficients in $\{0, 1\}$, eliminating general constant multiplications in the datapath.

4.4 Comparison between Reference and Proposed Algorithms

The two subplots in Figure 4.2 summarize the trade-offs that motivate the proposed inspection-factorization and Winograd polynomial modular schemes. In the top plot, the Winograd polynomial 2D variants use slightly more Hadamard products than the Toom–Cook base-



(a) Multiplication counts: Hadamard products and constant multiplications in Toom–Cook and Winograd polynomial 2D.



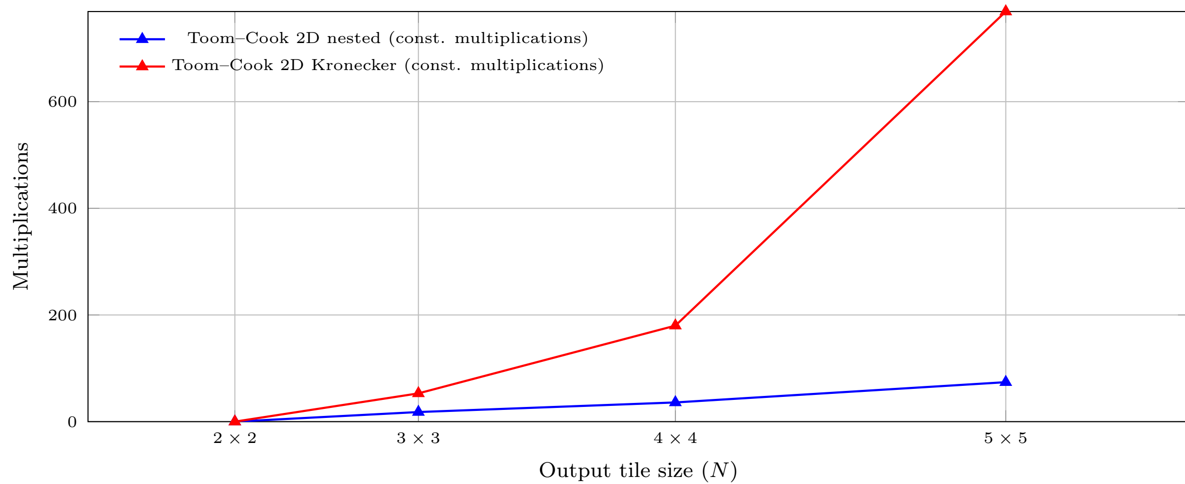
(b) Addition counts: sums in the transform matrices A and C for Toom–Cook and Winograd polynomial 2D.

> The top plot groups all multiplication-related costs (Hadamard products and constant multiplications), while the bottom plot shows only the additions in the transform matrices A and C.

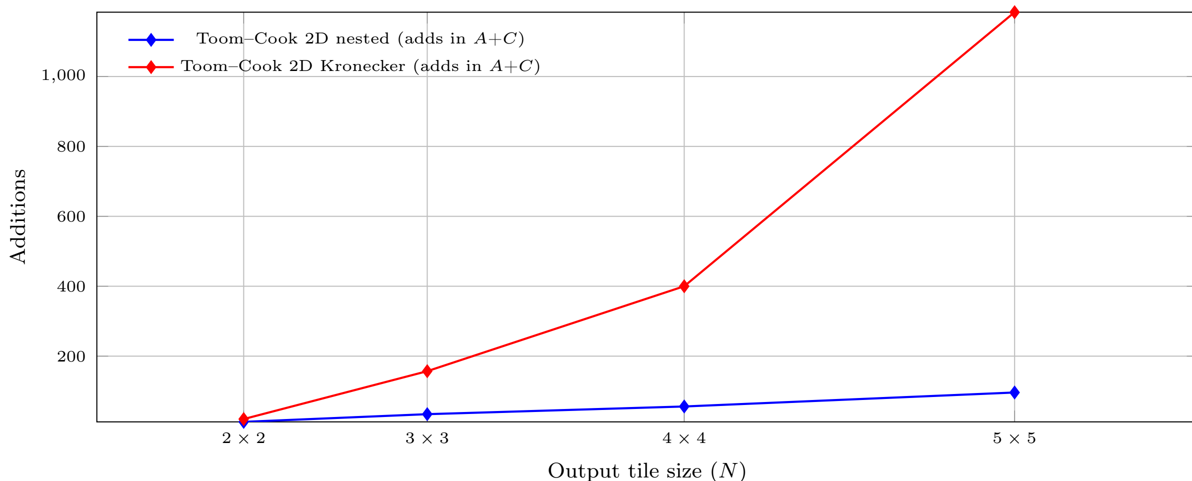
Figure 4.2: Comparison between Toom–Cook and Winograd polynomial in 2D Nested for output blocks from 2×2 up to 5×5.

line as the output block size $N = n^2$ grows, but they eliminate all constant multiplications. Once the constant factors are taken into account in the "all multiplications" curve, Toom–Cook requires more total multiplications (Hadamards plus constant products) than the Winograd polynomial schemes for all but the smallest case. The bottom plot shows that the addition counts in the transform matrices A and C are comparable, with Winograd polynomial generally requiring the same or fewer additions than Toom–Cook, especially for larger tiles. Together, these trends indicate that the extra Hadamards in the polynomial Winograd construction are more than compensated by the removal of constant multipliers, yielding lower overall multiplicative complexity with similar or reduced additive cost.

The comparison in Figure 4.3 focuses on two-dimensional Toom–Cook alone and contrasts the nested formulation with its Kronecker-lifted counterpart. For a fixed output tile size $N = n^2$, the Kronecker construction substantially increases both the number of constant mul-



(a) Constant-multiplication counts in the nested and Kronecker 2D Toom–Cook constructions.



(b) Addition counts in the transform matrices A and C for the nested and Kronecker 2D Toom–Cook constructions.

> The top plot shows how the Kronecker method substantially increases the number of constant multiplications in the transform stage, while the bottom plot highlights the corresponding growth in additions inside the A and C matrices.

Figure 4.3: Comparison between the nested and Kronecker 2D Toom-Cook constructions for output blocks from 2×2 up to 5×5.

tiplications in the transform stage and the number of additions inside the A and C matrices. While both variants share the same asymptotic behavior, the nested scheme achieves the desired two-dimensional factorization with far fewer scalar operations, making it more attractive as a reference implementation for hardware-oriented fast convolution in this work.

4.5 Conclusion

The algorithms developed in this chapter thus span from one-dimensional inspection-based factorization to multidimensional Kronecker constructions that can be directly mapped to hardware-friendly transform structures. In the next chapters, we instantiate these methods in software and hardware, quantify their computational and architectural costs, and compare them against the naïve and nested baselines identified in Chapter 3.

5. SOFTWARE IMPLEMENTATION

This chapter summarizes the fast convolution results implemented in software. We implemented and evaluated fast convolution algorithms in Python and C, including experiments on the RS5 RISC-V processor [Nunes et al., 2024]. These experiments quantify the trade-offs between naïve and transform-based convolutions when executed on a general-purpose core. From a tooling perspective, we describe a Python toolbox that supports the design, validation, and code generation of fast convolution algorithms.

The chapter is organized as follows. Section 5.1 details the software implementation on the RS5 processor and the structure of the Python toolbox used to prototype and analyze fast convolution algorithms. Section 5.2 presents the experimental results on the RS5, highlighting the performance and instruction-level trade-offs between naïve and fast convolution methods, and motivating the need for dedicated hardware accelerators.

5.1 Software implementation and toolbox

The software implementation of fast convolution algorithms was evaluated on the RS5 RISC-V processor [Nunes et al., 2024], simulated using Verilator. Table 5.1 presents the implemented convolution techniques, including a naïve convolution (N^9) and fast convolutions in both 1D and 2D forms, such as TCk^9, TCn^9, IFk^9, IFn^9, and MFn^4, employing the nested method (Equation (2.6)) where applicable.

Table 5.1: Number of multiplications and matrices' size.

Name	Type	Output	#Multip	A	B	C
N^9	Naïve	1	9	-	-	-
TCk^9	Toom-Cook 2D Kronecker	3×3	25	(9, 25)	[(5, 3), (5, 3)]	(25, 25)
TCn^9	Toom-Cook 2D Nested	3×3	25	[(3, 5), (3, 5)]	[(5, 3), (5, 3)]	[(5, 5), (5, 5)]
MFk^4	Winograd Minimal Filter 2D Kronecker	2×2	16	(4, 16)	[(4, 3), (4, 3)]	(16, 16)
IFk^9	Inspection Factorization 2D Kronecker	3×3	36	(9, 36)	[(3, 6), (3, 6)]	(36, 36)
IFn^9	Inspection Factorization 2D Nested	3×3	36	[(3, 6), (3, 6)]	[(3, 6), (3, 6)]	[(6, 6), (6, 6)]
MFn^4	Winograd Minimal Filter 2D Nested	2×2	16	[(2, 4), (2, 4)]	[(4, 3), (4, 3)]	[(4, 4), (4, 4)]
TC^3	Toom-Cook 1D	3	15	(3, 5)	(5, 3)	(5, 5)
TC^2	Toom-Cook 1D	2	12	(2, 4)	(4, 3)	(4, 4)
IF^3	Inspection Factorization 1D	3	18	(3, 6)	(3, 6)	(6, 6)

A Python package for fast convolutions has been developed. This package is designed for command-line use and includes Python implementations of various algorithms, such as Toom–Cook, inspection factorization, and nested convolutions. It provides a bridge between symbolic algorithm design, numerical validation, and code generation for both software and hardware implementations. In subsequent chapters, this toolbox is extended with hardware-aware cost models (e.g., numbers of multipliers, registers, buffer sizes, and memory accesses),

enabling systematic design-space exploration of multidimensional fast convolution algorithms under realistic accelerator constraints.

The package generates symbolic and numerical examples of fast convolutions in LaTeX format and exports them to PDF. It also facilitates code export to C, making it suitable for generic architectures. The export functionality generates matrix multiplication and Hadamard product functions that use loops, as well as optimized versions that eliminate them to improve performance. The code presented in Listings 5.1 and 5.2 illustrates the optimized version of the Toom–Cook 1D implementation. This optimized code computes the Hadamard product and matrix multiplications without control structures such as loops or conditional statements (if–else), which are typically required for these operations. Instead, all multiplications in the matrix operations are replaced with bit shifts, improving performance and efficiency.

```
1  void fast_conv(int *ms, const int *ma, const int *mgg, const int *mc, const int *md, int
       a_size, int c_size, int m_size) {
2
3      int *mss = (int *) malloc((m_size) * sizeof(int));
4      int *mdd = (int *) malloc((m_size) * sizeof(int));
5
6      ...
7
8      matrix_mul_shift_noloop_c(mdd, md);
9      hadamart_product_noloop(mss, mdd, mgg);
10     matrix_mul_shift_noloop_a(ms, mss);
11
12     ...
13 }
```

Code 5.1: Main function of optmized C code, non-essential lines have been removed.

```
1  void matrix_mul_shift_noloop_a(int *m_out, const int *m_in){
2      m_out[0] = + m_in[0] + m_in[1] + m_in[2] + m_in[3];
3      m_out[1] = + m_in[1] - m_in[2] + (m_in[3] << 1);
4      m_out[2] = + m_in[1] + m_in[2] + (m_in[3] << 2) + m_in[4];
5  }
6
7  void matrix_mul_shift_noloop_c(int *m_out, const int *m_in){
8      m_out[0] = + (m_in[0] << 1) - m_in[1] - (m_in[2] << 1) + m_in[3];
9      m_out[1] = - (m_in[1] << 1) - m_in[2] + m_in[3];
10     m_out[2] = + (m_in[1] << 1) - m_in[2] - (m_in[2] << 1) + m_in[3];
11     m_out[3] = - m_in[1] + m_in[3];
12     m_out[4] = + (m_in[1] << 1) - m_in[2] - (m_in[3] << 1) + m_in[4];
13 }
14
15 void hadamart_product_noloop(int *out, const int *in1, const int *in2){
16     out[0] = in1[0] * in2[0];
17     out[1] = in1[1] * in2[1];
18     out[2] = in1[2] * in2[2];
19     out[3] = in1[3] * in2[3];
20     out[4] = in1[4] * in2[4];
21 }
```

Code 5.2: Optmized C code for TC^3.

5.2 Experimental results on RS5

Table 5.2 presents the instruction profile to execute the convolutions. We used a 32×32 input feature map and a 3×3 kernel to provide a practical evaluation of the convolution operations. Instruction counting was enabled exclusively for the convolution to ensure accurate performance measurement. Compilation used the `-O2` optimization level (GCC) to enhance the performance of the generated code.

Table 5.2: Instruction Profiling for the Convolution Methods. The profiling of the Fast Convolution methods is expressed as a percentage w.r.t the N^9 method.

Name	clock cycles	addsub	shift	branch	mul
N^9	1,362,671	44,453	2,735	13,177	8,100
TCk^9	123%	120%	934%	45%	34%
TCn^9	133%	76%	383%	46%	35%
MFk^4	130%	73%	42%	45%	48%
IFk^9	119%	73%	38%	45%	49%
IFn^9	446%	58%	38%	46%	50%
MFn^4	152%	62%	42%	46%	51%
TC_3	174%	131%	395%	118%	61%
TC_2	195%	119%	86%	122%	71%
IF_3	490%	113%	70%	118%	74%

Initial implementations, not presented in Table 5.2, employed a fast convolution algorithm without decomposing the matrices A and C into powers of two. These implementations used a loop-based approach, applying the standard matrix multiplication algorithm, which consists of three nested loops to compute Cd and AS, and a single loop to compute $G \odot D$. These initial implementations were discarded due to a significant increase in operations, including multiplications.

Table 5.2 reveals that faster convolutions are slower than naïve ones (clock cycles column). This result was expected, given the higher computational complexity of fast convolution algorithms when implemented in software. *The most relevant outcome is the significant reduction in the number of multiplications from 50% (IFn^9 e IFk^9) by up to 65% (TCn^9 e TCk^9) for the 2D (3×3 output) case.*

The 2D versions reduced branch operations. Fast convolutions with output sizes of 2 and 2×2 lowered shift operations, as their matrices contained only 1, 0, and -1. Conversely, fast convolutions with output sizes of 3 and 3×3 saw increased shift operations by factors of 3 for 1D and 2D nested convolutions and 9 for 2D Kronecker convolutions. The nested convolution required fewer additions and shifts than the Kronecker approach but had a higher execution time.

Overall, the nested convolution method proved more advantageous than the Kronecker approach, requiring fewer additions and shifts. Although the 1D convolutions offer limited gains, they may still be helpful in resource-constrained environments.

This software evaluation shows that fast convolution methods reduce the number of multiplications, though at the cost of increased execution time. This explains why the MAC-based approach remains the most widely used method in software libraries. Overall, the RS5 experiments reinforce that simply reducing the number of multiplications is not sufficient to guarantee speedups on a general-purpose core. The additional additions, subtractions, and memory accesses introduced by the transform and inverse steps dominate execution time and energy, explaining why MAC- and GEMM-based implementations remain the de facto standard.

These observations contrast with the hardware results presented in Chapter 6, where the same algorithms can exploit dedicated datapaths, buffering schemes, and control logic. Together with the background and related work in Chapters 2 and 3 and the algorithmic developments in Chapter 4, the RS5 experiments therefore serve as a baseline and motivation for the specialized hardware architectures and unified algorithm–hardware framework developed in the remainder of the thesis.

6. HARDWARE IMPLEMENTATION OF THE WINOGRAD CONVOLUTION ACCELERATOR

This chapter describes the hardware implementation of the proposed Winograd-based convolution accelerator. It bridges the algorithmic framework introduced in Chapter 2 and Chapter 4 with an RTL architecture, highlighting how the design targets low-energy convolution under tight area and memory-bandwidth constraints typical of embedded and IoT systems. The chapter is organized around the main architectural blocks: the top-level accelerator system, the Winograd convolution core and its matrix-multiplication engine, the control unit and its scheduling strategy, and the parameterization and implementation flow used to explore different configurations.

6.1 Design Goals and Constraints

The primary design goal of the accelerator is to **minimize energy consumption** while preserving performance and area compared to a MAC-based convolution accelerator. The target deployment domain is embedded inference, particularly for mobile and IoT devices, where energy per inference and total on-chip area are more critical than peak throughput. In this context, simply increasing parallelism or clock frequency is rarely acceptable: any additional throughput that cannot be effectively utilized due to memory constraints is dissipated as dynamic power and often incurs unnecessary area overhead.

This work uses Winograd-style fast convolution to **meet the application performance requirements** and then trades the arithmetic savings for reductions in energy and storage. In other words, once the arithmetic complexity of convolution is reduced, the design does not attempt to saturate the additional arithmetic headroom with more parallel units; it uses that headroom to scale down the number of multipliers, the clock frequency, or both while still meeting the latency and throughput requirements of the target CNN.

A key constraint is that convolution in modern CNNs is typically **memory-bound**: the time spent fetching and storing input feature maps, filters, and outputs dominates the time spent performing arithmetic operations. For the intended embedded and IoT scenarios, external and even on-chip memory bandwidth is limited, and the accelerator must be designed to operate close to the memory traffic boundary. The goal is therefore not to build a compute engine that is significantly faster than the memory subsystem, but to tune the compute throughput to match the sustainable read/write rates of the memory hierarchy. When the accelerator operates at this boundary, additional arithmetic units remain underutilized while still consuming power and occupying area.

The Winograd filter form adopted in this work exploits spatial overlap between neighboring tiles to increase data reuse. For the chosen 2D Winograd configuration, each IFMAP

tile consumes a patch of the input feature map to produce an OFMAP tile. When sliding the window horizontally, consecutive IFMAP tiles require that only new columns be fetched for each new IFMAP tile. A similar overlap occurs in the vertical direction. In principle, this overlap could be exploited by a line-buffer organization that maintains several rows or columns of the input in on-chip memory, feeding the Winograd core as the window progresses. However, in the target technology and under the given area constraints, implementing a full line-buffered front-end was deemed too costly in terms of memory and control complexity.

Instead, the accelerator uses a more compact organization with a **single register group inside the convolution core** to hold the working set of a tile. At any point in time, only one Winograd tile is resident in this register group, and the core processes one convolution tile at a time. This choice eliminates the need for multiple replicated buffers and reduces the flip-flop count, at the cost of limiting spatial concurrency across tiles. Given that memory bandwidth is the dominant bottleneck, this reduction in concurrency does not affect throughput while positively impacting area and power.

Finally, the design allows the **number of multipliers** in the core to be adjusted to match the available memory bandwidth. If memory can only supply one tile every T cycles, there is no benefit in designing a core that consumes a tile in $T/2$ cycles. By instantiating fewer multipliers and accepting a larger number of cycles per tile, the accelerator aligns its computation rate with the memory subsystem and reduces both area and energy per operation. This trade-off between multiplier count, latency per tile, and memory bandwidth is exposed through RTL parameters and explored in the experimental evaluation.

6.2 Top-Level Architecture

The top-level hardware organization is encapsulated by the **System** module, which integrates the Winograd convolution core, the control logic, and the memory interface into a single unit.

6.2.1 External Interface

The **System** module provides a simple external interface intended for embedding within a larger design. At the control level, it exposes a clock and reset, configuration inputs (such as base addresses, strides, and layer dimensions), and status signals indicating when the accelerator is idle, busy, or has completed processing a layer. At the data level, the module interfaces with the memory subsystem through read and write ports that can be mapped either to on-chip SRAM blocks or to a system bus (e.g., an AXI or similar protocol).

The data interface is organized around a handshake-based protocol. For input feature maps and filters, the accelerator issues read requests with addresses and receives data words

with **valid** signals; for output tiles, it produces data words with **valid** signals and waits for **ready** signals from the memory subsystem or downstream logic. This organization decouples the schedule of the Winograd core from variations in memory latency, as long as the bandwidth remains within the design constraints.

Configuration registers specify the size of the input feature map, the number of input channels, the number of output channels, and the tile layout in memory. They also allow the system to select the specific Winograd configuration (e.g., tile and filter sizes) and the number of multiplier units to use in the core.

6.2.2 Internal Organization

Internally, the **System** module instantiates three main blocks: the convolution core, the control unit, and the memory/buffer interface. The convolution core implements the Winograd datapath and is responsible for transforming input tiles, performing the elementwise products with transformed filters, and applying the inverse transform to reconstruct output tiles. The control unit coordinates the progress of tiles through the core and orchestrates memory requests and write-backs.

Figure 6.1 illustrates the top-level organization. Input tiles and filter tiles are fetched from memory and written into the register bank inside the convolution core. After processing, output tiles are sent back to memory. The control unit manages this flow, ensuring that only one tile resides in the core at a time and that the core remains active whenever memory bandwidth allows. This streaming, tile-based organization is a direct consequence of the design goals: by limiting the amount of in-flight data and avoiding large on-chip buffers, the accelerator reduces its area while still exploiting the arithmetic savings of Winograd convolution.

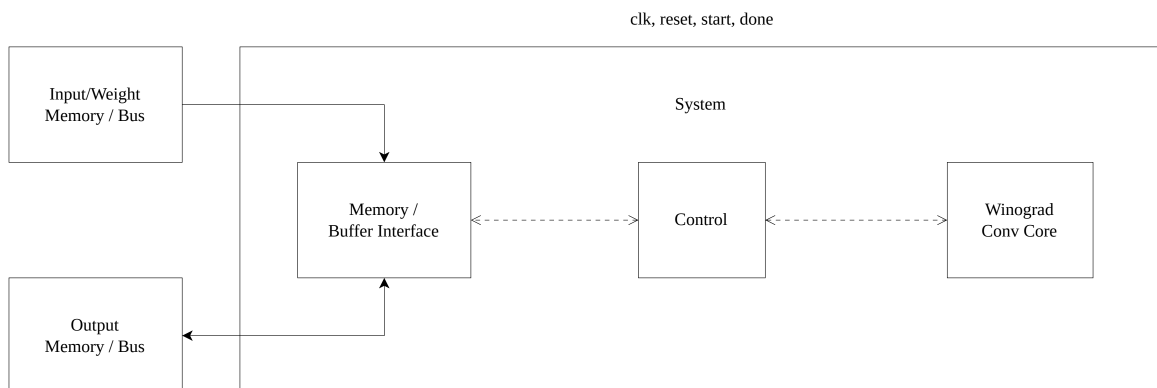


Figure 6.1: Top-level organization of the Winograd convolution accelerator.

Complementing this top-level view, Figure 6.2 illustrates the temporal behavior of the **System** through three FSMs: two within the *control* module (INPUT and OUTPUT) and one within the *convolution core* (CONV). To understand the operation as a pipeline, consider cycles 2-12 in the figure. After loading the tile data, the Input FSM activates the convolution

module (cycle 2). In parallel, the Output FSM reads partial data from the output memory. Subsequently, during cycles 3-6, the convolution is executed. One cycle before the convolution completes, new input data starts to be read, and at the end of the convolution, the result is written to a set of registers (CONV OUT state in the Output FSM). Next, input data are read in parallel while output data are written. After the input read completes, a new convolution cycle is initiated.

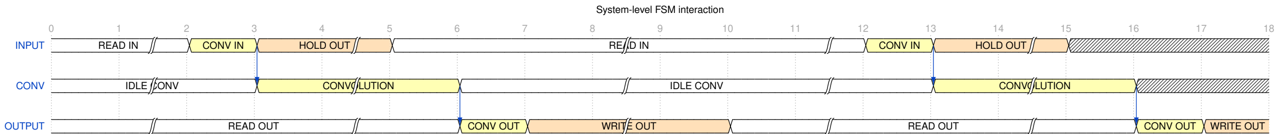


Figure 6.2: Approximate timing diagram for the joint evolution of the three FSMs: input, convolution, and output.

6.3 Convolution Core

The convolution core is the computational heart of the accelerator [1]. It implements the Winograd transform–multiply–inverse-transform cycle for one tile at a time via a simple request/response handshake: whenever the control unit presents an input tile and its associated filter coefficients and issues a request, the core returns the corresponding output tile and signals completion. Figure 6.3 summarizes this internal organization, highlighting the register bank, transform units, and Hadamard engine, and how data flows between them. This decouples the fine-grain datapath from the global schedule of tiles and layers.

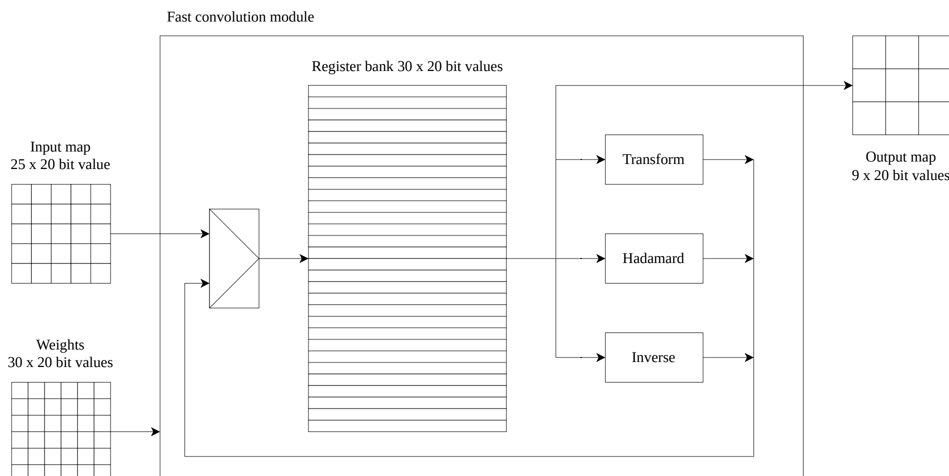


Figure 6.3: Block-level organization of the Winograd convolution core, showing the register bank, Transform, Hadamard and Inverse modules.

[1]The brief clip **"Sliding Window of 2D Fast Convolution Architecture"** walks through the transform–multiply–inverse transform pipeline sketched above, showing how input tiles flow through the three stages and why the register bank is reused across them, which complements the diagrams in this section. Available at https://youtu.be/WgymWFtuF30.

Internally, the core is organized around three functional blocks: a local register bank that holds the current tile and all intermediate values, a pair of linear-transform modules that implement the Winograd input and output transforms Transform and Inverse, and a Hadamard-product module built from a small number of shared multipliers. A finite-state controller sequences these blocks so that a single tile is loaded into the register bank, transformed into the Winograd domain, multiplied elementwise by the transformed filter, and finally brought back to the spatial domain. Only one tile is active in the core at any given time, which aligns with the design goal of operating at the boundary of available memory bandwidth rather than maximizing tile-level parallelism.

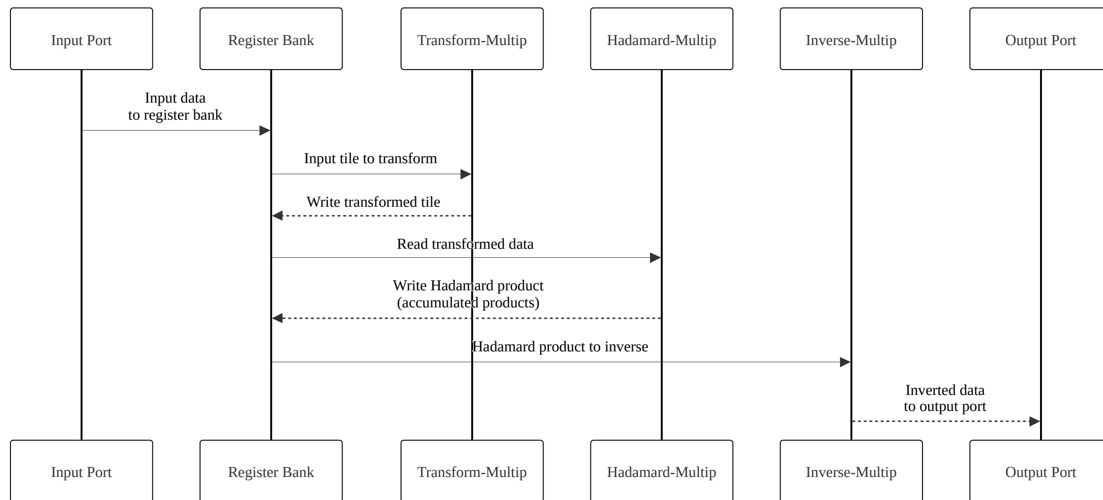
6.3.1 Winograd Datapath Implementation

The datapath in the convolution core implements the Winograd computation for a single tile, as introduced in Sections 2.4 and 4.1. In hardware, these equations are implemented by a fixed sequence of **Transform** (data transform), **Hadamard** (elementwise products precomputed with G), and **Inverse** (reconstruction of s) modules, independent of whether the underlying Winograd instance is nested or Kronecker; the difference lies only in the constant coefficients loaded into the transform networks. In both the nested and Kronecker cases, the filter-side transform G is precomputed offline and stored in memory, so the on-chip datapath only sees the transformed kernel.

When a new tile is requested, the corresponding input patch is first copied from the external interface into the local register bank. This register bank represents the single working set that the accelerator uses to hold intermediate values of a tile: it first stores the raw spatial-domain samples, then the transformed values, and finally the intermediate products that will be fed into the inverse transform.

Figure 6.4 provides a complementary sequence-level view of this datapath. It highlights how each tile is first written into the register bank and then repeatedly read and overwritten as it flows through the Transform, the Hadamard, and the Inverse modules. In this representation, the register bank appears as the central storage point that mediates all exchanges between functional blocks, explicitly showing the data's lifetime and the reuse of the same storage locations across the three phases of the Winograd computation.

The first stage of the Winograd computation is the input transform, implemented as a fixed network of additions, subtractions, and shifts that combine samples stored in the register bank into transformed coefficients. After applying this transform, the original samples in the register bank are overwritten by their transform-domain counterparts. Because the transform coefficients are constant for a given Winograd configuration, this stage does not require general-purpose multipliers and contributes primarily additions and bit shifts to the critical path.



> The diagram emphasizes how a single register bank holds the tile across all phases of the computation, mediating both intermediate results and the final spatial-domain output.

Figure 6.4: Sequence of data transfers between the register bank and the Transform, Hadamard, and Inverse modules for a single Winograd tile.

After the Hadamard product phase, the register bank contains the element-wise products of the transformed input tile and the transformed filter coefficients. The final Winograd stage is the inverse transform implemented by a second linear-transform network that mirrors the structure of the input transform. Feeding the register bank contents into this network produces an output tile in the spatial domain, which is then forwarded to the external interface. Because the inverse transform is combinational, the core can return the completed tile as soon as the transform has settled and immediately transition back to the idle configuration, ready to accept a new tile request.

This organization keeps all Winograd-specific linear algebra confined to two linear-transform blocks, while the surrounding control logic only needs to manage data movement through these blocks and the reuse of the single register bank across the different phases of the computation.

6.3.2 Transform and Inverse Matrix Generators

The Winograd-specific linear transforms used by the convolution core are implemented as constant-coefficient matrix–vector operators that implement the matrices $\mathbf{C}$ and $\mathbf{A}$ presented in Equation (2.6) for the nested 2D construction and in Equation (4.1) for the Kronecker-lifted variant. At a high level, these operators form shallow, feed-forward reduction networks that map spatial tiles into the Winograd domain and back.

For a given configuration, the forward transform applies for the 2D **nested** two transforms:

$$\delta = dC, \qquad D = C^T \delta$$

where d is the spatial tile and D is its representation in the Winograd domain. Architecturally, this mapping is computed in two one-dimensional stages: an input stage associated with C that projects the tile d onto an intermediate basis along one dimension, resulting in δ, followed by a stage associated with C^T that applies δ to the corresponding transform along the orthogonal dimension. After these two separable transform stages, the elementwise (Hadamard) product is performed to combine the transformed input with the transformed kernel:

$$S = G \odot D$$

Here, the entries of the transform-domain kernel matrix G are precomputed and stored, and they are multiplied elementwise with the transformed input coefficients D to produce S, the collection of products that is subsequently used by the inverse transform. The inverse transform follows the same pattern, implementing

$$\sigma = SA, \qquad s = A^T \sigma$$

where S contains the Hadamard products and s is the reconstructed spatial tile. Here, one stage associated with A combines S values along a first dimension, resulting in σ, and a second stage associated with A^T that applies σ to the corresponding transform along the orthogonal dimension, completing the reconstruction along the remaining dimension. In both cases, the transform pipeline exposes the separable structure of Transform (C^T and C) and Inverse (A and A^T) used in Section 2.4.

For the 2D **Kronecker** schemes of Equation (4.1), the same two same phases appear, but the data transform and inverse are expressed in vectorized form using Kronecker products with one stage each. In this case the full transform matrices are precomputed as

$$C = C_1 \otimes C_2, \qquad A = A_1 \otimes A_2,$$

and the computation proceeds as

$$D = C^{\mathsf{T}} \mathrm{vec}(d), \qquad S = G \odot D, \qquad s = A^{\mathsf{T}} S,$$

with C and A encoding the separable 2D transforms.

From the **reference** repertoire of Chapter 2, the hardware supports the 1D Toom–Cook schemes of Figures 2.5 and 2.6 (Toom–Cook-2 and Toom–Cook-3) and from the **proposed** repertoire of Chapter 4, the same datapath also implements the 1D inspection-factorization (IF) scheme of Equation (4.3) and the Winograd polynomial modular instance of Equation (4.5).

In all algorithms, the architecture implements the 2D Winograd form for a tile, with the specific choice of A, B, C, Q and the triplet (M, N, K) (input-tile size, output-tile size, and multiplication count) determined directly by the matrices listed in those tables. Changing from Toom–Cook to IF or to the Winograd polynomial scheme therefore amounts to swapping the precomputed transform matrices while keeping the same transform–Hadamard–inverse modules pipeline in hardware; the remainder of this section describes how that pipeline is organized at the register-transfer level.

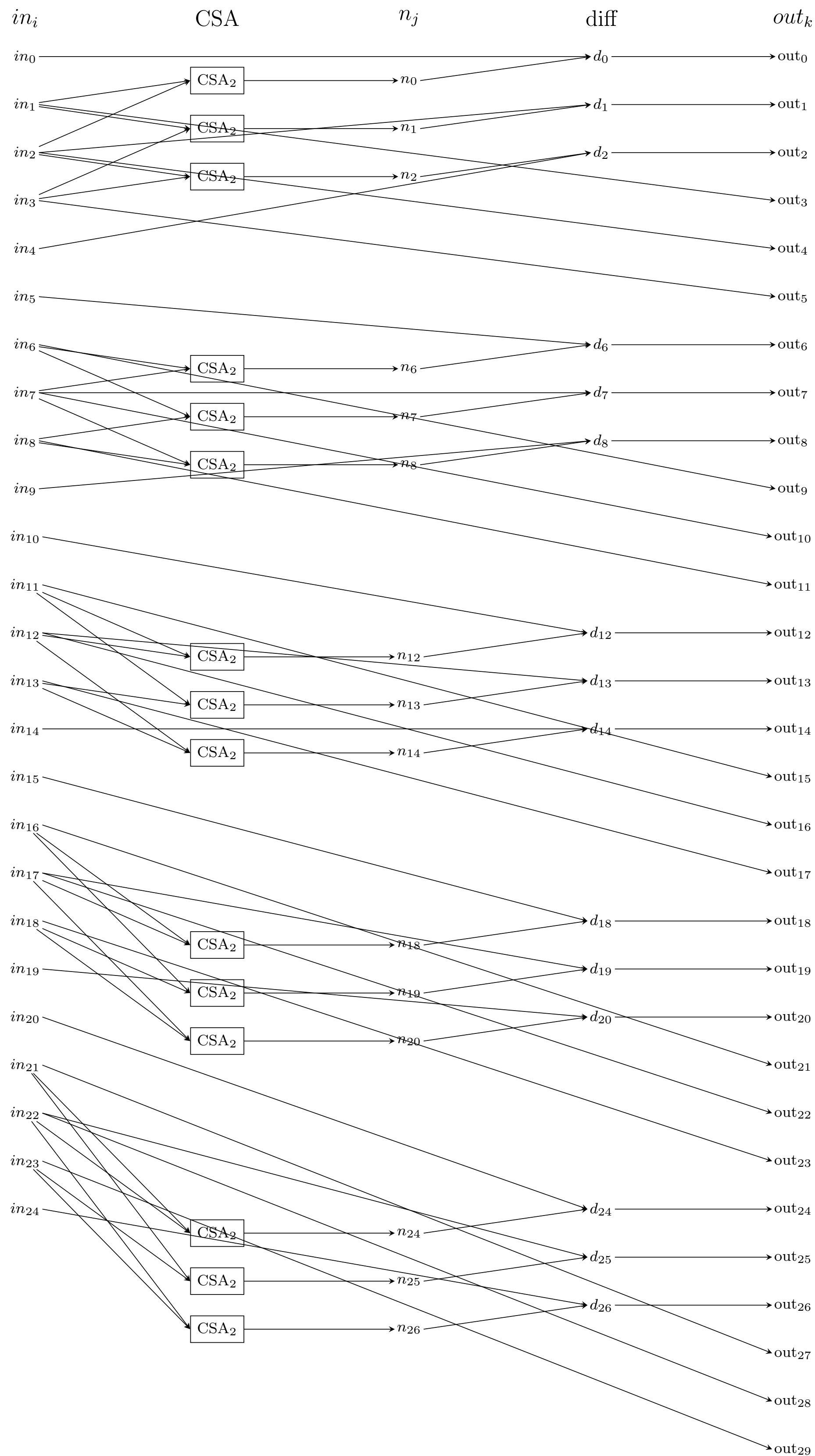
Transform and Inverse modules implement the families of linear combinations defined by the rows of C and A. Each stage is implemented as a constant-coefficient reduction network built from additions, subtractions, and bit shifts: small integer coefficients are decomposed into signed sums of powers of two so that multiplications by constants become combinations of shifts and additions. For each output component, terms with positive and negative sign are accumulated in two carry-save adder trees followed by a single subtraction, yielding shallow, regular networks in which most of the work is carried out by parallel additions and subtractions are confined to the final layer. This organization preserves a direct correspondence between the algebraic description of the transforms and their hardware realization—changing $\mathbf{C}$ or $\mathbf{A}$ only modifies the reduction coefficients, not the pipeline structure—and the two-stage separable mapping (C^{T} then C, and analogously for A) shortens critical paths and scales better to larger tiles than naïve matrix multiplication based on general-purpose multipliers. Figure 6.5 illustrates the internal organization of a representative C^{T} stage, showing how a subset of input samples associated with one row of $\mathbf{C}$ is reduced by small adder trees into the corresponding transform outputs.

6.3.3 Hadamard Products and Multiplier Sharing

The only true multiplications in the convolution core occur in the Hadamard-product phase, where the entries of the transformed input tile must be multiplied by the corresponding entries of the transformed filter. Instead of instantiating one multiplier per entry, the design uses a small set of shared multipliers and schedules the products across multiple cycles. This approach aligns with the overall design goal: the tile-level throughput should be matched to the memory bandwidth, so there is little benefit to gathering all products in a single cycle if the memory system cannot supply tiles at that rate.

The Hadamard engine is built from three elements:

- a counter that tracks the current iteration of the Hadamard loop;
- a multiplexer index generator that maps the counter to an array of indices selecting which entries of D are processed in the current cycle;



> The diagram shows how selected input samples in_i are grouped and combined by small carry-save adders (CSA) and subtractions to implement one structured row of the constant matrix C.

Figure 6.5: Internal organization of a representative `Transform MatrixC0` stage for the nested IF convolution.

- a small parallel array of identical **Multip** units, which implement the actual fixed-point multiplications between feature and weight entries in each cycle.

On each cycle of this phase, the index generator receives the current loop iteration and produces indices for all multipliers in the tile. These indices are used to select entries of the register bank and the corresponding entries of the weight vector. Each pair is fed into a multiplier, which computes a quantized signed product; internally, the **Multip** block first forms a wide product with $NBITS{+}QUANT$ bits and then discards the $QUANT$ least significant bits, which is equivalent to an arithmetic right shift by $QUANT$ bits. In other words, both input features and weights are represented in a signed fixed-point format with $NBITS$ total bits and $QUANT$ fractional bits, and the multiplier returns a value in the same format by truncating the fractional part of the full-precision product. This fixed-point scheme is consistent with the quantization used in the software and hardware implementations in Chapters 5 and 7. The resulting products are written back into the register bank at the specified positions, overwriting the previous values.

The duration of the Hadamard phase is determined by the number of iterations needed to process all entries of the tile using the available multipliers. For a given fast-convolution configuration with input-tile size M and output-tile size N, let K denote the total number of scalar products required by the algorithm for one transformed tile (the operation count of the Hadamard stage for that tile), and let P denote the number of multipliers. The number of Hadamard cycles is then:

$$N_{\text{cycles}} = \frac{K}{P},$$

so that the same algorithm can be mapped either to a wide, low-latency datapath (large P, small N_{cycles}) or to a narrower, multi-cycle datapath (small P, larger N_{cycles}) without changing the external tile interface. In the 2D IF configuration used throughout this chapter ($M = 5$, $N = 3$, $K = 6$), the Hadamard stage requires $K = 36$ products per tile. If the core instantiates $P = 6$ multipliers, the Hadamard phase therefore spans $N_{\text{cycles}} = 36/6 = 6$ cycles, or six multiplication stages, to process the full tile before the controller transitions to the inverse-transform modules.

6.3.4 State Machine of the Convolution Core

The sequencing of the convolution core is governed by a four-state FSM, depicted in Figure 6.6. Each state corresponds to one phase of the Winograd tile computation, and the transitions encode the progression from an idle core to a completed tile and back. Both the **Transform** and **Inverse** phases are implemented as combinational networks around the shared register bank, so the FSM spends one cycle in each of these states (to launch and store the result), whereas the **Hadamard** state may span several cycles, as determined by the ratio K/P discussed above (Section 6.3.3).

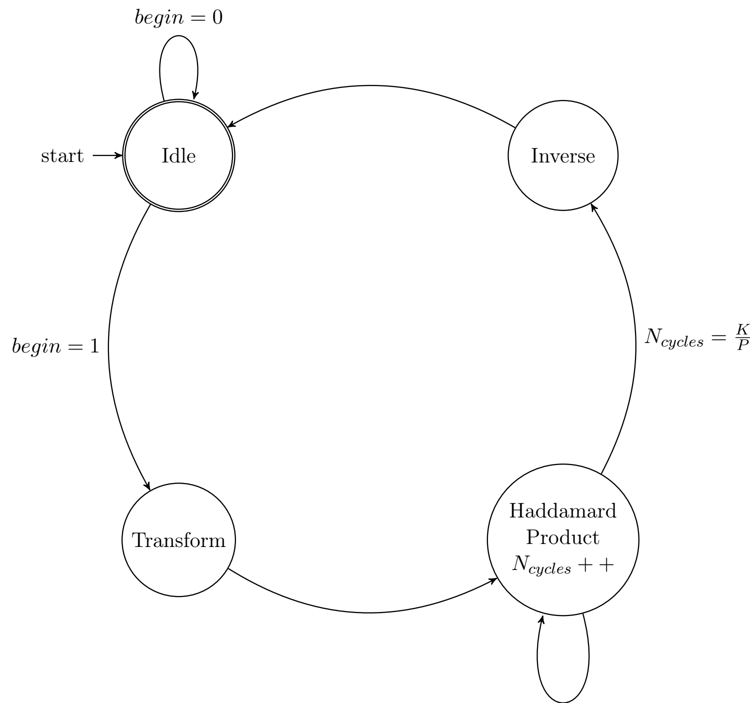


Figure 6.6: FSM for the fast convolution core.

In the **Idle** state, the core is ready to accept a new convolution request. As long as no new tile is requested, the machine stays in this state, and no internal registers are updated. When a new tile becomes available, the FSM leaves **Idle** and moves to the **Transform** state.

The **Transform** state represents the state in which the input tile is transformed into the Winograd domain using the matrix associated with C. In this phase, the spatial-domain samples stored in the internal register bank are combined by the transform module, and the resulting transformed values overwrite the original entries. Once the transform has been applied to the tile, the FSM advances to the **Hadamard Product** state.

In the **Hadamard Product** state, the core performs the elementwise products between the transformed input tile and the transformed filter coefficients. An internal counter tracks the number of subsets have been processed; as long as there remain unprocessed entries, the FSM stays in **Hadamard Product**. When all required products have been computed, the FSM exits the loop and transitions to the **Inverse** state.

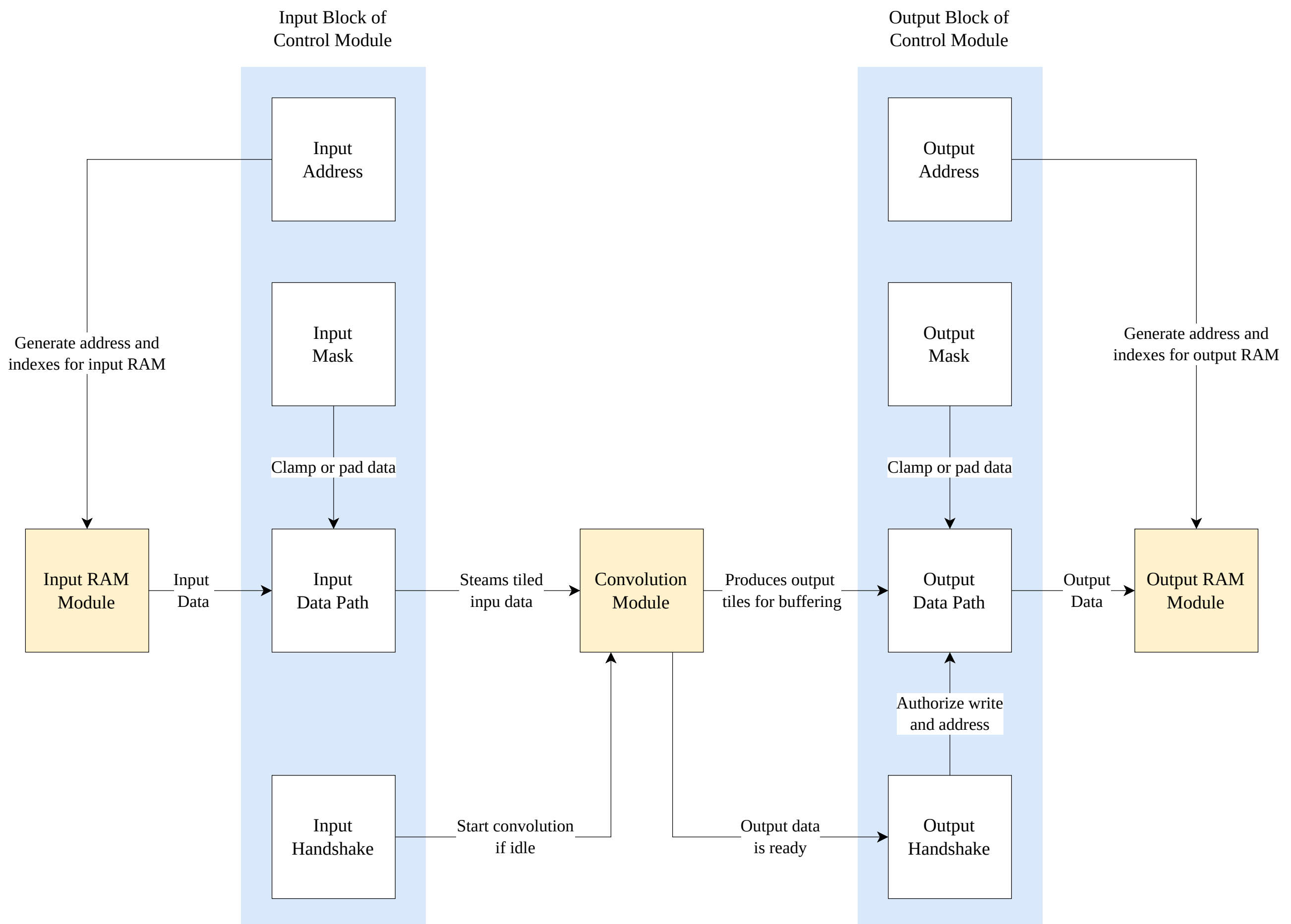
Finally, the **Inverse** state corresponds to the inverse Winograd transform, implemented by the module associated with A. In this phase, the products accumulated in the register bank are converted back to the spatial domain, producing the $M \times M$ output tile at the core output, where M denotes the linear size of an output tile. Once this transform has completed, the FSM returns to the **Idle** state, and the core is again ready to accept a new convolution request.

This FSM structure exposes the four algorithmic phases of a Winograd tile (load/-transform, Hadamard product, inverse transform, and idle) and reuses a single register bank

across all of them. As a result, the internal control logic adds only a small number of flip-flops and gates to the design, while still providing a structure to schedule the shared multipliers.

6.4 Control Unit and Scheduling

The **Control** module orchestrates the flow of tiles through the accelerator and sequences all memory accesses. Positioned between the external memory interfaces and the convolution core, it determines when to fetch weights and input tiles, when to initiate a Winograd convolution, and when and where to write back the resulting output tiles. To support these functions, the module employs two coupled FSMs: one managing the input side (weights and input features), and the other managing the output side (result write-back). Together, these FSMs implement the nested loops over sliding windows and channels that define a convolution layer, while enforcing the design choice of processing a single Winograd tile at a time within the core.



The input block generates addresses for the input RAM, clamps or pads samples, and streams tiled data to the convolution module. The symmetric output block buffers the results, applies output masking, and issues write operations to the output RAM.

Figure 6.7: High-level block diagram of the control module.

Figure 6.7 organizes the controller into two highlighted regions: an input block on the left and an output block on the right, separated by the convolution module for didactic purposes. The process starts at the input RAM module, whose addresses and window indices are produced by the **Input Address** sub-block; the **Input Mask** clamps or pads samples before they enter the **Input Data Path**, which streams tiled input data into the convolution module. The **Input Handshake** supervises the core and starts a new tile only when the convolution pipeline is idle. On the right-hand side, the **Output Data Path** receives output tiles from the core, the **Output Mask** formats or clips them as needed, and the **Output Handshake** authorizes each write while the **Output Address** block generates addresses and indices for the output RAM. In this way, the figure makes explicit that control flows from address generation on the input side to the final write-back into the output memory.

6.4.1 Functional Role of the Control Module

At a high-level, the control module plays three roles:

- **Layer-level control:** it receives a global start pulse, initializes internal counters and base addresses, and signals when the entire layer has been processed.
- **Tile-level scheduling:** it repeatedly fills an input window buffer with samples from the next spatial tile, associates them with the filter coefficients, and issues a start pulse to the convolution core whenever the buffer is full, and the core is idle.
- **Memory addressing and reuse:** it generates read and write addresses that implement a 2D sliding-window pattern over the IFMAP, taking advantage of overlapping columns between horizontally adjacent tiles to reduce input traffic, and mapping each OFMAP tile to the correct positions in the output feature map.

On the input side, the controller owns a local register bank for input features and a register bank for the kernel weights of the current input/output channel pair. On the output side, it maintains two small register banks: one that caches any existing output tile read back from memory and one that stores the freshly computed tile returned by the convolution core, so that both can be accumulated before write-back. The convolution core itself remains agnostic to the global layer structure: it simply consumes one input tile and its associated weights when requested and produces one output tile, while the control module is responsible for iterating over windows and channels and for combining contributions across input channels.

6.4.2 Input-side FSM

On the input side, control is expressed as an FSM that cycles through a set of states - Figure 6.8: **Idle**, **Weight**, **Read Input**, **Convolution**, **Hold Output**, **Hold Last Con-**

volution, and **End**. In the **Idle** configuration, the controller waits for a global start event; when it arrives, the address and window counters are reset, and the machine enters the **Weight** phase. In this phase, all coefficients of the filter kernel associated with the current input/output channel pair are streamed from memory into a local weight buffer until the full kernel footprint has been captured.

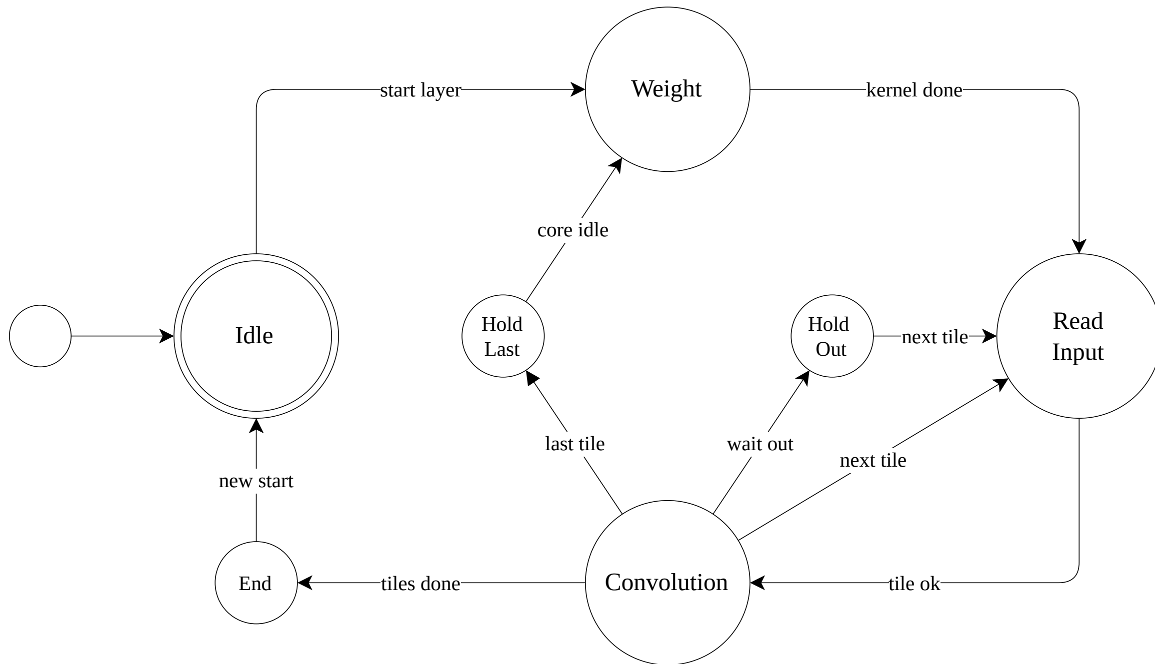


Figure 6.8: Input-side control state machine, showing the phases for loading weights, filling input windows, handing tiles to the convolution core, and coordinating the two hold phases across channels.

Once the channel's weights are available, the controller switches to the **Read Input** state. It walks over the input feature map in a 2D sliding-window pattern, repeatedly issuing memory reads and placing each sample into a specific position of the local window buffer according to a fixed mapping. This mapping implements the Winograd footprint: for the first window of each row, the controller gathers a full $N \times N$ patch of the IFMAP, which feeds the transform stage of the core, where N denotes the linear size of the input tile. Within each window, samples are fetched in column-major order (column by column, then row), which matches how overlapping columns are reused between horizontally adjacent tiles. For subsequent windows in the same row, the controller fetches only the $M - N$ new columns and reconstructs the rest of the patch from the overlapping samples already held in the buffer. In parallel, it maintains a hierarchy of counters that play the role of loop indices over horizontal window position, vertical window position, and input/output channel combinations. Internally, each input address is formed as a base pointer for the current window plus a row stride and a column offset; advancing horizontally increases the base pointer by M columns, while wrapping to the next row or to the next channel applies precomputed vertical and channel strides that reposition the base without changing the convolution core.

Viewed as nested loops, this behaviour corresponds to the following pattern: for each output channel, for each input channel that contributes to it, the controller iterates over all rows of windows in the feature map; for each such row, it steps through all window columns; and for each window it traverses all positions inside the $M \times M$ footprint to fill the local buffer. The innermost "pixel loop" is handled by the row and column indices within the tile, while the three outer loops over window row, window column, and channel combination are encoded in the window and channel counters that drive the base-address updates. The **Hold Output** and **Hold Last Convolution** states act as explicit waiting states in this schedule: they stall the input FSM whenever the output path or convolution core is temporarily busy, ensuring that no tile is dropped or overwritten while the corresponding results are still being consumed on the write side. At the borders of the feature map, the controller derives global row and column coordinates for each sample in the sliding window and uses these coordinates to decide whether a given access falls inside or outside the valid IFMAP region. When a window position extends beyond the feature-map boundaries, the corresponding samples are treated as zeros and are never read from memory, effectively implementing padding entirely within the control logic without changing the convolution core.

Whenever the window buffer is full for a given spatial position and channel pair and the convolution core indicates that it can accept new work, the controller reaches the **Convolution** phase: it hands the current window and kernel to the core and conceptually freezes them until the core has accepted the tile. Immediately after this handoff, the window and address counters are updated for the next spatial position. When moving one window to the right, the controller reuses the columns $N - M$ that overlap between neighboring tiles, or $(N - M) \times N$ samples, and fetches only the new M columns from memory, or $M \times N$ samples; when it reaches the last window in a row, it wraps to the first window of the next row by adjusting the base address vertically. When partial sums for an output tile are still resident in the write path, the controller may briefly enter a **Hold Output** or **Hold Last Convolution** phase to align the input schedule with the availability of the convolution core and the output accumulator.

Once all windows across all input and output channels have been handed to the core, the input-side FSM enters the **End** phase, in which it raises a layer-level done flag and stops issuing further memory requests. It only returns to the **Idle** configuration when a new global start event is received, at which point the same sequence of phases is replayed for the next convolution layer.

6.4.3 Output-side FSM

The output-side control is also expressed as a FSM, Figure 6.9, with the following states: **Idle** , **Read Output**, **Convolution Output**, **Write Output**, **End Channel**, and **Hold Weight**. Its role is to ensure that, for every spatial tile and for every output channel, the feature map stored in memory always reflects the accumulation of all contributing input

channels, and that tiles are written in an order consistent with the global layout of the output tensor.

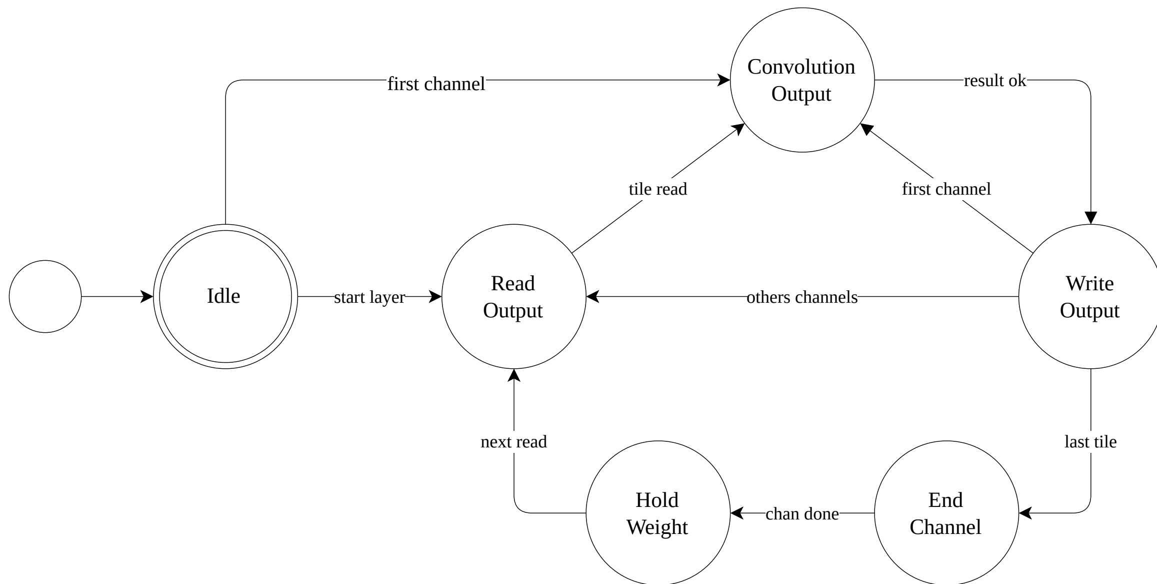


Figure 6.9: Output-side control FSM, showing the states for reading existing tiles, accepting new convolution results, writing accumulated tiles, and stepping across output channels.

The **Idle** state resets its internal counters at the beginning of a convolution channel. Once tiles start flowing through the datapath, a **Read Output** phase load the stored output tile from the output memory that will be accumulated with a new convolution output. Conceptually, this corresponds to reading the current partial sum for a given position and output channel and writing it locally so that it can be combined with the next tile produced by the Winograd core. A subsequent **Convolution Output** phase then receives an output tile from the core and stores it in a second local buffer; entry into this phase is gated by the result-side handshake described in Section 6.4.5, so the FSM only accepts a new tile when the core asserts that a complete result is ready and the output block is prepared to consume it. At this point, the information needed to update the corresponding output tile in memory is available.

In the **Write Output** phase, the controller walks over the tile footprint in row-major order and, for each element, forms the accumulated result (either a direct copy, in the single-channel case, or the sum of the newly computed value with the previous partial sum) and writes it back to memory. Output addresses are generated with the same pattern as on the input side: a base pointer for the current window is combined with a row stride, proportional to the full output feature-map width, and a column offset inside the tile. A hierarchy of counters tracks both the position inside the tile and the indices of the current window and channel within the layer. These counters drive the update of the base address and in-tile offsets so that the tile is always mapped to the correct coordinates in the output tensor, independently of the number of input channels that contribute to that position. When all elements of a tile have been committed, the same counters determine whether the next step is to advance horizontally

to the next window, wrap to the first window of the next row, or reset the spatial indices and move to the next output channel.

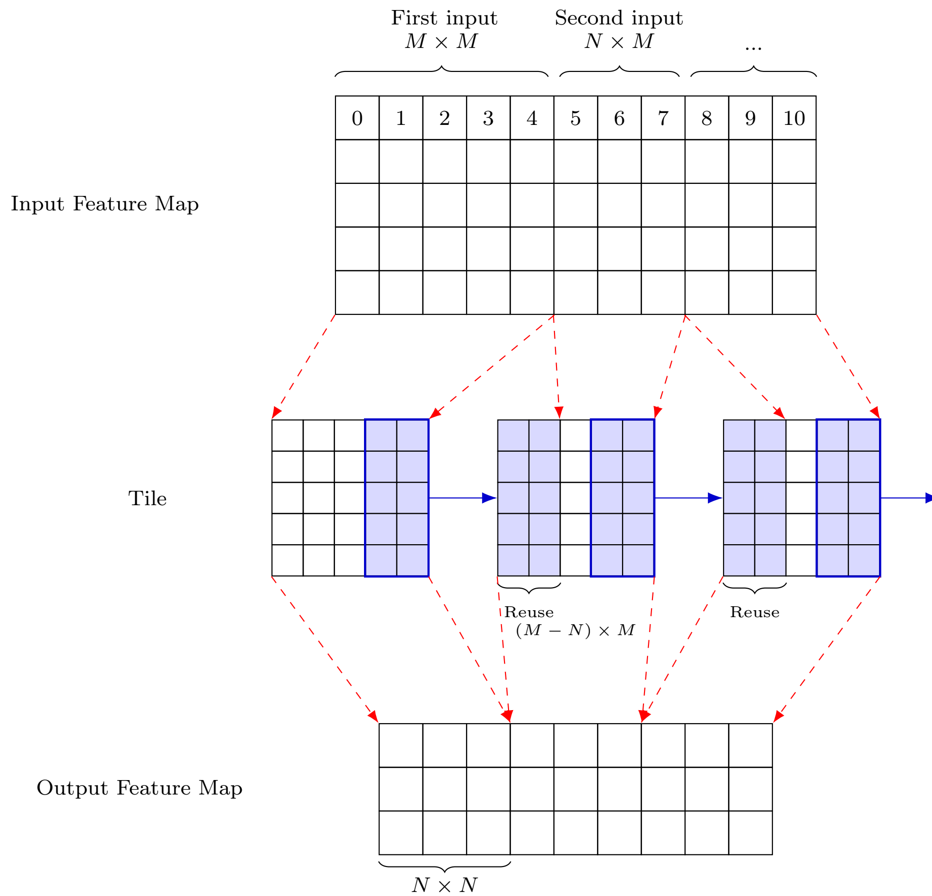
In loop form, the write side can be understood as iterating over all output channels, then over all rows of tiles, then over all tile columns, and finally over all positions inside each $M \times M$ output tile. The innermost loop over tile elements is implemented by the row and column indices within the tile, while the loops over tile row, tile column, and output channel are realized by the window and channel counters that update the base pointer of the output feature map. This structure ensures that, once all nested loops have completed, every valid spatial position and every output channel has been visited exactly once and holds the accumulated sum of all relevant input channels.

The **End Channel** phase deals with inter-channel bookkeeping and synchronization with the input-side control. When the last window of an output channel has been processed, the write-side logic resets the per-channel window counters and repositions its base pointer to the first tile of the next channel by applying a compile-time stride that corresponds to the full set of tiles that make up one output channel. If further accumulation is required for that channel, it may briefly enter a **Hold Weight** phase, waiting until the corresponding weights and input tiles have been loaded on the read side before starting the next sequence of **Read Output–Convolution Output–Write Output**. Once the counters indicate that all tiles of all channels have been emitted, the controller raises the global completion flag and returns to the **Idle** configuration, ready to serve the next convolution layer.

6.4.4 Interaction with Datapath and Memory

The two control FSMs provide the glue between the convolution core and the memory subsystem. Figure 6.10 illustrates this interaction using an IF/Toom–Cook configuration with $M = 5$ and $N = 3$. On the top, the **Input Feature Map** grid is indexed by column, and the braces labelled $M \times M$ and $N \times M$ highlight, respectively, the footprint of the first input tile and the additional columns that must be fetched to advance the sliding window horizontally. For the first window in a row, the controller reads $M \times M = 25$ samples from memory. For subsequent windows, it advances the window by N columns, reads only an $N \times M = 15$ samples at the new positions, and reuses the remaining $(M - N) \times M = 10$ samples from the previous window by keeping the last $(M - N) = 10$ columns in a vertical buffer local to the datapath. Below, the three **Tile** matrices correspond to successive Winograd tiles extracted from the IFMAP. Blue rectangles and the “Reuse” annotations emphasise that horizontally adjacent tiles share their border columns: when the window moves from **first tile** to **second tile**, only the new $N = 3$ columns need to be read, while the overlapping columns are provided from the buffer instead of being re-fetched from memory.

The vertical movement follows the same stride-N logic: when the controller finishes a row of windows and returns to the first column, it advances the IFMAP base address by



An $M \times M$ input window scans the IFMAP to feed Winograd tiles into the core, while an $N \times N$ footprint slides over the OFMAP to accumulate the corresponding output tiles at the correct spatial positions. In this illustration, we use $M = 5$ and $N = 3$, matching the input and output tile sizes of the Inspection Factorization (IF) and Toom–Cook-3 (TC^9) schemes discussed in Chapters 2 and 4.

Figure 6.10: Conceptual sliding-window view of the accelerator.

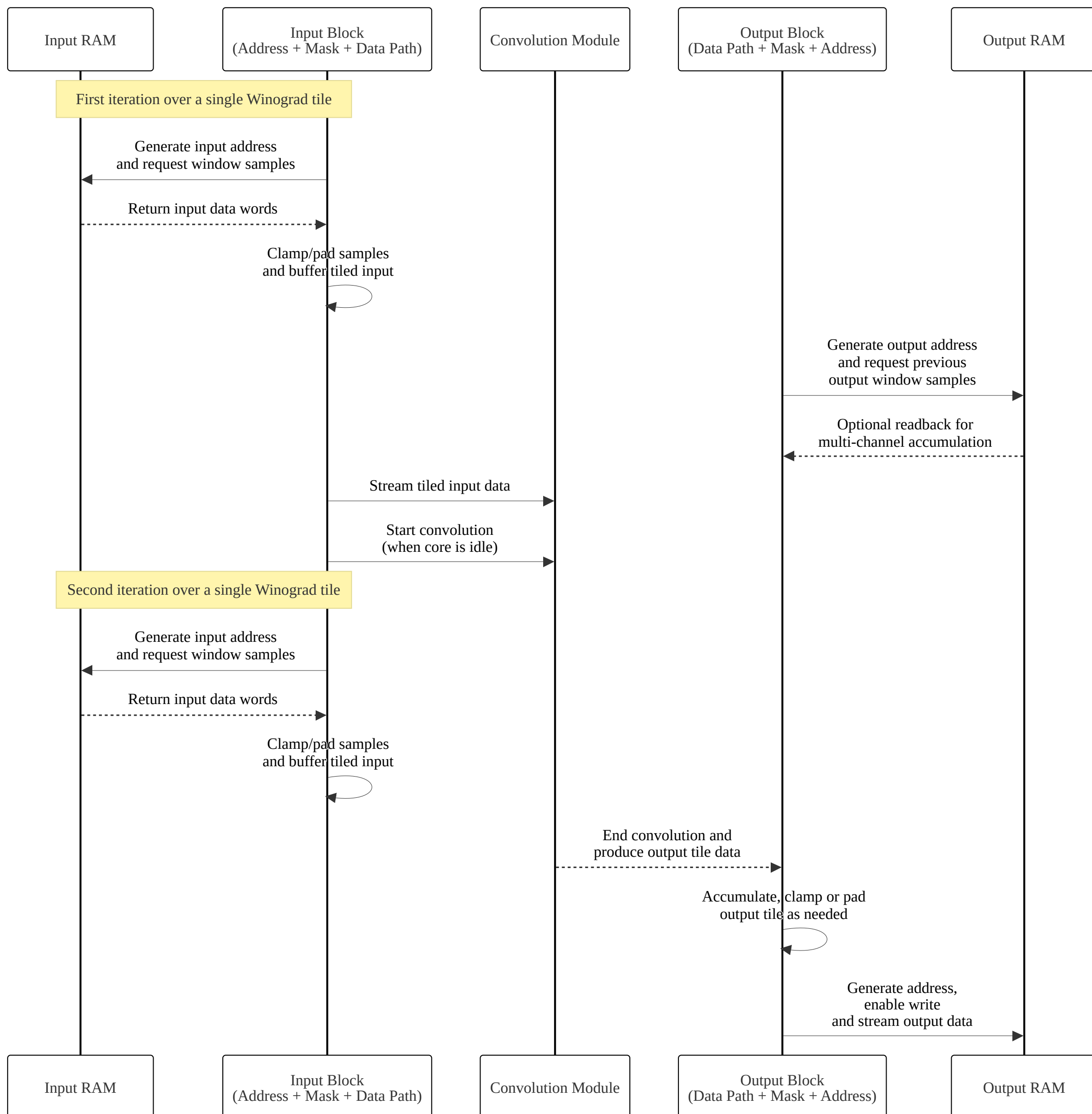
N rows and starts reading the next band of tiles. In the current implementation, however, no vertical reuse buffer is instantiated, so samples from different tile rows are always reloaded from memory; supporting vertical reuse would require an additional buffer to hold overlapping rows across tile rows.

Output Feature Map grid and the brace labeled $N \times N$ depict how each tile produced by the core is accumulated into a contiguous $N \times N$ region of the OFMAP. Dashed red arrows map ranges of input columns to the tiles that consume them and then from each tile to the corresponding $N \times N$ output patch, showing how the controller walks across an output row. Dashed blue arrows between tiles represent the reuse of transformed data across neighboring windows. Together, these elements make explicit that the controller implements convolution as a sequence of overlapping $M \times M$ input windows and $N \times N$ output footprints, while systematically reusing data horizontally to reduce memory traffic.

From the core's perspective, this logic simply presents self-contained tiles of input data and their associated kernels whenever the core is idle and ready, and signals when the sequence

of tiles for a given layer has been exhausted. The convolution core itself has no knowledge of the global layer topology or how tiles are sequenced in memory.

Figure 6.11 highlights this interaction from a data-centric perspective. The input block repeatedly generates addresses for the input RAM, clamps or pads samples, and assembles Winograd tiles that are streamed into the convolution core. In parallel, the output block receives completed tiles from the core, accumulates and formats them, and issues write commands and addresses to the output RAM. The sequence diagram also makes explicit that request generation for the next tile can overlap with the processing and write-back of the current one, exposing the decoupling between memory traffic and the internal Winograd pipeline.



> The input RAM is read under the supervision of the input block, tiled data is streamed into the convolution module, and the output block buffers, masks, and writes tiles into the output RAM while requests for subsequent tiles are already being prepared.

Figure 6.11: Sequence-level view of data movement across the control unit.

6.4.5 Handshake Protocol and Timing

The handshake protocol used in the control unit was introduced for three related reasons. First, it avoids unnecessary stalling of the control FSMs: instead of keeping a state artificially "busy" while the convolution core finishes its work, the handshake allows a state to leave as soon as a fire pulse has been issued, letting the FSM progress and prepare the next tile. Second, it ensures that each tile of input data and each convolution result is consumed exactly once. The fire and accept pulses act as unique tokens that mark the moment when a particular window or result is handed over, preventing the same data from being reused or overwritten inadvertently. Third, it decouples producers and consumers under back-pressure: if either the core or the output path is temporarily unable to accept new data, the handshake allows the other side to pause without losing data or violating ordering, so memory latency or core stalls do not propagate as protocol errors.

From a protocol perspective, both interfaces follow the same synchronous ready/-valid handshake: the tile-request interface between the input FSM and the convolution core implements a decoupled, latency-insensitive ready/valid handshake, while the result interface between the core and the output FSM follows the same pattern but with an implicit single-entry buffer that keeps each completed tile in a local register until it is consumed. In this way, every result is accepted once and consumer is not forced to operate at a fixed rate.

Figure 6.12 shows the handshake of the input FSM, with the goal of ensuring the correct synchronization between the input data and the convolution core. The protocol starts in label **a**. When **ready for input** rises while the input FSM is in **CONV IN**, the controller emits a single-cycle pulse (**a**→**b**) that starts the convolution. This pulse also asserts the internal busy register (**b**→**c**), so no additional requests can be issued until the current tile finishes. When the convolution core raises **convolution end**, the busy flag is cleared and the core returns to idle (**d**→**e**), which allows **ready for input** to rise again (**e**→**f**). The end pulse also re-arms the ready signal (**e**→**f**), closing the loop and guaranteeing exactly one fire per tile.

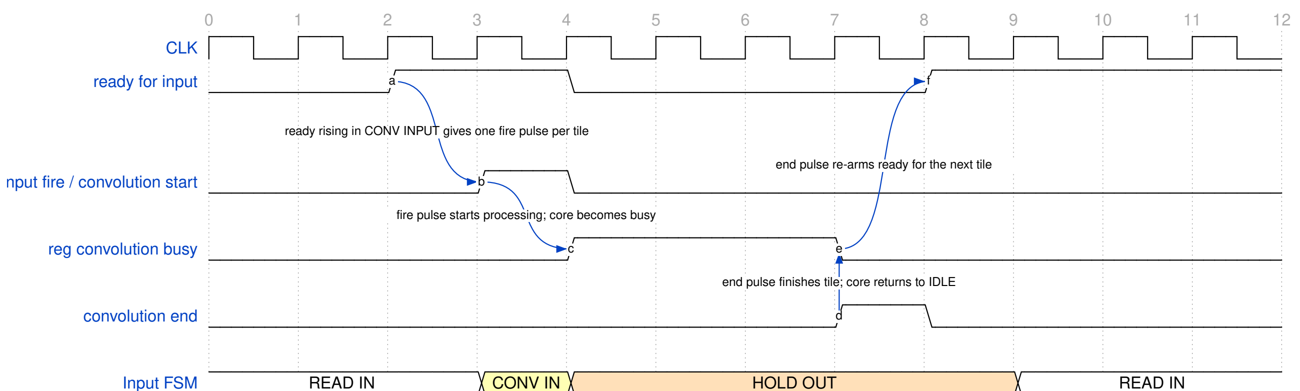


Figure 6.12: Input-side handshake between the control unit and the convolution core from a simplified behavioral trace.

Figure 6.13 shows the handshake of the output FSM, with the goal of ensuring the correct synchronization between the convolution core and the output data. A **result ready** pulse from the core stores the pending-result register (**a**→**b**). When the output FSM reaches **CONV OUT**, it asserts a single **result accept** pulse (**b**→**c**) and immediately clears the pending flag (**c**→**d**). If another result becomes ready while the FSM is not in **CONV OUT**, the pending flag remains asserted (**e**→**f**) until the next accept occurs (**f**→**g**), after which the pending flag returns to idle (**g**→**h**), ensuring that each completed tile is consumed exactly once.

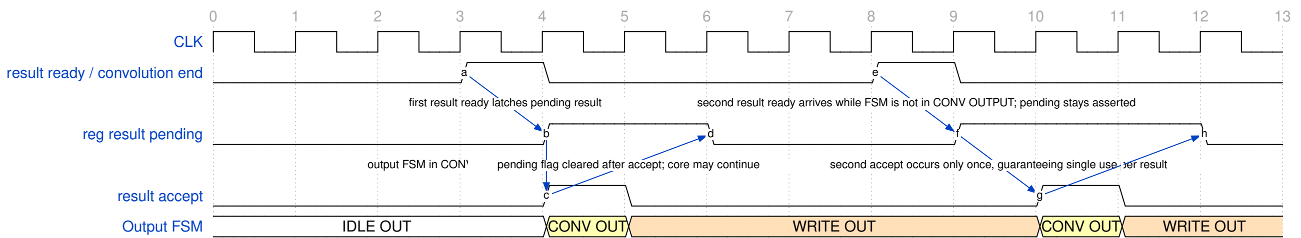


Figure 6.13: Output-side handshake timing between the convolution core, the output control block, and the output RAM interface, highlighting when completed tiles are committed to memory.

Figure 6.14 is a compact timing trace that groups actions of input memory, convolution and output memory into a single view for IF with six multipliers. The input side reads each tile in a burst of 15 cycles (**a**↔**b**) and then waits 4 cycles before the next read burst (**r**↔**s**). As soon as the read burst finishes, the core is fired in the same cycle (**a**→**x**), and the simplified **CONV** state spans 9 cycles (**c**↔**d**) before the **convolution end** pulse is raised (**d**→**e**). On the output side, each tile triggers a read of the OFMAP buffer (**e**↔**f**) followed by the corresponding write-back window (**g**↔**h**), with a 19-cycle gap between the read and write phases (**t**↔**v**). The edge labels indicate same-cycle triggers between signals, making explicit which events launch the core and which events commit the output tile.

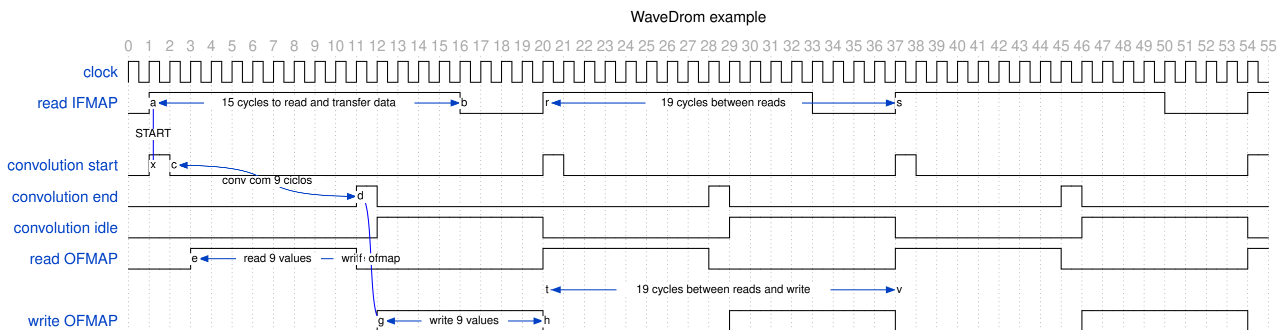


Figure 6.14: Approximate timing diagram for convolution core, input and output memory of IF with six multipliers

To maintain synchronization between the input and output control state machines, it is important that each has the same number of cycles to complete its operation. Figure 6.14 provides a concrete example: the output-side FSM consumes 19 cycles per tile, consisting of 9 cycles to read the OFMAP window, 1 cycle to store the convolution result, and 9 cycles to

write the updated window back to memory. The input-side FSM, by contrast, needs only 16 cycles: 15 to read the IFMAP tile and 1 to hand the tile to the convolution core. To prevent desynchronization, the input FSM must therefore insert 4 additional cycles in **HOLD OUTPUT** so that both controllers complete the tile in lockstep. This keeps the tile-level schedule aligned while explicitly acknowledging that the output side is the throughput bottleneck.

6.5 Conclusion

This chapter presented the hardware implementation of the proposed Winograd convolution accelerator. Starting from the design goals and constraints imposed by embedded and IoT deployment scenarios, this motivated the use of Winograd fast convolution not to maximize throughput, but to reduce arithmetic complexity and on-chip storage while operating at the boundary imposed by memory bandwidth. The chapter described the top-level **System** module, the internal organization of the Winograd convolution core and its matrix-multiplication engine, and the control unit responsible for scheduling tiles and managing register lifetimes. By explicitly exploiting tile overlap, limiting the number of in-flight tiles, and carefully aligning the computation rate with the available memory bandwidth, the design achieves significant savings in flip-flop usage and energy without sacrificing effective throughput.

7. RESULTS

This chapter presents the Power–Performance–Area (PPA) evaluation of the Convolutional Core (Section 7.1) and the complete system (Section 7.2). Performance is first evaluated analytically and then validated through RTL simulation.

The area is measured after logical synthesis using CADENCE *Genus*, with TSMC 28 nm technology and a clock constraint of 500 MHz, using the total cell area as the reference metric. Since routing is performed over standard cells, the reported total area does not directly reflect the effective silicon area.

Power estimates reported in this chapter follow an activity-annotated flow. The synthesized netlist is simulated in CADENCE *Xcelium* to capture switching activity, and the resulting activity data are back-annotated into *Genus* for power estimation. To obtain representative switching activity for power estimation, all experiments use inputs sampled from a normal distribution and quantized to 8 bits (word size 20 bits). Energy is computed as the product of the average power and the execution time.

Table 7.1 lists the fast-convolution algorithms evaluated in this chapter, and in the first row is the reference algorithm - *Naïve*. Each algorithm is identified by a symbol and specifies the tile dimensions used by the corresponding algorithm: an input tile of size $M \times M$ that produces an output tile of size $N \times N$. The pair (M, N) defines both the computational granularity of the datapath and the number of tiles required to cover a feature map. These parameters are used throughout the analytical and simulation-based evaluations.

Table 7.1: Evaluated algorithms with the sizes of the input M and the output N.

Symbol	Name	M	N
Naïve	Standard MAC-based convolution	3×3	1
IF^9	Inspection Factorization	5×5	3×3
TC^9	Toom Cook	5×5	3×3
WM^4	Winograd Minimal filter	4×4	2×2
TC^{16}	Toom Cook	6×6	4×4
WP^{16}	Winograd Polynomial	6×6	4×4

7.1 Convolutional Cores Evaluation

Subsections 7.1.1 to 7.1.5 present performance evaluations (analytical and simulated), area, power, and energy estimates. Subsection 7.1.6 concludes this section by discussing the limitations of the Kronecker method and the performance of the synthesis tools, showing that it is not necessary to implement CSA adders explicitly in the RTL code.

7.1.1 Analytical Performance Evaluation

We begin by considering an idealized operating regime to isolate the theoretical peak performance of the convolutional cores. The model assumes ideal memory behavior: input samples and transformed coefficients are available without latency, and result write-back incurs no delay. Under these conditions, the execution cycle count is determined by the internal computational resources, specifically the degree of multiplier parallelism and the scheduling constraints (steps executed in the FSMs that control the cores).

This analytical model, therefore, establishes a *compute-bound upper limit* on achievable throughput. Any practical implementation, subject to finite memory bandwidth, buffering requirements, and control overhead, can at best attain this performance and will generally exhibit lower throughput.

The compute-only upper bound is summarized by the total-cycle functions for fast convolution:

$$C_{\text{fast}} = C_{\text{in}}\, C_{\text{out}} \left(\frac{F_{\text{map}}}{N}\right)^2 \left[\left(\frac{K^2}{M_{\text{AC}}}\right) + 2\right],$$

and, for the Naïve baseline:

$$C_{\text{naive}} = 27 C_{\text{in}}\, C_{\text{out}} \left(\frac{F_{\text{map}}}{3}\right)^2.$$

where: C_{fast} is the cycle count for fast convolution; C_{naive} is the cycle count for the Naïve baseline; C_{in} is the number of input channels; C_{out} is the number of output channels; F_{map} is the output feature-map size (one dimension); N is the output tile size of the fast convolution algorithm; K is the transform-domain tile size (one dimension); M_{AC} is the number of parallel MAC units. The detailed derivation of these expressions is provided in Chapter D.

Table 7.2 presents the algorithms evaluated in this chapter in the first column. Subsequently, the term K^2 is reported for each algorithm, along with the possible number of MAC units used in their implementations. The number of MACs must be divisible by K^2 to avoid MAC underutilization. The last column illustrates how the K^2 multiplications are performed based on the number of MACs, defined as the product of the number of MACs and the number of iterations in the control FSM (Hadamard states).

Table 7.2: Evaluated algorithms, and the number of configurations of each one, defined by the number of MACs.

Algorithm	K^2	MACs	Example
IF^9	36	{1, 2, 3, 4, 6, 9, 12, 18, 36}	12 MACs and 3 Hadamard states
TC^9	25	{1, 5, 25}	5 MACs and 5 Hadamard states
WM^4	16	{1, 2, 4, 8, 16}	8 MACs and 2 Hadamard states
TC^{16}	36	{1, 2, 3, 4, 6, 9, 12, 18, 36}	6 MACs and 6 Hadamard states
WP^{16}	64	{1, 2, 4, 8, 16, 32, 64}	16 MACs and 4 Hadamard states

Figure 7.1 plots the performance of the algorithms according to the analytical expressions. The reference is the Naïve convolution, represented by a red line with a constant execution time. It can be observed that beyond a certain number of MAC units, performance no longer decreases proportionally. Architecturally, this indicates that allocating area to additional multipliers beyond this "saturation" point is unlikely to yield proportional performance gains; therefore, such MAC configurations should be avoided.

Figure 7.1 establishes the compute-bound ceiling of the convolutional cores. The key result is not that "fast schemes are faster", but that the cores exhibit a performance saturation point: beyond a moderate number of multipliers, the reduction in execution cycles per additional multiplier becomes marginal.

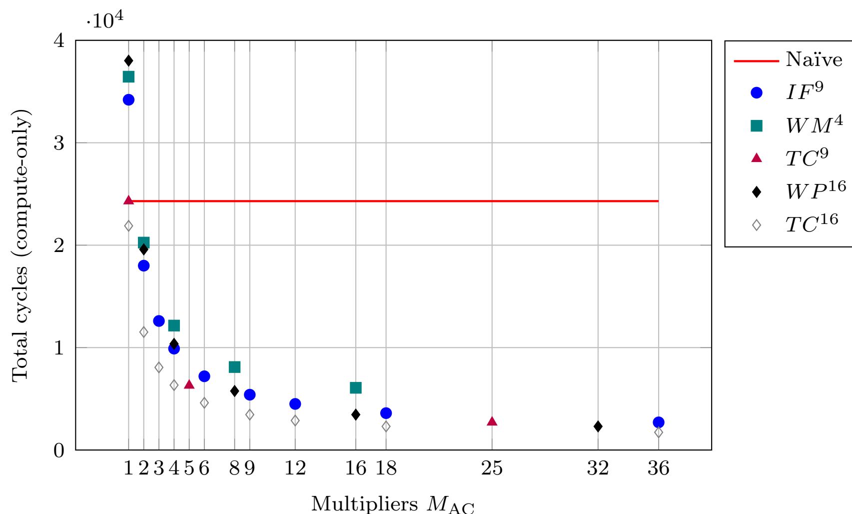


Figure 7.1: Compute-only total cycles for for $F_{\text{map}} = 32$ and $C_{\text{in}} = C_{\text{out}} = 3$, compared against the Naïve baseline.

Table 7.3 reports the performance ratios, using the Naïve algorithm as a reference, at two configurations per algorithm. In the *Balanced* case, the number of multipliers corresponds to the MAC count at which performance starts to "saturate". The second configuration corresponds to the maximum MAC count allowed by the algorithms. The values corresponding to the *Balanced* cases indicate the trade-off between performance and silicon cost. Beyond this number of MAC units, further performance gains incur a large silicon cost.

Table 7.3: Analytical compute-only ratio summary for the evaluated fast-convolution algorithms. Ratio corresponds to the performance improvement w.r.t. the Naïve algorithm.

	Balanced		Max MAC number	
Algorithm	MACs	Ratio	MACs	Ratio
WM^4	8	67%	16	75%
IF^9	6	70%	36	89%
TC^9	5	74%	25	88%
WP^{16}	8	76%	32	90%
TC^{16}	6	81%	36	93%

7.1.2 Simulated Performance Evaluation

Figure 7.2 presents measured cycle counts for the convolution cores obtained via RTL simulation. Note that this graph does not include all MAC configurations due to limitations of the RTL implementation. Additionally, the convolution cores support only a single channel, whereas multi-channel support is implemented in the control unit. Therefore, the plotted values are scaled by 9 (3×3) to account for three input and three output channels, enabling a direct comparison with the analytical evaluation.

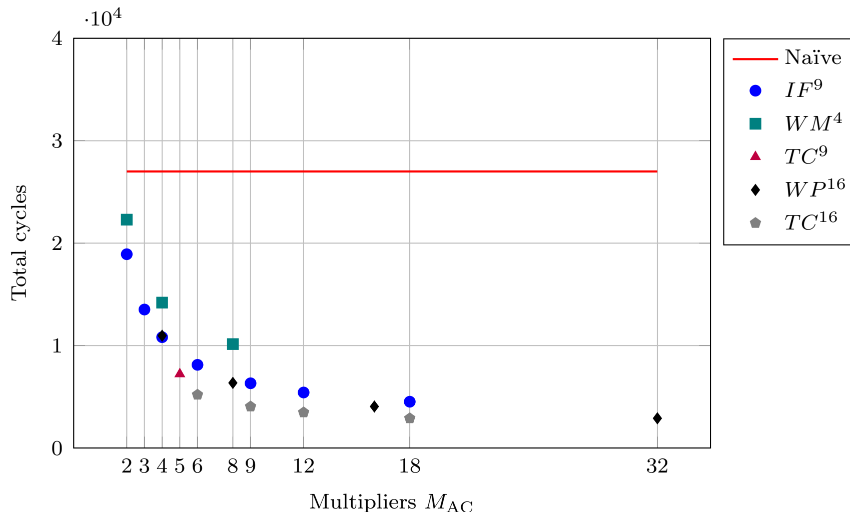


Figure 7.2: Simulated performance for the evaluated algorithms (y-axis), and number of MACs (x-axis). The reference, Naïve, requires 27000 cycles to execute.

This graph validates the analytical model: performance improves with an increasing multiplier count, then flattens as the number of MAC units increases.

Table 7.4 reports the performance ratios using the Naïve algorithm as a reference, following the same approach as Table 7.3, which relies on the analytical model. The simulated values are lower; however, the relative ordering across algorithms and the performance follow the same trend. This difference is explained by the idle cycles required for data movement. The analytical model can thus be used to compare algorithms under a common compute-only reference, without requiring RTL implementations for every configuration.

Table 7.4: Simulated compute-only ratio summary for the evaluated fast-convolution families.

	Balanced		Max MAC number	
Name	MACs	Ratio	MACs	Ratio
WM^4	8	62%	-	-
TC^9	5	72%	-	-
IF^9	6	69%	18	83%
WP^{16}	8	76%	32	89%
TC^{16}	6	80%	18	89%

This first set of results demonstrates the potential of fast convolution algorithms to reduce the number of cycles required to execute the convolution, which is the fundamental premise of this Dissertation.

7.1.3 Area

Figure 7.3 presents the total cell area, and as expected, the area increases with the multiplier count. The graph highlights the following observations:

- The Winograd minimal filter (WM^4) exhibits a smaller area than the Naïve baseline across all evaluated multiplier counts. This result indicates that this algorithm can deliver higher throughput at a lower area than a straightforward MAC-based implementation.
- The IF^9 configuration with up to six multipliers also presents a smaller area than the Naïve convolution while simultaneously delivering higher throughput and lower area.

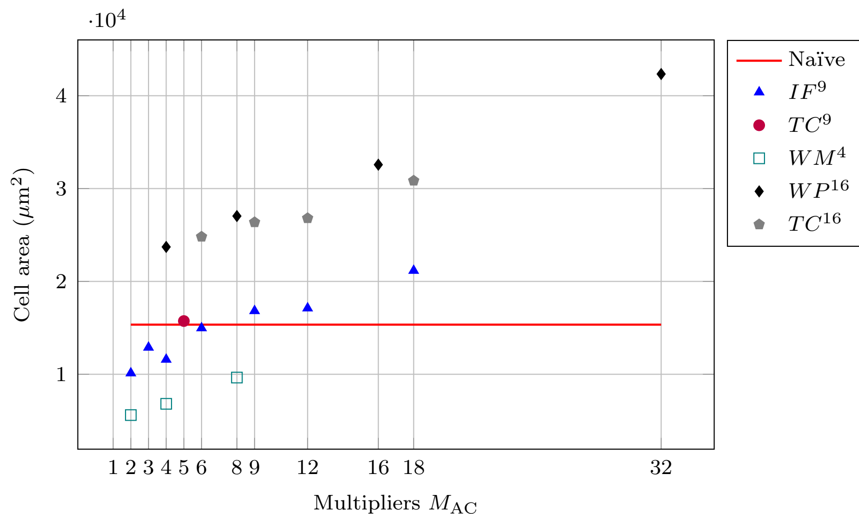


Figure 7.3: Total ell area per configuration (μm^2) as a function of the multiplier count.

Fast convolution algorithms trade multiplications for transforms, and the cost of these transforms is not limited to adders: it also includes intermediate storage, such as register banks. As K^2 increases, both the number of registers and the control complexity grow. Consequently, when a scheme reduces the number of multiplications but significantly increases K^2, it may become area-inefficient, even if its arithmetic count appears attractive in analytical form. This explanation clarifies why WP^{16} and TC^{16} exhibit a larger area than the reference naïve convolution.

7.1.4 Power

Figure 7.4 presents the estimated power consumption as a function of the number of MAC units. Power dissipation closely tracks cell area, as both dynamic and leakage power scale primarily with the total number of cells rather than solely with the number of multipliers. Among the evaluated schemes, the Winograd minimal filter (WM^4) stands out as the only algorithm that consistently remains below the Naïve baseline across all configurations, establishing it as an option when performance gains must not come at the expense of area and power.

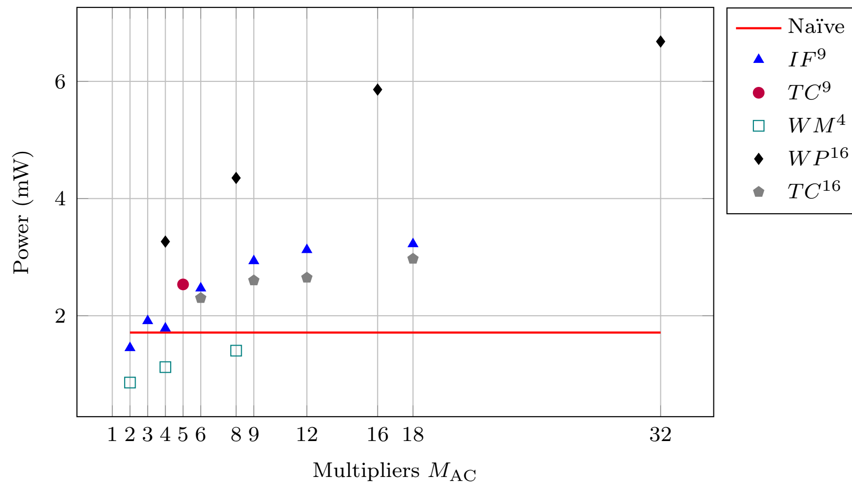


Figure 7.4: Power consumption (mW) as a function of the multiplier count.

This plot also supports a system-level warning: power consumption increases with parallelism. If the datapath is not kept busy, for example, due to bandwidth limitations, the accelerator may operate for longer periods at higher power levels, directly degrading energy efficiency. The system-level evaluation confirms that narrow bus configurations indeed underutilize the convolution core.

7.1.5 Energy

Figure 7.5 summarizes the energy consumption as a function of the number of MACs. Despite the higher power dissipation, the reduction in the number of cycles required to execute the convolution makes all fast convolution algorithms more energy-efficient than the Naïve convolution.

Figure 7.6 plots average power (y-axis) against execution time expressed in cycles (x-axis) for the convolution core, i.e., the two quantities whose product determines energy consumption. Consequently, the position of each point directly reflects how a configuration attains energy efficiency: by shortening execution time (moving left), by lowering power (moving down), or by combining both effects. This highlights that different fast-convolution configurations can achieve low energy through different mechanisms. Some algorithms primarily reduce the cycle

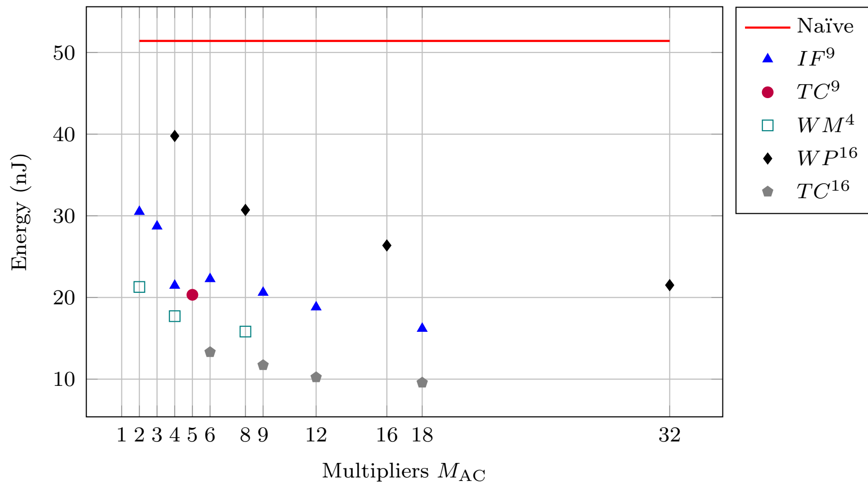
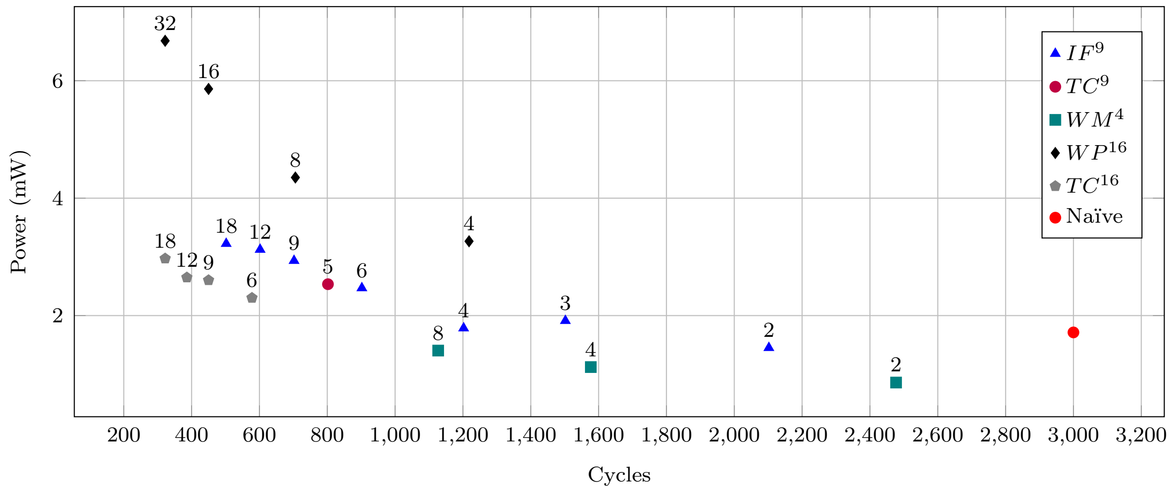


Figure 7.5: Energy evaluation as a function of the multiplier count.

count, often at the cost of higher power dissipation. In contrast, others reduce power while providing more modest cycle reductions. The most energy-competitive designs concentrate toward the lower-left region (e.g., TC^{16} and IF^9). The Naïve reference, in contrast, lies far to the right, indicating a substantially longer execution, which makes its energy less competitive even without being the highest-power point.



Each point is labeled with the algorithm and multiplier count. The reported energies are scaled by 9 to account for three input and three output channels, since the convolution core operates on a single channel.

Figure 7.6: Convolution-core power–cycle scatter plot.

Table 7.5 reports normalized ratios of performance, cell area, power, and energy relative to the Naïve baseline. The results are organized into two operating points: a *Balanced* configuration and a *Max MAC number* configuration. Positive values indicate improvements over the Naïve convolution, whereas negative values indicate degradations.

The normalized PPA ratios in Table 7.5 provide a rationale for the literature clustering shown in Figure 3.3. The WM^4 algorithm (2×2 in Figure 3.3) achieves consistent reductions in area and power while preserving the energy gains of fast convolution, which explains its prominence in efficiency-constrained designs. The TC^{16} family (4×4) emphasizes higher per-

Table 7.5: Evaluation of convolution algorithms. Normalized ratios of performance (simulation), cell area, power, and energy. All values are reported relative to the Naïve baseline under the same workload and clock constraint.

	Balanced					Max MAC number				
Name	MACs	Perf.	Area	Power	Energy	MACs	Perf.	Area	Power	Energy
WMn^4	8	62%	37%	18%	69%	-	-	-	-	-
TCn^9	5	69%	-3%	-48%	60%	-	-	-	-	-
IFn^9	6	72%	2%	-44%	57%	18	83%	-38%	-88%	69%
TCn^{16}	6	76%	-76%	-34%	74%	18	89%	-101%	-73%	81%
WPn^{16}	8	80%	-176%	-154%	40%	32	89%	-176%	-290%	58%

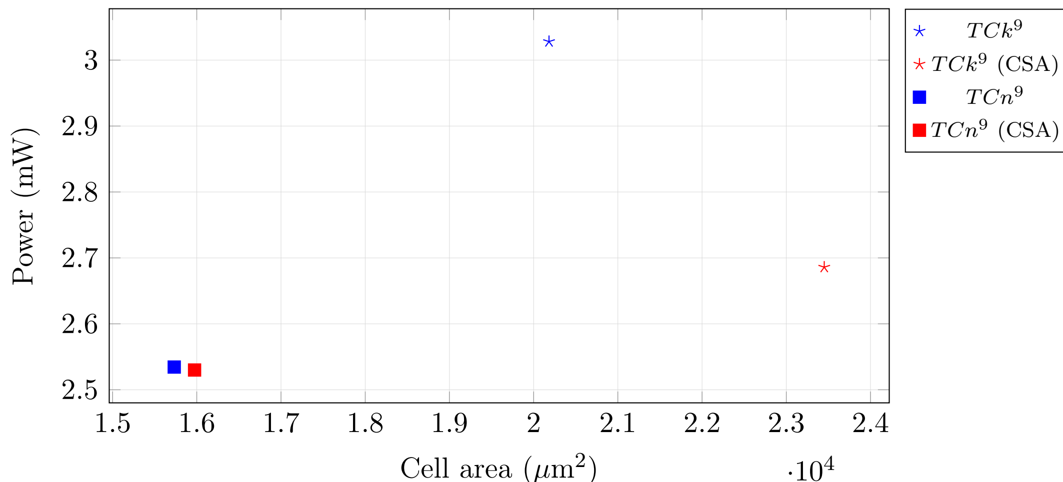
formance and energy improvements at the expense of increased area and power, consistent with its prevalence in throughput-oriented designs. The limited adoption of TC^9 (3×3) reflects its narrower design-space flexibility, *whereas the* IF^9 *proposed here offers an alternative with comparable performance–energy trade-offs, positioning it as a practical middle ground between* WM^4 *and* TC^{16}*.*

7.1.6 Kronecker Binding and CSA Implementations

Figure 7.7 isolates two independent architectural choices within the same Toom-Cook family: the binding strategy (nested vs. Kronecker) and the use of a carry-save adder (CSA) variant. Toom-Cook was selected for this case study because, unlike IF, WM, and WP, its forward and inverse transforms involve multiplications by constants other than 0 and 1. This property makes Toom-Cook a suitable case for evaluating whether CSA can effectively restructure constant multiplications into addition chains, and whether Kronecker binding exposes additional opportunities to absorb these constant operations into summation networks.

The results show that Kronecker binding increases both cell area and power, and that explicitly introducing CSA adders into the RTL code does not yield benefits. The additional logic and state required by Kronecker/CSA outweigh the potential advantages of simplifying constant multiplications into sums. For this reason, the remaining experiments in this work adopt nested binding without explicit CSA in the RTL code.

A common intuition is that schemes with simpler transforms, in which additions and subtractions are the predominant arithmetic operations, are preferable because multipliers are often viewed as the primary cost driver. The results indicate that this intuition does not necessarily hold in a modern ASIC flow: multiplier implementations are highly optimized for both energy and timing, and synthesis tools effectively reduce multiplications by constants, a central component of Toom–Cook transforms, particularly at larger sizes. Consequently, schemes such as IF and WP, whose main advantage lies in simpler transform arithmetic



> Markers encode the scheme family (star for TCk^9, square for TCn^9) and color encodes the CSA variant (red for CSA, blue otherwise).

Figure 7.7: Cell area (x-axis) versus power (y-axis) for the TCn^9 and TCk^9 configurations.

rather than aggressive multiplication reduction, may become less competitive when their larger K^2 footprint dominates area and power.

7.2 System level: Control and Convolution

7.2.1 Analytical Performance Evaluation

The core-level evaluation provides an upper bound on computational performance. System-level behavior, however, is constrained by the slowest stage in the end-to-end pipeline: bus throughput, memory latency, buffering, and control overhead can dominate even when the core is fast.

This section introduces the analytical performance models used to compare complete accelerators (control and core) against the Naïve MAC-based convolution baseline. The goal is not to model every microarchitectural detail but to capture the dominant factors that determine whether a configuration is compute-bound or memory-bound.

We first establish a Naïve-convolution reference based on the stride-1 environment of Juracy [2022]; Silva [2023], which was also adopted in Chapter 3 as the main RTL baseline. Compared with the original formulation, the model used in this work is explicitly parameterized by the number of input/output channels and an effective memory-latency factor.

For a square output feature map of size $F_{\text{map}} \times F_{\text{map}}$, the latency model is

$$f(F_{\text{map}}, C_{\text{in}}, C_{\text{out}}, L_{\text{mem}}) = 3\big(2 + F_{\text{map}}\big)F_{\text{map}}\, C_{\text{in}} C_{\text{out}} \quad \text{cycles per output feature map} \tag{7.1}$$

where: C_{in} and C_{out} denote the numbers of input and output channels, and L_{mem} is set to 1. The multiplicative constant 3 accounts for the number of memory reads per output: with stride-1, each 3×3 window reuses six of the nine values, so only three new samples need to be fetched. The additive constant 2 captures the two initial cycles per row required to prime the systolic array before steady streaming.

In the configuration used throughout this chapter ($C_{\text{in}} = C_{\text{out}} = 3$ and $L_{\text{mem}} = 1$), Equation (7.1) reduces to

$$
\begin{aligned}
f(F_{\text{map}}) &= f\Big(F_{\text{map}}, 3, 3, 0\Big) \\
&= 3\Big(2 + F_{\text{map}}\Big)F_{\text{map}} \cdot 9 \\
&= 27\Big(2 + F_{\text{map}}\Big)F_{\text{map}} \\
&= 27F_{\text{map}}^2 + 54F_{\text{map}} \quad \text{cycles per output feature map}
\end{aligned}
$$

We now summarize the analytical model used for the fast-convolution accelerator; the full derivation is provided in Annex D. The model uses the following parameters:

- M: linear input-tile size.
- N: linear output-tile size, such that each tile produces an $N \times N$ output block.
- w_{size}: effective word size (in samples per cycle) used to stream both input and output tiles.
- C_{in}, C_{out}: numbers of input and output channels, respectively.
- F_{ifmap}: linear size of the input feature map; for stride-1 convolution with $L = 3$, $F_{\text{map}} = F_{\text{ifmap}} - 2$.
- K: linear size of the transformed kernel, requiring K^2 elementwise products per tile.
- M_{AC}: number of MAC units.
- L_{mem}: memory latency.

Equation (7.2) presents the analytical performance model for the fast convolution algorithms. We neglect channel transient effects, such as the end of a row and channel changes.

$$
\tilde{f}_{\text{fast}}(F_{\text{ifmap}}) = C_{\text{in}}C_{\text{out}} \max\left(\frac{(2N^2+1)}{w_{\text{size}}} L_{\text{mem}}, \frac{K^2}{M_{\text{AC}}} + 2\right) \left(\frac{F_{\text{ifmap}} - 2}{N}\right)^2. \tag{7.2}
$$

Observe in Equation (7.2) the $\max(\cdot)$ operator, which explicitly separates memory bandwidth from computation. If the memory-bandwidth term dominates, the accelerator is *memory/bus-bound*, and compute resources are underutilized. Conversely, if the computation term dominates, the accelerator is *compute-bound*, and additional bandwidth will not improve performance. This property makes the model particularly useful for architectural exploration, as it directly identifies the dominant performance lever: w_{size} (and L_{mem}) versus M_{AC}.

To explore this trade-off, we consider three bus-width configurations for each convolution algorithm: (*i*) $w_{\text{size}} = 1$, corresponding to a 20-bit bus (the word size selected for the convolution cores); (*ii*) $w_{\text{size}} = M$, where M is the input tile size; (*iii*) $w_{\text{size}} = M \times N$, corresponding to the full tile size. Table 7.6 summarizes the configurations for cases (*ii*) and (*iii*). We focus on $w_{\text{size}} = M$ as the representative evaluated case because its implementation is simpler (see below Figure 7.9), it enables reading a tile column in a single cycle, and it can be applied in both steady-state and transient regimes.

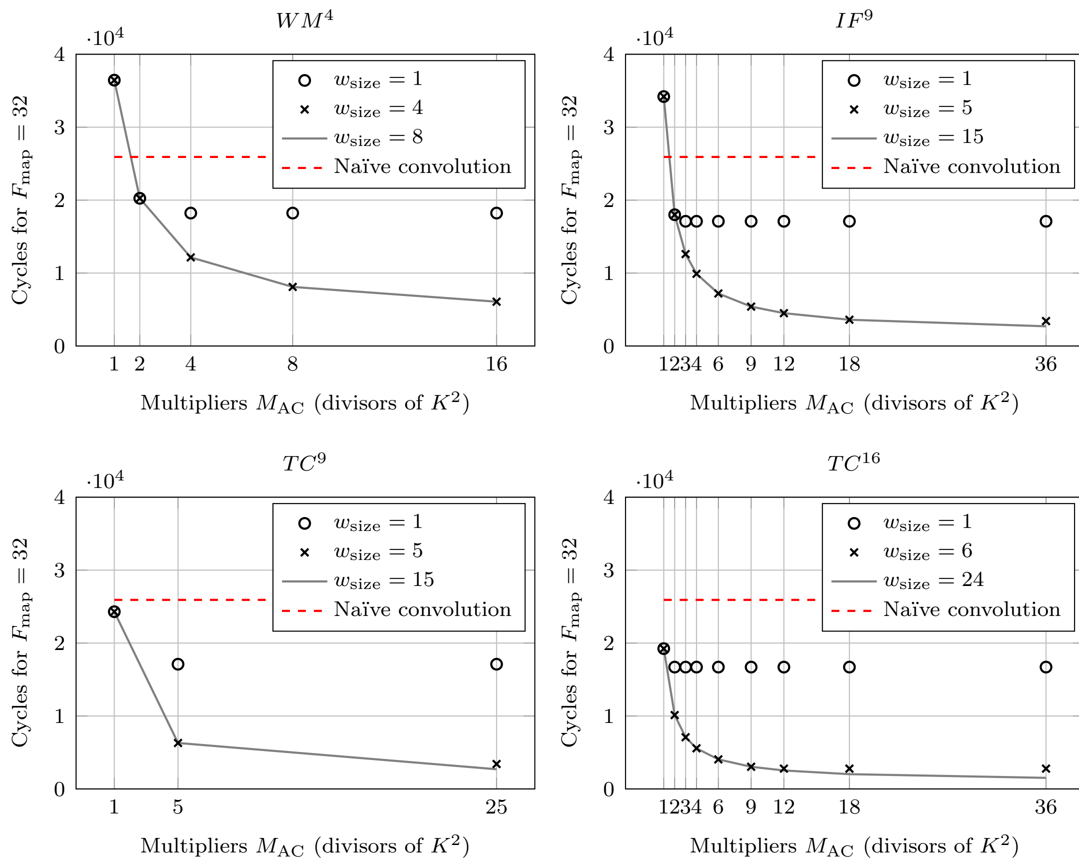
Table 7.6: Word-size settings for the bus-width study, where M is the input tile size and N is the output tile size. Bits corresponds to the bus width, which is the $20 \times w_{size}$, where 20 is the reference word size.

Algorithm	M	N	$w_{size} = M$	Bits	$w_{size} = M \times N$	Bits
WM^4	4	2	4	80	8	160
TC^9	5	3	5	100	15	300
IF^9	5	3	5	100	15	300
TC^{16}	6	4	6	120	24	480
WP^{16}	6	4	6	120	24	480

Figure 7.8 summarizes a system-level word size evaluation. The figure is organized into four subplots (WM^4, IF^9, TC^9, and TC^{16}), each reporting the total cycle count for $F_{\text{map}} = 32$ as a function of the number of multipliers M_{AC}. Within each subplot, different markers represent different effective bus widths w_{size}, while the dashed horizontal line provides the Naïve convolution.

The plots show that when $w_{\text{size}} = 1$, fast convolution algorithms achieve only modest performance improvements, even when the MAC count increases. This behavior is consistent with a regime in which the core frequently waits for input feature-map data, so that part of the theoretical compute advantage is not fully translated into cycle reduction. As w_{size} increases, data delivery becomes less restrictive, utilization improves, and the resulting speedup becomes more pronounced, reaching reductions ranging from 62% to 89% in the simulated convolution core. From an architectural perspective, the figure indicates that fast convolution is most beneficial when paired with a memory subsystem capable of supplying several words simultaneously; otherwise, achievable gains may be limited by bandwidth rather than compute.

The implementation with $w_{size} = M$ is straightforward, as it only requires organizing the memory contents in a column-major order rather than row-major order. Figure 7.9 below illustrates, on the left, the addresses to be read with the convolution window sliding along the X axis, then along the Y axis. Column-wise memory organization implies storing the data in transposed form, so that the convolution window first moves along the Y axis and then along the X axis. This arrangement enables reading an entire column in a single memory cycle. For example, the first read returns the contents of addresses $\{1, 9, 17, 36\}$.



Each subplot reports the total cycle count for $F_{\text{map}} = 32$. Different markers correspond to different bus widths – w_{size}; the red dashed line is the Naïve convolution baseline.

Figure 7.8: Analytic performance (y-axis), varying the MAC number (x-axis), for different bus widths.

7.2.2 Simulated Performance Evaluation

Figure 7.10 aggregates the system-level results by plotting the total cycle count (y-axis) against the number of MACs (x-axis), while Table 7.7 complements the plot by reporting cycle ratios normalized to the Naïve baseline for the same algorithm under different multiplier counts and word sizes. The Naïve baseline is shown as a horizontal reference line.

Two trends are visible in the figure:

- $w_{\text{size}} = 1$: the total cycle count varies only weakly with M_{AC}, suggesting that simply increasing arithmetic resources does not translate proportionally into performance gains.
- $w_{\text{size}} = M$: the speedup increases, indicating lower cycle counts for the same family and comparable multiplier counts.

The data demonstrate that system-level runtime reflects data movement and control effects in addition to arithmetic throughput. Consequently, comparisons between fast-convolution families depend on word size and the resulting datapath utilization, rather than on multiplier count alone.

Standard memory addressing by rows | **Memory contents organized by columns**

Tile 1

1	**2**	**3**	**4**	5	6	7	8
9	**10**	**11**	**12**	13	14	15	16
17	**18**	**19**	**20**	21	22	23	24
25	**26**	**27**	**28**	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

1	**9**	**17**	**25**	33	41	49	57
2	**10**	**18**	**26**	34	42	50	58
3	**11**	**19**	**27**	35	43	51	59
4	**12**	**20**	**28**	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Tile 2

1	2	**3**	**4**	**5**	**6**	7	8
9	10	**11**	**12**	**13**	**14**	15	16
17	18	**19**	**20**	**21**	**22**	23	24
25	26	**27**	**28**	**29**	**30**	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	**11**	**19**	**27**	35	43	51	59
4	**12**	**20**	**28**	36	44	52	60
5	**13**	**21**	**29**	37	45	53	61
6	**14**	**22**	**30**	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Tile 3

1	2	3	4	**5**	**6**	**7**	**8**
9	10	11	12	**13**	**14**	**15**	**16**
17	18	19	20	**21**	**22**	**23**	**24**
25	26	27	28	**29**	**30**	**31**	**32**
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	**13**	**21**	**29**	37	45	53	61
6	**14**	**22**	**30**	38	46	54	62
7	**15**	**23**	**31**	39	47	55	63
8	**16**	**24**	**32**	40	48	56	64

Tile 4

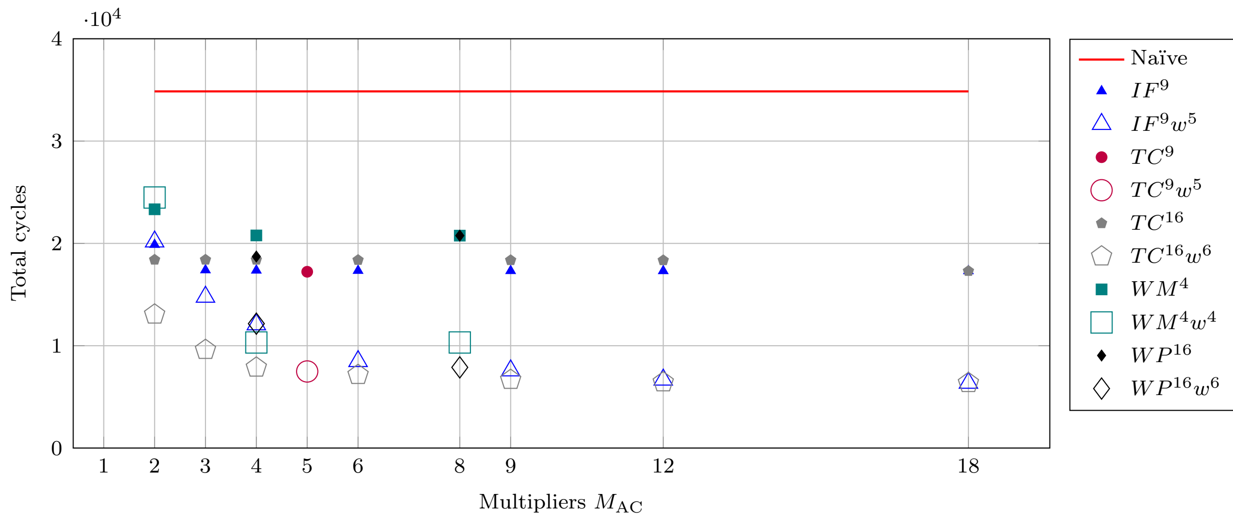
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	**18**	**19**	**20**	21	22	23	24
25	**26**	**27**	**28**	29	30	31	32
33	**34**	**35**	**36**	37	38	39	40
41	**42**	**43**	**44**	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

1	9	**17**	**25**	**33**	**41**	49	57
2	10	**18**	**26**	**34**	**42**	50	58
3	11	**19**	**27**	**35**	**43**	51	59
4	12	**20**	**28**	**36**	**44**	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Figure 7.9: Convolution with row- and column-major order memory organizations.

7.2.3 Area

Figure 7.11 summarizes the system-level cell area across the evaluated configurations and multiplier counts. Across all configurations, system-level implementations of fast schemes are larger than the Naïve baseline, even when the corresponding core datapath (e.g., WM^4) is smaller than the Naïve design in isolation. This pattern indicates that the dominant contributor at the system level is the control logic and registers overhead rather than the datapath itself.



> Total cycle count ($\times 10^4$) is plotted against the number of multipliers M_{AC}. The Naïve baseline is shown as a horizontal line. Each marker corresponds to one fast-convolution family and configuration (e.g., IF^9, TC^9, TC^{16}, WM^4, WP^{16}), and open markers denote the wider-bus cases listed in the legend (e.g., $w_{\mathrm{size}} = 4$, 5, or 6).

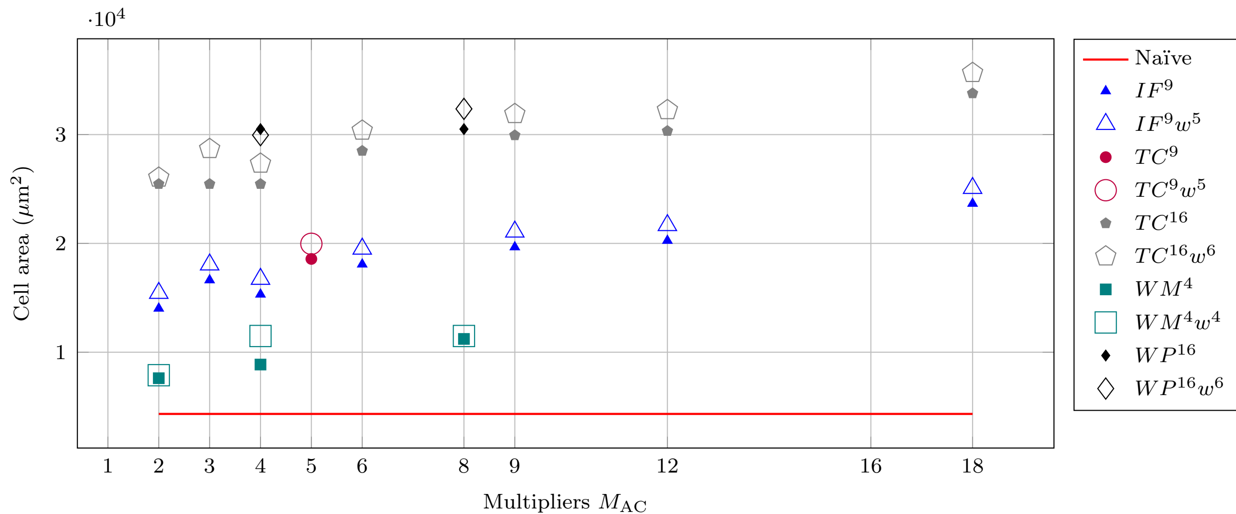
Figure 7.10: Simulated system-level runtime versus multiplier number.

Table 7.7: System-level cycle ratios (normalized to the Naïve baseline) comparing each algorithm across multiplier counts and word sizes.

	$w_{size} = 1$		Selected w_{size}		
Name	MACs	Ratio	MACs	w_{size}	Ratio
WM^4	8	40%	8	4	70%
TC^9	5	51%	5	5	79%
IF^9	6	50%	6	5	76%
IF^9	18	50%	18	5	82%
TC^{16}	6	47%	6	6	79%
TC^{16}	18	50%	18	6	82%
WP^{16}	8	40%	8	6	77%
WP^{16}	32	47%	32	6	81%

This dissertation adopted a conservative, generic control organization to enable broad comparisons across algorithms. A direct consequence is that the control layer is not tailored to any particular fast scheme and therefore tends to penalize families that require larger transform-domain states and more structured data movement. In practice, this effect is more pronounced for IF and, even more so, for WP, which already requires a large internal register file ($K^2 = 64$). For this reason, the reported system-level area, and the corresponding energy trends, for IF and WP are best interpreted as conservative lower bounds on their potential competitiveness under a controller optimized for their specific access patterns.

Finally, the open-marker points indicate that increasing w_{size} has only a modest impact on area relative to $w_{\mathrm{size}} = 1$. This observation suggests that bandwidth upgrades are comparatively inexpensive in terms of area. It supports the assumption that improving w_{size} is



> Total cell area (μm^2) is plotted against the number of multipliers M_{AC}. The Naïve baseline is shown as a horizontal line. Each marker corresponds to one fast-convolution family and configuration (e.g., IF^9, TC^9, TC^{16}, WM^4, WP^{16}), and open markers denote the wider-bus cases listed in the legend (e.g., $w_{size} = 4$, 5, or 6).

Figure 7.11: System-level cell area versus multiplier count.

often preferable to aggressively scaling M_{AC} once control and data movement dominate system costs.

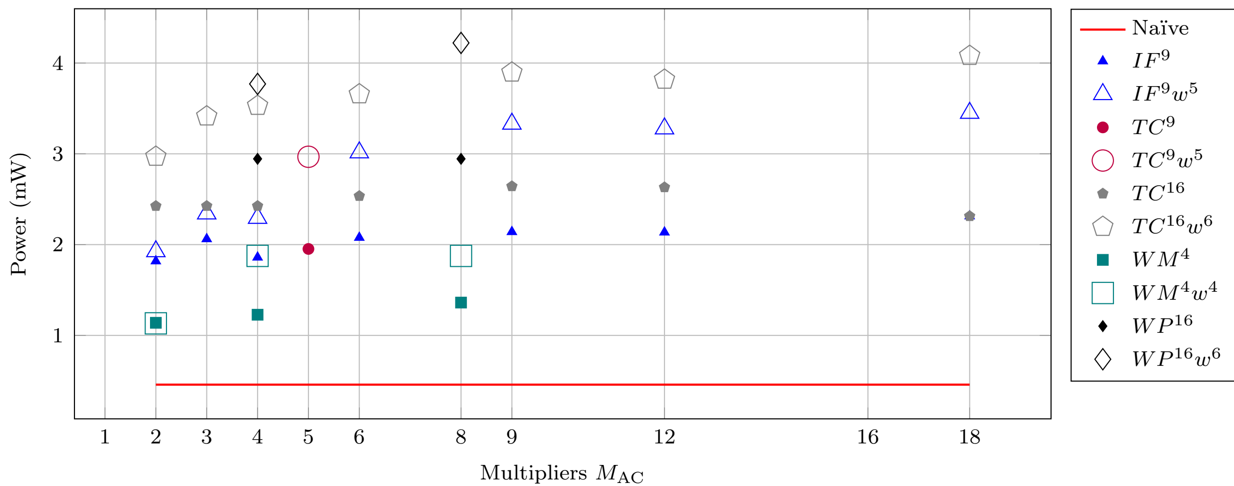
7.2.4 Power

As shown in Figure 7.12, average power increases with M_{AC} for all fast-convolution configurations. The Naïve baseline remains lower because it uses fewer resources in this system configuration. In this evaluation flow, power is driven by cell area and switching activity; consequently, differences across points reflect both datapath size and controller activity. In addition, increasing w_{size} tends to raise power more noticeably than it increases area, indicating that bandwidth scaling carries a stronger power penalty than an area penalty. Therefore, system-level energy comparisons must account for whether reductions in cycle count compensate for increased power.

7.2.5 Energy

As shown in Figure 7.13, increasing the multiplier count reduces total energy consumption, but this effect becomes more visible when w_{size} is larger, as higher bandwidth improves utilization and amplifies the energy gains from additional parallelism.

As shown in Figure 7.14, fast-convolution configurations reduce cycle count relative to the Naïve baseline, but energy varies across operating points. Points with larger M_{AC} tend to shift left (fewer cycles) and upward (higher power), while wider-bus configurations



> Average power (mW) is plotted against the number of multipliers M_{AC}. The Naïve baseline is shown as a horizontal line. Each marker corresponds to one fast-convolution family and configuration (e.g., IF^9, TC^9, TC^{16}, WM^4, WP^{16}), and open markers denote the wider-bus cases listed in the legend (e.g., $w_{\text{size}} = 4$, 5, or 6).

Figure 7.12: System-level power versus multiplier count.

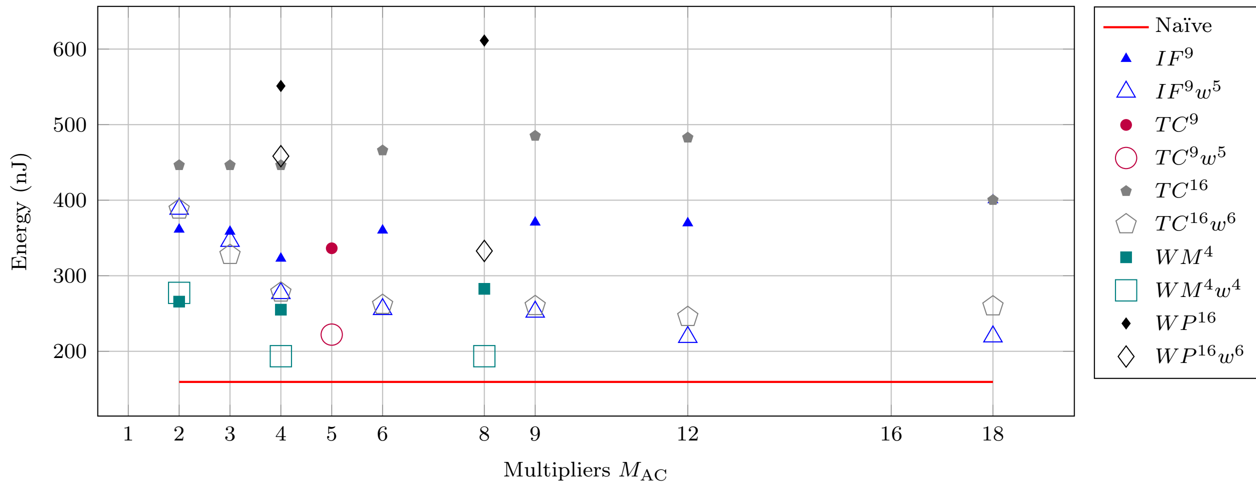
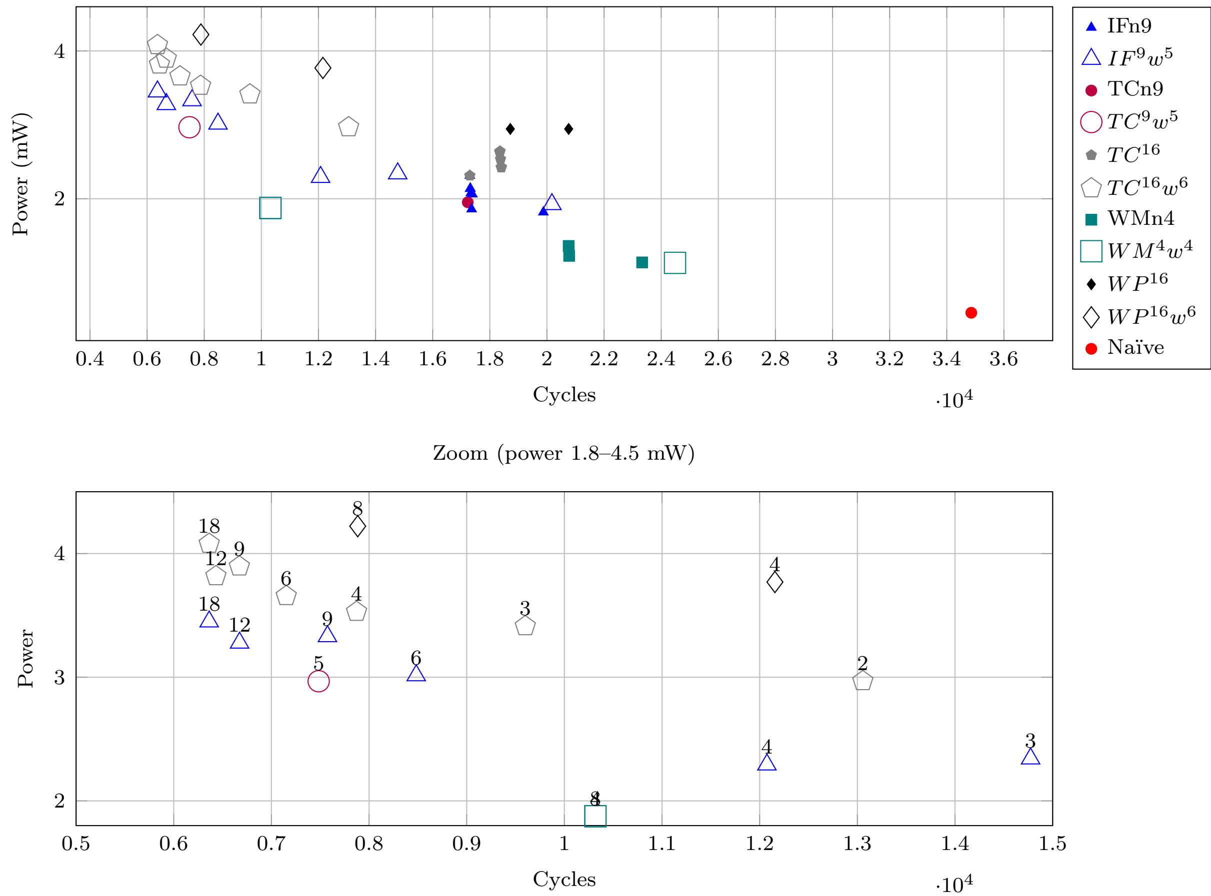


Figure 7.13: System-Level Energy per operation in nJ as a function of the multiplier count.

shift left by reducing idle time in the datapath. The zoomed panels focus exclusively on fast-convolution configurations with selected w_{size}. Therefore, system-level energy results must be interpreted together with bus width and multiplier utilization. Increasing w_{size} reduces idle cycles and shortens execution time, which can lower total energy even if instantaneous power rises modestly. By contrast, when w_{size} is small, the datapath is frequently starved, so the system runs longer at a relatively high power level, increasing the energy cost. The practical implication is that bandwidth and utilization must be considered alongside multiplier count: improving data delivery is often a more effective path to energy efficiency than scaling arithmetic resources in isolation.

Table 7.8 summarizes the same system-level trends in numerical form, comparing fast-convolution families under different w_{size} for performance, area, power, and energy. The table reinforces the conclusions drawn from the plots. It explicitly shows that IF^9 with 18 MACs



Power (mW) is plotted against total cycle count ($\times 10^4$). Each point corresponds to one configuration ($M_{\text{AC}}, w_{\text{size}}$) of IF[9], TC[9], or WM[4]. The Naïve baseline is included for reference. Numeric labels next to points indicate the multiplier count M_{AC}. Each marker corresponds to one fast-convolution family and configuration (e.g., IF^9, TC^9, TC^{16}, WM^4, WP^{16}), and open markers denote the wider-bus cases listed in the legend (e.g., $w_{\text{size}} = 4$, 5, or 6). The zoomed panels isolate the region of interest, containing only architectures with $w_{\text{size}} > 1$.

Figure 7.14: System-level power versus runtime.

and $w_{\text{size}} = 5$ achieves the best performance among the evaluated configurations while also delivering the second-lowest energy consumption.

Table 7.8: System-level normalized ratios for performance (simulation), cell area, power, and energy, with emphasis on comparing algorithms across different w_{size} configurations. All values are reported relative to the Naïve baseline under the same workload and clock constraint.

	$w_{size} = 1$					Selected w_{size}					
Name	MACs	Perf.	Area	Power	Energy	MACs	w_{size}	Perf.	Area	Power	Energy
WMn^4	8	40%	-159%	-198%	-77%	8	4	70%	-166%	-310%	-21%
TCn^9	5	51%	-329%	-327%	-111%	5	5	79%	-361%	-549%	-39%
IFn^9	6	50%	-318%	-354%	-126%	6	5	76%	-351%	-559%	-60%
IFn^9	18	50%	-447%	-406%	-151%	18	5	82%	-480%	-654%	-38%
TCn^{16}	6	47%	-559%	-454%	-192%	6	6	79%	-602%	-699%	-64%
TCn^{16}	18	50%	-681%	-406%	-151%	18	6	82%	-724%	-792%	-63%
WPn^{16}	8	40%	-605%	-544%	-283%	8	6	77%	-648%	-823%	-109%
WPn^{16}	32	47%	-936%	-739%	-347%	32	6	81%	-979%	-1056%	-116%

7.3 Final remarks

This chapter presented a systematic PPA evaluation of the naïve baseline and the proposed fast-convolution accelerators, combining analytical modeling with RTL-based measurements at both convolution-core and system levels.

The chapter shows, via the analytical model and RTL validation, that performance scaling with the number of M_{AC} reaches saturation: beyond a point, adding MACs yields only small cycle reductions. This is captured by the *Balanced* operating point, where further parallelism becomes marginal, and confirmed by RTL results that follow the same trend despite higher absolute cycles due to idle time and control/data-movement overhead. Hence, M_{AC} should be increased only until the architecture can keep the datapath fed.

At the convolution-core level, all fast-convolution configurations reduce energy relative to the naïve baseline, since fewer cycles compensate the power increase. At the system level, total time is set by the slowest stage (bus, memory, or control), so under a narrow bus the cycle count changes little with M_{AC}, indicating a bus/memory-bound regime. In that case, increasing the effective bus width w_{size} is usually more effective than scaling M_{AC}, because it improves data delivery and reduces idle cycles. Therefore, fast convolution meets its potential only when bandwidth and control are not constraining.

Minimizing multiplications alone does not predict cost: as K^2 grows, transform-domain state and structured data movement increase, and registers, routing, and control can dominate area, power, and energy. For ASIC design, each algorithm must be evaluated based on the combined costs of datapath, state, and control; multiplication count alone is not a reliable proxy.

By deliberately adopting a generic and conservative controller to enable broad comparisons, the chapter introduces a systematic bias against families that require a larger transform-domain state and more structured data movement, particularly IF and, most notably, WP. Therefore, system-level results for IF/WP should be interpreted as conservative estimates. This observation motivates future work on algorithm-aware control, transfer–compute overlap, and buffering optimization to better approach core-level potential at the system level.

Within the operational ranking, IF^9 represents the most practical "middle ground" when the objective is neither strict minimum cost (where WM^4 dominates) nor maximum throughput (where TC^{16} leads), but rather a constrained design that balances performance, energy, and implementability across operating points. IF^9 offers a large set of multiplier counts, which enables fine-grained matching to area, clock, and bandwidth constraints, in contrast to the coarser design options of TC^9. At the core level, IF^9 provides competitive trade-offs alongside WM^4. It can also remain smaller than the naïve baseline for up to 6 MACs while still delivering acceleration, making it the only alternative beyond WM^4 that avoids an immediate area penalty.

At the system level, IF^9 is particularly advantageous under bandwidth constraints. In bus-limited regimes, it can achieve higher performance and lower energy consumption than TC^{16}. These results indicate that IF^9 should be understood as a constraint-driven design choice rather than a universally optimal solution.

8. CONCLUSIONS AND FUTURE WORK

This dissertation investigated the performance of fast convolution algorithms implemented as hardware accelerators using a unified methodology. The central question is not whether a scheme reduces multiplications in isolation, but whether the complete system, including transforms, transform domain state, data movement, and control, can achieve performance and energy gains under realistic architectural constraints. To address this question, the study employed a common architectural template and a generic control organization, which enabled a systematic comparison across algorithm families and configurations. The analysis provided both a ranked assessment of fast convolution instances under the evaluated assumptions and a characterization of the constraints that determine when and why fast convolution yields system-level gains.

The first conclusion is that **arithmetic optimality alone does not predict accelerator efficiency**. Although fast convolution reduces the number of multiplications, it introduces structured transforms and additional intermediate values that must be stored, controlled, and transferred. These effects are not secondary. They determine the critical path, register utilization, and the bandwidth required between stages. Therefore, in hardware design, the relevant metric is the combined cost of *(i) transform cost*, *(ii) registered data*, and *(iii) control overhead*, rather than the multiplication count alone. This dissertation quantified the combined cost by evaluating multiple algorithm families within a shared framework and distinguishing core-level effects from system-level effects.

At the **core level**, fast convolution configurations can reduce execution time and energy, but they do so in different ways. Some configurations mainly reduce cycles (often with higher average power), while others reduce power at similar runtimes. This is visible when power is plotted against time: energy-competitive architectures cluster in the lower-left region, while the naïve reference stays far to the right due to its much longer execution time. A key pattern is that **lower energy is not only about lower power**. In several cases, a design that increases power still reduces energy if the cycle reduction is large enough. This reinforces that energy is a joint outcome of performance and power, not a single metric in isolation.

A second **core level** observation is that **increasing computational resources eventually reaches saturation**. Increasing the number of multipliers or the internal bandwidth improves throughput only until the computation becomes limited by the convolution core states. Beyond that point, additional compute resources remain underutilized, and performance flattens. In practice, designers must codesign the architecture with the selected algorithm. A mismatch between computational parallelism and data movement increases area and power consumption while providing little performance benefit. This dissertation employs analytical models and empirical evaluation to identify these saturation regimes, which are critical for selecting an appropriate operating point rather than implementing an over-provisioned datapath.

At the **system level the cost reflects architecture and control, not only the datapath**. When the core is embedded in a complete system with control, buffering, and structured data movement, all fast-convolution configurations become larger than the naïve baseline, even when the corresponding core is smaller. Fast schemes require more registers for intermediate results and more complex addressing and control logic. Under a generic controller, these requirements translate into a larger register area and extra control logic. As a result, a fast core that appears attractive in isolation may lose some of its advantage when system overhead is included.

This system-level effect is particularly evident for families that require large registers to store intermediate data. In this dissertation, the control organization was deliberately designed to be generic in order to enable fair comparisons. This decision has a direct consequence: **the control module is not optimized for any specific scheme**. As a result, schemes that require a larger structured state and stricter ordering incur a penalty. In practical terms, this primarily affects IF and WP, not because their arithmetic performance is inferior, but because their implementations depend more heavily on the control module. Therefore, the system-level area and energy results reported for IF and WP should be interpreted as **lower bounds on their potential competitiveness** under a controller tailored to their dataflow. This reframes the research question from "which scheme is best " to "which scheme remains superior under a given control and memory organization, and to what extent can specialization recover lost efficiency."

A further conclusion concerns the distinction between "best" and "balanced" configurations. Across the explored design space, selecting the minimum-energy point within a family often required additional resources, increasing area and sometimes power consumption, but reducing total energy by shortening runtime. In contrast, balanced configurations kept area and power closer to the baseline, at the cost of higher energy than the minimum achievable configuration. Thus, **fast convolution does not correspond to a single operating point**. Each family defines a continuum of trade-offs, and the appropriate choice depends on the primary constraint: area, power, or energy.

The dissertation also clarified a practical limitation for generalization. The evaluation intentionally did not use quantization above 20 bits. This matters because some families, notably Toom-Cook variants, can require multiplications by large constants as tile size increases, which increases the internal word length. In contrast, families whose transforms are mainly additions (such as IF and WP in the evaluated forms) do not face the same issue. Since quantization was not a focus, the dissertation does not claim superiority under low-precision regimes. Instead, it isolates architectural and control-driven effects under a consistent precision assumption, leaving quantization-aware conclusions to future work.

Putting these results together, the main practical implication is the following. **Fast convolution can improve performance and energy, but only when the architecture matches the algorithm dataflow and state requirements**. The configurations

that performed well in this dissertation are those that (i) reduce cycles meaningfully, (ii) keep transform-domain state manageable, and (iii) fit the generic control organization without excessive overhead. Conversely, when a scheme expands state (e.g., large K^2) or demands tightly structured movement of intermediates, a generic controller can increase costs and penalize expected gains. This explains why a complete-system evaluation is necessary: it shows whether the algorithmic advantage holds up under the required buffering, timing, and control.

This dissertation provides a systematic perspective that extends beyond the specific designs evaluated. First, it demonstrates that core-only PPA trends can be misleading and require system-level evaluation. Second, it defines energy as a joint function of runtime and power, which prevents conclusions based on either metric in isolation. Third, it identifies control and data movement as central design axes for fast convolution and suggests that further gains may arise from **architecture-aware algorithms**.

Inspection Factorization, the main contribution of this dissertation, was evaluated. It supports multiple configurations by varying the number of MACs. In the core evaluation, IF^9 provides comparable performance–energy trade-offs and behaves as a practical middle ground between WM^4 and TC^{16} across operating points. In the system evaluation, IF^9 with 18 MACs and $w_{\text{size}} = 5$ achieves the best performance among the evaluated configurations while also delivering the second-lowest energy consumption.

To improve reproducibility and reuse, the implementation artifacts of this dissertation are publicly available in three repositories. The repository https://github.com/tarsioonofrio/fast-convolution-rtl provides the algorithm-design and code-generation toolbox used to implement fast convolution methods from Chapters 2 and 4 and to emit C and SystemVerilog artifacts consumed in Chapters 5 and 6. The repository https://github.com/tarsioonofrio/FastConv_SystemVerilog contains the synthesizable hardware implementation and automation flow (simulation, synthesis, and report consolidation), and also the stored result files used in Chapter 7. The repository https://github.com/tarsioonofrio/fast-conv-RS5 contains the RS5 software implementation and benchmarking flow used in Chapter 5, including optimized and non-optimized execution variants, consolidated report generation, and the stored result files used in Chapter 5.

The following directions for future work build on these findings, with emphasis on control specialization, memory-system integration, and additional dimensions (such as quantization) that can further reshape the trade-offs observed in this study.

- **Broader empirical evaluation on CNN workloads.** We will extend the evaluation beyond synthetic kernels by running benchmarks across representative CNN models and layer types (e.g., ResNet/MobileNet/EfficientNet-style blocks), sweeping $(C_{\text{in}}, C_{\text{out}})$, feature-map sizes, and batch sizes. This includes end-to-end inference measurements and layer-wise breakdowns to identify when each family is compute-bound versus system-bound, and to validate whether the per-kernel PPA trends observed here persist under realistic networks and dataflows.

- **FPGA prototype for real-world measurements.** We plan to prototype the accelerator on FPGA to measure throughput, latency, power, and energy under realistic I/O and clocking. The prototype will be used to (i) validate the cycle model against hardware execution, (ii) quantify the impact of platform memory and interconnect limits, and (iii) collect power/energy measurements with board-level instrumentation (when available) to complement the RTL-based estimates.

- **Quantized inference and numerical robustness of transforms.** We will evaluate quantized models to measure how transform and inverse stages affect inference metrics under fixed-point constraints. This is relevant because WM, IF, and WP avoid constant multiplications in the transforms, while TC may introduce larger constant factors. We will study bit growth, scaling/rounding strategies, saturation policies, and per-layer calibration to determine how numerical effects interact with each family's PPA and whether quantization shifts the ranking.

- **Algorithm-aware control to reduce system-level bias against IF and WP.** The current controller was intentionally generic to enable broad comparisons, but it penalizes families with larger transform-domain state and structured movement (notably IF and WP). As future work, we will redesign control to be algorithm-aware, focusing on reducing idle cycles through tighter control, improved burst formation, transfer–compute overlap, and registering strategies to each family's access pattern. The goal is to expose the potential of IF/WP at system level rather than the conservative point captured here.

- **Reducing internal storage from $K^2 \to K$ and $M^2 \to M$.** We will investigate architectural alternatives that reduce transform-domain storage by avoiding full materialization of K^2 (and analogously M^2) intermediate states. Candidate directions include streaming/in-place transforms to avoid storing full transform-domain blocks, partial evaluation with on-the-fly recombination to consume intermediate results as soon as they are produced, logic reuse to trade small recomputation for large register banks when recomputing cheap intermediates is preferable to keeping large buffers, and reducing the size of large register arrays by optimizing the dataflow and eliminating unnecessary transform-domain state. The objective is to lower area and power, improving scalability for larger kernels and making higher-K^2 families more competitive.

REFERENCES

Agarwal, R. and Cooley, J. (1977). New algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, page 392–410. https://ieeexplore.ieee.org/document/1162981.

Ahmad, A. and Pasha, M. A. (2019). Towards Design Space Exploration and Optimization of Fast Algorithms for Convolutional Neural Networks (CNNs) on FPGAs. In *ACM/IEEE Design, Automation Test in Europe Conference (DATE)*, pages 1106–1111. https://doi.org/10.23919/DATE.2019.8715272.

Ahmad, A. and Pasha, M. A. (2020). Ffconv: An fpga-based accelerator for fast convolution layers in convolutional neural networks. *ACM Trans. Embed. Comput. Syst.*, page 15:1–15:24. https://doi.org/10.1145/3380548.

Alam, S. A., Anderson, A., Barabasz, B., and Gregg, D. (2022). Winograd convolution for deep neural networks: Efficient point selection. http://arxiv.org/abs/2201.10369.

Aydonat, U., O'Connell, S., Capalija, D., Ling, A., and Chiu, J. (2017). An OpenCL™ Deep Learning Accelerator on Arria 10. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 55–64. https://doi.org/10.1145/3020078.3021738.

Bao, C., Xie, T., Feng, W., Chang, L., and Yu, C. (2020). A Power-Efficient Optimizing Framework FPGA Accelerator Based on Winograd for YOLO. *IEEE Access*, pages 94307–94317. https://doi.org/10.1109/ACCESS.2020.2995330.

Blahut, R. E. (2010). *Fast Algorithms for Signal Processing.* Cambridge University Press. https://www.cambridge.org/core/books/fast-algorithms-for-signal-processing/E35F46175F3558980873443CA8A6A4FD.

Breitzman, A. F. and Johnson, J. R. (2004). Automatic derivation and implementation of fast convolution algorithms. *Journal of Symbolic Computation*, page 261–293. https://www.sciencedirect.com/science/article/pii/S0747717103001093.

Burrus, C. and Selesnick, I. (1995). On Programs for Prime Length FFTs and Circular Convolution. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1137–1140. https://doi.org/10.1109/ICASSP.1995.480436.

Cariow, A. and Paplinski, J. P. (2020). Some algorithms for computing short-length linear convolution. *Electronics*, page 2115. https://www.mdpi.com/2079-9292/9/12/2115.

Cariow, A. and Paplinski, J. P. (2021). Algorithmic structures for realizing short-length circular convolutions with reduced complexity. *Electronics*, page 2800. https://www.mdpi.com/2079-9292/10/22/2800.

Cariow, A., Papliński, J. P., and Makowska, M. (2023). Vlsi-friendly filtering algorithms for deep neural networks. *Applied Sciences*, page 9004. https://www.mdpi.com/2076-3417/13/15/9004.

Chang, J.-W., Ahn, S., Kang, K.-W., and Kang, S.-J. (2020). Towards Design Methodology of Efficient Fast Algorithms for Accelerating Generative Adversarial Networks on FPGAs. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 283–288. https://doi.org/10.1109/ASP-DAC47756.2020.9045214.

Chen, Y., Xie, Y., Song, L., Chen, F., and Tang, T. (2020). A survey of accelerator architectures for deep neural networks. *Engineering*, page 264–274. https://www.sciencedirect.com/science/article/pii/S2095809919306356.

Cheng, C. and Parhi, K. K. (2020). Fast 2d convolution algorithms for convolutional neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, page 1678–1691. https://ieeexplore.ieee.org/document/8966288.

Cook, S. A. and Aanderaa, S. O. (1969). On the minimum computation time of functions. *Transactions of the American Mathematical Society*, pages 291–314. https://doi.org/10.1090/S0002-9947-1969-0249212-8.

Dally, W. J., Turakhia, Y., and Han, S. (2020). Domain-specific hardware accelerators. *Communications of the ACM*, pages 48–57. https://doi.org/10.1145/3361682.

Deng, H., Wang, J., Ye, H., Xiao, S., Meng, X., and Yu, Z. (2021). 3D-VNPU: A Flexible Accelerator for 2D/3D CNNs on FPGA. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–185. https://doi.org/10.1109/FCCM51124.2021.00029.

Di, X., Yang, H.-G., Jia, Y., Huang, Z., and Mao, N. (2020). Exploring efficient acceleration architecture for winograd-transformed transposed convolution of gans on fpgas. *Electronics*, page 286. https://www.mdpi.com/2079-9292/9/2/286.

DiCecco, R., Lacey, G., Vasiljevic, J., Chow, P., Taylor, G., and Areibi, S. (2016). Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks. In *International Conference on Field-Programmable Technology (FPT)*, pages 265–268. https://doi.org/10.1109/FPT.2016.7929549.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Huang, Y., Shen, J., Wang, Z., Wen, M., and Zhang, C. (2018). A High-efficiency FPGA-based Accelerator for Convolutional Neural Networks using Winograd Algorithm. *Journal of Physics: Conference Series*, pages 1–12. https://dx.doi.org/10.1088/1742-6596/1026/1/012019.

Jiang, J., Chen, X., and Tsui, C.-Y. (2023). Accelerating Large Kernel Convolutions with Nested Winograd Transformation. In *IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. https://doi.org/10.1109/VLSI-SoC57769.2023.10321932.

Ju, C. and Solomonik, E. (2019). Derivation and Analysis of Fast Bilinear Algorithms for Convolution. *CoRR*, pages 1–34. http://arxiv.org/abs/1910.13367.

Juracy, L. R. (2022). A framework for fast architecture exploration of convolutional neural network accelerators. https://tede2.pucrs.br/tede2/bitstream/tede/10437/2/LEONARDO_REZENDE_JURACY_TES.pdf.

Juracy, L. R., Amory, A. M., and Moraes, F. (2022). A fast, accurate, and comprehensive ppa estimation of convolutional hardware accelerators. *IEEE Transactions on Circuits and Systems I-regular Papers*, page 1–14.

Juracy, L. R., Garibotti, R., and Moraes, F. G. (2023). From cnn to dnn hardware accelerators: A survey on design, exploration, simulation, and frameworks. *Foundations and Trends textsuperscript textregistered in Electronic Design Automation*, page 270–344. https://www.nowpublishers.com/article/Details/EDA-060.

Juracy, L. R., Moreira, M. T., Moraes Amory, A., and Moraes, F. (2021a). A tensorflow and system simulator integration approach to estimate hardware metrics of convolution accelerators. *Latin American Symposium on Circuits and Systems*, page 1–4.

Juracy, L. R., Moreira, M. T., Morais Amory, A., Hampel, A. F., and Moraes, F. G. (2021b). A high-level modeling framework for estimating hardware metrics of cnn accelerators. *IEEE Transactions on Circuits and Systems I: Regular Papers*, page 4783–4795. https://doi.org/10.1109/TCSI.2021.3101176.

Kala, S., Jose, B. R., Mathew, J., and Nalesh, S. (2019a). High-performance cnn accelerator on fpga using unified winograd-gemm architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, page 2816–2828. https://ieeexplore.ieee.org/document/8856248/.

Kala, S., Mathew, J., Jose, B. R., and Nalesh, S. (2019b). UniWiG: Unified Winograd-GEMM Architecture for Accelerating CNN on FPGAs. In *International Conference on VLSI Design (VLSID)*, pages 209–214. https://doi.org/10.1109/VLSID.2019.00055.

Kala, S. and Nalesh, S. (2020). Efficient cnn accelerator on fpga. *IETE Journal of Research*, page 733–740. https://doi.org/10.1080/03772063.2020.1821797.

Kala, S., Paul, D., Jose, B., and Nalesh, S. (2018). Design Space Exploration of Convolution Algorithms to Accelerate CNNs on FPGA. In *International Symposium on Embedded Computing and System Design (ISED)*, pages 21–25. https://doi.org/10.1109/ISED.2018.8704043.

Karatsuba, A. and Ofman, Y. (1963). Multiplication of Multidigit Numbers on Automata. *Soviet physics. Doklady*, pages 595–596. https://www.researchgate.net/publication/234346907_Multiplication_of_Multidigit_Numbers_on_Automata.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). ImageNet classification with Deep Convolutional neural networks. *Communications of the ACM*, pages 84–90. https://doi.org/10.1145/3065386.

Lavin, A. and Gray, S. (2016). Fast Algorithms for Convolutional Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021. https://doi.org/10.1109/CVPR.2016.435.

Li, J., Liang, Y., Yang, Z., and Li, X. (2025). An efficient convolutional neural network accelerator design on FPGA using the layer-to-layer unified input winograd architecture. *Electronics*. Publisher: Multidisciplinary Digital Publishing Institute (MDPI).

Li, M., Li, P., Yin, S., Chen, S., Li, B., Tong, C., Yang, J., Chen, T., and Yu, B. (2024). WinoGen: A highly configurable winograd convolution IP generator for efficient CNN acceleration on FPGA. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6. https://dl.acm.org/doi/10.1145/3649329.3657392.

Liang, Y., Lu, L., Xiao, Q., and Yan, S. (2020). Evaluating fast algorithms for convolutional neural networks on fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, page 857–870. https://doi.org/10.1109/TCAD.2019.2897701.

Liu, X., Chen, Y., Hao, C., Dhar, A., and Chen, D. (2021). WinoCNN: Kernel Sharing Winograd Systolic Array for Efficient Convolutional Neural Network Acceleration on FPGAs. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 258–265. https://doi.org/10.1109/ASAP52443.2021.00045.

Lu, L. and Liang, Y. (2018). SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. https://doi.org/10.1145/3195970.3196120.

Lu, L., Liang, Y., Xiao, Q., and Yan, S. (2017). Evaluating fast algorithms for convolutional neural networks on fpgas. In *Proc. - IEEE Annu. Int. Symp. Field-Program. Cust. Comput. Mach., FCCM*, page 101–108. https://doi.org/10.1109/FCCM.2017.64.

Lv, P., Liu, W., and Li, J. (2020). A fpga-based accelerator implementaion for yolov2 object detection using winograd algorithm. In *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, page 1894–1898. https://ieeexplore.ieee.org/document/9421655/.

McClellan, J. H. and Rader, C. M. (1979). *Number Theory in Digital Signal Processing*. Prentice Hall.

Moolchandani, D., Kumar, A., and Sarangi, S. R. (2021). Accelerating CNN Inference on ASICs: A Survey. *Journal of Systems Architecture*, pages 1–26. https://doi.org/10.1016/j.sysarc.2020.101887.

Myers, D. G. (1990). *Digital Signal Processing: Efficient Convolution and Fourier Tranform Techniques*. Prentice Hall.

Nugroho, D. H. T., Adiono, T., Syafalni, I., Sutisna, N., Mulyawan, R., and Ahmadi, N. (2021). Hardware resource reduction using winograd approach on yolov3-tiny. In *2021 15th International Conference on Telecommunication Systems, Services, and Applications (TSSA)*, page 1–4. https://ieeexplore.ieee.org/document/9768243/.

Nunes, W. A., Dal Zotto, A. E., Borges, C. d. S., and Moraes, F. G. (2024). RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems. In *IEEE Latin America Symposium on Circuits and Systems (LASCAS)*, pages 1–5. https://doi.org/10.1109/lascas60203.2024.10506171.

Nussbaumer, H. (1978). New algorithms for convolution and DFT based on polynomial transforms. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 638–641. https://doi.org/10.1109/ICASSP.1978.1170580.

Nussbaumer, H. J. (1982). *Fast Fourier Transform and Convolution Algorithms*. Springer. https://link.springer.com/book/10.1007/978-3-662-00551-4.

Parhi, K. K. (1999). *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley.

Pitas, I. (2021). Optimal Multidimensional Cyclic Convolution Algorithms For Deep Learning And Computer Vision Applications. In *International Conference on Autonomous Systems (ICAS)*, pages 1–5. https://doi.org/10.1109/ICAS49788.2021.9551198.

Podili, A., Zhang, C., and Prasanna, V. (2017). Fast and efficient implementation of Convolutional Neural Networks on FPGA. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 11–18. https://doi.org/10.1109/ASAP.2017.7995253.

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., and Kepner, J. (2020). Survey of machine learning accelerators. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, page 1–12. https://doi.org/10.1109/HPEC43674.2020.9286149.

Selesnick, I. and Burrus, C. (1993). Multidimensional Mapping Techniques for Convolution. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 288–291. https://doi.org/10.1109/ICASSP.1993.319493.

Selesnick, I. and Burrus, C. (1994). Extending Winograd's small convolution algorithm to longer lengths. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 449–452. https://doi.org/10.1109/ISCAS.1994.408999.

Selesnick, I. and Burrus, C. (1996). Automatic generation of prime length fft programs. *IEEE Transactions on Signal Processing*, 44(1):14–24. https://doi.org/10.1109/78.482008.

Shawahna, A., Sait, S. M., and El-Maleh, A. (2019). FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. *IEEE Access*, pages 7823–7859. https://doi.org/10.1109/ACCESS.2018.2890150.

Shen, J., Huang, Y., Wang, Z., Qiao, Y., Wen, M., and Zhang, C. (2018). Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 97–106. https://doi.org/10.1145/3174243.3174257.

Shen, J., Huang, Y., Wen, M., and Zhang, C. (2020). Toward an efficient deep pipelined template-based architecture for accelerating the entire 2-d and 3-d cnns on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, page 1442–1455. https://doi.org/10.1109/TCAD.2019.2912894.

Silva, T. O. C. d. (2023). Ambiente para exploração de CNNS em nível RTL. https://primo-pmtna01.hosted.exlibrisgroup.com/permalink/f/1paomtm/ds_tccs10923/26401.

Silvano, C., Ielmini, D., Ferrandi, F., Fiorin, L., Curzel, S., Benini, L., Conti, F., Garofalo, A., Zambelli, C., Calore, E., Schifano, S., Palesi, M., Ascia, G., Patti, D., Petra, N., De Caro, D., Lavagno, L., Urso, T., Cardellini, V., Cardarilli, G. C., Birke, R., and Perri, S. (2025). A survey on deep learning hardware accelerators for heterogeneous hpc platforms. *ACM Comput. Surv.*, page 286:1–286:39. https://dl.acm.org/doi/10.1145/3729215.

Tolimieri, R., An, M., and Lu, C. (1997). *Mathematics of Multidimensional Fourier Transform Algorithms*. Springer. http://link.springer.com/10.1007/978-1-4612-1948-4.

Tolimieri, R., An, M., and Lu, C. (2013). *Algorithms for Discrete Fourier Transform and Convolution*. Springer.

Toom, A. L. (1963). The complexity of a scheme of functional elements simulating the multiplication of integers. In *Doklady Akademii Nauk*, page 496–498. https://www.mathnet.ru/eng/dan27978.

Wang, H., Lin, J., Xie, Y., Yuan, B., and Wang, Z. (2018). Efficient Reconfigurable Hardware Core for Convolutional Neural Networks. In *Asilomar Conference on Signals, Systems, and Computers (ACSSC)*, pages 777–781. https://doi.org/10.1109/ACSSC.2018.8645259.

Wang, X., Wang, C., Cao, J., Gong, L., and Zhou, X. (2020). Winonn: Optimizing fpga-based convolutional neural network accelerators using sparse winograd algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, page 4290–4302. https://doi.org/10.1109/TCAD.2020.3012323.

Winograd, S. (1976). Some bilinear forms whose multiplicative complexity depends on the field of constants. *Mathematical systems theory*, page 169–180. https://doi.org/10.1007/BF01683270.

Winograd, S. (1978). On computing the discrete fourier transform. *Mathematics of Computation*, page 175–199. https://www.ams.org/mcom/1978-32-141/S0025-5718-1978-0468306-4/.

Winograd, S. (1987). *Arithmetic Complexity of Computations (CBMS-NSF Regional Conference Series in Applied Mathematics)*. Society for Industrial and Applied Mathematics. http://gen.lib.rus.ec/book/index.php?md5=48021843a961a60f54b87d2723901e70.

Xiao, Q., Liang, Y., Lu, L., Yan, S., and Tai, Y.-W. (2017). Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *Design Automation Conference (DAC)*, pages 1–6. https://doi.org/10.1145/3061639.3062244.

Xie, K., Lu, Y., He, X., Yi, D., Dong, H., and Chen, Y. (2024). Winols: A large-tiling sparse winograd CNN accelerator on FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, pages 1–24. Publisher: Association for Computing Machinery.

Yang, C., Wang, Y., Wang, X., and Geng, L. (2019). Wra: A 2.2-to-6.3 tops highly unified dynamically reconfigurable accelerator using a novel winograd decomposition algorithm for convolutional neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, page 3480–3493. https://doi.org/10.1109/TCSI.2019.2928682.

Yang, T., Liao, Y., Shi, J., Liang, Y., Jing, N., and Jiang, L. (2020). A Winograd-Based CNN Accelerator with a Fine-Grained Regular Sparsity Pattern. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 254–261. https://doi.org/10.1109/FPL50879.2020.00050.

Ye, H., Zhang, X., Huang, Z., Chen, G., and Chen, D. (2020). HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. https://doi.org/10.1109/DAC18072.2020.9218684.

Yepez, J. and Ko, S.-B. (2020). Stride 2 1-d, 2-d, and 3-d winograd for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, page 853–863. https://doi.org/10.1109/TVLSI.2019.2961602.

Yu, J., Hu, Y., Ning, X., Qiu, J., Guo, K., Wang, Y., and Yang, H. (2020). Instruction Driven Cross-layer CNN Accelerator with Winograd Transformation on FPGA. In *International*

Conference on Field Programmable Technology (ICFPT), pages 227–230. https://doi.org/10.1109/FPT.2017.8280147.

Zhai, J., Li, B., Lv, S., and Zhou, Q. (2023). Fpga-based vehicle detection and tracking accelerator. *Sensors*, page 2208. https://www.mdpi.com/1424-8220/23/4/2208.

Zhao, Y., Wang, D., and Wang, L. (2019). Convolution accelerator designs using fast algorithms. *Algorithms*, page 112. https://www.mdpi.com/1999-4893/12/5/112.

Zhuge, C., Liu, X., Zhang, X., Gummadi, S., Xiong, J., and Chen, D. (2018). Face Recognition with Hybrid Efficient Convolution Algorithms on FPGAs. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 123–128. https://doi.org/10.1145/3194554.3194597.

APPENDIX A – FOUNDATIONS OF DISCRETE CONVOLUTION AND FAST MULTIPLICATION

This appendix introduces the mathematical foundations required to develop and understand the Dissertation. We follow a linear progression, beginning with multiplication and fast multiplication, then moving to linear and circular convolution, fast convolution (including Winograd and Toom–Cook methods), filtering, and multidimensional algorithms. The goal is to provide a unified algebraic view of convolution that will later support the algorithm design in Chapter 4, the software implementation and evaluation in Chapter 5, the hardware architectures in Chapter 6, and the experimental results presented in Chapter 7.

A.1 Multiplication

To understand the foundations of fast convolution, it is essential first to introduce the history of fast multiplication methods. These methods have significantly influenced the development of efficient convolution techniques. Therefore, we examine the evolution of multiplication methods, starting with traditional approaches. This temporal perspective leads us to the fast convolution methods. It is important to note that multiplication and convolution are deeply related mathematical operations, and this relationship becomes evident throughout the text.

A.1.1 Naive Multiplication

Until the 1960s, integer multiplication was performed using the traditional school method. This method, while straightforward, required N^2 multiplications and $2N-1$ additions, making it computationally expensive for large values of N. This limitation stimulated the development of more efficient techniques, such as the Karatsuba algorithm, which is explored next.

The traditional method for multiplying two numbers, commonly known as the schoolbook method, involves a straightforward approach widely taught in elementary education. Consider two numbers, x and y, each with two digits represented as (x_1, x_0) and (y_1, y_0), respectively. The process of multiplying these two numbers involves calculating the partial products x_0y_0, x_0y_1, x_1y_0, and x_1y_1, and then summing them with the appropriate carries. This method requires N^2 multiplications and $2N-1$ additions, where N is the number of digits in each number. The steps are symbolically represented below.

$$\begin{array}{cccc} & & x_1 & x_0 \\ \times & & y_1 & y_0 \\ \hline & & x_1y_0 & x_0y_0 \\ + & x_1y_1 & x_0y_1 & \\ \hline & x_1y_1 & x_1y_0 + x_0y_1 & x_0y_0 \end{array}.$$

To illustrate this method with a numerical example, consider multiplying 12 and 23. Here, $x = 12$ with digits $x_1 = 1$ and $x_0 = 2$, and $y = 23$ with digits $y_1 = 2$ and $y_0 = 3$:

$$\begin{array}{cccc} & & 1 & 2 \\ \times & & 2 & 3 \\ \hline & & 3 & 6 \\ + & 2 & 4 & \\ \hline & 2 & 7 & 6 \end{array}.$$

This operation can be broken down into the following partial products:

$$x_0y_0 = 2 \times 3 = 6$$
$$x_1y_0 + x_0y_1 = 1 \times 3 + 2 \times 2 = 7$$
$$x_1y_1 = 1 \times 2 = 2$$

.

Summing these products together gives the final result:

$$276 = x_1y_1 \times 10^2 + (x_1y_0 + x_0y_1) \times 10^1 + x_0y_0 \times 10^0.$$

This method, although simple, forms the foundation for understanding more advanced multiplication techniques, such as the Karatsuba algorithm, which optimizes the number of required operations.

A.1.2 Karastuba Multiplication

Karatsuba and Ofman [1963] demonstrated that it is possible to multiply two numbers using fewer multiplications, specifically 3 multiplications and 4 additions, rather than the traditional 4 multiplications and 3 additions. His approach cleverly reduces the required operations by reusing certain products to calculate the necessary terms. Specifically, Karatsuba showed that the internal products $x_1y_0 + x_0y_1$ can be computed without directly multiplying these terms. Instead, these products are derived by calculating the difference between the sum of the external products, x_0y_0 and x_1y_1, and the product of the differences between the digits of x and y, denoted as $(x_1 - x_0)$ and $(y_1 - y_0)$.

The process is expressed as follows by Winograd [1987]:

$$\begin{aligned} x_0y_0 &= x_0y_0 \\ x_0y_1 + x_1y_0 &= x_0y_0 + x_1y_1 - (x_1 - x_0)(y_1 - y_0) \\ x_1y_1 &= x_1y_1 \end{aligned}$$

.

To illustrate this method with a numerical example, consider multiplying 12 and 23. Using Karatsuba's method, we calculate:

$$\begin{aligned} x_0y_0 &= 2 \times 3 \\ x_0y_1 + x_1y_0 &= 2 \times 3 + 1 \times 2 - (2 - 1)(3 - 2) \\ x_1y_1 &= 1 \times 2 \end{aligned}$$

.

These intermediate steps yield the following products:

$$\begin{aligned} x_0y_0 &= 6 \\ x_0y_1 + x_1y_0 &= 7 \\ x_1y_1 &= 2 \end{aligned}$$

.

Finally, the results are combined by summing the products, with appropriate shifts corresponding to their positional significance in base 10:

$$x_0y_0 \times 10^2 + (x_0y_1 + x_1y_0) \times 10^1 + x_1y_1 \times 10^0 = 276$$

.

This approach reuses the external products x_0y_0 and x_1y_1 to calculate the internal product $x_0y_1 + x_1y_0$, effectively reducing the operation by one multiplication but introducing an additional addition. The result is a more efficient multiplication algorithm that lays the groundwork for further optimization in computational mathematics.

Building on the foundations laid by the Karatsuba algorithm, further advancements were made in multiplication. Toom [1963] proposed an algorithm capable of handling more than two-digit operands. Subsequently, Cook and Aanderaa [1969] refined this approach, leading to the widely recognized Toom-Cook algorithm. This method represents a significant step forward in fast multiplication techniques.

A.2 Convolution

Convolution is a fundamental operation in mathematics and digital signal processing, central to numerous applications such as digital filtering, data correlation, and, in particular, convolutional neural networks (CNNs). Convolution can be viewed as summation, a matrix multiplication and a polynomial product [Blahut, 2010; Breitzman and Johnson, 2004].

A.2.1 Linear Convolution

The direct computation of the convolution for two data sequences, $d = (d_0, \ldots, d_{N-1})$ and $g = (g_0, \ldots, g_{L-1})$ produces:

$$s = (s_0, \ldots, s_{N+L-2}) = (d_0 g_0, \ldots, d_0 g_{N-1}, \ldots, d_{N-1} g_0, \ldots, d_{N-1} g_{L-1})..$$

where N is the length of the data, L is the length of the filter, and $M = L + N - 1$ is the length of the output sequence.

A graphical symbolic convolution of four steps is executed in Figure A.1 where d is the fixed sequence data and g is reversed and moved step by step to the right.

$$\begin{array}{ccccc} s_0 & & & & \\ \hline & & & d_0 & d_1 \\ \times & g_2 & g_1 & g_0 & \\ & & & d_0g_0 & \\ \hline = d_0g_0 & & & & \end{array} \quad \begin{array}{cccc} s_1 & & & \\ \hline & & d_0 & d_1 \\ \times & g_2 & g_1 & g_0 \\ & & d_0g_1 & d_1g_0 \\ \hline = d_0g_1 + d_1g_0 & & & \end{array} \quad \begin{array}{cccc} s_2 & & & \\ \hline & d_0 & d_1 & \\ \times & g_2 & g_1 & g_0 \\ & d_0g_2 & d_1g_1 & \\ \hline = d_0g_2 + d_1g_1 & & & \end{array} \quad \begin{array}{ccccc} s_3 & & & & \\ \hline & d_0 & d_1 & & \\ \times & & g_2 & g_1 & g_0 \\ & & d_1g_2 & & \\ \hline = d_1g_2 & & & & \end{array}$$

Figure A.1: Graphical symbolic convolution.

where d has size 2 and g has size 3, and the output s is a signal of size 4.

A graphical numeric convolution [1]. with the same size of Figure A.1 is executed in Figure A.2 where $d = [1, 2]$ and $g = [1, 2, 3]$:

$$\begin{array}{ccccc} s_0 & & & & \\ \hline & & & 1 & 2 \\ \times & 3 & 2 & 1 & \\ & & & 1 & \\ \hline = 1 & & & & \end{array} \quad \begin{array}{cccc} s_1 & & & \\ \hline & & 1 & 2 \\ \times & 3 & 2 & 1 \\ & & 2 & 2 \\ \hline = 4 & & & \end{array} \quad \begin{array}{cccc} s_2 & & & \\ \hline & 1 & 2 & \\ \times & 3 & 2 & 1 \\ & 3 & 4 & \\ \hline = 7 & & & \end{array} \quad \begin{array}{ccccc} s_3 & & & & \\ \hline & 1 & 2 & & \\ \times & & 3 & 2 & 1 \\ & & 6 & & \\ \hline = 6 & & & & \end{array}$$

Figure A.2: Graphical numeric convolution.

[1] The short clip **"Graphical 1D Convolution"** animates a numerical convolution while emphasizing how the filter window slides across the data, reinforcing the pictorial method introduced earlier. It is available at https://youtu.be/4YMlgXJsSpA.

Equation A.1 presents the naïve 1D convolution using the summation notation [2].

$$s_i = (d * g)_i = \sum_{k=0}^{N-1} g_{[i+k]} \cdot d_k \tag{A.1}$$

In this naïve implementation of the convolution, there are $K = L \times N$ multiplications and $A = L + N - 1$ additions. If $N = 2$ and $L = 3$ for linear convolution in summation notation:

$$\begin{aligned} s_0 &= d_0 g_0 \\ s_1 &= d_0 g_1 + d_1 g_0 \\ s_2 &= d_0 g_2 + d_1 g_1 \\ s_3 &= d_1 g_2 \end{aligned}$$

.

The linear convolution is equivalent to the following matrix vector multiplication:

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_{L+N-2} \end{bmatrix} = \begin{bmatrix} d_0 & & & \\ d_1 & d_0 & & \\ \vdots & d_1 & \ddots & \\ d_{N-1} & \vdots & \ddots & d_0 \\ & d_{N-1} & & d_1 \\ & & \ddots & \vdots \\ & & & d_{N-1} \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{L-1} \end{bmatrix} \tag{A.2}$$

where the dimensions of the matrices are $s_{[L+N-2]}$, $d_{[L+N-2,L-1]}$ and $g_{[L-1]}$. Using N and L from the previous example to make a linear convolution in matrix notation:

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} d_0 g_0 \\ d_0 g_1 + d_1 g_0 \\ d_0 g_2 + d_1 g_1 \\ d_1 g_2 \end{bmatrix} = \begin{bmatrix} d_0 & & \\ d_1 & d_0 & \\ & d_1 & d_0 \\ & & d_1 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}$$

The linear convolution can also be expressed as a polynomial multiplication:

$$d(x) = \sum_{i=0}^{N-1} d_i x^i \quad g(x) = \sum_{i=0}^{L-1} g_i x^i \tag{A.3}$$

[2]The clip **"Naïve Convolution Math"** steps through the standard summation form, highlighting each multiplication–accumulation that contributes to the output and thereby linking the algebra with the procedural description in the paragraph. Watch it at https://youtu.be/itwrJ3AFF1M.

Where $d(x) \cdot g(x)$ produces $s = (sx^0, sx^1, sx^2 \ldots, sx^{L+N-2})$. Taking the same N and L from the previous examples to make a linear convolution in polynomial notation:

$$
\begin{aligned}
d(x) &= \sum_{i=0}^{1} d_i x^i = d_0 x^0 + d_1 x^1 \\
g(x) &= \sum_{i=0}^{2} g_i x^i = g_0 x^0 + g_1 x^1 + g_2 x^2 \\
s(x) &= (d_0 g_0) x^0 + (d_0 g_1 + d_1 g_0) x^1 + (d_0 g_2 + d_1 g_1) x^2 + (d_1 g_2) x^3
\end{aligned}
$$

A.2.2 Circular Convolution

Another form of convolution is the circular convolution, which represents a specific instance of periodic convolution. This operation entails the convolution of two sequences of equal length, resulting in an output sequence of the same length.

Given two sequences $d = (d_0, \ldots, d_{N-1})$ and $g = (g_0, \ldots, g_{L-1})$, the output sequence $s = (s_0, \ldots, s_{N-1})$ is given by $s = d \circledast g$:

$$
(d \circledast g)_i = \sum_{k=0}^{N-1} g_{[k]} \cdot d_{[(i-k) \bmod N]}, \quad 0 \leq i < N \tag{A.4}
$$

If $N = 2$ and $L = 3$ for circular convolution in summation notation:

$$
\begin{aligned}
s_0 &= d_0 g_0 + d_0 g_2 + d_1 g_1 \\
s_1 &= d_0 g_1 + d_1 g_0 + d_1 g_2
\end{aligned}
$$

.

Circular convolution is equivalent to the following matrix vector multiplication:

$$
\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_{N-2} \\ s_{N-1} \end{bmatrix} = \begin{bmatrix} d_0 & d_{N-1} & d_{N-2} & \cdots & d_1 \\ d_1 & d_0 & d_{N-1} & \cdots & d_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d_{N-2} & d_{N-3} & \cdots & d_0 & d_{N-1} \\ d_{N-1} & d_{N-2} & \cdots & d_1 & d_0 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{L-2} \\ g_{L-1} \end{bmatrix} \tag{A.5}
$$

where the dimensions of the matrices are $s_{[N-1]}$, $d_{[N-1,L-1]}$ and $g_{[L-1]}$. Using N and L from the previous example to make a cyclic convolution in matrix notation:

$$\begin{bmatrix} s_0 \\ s_1 \end{bmatrix} = \begin{bmatrix} d_0g_0 + d_1g_1 + d_0g_2 \\ d_1g_0 + d_0g_1 + d_1g_2 \end{bmatrix} = \begin{bmatrix} d_0 & d_1 & d_0 \\ d_1 & d_0 & d_1 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} \tag{A.6}$$

And the cyclic convolution can also be expressed as a polynomial multiplication:

$$s(x) = g(x) \cdot d(x) \pmod{x^N - 1} \tag{A.7}$$

Taking the same N and L from the previous examples to make a cyclic convolution in polynomial notation:

$$\begin{aligned} s(x) &= (d_0 + d_1x)(g_0 + g_1x + g_2x^2) \pmod{x^2 - 1} \\ &= (d_0g_0 + d_0g_1x + d_0g_2x^2 + d_1g_0x + d_1g_1x^2 + d_1g_2x^3) \pmod{x^2 - 1} \\ &= (d_0g_0 + d_1g_1 + d_0g_2)x^0 + (d_1g_0 + d_0g_1 + d_1g_2)x^1 \end{aligned}$$

.

A.3 Vandermonde Matrix

A Vandermonde matrix $V_{i,j}$ is a type of matrix characterized by rows that follow a geometric progression, where i corresponds to β_i, and j represents the power index. If $V_{4,3}$ and $\beta = [\beta_0, \beta_1, \beta_2, \beta_3] = [0, -1, 1, 2]$, the Vandermonde matrix is:

$$V = \begin{bmatrix} \beta_0^0 & \beta_0^1 & \beta_0^2 \\ \beta_1^0 & \beta_1^1 & \beta_1^2 \\ \beta_2^0 & \beta_2^1 & \beta_2^2 \\ \beta_3^0 & \beta_3^1 & \beta_3^2 \end{bmatrix} = \begin{bmatrix} 0^0 & 0^1 & 0^2 \\ (-1)^0 & (-1)^1 & (-1)^2 \\ 1^0 & 1^1 & 1^2 \\ 2^0 & 2^1 & 2^2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix}.$$

A.4 Toom–Cook Algorithm and Matrix Construction

One of the key algorithms developed by Winograd for fast convolution is the Toom–Cook algorithm [Blahut, 2010; Winograd, 1987; Myers, 1990; Tolimieri et al., 2013], originally introduced for fast polynomial multiplication. For a 1D convolution between a data vector of length N and a filter of length L, Toom–Cook achieves the minimal number of scalar multiplications, given by $K = L + N - 1$, in contrast with the quadratic growth $K = W^2$ of the naïve convolution when L and N have the same width W.

In the matrix formulation, Toom–Cook is described by four matrices A, B, C, and Q such that the convolution can be written as $s = C\Big[(Q \odot Bg) \odot (Ad)\Big]$, where d and g are the input data and filter, respectively, and $\odot$ denotes element-wise multiplication. This section details how these matrices are constructed from a set of interpolation points using Vandermonde matrices and Lagrange interpolation.

We start from a vector of interpolation points

$$\beta = [\beta_0, \beta_1, \ldots, \beta_{k-2}, \triangle], \quad \beta_i \in \mathbb{Z}, \quad i = 0, 1, \ldots, k-2, \tag{A.8}$$

where the last entry $\triangle$ denotes an "infinite" evaluation point used to capture the highest-order term. Rational choices $\beta_i \in \mathbb{Q}$ are also possible, but in this Dissertation we restrict ourselves to integer nodes to keep the resulting matrices hardware-friendly.

Matrices A and B are Vandermonde matrices $V_{i,j}$, and matrix C is the inverse Vandermonde matrix $V^{-1}_{i,j}$, all constructed from the values in β. The definition of Vandermonde matrices was introduced in Section A.3; here we extend it to handle the special symbol $\triangle$. For rows indexed by $i \leq k-2$, the entries of V follow the usual rule

$$V_{i,j} = \beta_i^j, \tag{A.9}$$

while the last row, associated with $\beta_{k-1} = \triangle$, is defined as

$$V_{k-1,j} = \begin{cases} 0 & \text{if } j < n-1, \\ 1 & \text{if } j = n-1, \end{cases} \tag{A.10}$$

where n is the number of columns. In other words, the last row is all zeros except for a single 1 in the last column, corresponding to the contribution of the highest-degree term.

For example, if $V_{5,5}$ and

$$\beta = [\beta_0, \beta_1, \beta_2, \beta_3, \beta_4] = [0, 1, -1, 2, \triangle],$$

the associated Vandermonde matrix is

$$V = \begin{bmatrix} \beta_0^0 & \beta_0^1 & \beta_0^2 & \beta_0^3 & \beta_0^4 \\ \beta_1^0 & \beta_1^1 & \beta_1^2 & \beta_1^3 & \beta_1^4 \\ \beta_2^0 & \beta_2^1 & \beta_2^2 & \beta_2^3 & \beta_2^4 \\ \beta_3^0 & \beta_3^1 & \beta_3^2 & \beta_3^3 & \beta_3^4 \\ \beta_4^0 & \beta_4^1 & \beta_4^2 & \beta_4^3 & \beta_4^4 \end{bmatrix} = \begin{bmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ (-1)^0 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ 2^0 & 2^1 & 2^2 & 2^3 & 2^4 \\ \triangle^0 & \triangle^1 & \triangle^2 & \triangle^3 & \triangle^4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

When $k > 3$, as in the $\{M_5, N_3, L_3, K_5\}$ example of Figure 2.6, the entries of $C = V^{-1}$ are typically rational numbers, belonging to $\mathbb{Q}$ rather than to $\mathbb{Z}$. For $V_{5,5}$ above, the inverse

Vandermonde matrix is

$$V^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & -\frac{1}{3} & -\frac{1}{6} & 2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{6} & \frac{1}{6} & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

This contrasts with the smaller Karatsuba case $k = 3$ in Figure 2.4, where all entries of C are integers.

From an implementation perspective, the presence of fractional coefficients is undesirable, because it implies run-time divisions or non-trivial constant multipliers. To mitigate this, it is convenient to separate numerator and denominator contributions and collect the denominators into a diagonal matrix Q that is applied together with the transformed filter. One way to do this is to compute the inverse interpolation using Lagrange polynomials in two phases.

We adopt a Lagrange formulation in which the numerator polynomials produce the rows of C^{T} and the denominators are collected in the vector Q. The Lagrange basis polynomial associated with node β_i is

$$s_i(x) = \frac{\prod_{j \neq i} (x - \beta_j)}{\prod_{j \neq i} (\beta_i - \beta_j)}. \tag{A.11}$$

For each $\beta_i \in \{\beta_0, \beta_1, \ldots, \beta_{k-2}\}$, we define

$$C_i^{\mathsf{T}}(x) = \prod_{\substack{j=0 \\ j \neq i}}^{k-2} (x - \beta_j), \tag{A.12}$$

which corresponds to the numerator of Equation (A.11), and

$$Q_i = \prod_{\substack{j=0 \\ j \neq i}}^{k-2} \frac{1}{(\beta_i - \beta_j)}, \quad i = 0, 1, \ldots, k-2, \tag{A.13}$$

which collects the denominators. The last components, associated with $\beta_{k-1} = \triangle$, are set to $C_{k-1}^{\mathsf{T}} = 1$ and $Q_{k-1} = 1$.

For the running example $\beta = [0, 1, -1, 2, \triangle]$, the polynomials $C_i^{\mathsf{T}}(x)$ become

$$\begin{aligned}
C_0^{\mathsf{T}}(x) &= (x+1)(x-1)(x-2), \\
C_1^{\mathsf{T}}(x) &= x(x+1)(x-2), \\
C_2^{\mathsf{T}}(x) &= x(x-1)(x-2), \\
C_3^{\mathsf{T}}(x) &= x(x+1)(x-1), \\
C_4^{\mathsf{T}}(x) &= x(x+1)(x-1)(x-2),
\end{aligned}$$

which, when expanded and organized by powers of x, yield the matrix representation

$$\begin{array}{rcccccc} C_0^\mathsf{T} = & 2 & -x & -2x^2 & x^3 & \\ C_1^\mathsf{T} = & & -2x & -x^2 & x^3 & \\ C_2^\mathsf{T} = & & +2x & -3x^2 & x^3 & \\ C_3^\mathsf{T} = & & x & & -x^3 & \\ C_4^\mathsf{T} = & & +2x & -x^2 & -2x^3 & x^4 \end{array} \qquad C^\mathsf{T} = \begin{bmatrix} 2 & -1 & -2 & 1 & 0 \\ 0 & -2 & -1 & 1 & 0 \\ 0 & 2 & -3 & 1 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 \end{bmatrix}.$$

Transposing C^T gives the matrix C, which can be interpreted as rearranging the coefficients according to the powers of x:

$$\begin{array}{rccccc} C_0 = & 2 & & & & \\ C_1 = & -x & -2x & +2x & x & +2x \\ C_2 = & -2x^2 & -x^2 & -3x^2 & & -x^2 \\ C_3 = & x^3 & x^3 & x^3 & -x^3 & -2x^3 \\ C_4 = & & & & & x^4 \end{array}.$$

Evaluating the polynomials at $x = 1$ recovers the concrete integer matrix

$$C = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ -1 & -2 & +2 & -1 & +2 \\ -2 & -1 & -3 & 0 & -1 \\ 1 & 1 & 1 & 1 & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

while the denominators yield

$$\begin{aligned} Q_0 &= \frac{1}{(0-1)(0-1)(0+2)}, \\ Q_1 &= \frac{1}{(-1+0)(-1-1)(-1+2)}, \\ Q_2 &= \frac{1}{(-2-0)(-2+1)(-1-2)}, \\ Q_3 &= \frac{1}{(1+0)(1+1)(1+2)}, \\ Q_4 &= 1, \end{aligned}$$

which we collect in

$$Q = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{6} \\ \frac{1}{6} \\ 1 \end{bmatrix}.$$

This construction explains the structure of the concrete Toom–Cook matrices used in Chapter 2, in particular the Karatsuba case $\{M_3, N_2, L_2, K_3\}$ (Figure 2.4) and the higher-order schemes $\{M_4, N_2, L_3, K_4\}$ and $\{M_5, N_3, L_3, K_5\}$ (Figures 2.5 and 2.6) that serve as 1D reference algorithms for this Dissertation.

A.5 Multidimensional Algorithms

Fast Fourier Transform (FFT) algorithms emerged as practical tools for performing fast convolutions during the same period. A wide range of algorithms were developed to optimize the execution of multidimensional transforms, using methods such as iterated and nested approaches, which were also employed to compute fast convolutions. Later, some of these algorithms were described using Kronecker product notation by Blahut [2010] and Tolimieri et al. [1997].

In the 1970s, Agarwal and Cooley [1977] pioneered a range of techniques related to fast convolution. They developed a series of short convolution matrices, though these innovations initially lacked a unified theoretical framework. One-dimensional short circular convolution methods were extended to longer convolutions by transforming them into multidimensional convolution problems using CRT (Chinese Remainder Theorem) and Krockener product. Winograd developed a similar method for multidimensional transforms [Blahut, 2010; Winograd, 1978].

Multidimensional arrays are typical in signal processing, particularly in image processing and, more recently, in CNNs. These arrays naturally extend the complexity of operations such as convolutions into multiple dimensions. Specifically, two-dimensional convolution is a fundamental operation that processes pairs of two-dimensional arrays. Equation A.14 presents the naïve 2D convolution.

$$s[i_1 i_2] = (d * * g)[i_1 i_2] = \sum_{k_1=0}^{W_1-1} \sum_{k_2=0}^{W_2-1} d[i_1 - k_1, i_2 - k_2] g[k_1 k_2]. \tag{A.14}$$

The simplest way to process 2D data using 1D fast convolution is to treat each function of the second dimension as a separate fast convolution and process the data in 2D as if it were 1D, executing each fast convolution individually. However, more efficient methods are available for data processing, resulting in faster execution times and reduced energy consumption.

$$s = \sum_{k=0}^{R} F(g_k, d_k). \tag{A.15}$$

where: F is the fast convolution algorithm, R is the number of rows of the kernel and image.

A.6 Background: Kronecker product

The Kronecker product, represented by $\otimes$, is a mathematical operation that takes two matrices of arbitrary dimensions and produces a block matrix [Blahut, 2010; Breitzman and Johnson, 2004].

$$C_{mr,ns} = A_{m,n} \otimes B_{r,s}.$$

where A is an $m \times n$ matrix and B is a $r \times s$ matrix, and the Kronecker product $A \otimes B$ is the $mr \times ns$ block matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1s} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1s} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2s} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2s} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{r1} & a_{11}b_{r2} & \cdots & a_{11}b_{rs} & \cdots & \cdots & a_{1n}b_{r1} & a_{1n}b_{r2} & \cdots & a_{1n}b_{rs} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1s} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1s} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2s} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2s} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{r1} & a_{m1}b_{r2} & \cdots & a_{m1}b_{rs} & \cdots & \cdots & a_{mn}b_{r1} & a_{mn}b_{r2} & \cdots & a_{mn}b_{rs} \end{bmatrix}.$$

Consider as an example, A and B two 2×2 matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 2 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \\ 3 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 4 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \end{bmatrix} = \left[\begin{array}{cc|cc} 1\times 0 & 1\times 5 & 2\times 0 & 2\times 5 \\ 1\times 6 & 1\times 7 & 2\times 6 & 2\times 7 \\ \hline 3\times 0 & 3\times 5 & 4\times 0 & 4\times 5 \\ 3\times 6 & 3\times 7 & 4\times 6 & 4\times 7 \end{array}\right] = \left[\begin{array}{cc|cc} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ \hline 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{array}\right].$$

A.7 Power-of-two factorization

To reduce the number of multiplications in Toom–Cook 2D algorithms compared to the naïve method, it is essential to increase N, as shown in Figure 2.8. However, when $N > 4$ in 2D or $N > 2$ in 1D convolution, matrices A and C contain values other than 1, -1, and 0, as shown in Figure 2.6. This behavior contrasts with the case in which $N \leq 4$ in 2D or $N \leq 2$ in 1D convolution, as depicted in Figure 2.5.

Although fast convolution methods can significantly optimize computations by reducing the number of multiplications in $S = G \odot D$ (Equation 2.2) for higher values of N, the need for multipliers in the matrix operations involving A and C, particularly when incorporating constants that are not powers of 2, introduces trade-offs. These trade-offs must be carefully balanced, as they diminish the overall reduction in multiplications that fast convolution techniques initially offer.

To reduce the cost of multiplication, we adopt a method that avoids increasing the multiplication count by decomposing constants into powers of two. For example, integer 18 can be expressed as $18 = 16 + 2 = 2^4 + 2^1$. This allows multiplication by 18 using bit shifts and additions instead of multipliers.

For example, consider matrix B from Figure 2.6. Figure A.3 presents matrix B decomposed into powers of two constants. By employing this decomposition, we efficiently use bit shifts and additions rather than multipliers, simplifying the multiplication process in both scalar and matrix contexts.

$$B = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2^0 & 0 & 0 \\ 2^0 & 2^0 & 2^0 \\ 2^0 & -2^0 & 2^0 \\ 2^0 & 2^1 & 2^2 \\ 0 & 0 & 2^0 \end{bmatrix}.$$

Figure A.3: Matrix B decomposed into powers of two constants.

A.8 Agarwal–Cooley and split-nesting

The Agarwal-Cooley algorithm employs the tensor product to create long cyclic convolutions from smaller cyclic convolutions [Breitzman and Johnson, 2004]. It decomposes a one-dimensional cyclic convolution of size $n = n' \times n''$ into a multidimensional cyclic convolution of size $n' \times n''$, provided that n' and n'' are coprime. This approach is utilized to combine an n'-point fast convolution with M' multiplications and an n''-point fast convolution with M'' multiplications, resulting in a multidimensional convolution of size $n' \times n''$ with $M' \times M''$ multiplications. The algorithm employs CRT for integers to map a one-dimensional cyclic convolution into a multidimensional cyclic convolution, differing from the application of the CRT in the context of polynomials as used in the Winograd algorithm. While it does not minimize the number of multiplications to the extent of the Winograd algorithm, it mitigates the tendency for an excessive increase in additions for large n. The Agarwal-Cooley algorithm is recommended for decomposing long cyclic convolutions into shorter ones, which can subsequently be handled using the Winograd or Toom-Cook algorithms [Agarwal and Cooley, 1977; Blahut, 2010].

The split-nesting algorithm was introduced by Nussbaumer [1978] and is directly derived from the Agarwal-Cooley algorithm [Breitzman and Johnson, 2004]. This algorithm

efficiently computes large one-dimensional convolutions utilizing a limited number of small fast convolutions. It serves as a generalization of the method proposed by Agarwal and Cooley. In particular, this approach significantly reduces the number of additions compared to the Agarwal-Cooley algorithm. The development of this technique is elaborated on Breitzman and Johnson [2004], where it is shown to leverage properties of the tensor product.

APPENDIX B – SYSTEMATIC REVIEW TABLES

This appendix consolidates the per-paper information used in the systematic review of Winograd-based convolution accelerators. The first table summarizes, for each reference, how fast convolution is generated, how it is bound to convolution, and which implementation language is reported. The second table details which Winograd output size $m \times m$ are explicitly implemented in each work.

Reference	Generator	Bind	Language
Zhuge et al. [2018]	Toom-Cook	Nest	HSL
Zhao et al. [2019]			
Zhai et al. [2023]		Nest	HSL
Yu et al. [2020]	Toom-Cook	Nest	
Yepez and Ko [2020]	Toom-Cook	Nest	
Ye et al. [2020]	Toom-Cook	Nest	HSL
Yang et al. [2019]		Nest	
Yang et al. [2020]	Toom-Cook	Nest	
Xie et al. [2024]	Toom-Cook	Nest	HDL
Xiao et al. [2017]		Nest	HSL
Wang et al. [2020]		Nest	HDL
Shen et al. [2018]	Toom-Cook	Nest	HSL
Shen et al. [2020]	Toom-Cook	Nest	HSL
Podili et al. [2017]			
Lu and Liang [2018]		Nest	HSL
Li et al. [2024]		Nest	HSL
Liu et al. [2021]	Toom-Cook	Nest	HSL
Li et al. [2025]	Toom-Cook		HDL
Liang et al. [2020]	Toom-Cook	Nest	HSL
Kala et al. [2019b]		Nest	
Kala et al. [2019a]		Nest	
Kala and Nalesh [2020]		Nest	
Kala et al. [2018]	Toom-Cook	Nest	
Huang et al. [2018]		Nest	
Di et al. [2020]		Nest	HSL
DiCecco et al. [2016]	Toom-Cook	Nest	
Deng et al. [2021]	Toom-Cook	Nest	
Chang et al. [2020]	Toom-Cook	Nest	HSL
Bao et al. [2020]	Toom-Cook	Nest	HSL
Ahmad and Pasha [2020]		Nest	
Ahmad and Pasha [2019]	Toom-Cook		
Aydonat et al. [2017]			HSL
Lu et al. [2017]	Toom-Cook	Nest	HSL

Table B.1: Winograd algorithms and implementation details for each article in the systematic review.

Reference	2x2	3x3	4x4	5x5	6x6	7x7	8x8
Zhuge et al. [2018]	X		X				
Zhao et al. [2019]	X						
Zhai et al. [2023]							
Yu et al. [2020]	X						
Yepez and Ko [2020]	X		X				
Ye et al. [2020]	X		X				
Yang et al. [2019]			X				
Yang et al. [2020]	X						
Xie et al. [2024]	X		X		X		
Xiao et al. [2017]			X				
Wang et al. [2020]			X				
Shen et al. [2018]	X						
Shen et al. [2020]	X						
Podili et al. [2017]	X						
Lu and Liang [2018]	X						
Lu et al. [2017]	X						
Li et al. [2024]	X		X		X		
Liu et al. [2021]	X		X				
Li et al. [2025]			X				
Liang et al. [2020]			X	X	X	X	X
Kala et al. [2019b]	X	X	X		X		
Kala et al. [2019a]	X						
Kala and Nalesh [2020]	X	X	X		X		
Kala et al. [2018]	X	X	X		X		
Huang et al. [2018]	X		X				
Di et al. [2020]	X						
DiCecco et al. [2016]	X						
Deng et al. [2021]	X						
Chang et al. [2020]	X						
Bao et al. [2020]	X						
Aydonat et al. [2017]			X				
Ahmad and Pasha [2020]			X				
Ahmad and Pasha [2019]	X	X	X	X	X	X	

Table B.2: Winograd output size $m \times m$ for 3×3 kernel implemented by each article. A mark "X" indicates that the corresponding configuration is explicitly reported in the paper.

APPENDIX C – PROPOSED FAST CONVOLUTION ALGORITHMS

This appendix collects the algebraic background and constructive proofs underlying the proposed fast convolution algorithms of Chapter 4. While the main chapter focuses on the high-level ideas, design choices, and architectural implications of the methods, here we present the complete derivations and worked micro-examples that motivate the final A, B, C matrices. The goal is twofold: (i) to make the Dissertation self-contained for readers interested in the polynomial and CRT machinery that sits behind the algorithms, and (ii) to provide a precise reference for future extensions or reimplementations of the proposed schemes.

C.1 Generating Fast Linear Convolution Algorithms via CRT

This section presents a constructive method to **generate** Winograd-style fast linear convolution algorithms in the bilinear form

$$\mathbf{h} \;=\; C\Big((B\mathbf{g}) \odot (A\mathbf{d})\Big),$$

where $\odot$ denotes the Hadamard (componentwise) product. The construction is systematic: given a factorization of a suitable modulus polynomial, it produces the matrices A, B, C and therefore a complete algorithm.

Definition (Monomial, polynomial, degree, leading coefficient, monic). *A* ***monomial*** *is a term of the form* x^k *(e.g.,* $1 = x^0, x, x^2$*). A* ***polynomial*** *over a field* $\mathbb{F}$ *is a finite sum of monomials,* $p(x) = a_0 + a_1x + \cdots + a_rx^r$ *with* $a_i \in \mathbb{F}$*. The* ***degree*** $\deg(p)$ *is the largest exponent with nonzero coefficient. The* ***leading coefficient*** *is the coefficient of* $x^{\deg(p)}$*. A polynomial is* ***monic*** *if its leading coefficient is* 1*.*

Definition (Polynomial gcd and coprime factors). *For polynomials* $p(x), q(x) \in \mathbb{F}[x]$, $\gcd(p, q)$ *denotes their greatest common divisor (a polynomial of maximum degree that divides both). We say* p *and* q *are* ***coprime*** *if* $\gcd(p, q) = 1$*, i.e., they share no nontrivial common factor.*

Definition (Standard monomial basis). *To represent polynomials as vectors, we use the* ***standard monomial basis*** $\{1, x, x^2, \ldots, x^{L-1}\}$ *for polynomials of degree* $< L$*. For example,* $D(x) = 3 + 5x - 2x^2$ *corresponds to the coefficient vector* $\mathbf{D} = [3,\ 5,\ -2]^T$ *in this basis.* ***Notation.*** *We denote by* $\operatorname{coeff}(D(x))$ *the column vector of coefficients of* $D(x)$ *in this basis (ordered*

from x^0 to x^{L-1}). For instance,

$$\mathrm{coeff}(3 + 5x - 2x^2) = \begin{bmatrix} 3 \\ 5 \\ -2 \end{bmatrix}.$$

Important: *the vector length is always the dimension of the target basis. If needed, higher-order terms are truncated by modular reduction and missing terms are padded with zeros.*

Problem model

Let the input (data) and kernel (filter) be represented as polynomials:

$$d(x) = \sum_{i=0}^{n-1} d_i x^i, \qquad g(x) = \sum_{j=0}^{m-1} g_j x^j.$$

Their linear convolution equals the coefficient vector of

$$h(x) = d(x)g(x), \qquad \deg(h) = n + m - 2.$$

Stage 1: Choosing a modulus and why coprime factorization matters

We choose a **monic** modulus polynomial $m(x)$ such that

$$\deg(m) = n + m - 2, \qquad m(x) = \prod_{i=1}^{t} p_i(x), \qquad \gcd(p_i, p_j) = 1 \ (i \neq j).$$

The coprime condition $\gcd(p_i, p_j) = 1$ is essential: it guarantees that CRT recombination is **unique**, i.e., that a polynomial modulo $m(x)$ is uniquely determined by its residues modulo the factors $p_i(x)$. Without coprimality, residues are not independent and the inverse CRT map is not well-defined.

Stage 2: Reduction modulo $p(x)$ as a linear map

Fix one factor $p(x) = p_i(x)$. The operation "take the remainder modulo $p(x)$" is linear in the polynomial coefficients. Therefore, there exists a matrix M_p such that for any polynomial $D(x)$ represented in the standard monomial basis,

$$\mathrm{coeff}\Big(D(x) \bmod p(x)\Big) = M_p\, \mathrm{coeff}(D(x)).$$

Constructing M_p (column-by-column).

For each monomial basis element x^j (column j), compute the remainder $r_j(x) = x^j \bmod p(x)$ and express it in the basis $\{1, x, \dots, x^{\deg(p)-1}\}$. Place these coefficients as column j of M_p. By linearity, M_p correctly maps any coefficient vector to the coefficient vector of the remainder.

Stage 3: CRT reduction as a stacked transform

Given $m(x) = \prod_i p_i(x)$, define the CRT reduction matrix

$$R = \begin{bmatrix} M_{p_1} \\ M_{p_2} \\ \vdots \\ M_{p_t} \end{bmatrix}.$$

Then $R\mathbf{d}$ is the concatenation of the residue coefficient vectors of $d(x)$ modulo each factor:

$$R\mathbf{d} = \begin{bmatrix} \mathrm{coeff}(d \bmod p_1) \\ \vdots \\ \mathrm{coeff}(d \bmod p_t) \end{bmatrix}, \qquad R\mathbf{g} = \begin{bmatrix} \mathrm{coeff}(g \bmod p_1) \\ \vdots \\ \mathrm{coeff}(g \bmod p_t) \end{bmatrix}.$$

Definition (Direct sum $\oplus$). *For matrices X and Y, their **direct sum** is the block-diagonal matrix*

$$X \oplus Y = \begin{bmatrix} X & 0 \\ 0 & Y \end{bmatrix}.$$

For vectors $\mathbf{u}$ and $\mathbf{v}$, $\mathbf{u} \oplus \mathbf{v} = [\mathbf{u}^T \ \mathbf{v}^T]^T$ is concatenation. In our context, $\oplus$ encodes independent CRT subproblems that run side-by-side without cross-coupling.

Stage 4: Factor-level modular multipliers and the role of $\oplus$

For each factor $p_i(x)$, we construct a bilinear algorithm (C_i, A_i, B_i) that computes multiplication **modulo** $p_i(x)$:

$$\mathbf{w}_i = C_i\Big((B_i\mathbf{v}_i) \odot (A_i\mathbf{u}_i)\Big) \equiv \mathrm{coeff}\Big(u_i(x)v_i(x) \bmod p_i(x)\Big).$$

A systematic way to obtain such a block is:

1. compute the ordinary product $u_i(x)v_i(x)$ using any bilinear scheme;

2. apply the linear reduction matrix M_{p_i} to the output coefficients, which can be absorbed into the recombination matrix C_i.

Because factor-level subproblems are independent, we combine them using direct sums:

$$A_{\oplus} = \bigoplus_{i=1}^{t} A_i, \quad B_{\oplus} = \bigoplus_{i=1}^{t} B_i, \quad C_{\oplus} = \bigoplus_{i=1}^{t} C_i.$$

This block-diagonal structure exactly captures "run all modular multiplications in parallel" in the CRT domain.

Stage 5: CRT recombination (R^{-1}) via idempotent selectors

CRT provides an inverse linear map R^{-1} that reconstructs a polynomial modulo $m(x)$ from its residues modulo $p_i(x)$. A constructive recombination uses **idempotent selector** polynomials. Define

$$m_i(x) = \frac{m(x)}{p_i(x)}.$$

Since $\gcd(m_i, p_i) = 1$, there exists $s_i(x)$ such that

$$m_i(x)s_i(x) \equiv 1 \pmod{p_i(x)}.$$

Define the selector

$$e_i(x) \equiv m_i(x)s_i(x) \pmod{m(x)}.$$

Then $e_i(x)$ satisfies:

$$e_i(x) \equiv 1 \pmod{p_i(x)}, \qquad e_i(x) \equiv 0 \pmod{p_j(x)}\ (j \neq i).$$

Given residues $a_i(x)$, the recombined polynomial modulo $m(x)$ is

$$a(x) \equiv \sum_{i=1}^{t} a_i(x)e_i(x) \pmod{m(x)}.$$

Because multiplication by $e_i(x)$ modulo $m(x)$ is linear in coefficients, the above recombination corresponds to a fixed matrix R^{-1} acting on the stacked residue vectors.

Stage 6: A bilinear algorithm for multiplication modulo $m(x)$

Putting the pieces together, multiplication modulo $m(x)$ is:

$$\mathbf{c} = R^{-1}\, C_{\oplus}\Big(\big(B_{\oplus}(R\mathbf{g})\big) \odot \big(A_{\oplus}(R\mathbf{d})\big)\Big),$$

which defines a generated triple (C_m, A_m, B_m) such that $c(x) \equiv d(x)g(x) \pmod{m(x)}$.

Stage 7: Tolimieri linear-from-cyclic correction

We now recover the **full linear product** from the modular result. Let $c(x) \equiv d(x)g(x) \pmod{m(x)}$. By definition of remainder, there exists $q(x)$ such that

$$d(x)g(x) = c(x) + q(x)m(x).$$

Since $\deg(dg) = \deg(m) = n + m - 2$ and $\deg(c) \leq \deg(m) - 1$, $q(x)$ cannot have degree ≥ 1; therefore $q(x)$ is a scalar $t \in \mathbb{F}$. Comparing leading terms and using that $m(x)$ is monic gives

$$t = d_{n-1}g_{m-1}.$$

Hence the Tolimieri identity:

$$d(x)g(x) = c(x) + \left(d_{n-1}g_{m-1}\right) m(x).$$

Operationally, we append one extra Hadamard component computing $t = d_{n-1}g_{m-1}$ (often referred to as an "∞ channel" since it corresponds to extracting leading coefficients), and then apply a fixed linear map that adds $t\,m(x)$ into the coefficient vector.

Stage 8: Selection and padding (wiring)

If the internal CRT/modular representation uses coefficient vectors larger than the original lengths n and m, we apply sparse embedding (zero-padding) and selection matrices. These are purely structural (0/1) maps and do not alter correctness.

Worked micro-example (selector intuition).

For $m(x) = (x - 1)(x + 1)$, define $p_1 = x - 1$ and $p_2 = x + 1$. Then $m_1 = x + 1$ and $m_2 = x - 1$. One can choose $s_1 = \frac{1}{2}$ since $(x + 1)\frac{1}{2} \equiv 1 \pmod{x - 1}$ (evaluate at $x = 1$), and $s_2 = -\frac{1}{2}$ since $(x - 1)(-\frac{1}{2}) \equiv 1 \pmod{x + 1}$ (evaluate at $x = -1$). Thus

$$e_1(x) \equiv \frac{x + 1}{2} \pmod{m(x)}, \qquad e_2(x) \equiv \frac{1 - x}{2} \pmod{m(x)}.$$

Indeed, $e_1(1) = 1, e_1(-1) = 0$ and $e_2(1) = 0, e_2(-1) = 1$, i.e., each e_i acts as a residue selector. The recombination $a(x) \equiv a_1(x)e_1(x) + a_2(x)e_2(x) \pmod{m(x)}$ then reconstructs the unique polynomial modulo $m(x)$ matching both residues.

Algorithm C.1 GenerateFastLinearConvolution via CRT (Breitzman) + correction (Tolimieri)

Require: lengths n, m; coprime factors $\{p_i(x)\}_{i=1}^{t}$ with monic $m(x) = \prod_i p_i(x)$ and $\deg(m) = n + m - 2$

Ensure: matrices (C, A, B) such that $\mathbf{h} = C((B\mathbf{g}) \odot (A\mathbf{d}))$ equals the linear convolution

1: Build $m(x) \leftarrow \prod_{i=1}^{t} p_i(x)$ (monic), verify $\deg(m) = n + m - 2$
2: **for** each factor $p_i(x)$ **do**
3: Construct reduction matrix M_{p_i} such that $\text{coeff}(D \bmod p_i) = M_{p_i}\text{coeff}(D)$
4: Construct a bilinear modular multiplier (C_i, A_i, B_i) for multiplication modulo $p_i(x)$
5: **end for**
6: Build CRT reduction $R \leftarrow [M_{p_1}; \ldots ; M_{p_t}]$
7: Combine blocks: $(C_\oplus, A_\oplus, B_\oplus) \leftarrow \bigoplus_{i=1}^{t}(C_i, A_i, B_i)$
8: Construct CRT recombination R^{-1} (e.g., via selectors $e_i(x)$) as a linear coefficient map
9: Define modular product: $\mathbf{c} \leftarrow R^{-1}C_\oplus\Big((B_\oplus(R\mathbf{g})) \odot (A_\oplus(R\mathbf{d}))\Big)$
10: Append scalar channel $t \leftarrow d_{n-1}g_{m-1}$ (extend transforms to output d_{n-1} and g_{m-1})
11: Apply fixed linear correction implementing $h(x) = c(x) + t\, m(x)$ (absorb into final C)
12: Apply padding/selection matrices if required
13: **return** (C, A, B)

C.2 Linear 2×2 convolution via Evaluate–Hadamard–Interpolate (the A, B, C form)

This section is intentionally procedural: each stage states input, operation, and output.

Problem statement.

Input polynomials:

$$d(x) = d_0 + d_1 x, \qquad g(x) = g_0 + g_1 x.$$

Target product:

$$h(x) = d(x)g(x) = d_0 g_0 + (d_0 g_1 + d_1 g_0)x + d_1 g_1 x^2 = h_0 + h_1 x + h_2 x^2.$$

Vector form:

$$\mathbf{d} = \begin{bmatrix} d_0 \\ d_1 \end{bmatrix}, \quad \mathbf{g} = \begin{bmatrix} g_0 \\ g_1 \end{bmatrix}, \quad \mathbf{h} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix}.$$

Stage 1: Choosing a modulus and coprime factorization

Input: output degree is 2.

Operation: choose a monic polynomial with degree 2 and simple coprime factors.

$$m(x) = x(x-1), \qquad p_1(x) = x, \qquad p_2(x) = x - 1.$$

Output: a CRT domain with two scalar channels (roots 0 and 1).

Stage 2: Reduction modulo each factor (matrix construction)

Input: generic quadratic

$$u(x) = u_0 + u_1 x + u_2 x^2, \qquad \mathbf{u} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix}.$$

The monomials 1, x, and x^2 are the basis elements of this input space.

Operation A (mod x): compute remainders of basis monomials.

$$1 \bmod x = 1, \quad x \bmod x = 0, \quad x^2 \bmod x = 0.$$

So the map is

$$M_x = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}.$$

Operation B (mod $x-1$): evaluate basis monomials at $x = 1$.

$$1 \bmod (x-1) = 1, \quad x \bmod (x-1) = 1, \quad x^2 \bmod (x-1) = 1.$$

So

$$M_{x-1} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}.$$

Output: two explicit reduction matrices.

Stage 3: CRT reduction as stacked transform

Input: M_x and M_{x-1}.

Operation: stack them.

$$R = \begin{bmatrix} M_x \\ M_{x-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

Output: for any quadratic $u(x)$,

$$R\mathbf{u} = \begin{bmatrix} u(0) \\ u(1) \end{bmatrix} = \begin{bmatrix} u_0 \\ u_0 + u_1 + u_2 \end{bmatrix}.$$

Stage 4: Evaluation channels and Hadamard products

Input: $\mathbf{d}, \mathbf{g}$.

Operation A (define transforms): The transform matrices are obtained from the three chosen evaluation channels for degree-1 polynomials:

channel	$d(x) = d_0 + d_1 x$	row vector on $[d_0\ d_1]^T$
$x = 0$	$d(0) = d_0$	$[1\ 0]$
$x = 1$	$d(1) = d_0 + d_1$	$[1\ 1]$
$x = \infty$	leading-term channel $= d_1$	$[0\ 1]$

Stacking these rows gives the forward transform matrix. Since data and kernel use the same channels, the two forward transforms are equal, so $A = B$.

$$A = B = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Operation B (apply A to d):

$$A\mathbf{d} = \begin{bmatrix} 1 \cdot d_0 + 0 \cdot d_1 \\ 1 \cdot d_0 + 1 \cdot d_1 \\ 0 \cdot d_0 + 1 \cdot d_1 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_0 + d_1 \\ d_1 \end{bmatrix}.$$

Operation C (apply B to g):

$$B\mathbf{g} = \begin{bmatrix} 1 \cdot g_0 + 0 \cdot g_1 \\ 1 \cdot g_0 + 1 \cdot g_1 \\ 0 \cdot g_0 + 1 \cdot g_1 \end{bmatrix} = \begin{bmatrix} g_0 \\ g_0 + g_1 \\ g_1 \end{bmatrix}.$$

Operation D (Hadamard):

$$\boldsymbol{\mu} = (A\mathbf{d}) \odot (B\mathbf{g}) = \begin{bmatrix} \mu_0 \\ \mu_1 \\ \mu_2 \end{bmatrix} = \begin{bmatrix} d_0 g_0 \\ (d_0 + d_1)(g_0 + g_1) \\ d_1 g_1 \end{bmatrix}.$$

Output:

$$\mu_0 = h(0), \qquad \mu_1 = h(1), \qquad \mu_2 = t = d_1 g_1.$$

Stage 5: Recombination of the finite channels ($x = 0$ and $x = 1$)

Input:

$$m_0 = \mu_0 = h(0), \qquad m_1 = \mu_1 = h(1).$$

Operation: reconstruct the degree-< 2 intermediate polynomial

$$c(x) = c_0 + c_1 x$$

from

$$c(0) = m_0, \qquad c(1) = m_1.$$

Substituting $x = 0$ and $x = 1$ gives the explicit linear system

$$\begin{cases} c_0 = m_0, \\ c_0 + c_1 = m_1. \end{cases}$$

Solving it step by step:

$$c_0 = m_0, \qquad c_1 = m_1 - c_0 = m_1 - m_0.$$

Output:

$$c(x) = m_0 + (m_1 - m_0)x.$$

This stage uses only the finite channels. The leading-term channel $m_\infty = \mu_2 = d_1 g_1$ is incorporated in Stage 6.

Stage 6: Tolimieri correction for full linear result

Input: $c(x)$ and $t = \mu_2$.

Operation:

$$h(x) = c(x) + t\, m(x), \qquad m(x) = x(x-1) = x^2 - x.$$

Substitute and expand:

$$h(x) = \mu_0 + (\mu_1 - \mu_0)x + \mu_2(x^2 - x) = \mu_0 + (-\mu_0 + \mu_1 - \mu_2)x + \mu_2 x^2.$$

Output coefficients:

$$h_0 = \mu_0, \qquad h_1 = -\mu_0 + \mu_1 - \mu_2, \qquad h_2 = \mu_2.$$

Stage 7: Final matrix C (from formulas to matrix rows)

Input: $\boldsymbol{\mu} = \begin{bmatrix}\mu_0 & \mu_1 & \mu_2\end{bmatrix}^T$.
Operation: encode each formula as one row.

$$\text{Row 1 } (h_0): \; [1\ 0\ 0], \quad \text{Row 2 } (h_1): \; [-1\ 1\ -1], \quad \text{Row 3 } (h_2): \; [0\ 0\ 1].$$

Hence

$$C = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}.$$

Output check:

$$C\boldsymbol{\mu} = \begin{bmatrix} \mu_0 \\ -\mu_0 + \mu_1 - \mu_2 \\ \mu_2 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \end{bmatrix}.$$

Stage 8: Complete flow and compact operator

Flow:

$$\mathbf{d} \xrightarrow{A} A\mathbf{d}, \quad \mathbf{g} \xrightarrow{B} B\mathbf{g}, \quad \xrightarrow{\odot} \boldsymbol{\mu}, \quad \xrightarrow{C} \mathbf{h}.$$

Compact bilinear form:

$$\mathbf{h} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}}_{C} \left(\underbrace{\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}}_{A} \mathbf{d} \odot \underbrace{\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}}_{B} \mathbf{g} \right).$$

Equivalent diagonal-middle form:

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} g_0 & 0 & 0 \\ 0 & g_0 + g_1 & 0 \\ 0 & 0 & g_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \end{bmatrix}.$$

Numerical mini-check (2×2)

Input selection:

$$\mathbf{d} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \Rightarrow d(x) = 1 + 2x, \qquad \mathbf{g} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \Rightarrow g(x) = 2 + 3x.$$

Step 1 (evaluations):

$$A\mathbf{d} = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}, \qquad B\mathbf{g} = \begin{bmatrix} 2 \\ 5 \\ 3 \end{bmatrix}.$$

Passo 2 (Hadamard):

$$\boldsymbol{\mu} = (A\mathbf{d}) \odot (B\mathbf{g}) = \begin{bmatrix} 2 \\ 15 \\ 6 \end{bmatrix}.$$

Step 3 (final interpolation):

$$\mathbf{h} = C\boldsymbol{\mu} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 15 \\ 6 \end{bmatrix} = \begin{bmatrix} 2 \\ 7 \\ 6 \end{bmatrix}.$$

Direct verification:

$$(1 + 2x)(2 + 3x) = 2 + 7x + 6x^2,$$

que coincide com $\mathbf{h} = [2, 7, 6]^T$.

C.3 Linear 2×3 convolution via Evaluate–Hadamard–Interpolate (the A, B, C form)

This section uses the same micro-step structure as 2×2, now with one extra kernel coefficient.

Problem statement.

$$d(x) = d_0 + d_1 x, \qquad g(x) = g_0 + g_1 x + g_2 x^2.$$

$$h(x) = d(x)g(x) = h_0 + h_1 x + h_2 x^2 + h_3 x^3.$$

Vector form:

$$\mathbf{d} = \begin{bmatrix} d_0 \\ d_1 \end{bmatrix}, \quad \mathbf{g} = \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}, \quad \mathbf{h} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix}.$$

Stage 1: Choosing a modulus and coprime factorization

Input: output degree is 3.

Operation: choose monic degree-3 polynomial with simple linear factors.

$$m(x) = x(x-1)(x+1) = x^3 - x,$$

$$p_1(x) = x, \qquad p_2(x) = x - 1, \qquad p_3(x) = x + 1.$$

Output: CRT domain with three scalar residue channels (at $0, 1, -1$).

Stage 2: Reduction modulo each factor (matrix construction)

Input: generic quadratic

$$u(x) = u_0 + u_1 x + u_2 x^2, \qquad \mathbf{u} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix}.$$

Operation A (mod x):

$$1 \bmod x = 1, \quad x \bmod x = 0, \quad x^2 \bmod x = 0 \Rightarrow M_x = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}.$$

Operation B (mod $x-1$):

$$1 \mapsto 1, \quad x \mapsto 1, \quad x^2 \mapsto 1 \Rightarrow M_{x-1} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}.$$

Operation C (mod $x+1$):

$$1 \mapsto 1, \quad x \mapsto -1, \quad x^2 \mapsto 1 \Rightarrow M_{x+1} = \begin{bmatrix} 1 & -1 & 1 \end{bmatrix}.$$

Output: three explicit reduction rows.

Stage 3: CRT reduction matrix and output meaning

Input: M_x, M_{x-1}, M_{x+1}.

Operation: stack rows.

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix}.$$

Output:

$$R\mathbf{u} = \begin{bmatrix} u(0) \\ u(1) \\ u(-1) \end{bmatrix} = \begin{bmatrix} u \bmod x \\ u \bmod (x-1) \\ u \bmod (x+1) \end{bmatrix}.$$

Stage 4: Evaluation channels and Hadamard products

Input: $\mathbf{d}, \mathbf{g}$.

Operation A (define transforms): As in the 2×2 case, A and B are built row-by-row from the selected channels $x \in \{0, 1, -1, \infty\}$.

For $d(x) = d_0 + d_1 x$:

channel	$d(x) = d_0 + d_1 x$	row vector on $[d_0\ d_1]^T$
$x = 0$	$d(0) = d_0$	$[1\ 0]$
$x = 1$	$d(1) = d_0 + d_1$	$[1\ 1]$
$x = -1$	$d(-1) = d_0 - d_1$	$[1\ -1]$
$x = \infty$	leading-term channel $= d_1$	$[0\ 1]$

For $g(x) = g_0 + g_1 x + g_2 x^2$:

channel	$g(x) = g_0 + g_1 x + g_2 x^2$	row vector on $[g_0\ g_1\ g_2]^T$
$x = 0$	$g(0) = g_0$	$[1\ 0\ 0]$
$x = 1$	$g(1) = g_0 + g_1 + g_2$	$[1\ 1\ 1]$
$x = -1$	$g(-1) = g_0 - g_1 + g_2$	$[1\ -1\ 1]$
$x = \infty$	leading-term channel $= g_2$	$[0\ 0\ 1]$

Stacking the rows gives A and B:

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

Operation B (apply A):

$$A\mathbf{d} = \begin{bmatrix} 1 \cdot d_0 + 0 \cdot d_1 \\ 1 \cdot d_0 + 1 \cdot d_1 \\ 1 \cdot d_0 - 1 \cdot d_1 \\ 0 \cdot d_0 + 1 \cdot d_1 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_0 + d_1 \\ d_0 - d_1 \\ d_1 \end{bmatrix}.$$

Operation C (apply B):

$$B\mathbf{g} = \begin{bmatrix} 1 \cdot g_0 + 0 \cdot g_1 + 0 \cdot g_2 \\ 1 \cdot g_0 + 1 \cdot g_1 + 1 \cdot g_2 \\ 1 \cdot g_0 - 1 \cdot g_1 + 1 \cdot g_2 \\ 0 \cdot g_0 + 0 \cdot g_1 + 1 \cdot g_2 \end{bmatrix} = \begin{bmatrix} g_0 \\ g_0 + g_1 + g_2 \\ g_0 - g_1 + g_2 \\ g_2 \end{bmatrix}.$$

Operation D (Hadamard):

$$\boldsymbol{\mu} = (A\mathbf{d}) \odot (B\mathbf{g}) = \begin{bmatrix} \mu_0 \\ \mu_1 \\ \mu_2 \\ \mu_3 \end{bmatrix} = \begin{bmatrix} d_0 g_0 \\ (d_0 + d_1)(g_0 + g_1 + g_2) \\ (d_0 - d_1)(g_0 - g_1 + g_2) \\ d_1 g_2 \end{bmatrix}.$$

Output channels:

$$\mu_0 = h(0), \qquad \mu_1 = h(1), \qquad \mu_2 = h(-1), \qquad \mu_3 = t = d_1 g_2.$$

Stage 5: Quadratic interpolation from $\{0, 1, -1\}$

Input: residues μ_0, μ_1, μ_2.

Operation: reconstruct

$$c(x) = c_0 + c_1 x + c_2 x^2$$

with

$$c(0) = \mu_0, \qquad c(1) = \mu_1, \qquad c(-1) = \mu_2.$$

Line-by-line:

$$c_0 = \mu_0,$$

$$c_0 + c_1 + c_2 = \mu_1, \qquad c_0 - c_1 + c_2 = \mu_2,$$

$$\Rightarrow c_1 = \frac{\mu_1 - \mu_2}{2}, \qquad c_2 = \frac{\mu_1 + \mu_2}{2} - \mu_0.$$

Matrix form:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} \\ -1 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}}_{V^{-1}} \begin{bmatrix} \mu_0 \\ \mu_1 \\ \mu_2 \end{bmatrix}.$$

Output: modulo-m polynomial $c(x)$.

Stage 6: Tolimieri correction for full cubic output

Input: $c(x)$ and $t = \mu_3$.

Operation:

$$h(x) = c(x) + t\, m(x), \qquad m(x) = x^3 - x.$$

Expand:

$$h(x) = c_0 + (c_1 - t)x + c_2 x^2 + t x^3.$$

Hence

$$h_0 = c_0, \qquad h_1 = c_1 - t, \qquad h_2 = c_2, \qquad h_3 = t.$$

Substitute Stage 5 formulas:

$$h_0 = \mu_0, \qquad h_1 = \frac{\mu_1 - \mu_2}{2} - \mu_3, \qquad h_2 = \frac{\mu_1 + \mu_2}{2} - \mu_0, \qquad h_3 = \mu_3.$$

Stage 7: Final matrix C (row-by-row construction)

Input: $\boldsymbol{\mu} = \begin{bmatrix} \mu_0 & \mu_1 & \mu_2 & \mu_3 \end{bmatrix}^T$.

Operation: convert each h_i formula into one matrix row:

$$\text{Row 1 } (h_0): \ [1\ 0\ 0\ 0],$$

$$\text{Row 2 } (h_1): \ [0\ \tfrac{1}{2}\ -\tfrac{1}{2}\ -1],$$

$$\text{Row 3 } (h_2): \ [-1\ \tfrac{1}{2}\ \tfrac{1}{2}\ 0],$$

$$\text{Row 4 } (h_3): \ [0\ 0\ 0\ 1].$$

Thus

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} & -1 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Output check:

$$C\boldsymbol{\mu} = \begin{bmatrix} \mu_0 \\ \frac{\mu_1-\mu_2}{2} - \mu_3 \\ \frac{\mu_1+\mu_2}{2} - \mu_0 \\ \mu_3 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix}.$$

Stage 8: Complete flow and compact operator

Flow:

$$\mathbf{d} \xrightarrow{A} A\mathbf{d}, \quad \mathbf{g} \xrightarrow{B} B\mathbf{g}, \quad \xrightarrow{\odot} \boldsymbol{\mu}, \quad \xrightarrow{C} \mathbf{h}.$$

Compact bilinear form:

$$\mathbf{h} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} & -1 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{C} \left(\underbrace{\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}}_{A} \mathbf{d} \odot \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix}}_{B} \mathbf{g} \right).$$

Equivalent diagonal-middle form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} & -1 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} g_0 & 0 & 0 & 0 \\ 0 & g_0+g_1+g_2 & 0 & 0 \\ 0 & 0 & g_0-g_1+g_2 & 0 \\ 0 & 0 & 0 & g_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \end{bmatrix}.$$

Numerical mini-check (2×3)

Input selection:

$$\mathbf{d} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \Rightarrow d(x) = 1 + 2x, \qquad \mathbf{g} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \Rightarrow g(x) = 1 + 2x + 3x^2.$$

Step 1 (evaluations):

$$A\mathbf{d} = \begin{bmatrix} 1 \\ 3 \\ -1 \\ 2 \end{bmatrix}, \qquad B\mathbf{g} = \begin{bmatrix} 1 \\ 6 \\ 2 \\ 3 \end{bmatrix}.$$

Passo 2 (Hadamard):

$$\boldsymbol{\mu} = (A\mathbf{d}) \odot (B\mathbf{g}) = \begin{bmatrix} 1 \\ 18 \\ -2 \\ 6 \end{bmatrix}.$$

Step 3 (final interpolation):

$$\mathbf{h} = C\boldsymbol{\mu} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} & -1 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 18 \\ -2 \\ 6 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 7 \\ 6 \end{bmatrix}.$$

Direct verification:

$$(1 + 2x)(1 + 2x + 3x^2) = 1 + 4x + 7x^2 + 6x^3,$$

que coincide com $\mathbf{h} = [1, 4, 7, 6]^T$.

APPENDIX D – ADDITIONAL RESULTS

This appendix collects supplementary analyses that support the results in Chapter 7. In particular, it documents the derivation of the fast-convolution analytical model used in the main chapter.

D.1 Derivation of the fast-convolution analytical model

We derive a cycle model that decomposes the latency per output feature map into channel, transient, and steady-state contributions. Let M be the linear input-tile size and N the linear output-tile size (so that each tile produces an $N \times N$ output block). Let B_{width} denote the effective bus width (samples per cycle) used to stream tiles. Let C_{in} and C_{out} denote the numbers of input and output channels. Let F_{ifmap} be the linear size of the input feature map, and let $F_{\text{map}} = F_{\text{ifmap}} - 2$ for stride-1 convolution with $L = 3$. Let K denote the linear size of the transformed kernel (so that K^2 elementwise products are required per tile), let M_{AC} be the number of MAC units, and let L_{mem} be the effective memory latency.

Assuming F_{map} is aligned with the tile size N (i.e., F_{map}/N tiles per output dimension), the total cycle count to produce the full output feature map is modeled as the number of input–output channel pairs multiplied by the sum of three phase-specific contributions. We adopt a continuous approximation: ratios such as F_{map}/N and M^2/B_{width} are treated as real-valued factors, even though the RTL implementations use integer arithmetic with ceiling operations to account for partial tiles and non-divisible bus widths. This keeps the analytical expressions tractable while preserving the dominant scaling trends observed in cycle-accurate simulations:

$$f_{\text{fast}} = C_{\text{in}} C_{\text{out}} \Big(f_{\text{channel}} + f_{\text{transient}} + f_{\text{steady}} \Big). \tag{D.1}$$

This decomposition implicitly assumes that each input–output channel pair processes its own copy of the input tiles, with no inter-output-channel reuse of input data. It is therefore a pessimistic model that treats the input traffic as scaling linearly with C_{out}.

Channel term.

The term f_{channel} captures the cost of loading the transformed-kernel tile for each input–output channel pair. Since this transfer is memory-bound, it scales with the effective latency and with the number of elements in the transformed kernel:

$$f_{\text{channel}}(K, L_{\text{mem}}) = K^2 \, L_{\text{mem}}. \tag{D.2}$$

Transient term.

The transient term $f_{\text{transient}}$ models the initial filling of the input buffers before steady streaming is possible. Along one output dimension there are F_{map}/N windows, and each window consumes an $M \times M$ input tile. Assuming each input sample is transferred once through the bus, the transient cost is

$$f_{\text{transient}}(M, N, B_{\text{width}}, L_{\text{mem}}, F_{\text{map}}) = \frac{M^2}{B_{\text{width}}} \frac{F_{\text{map}}}{N} L_{\text{mem}}. \tag{D.3}$$

Steady-state term.

The steady-state term f_{steady} describes the streaming regime, in which consecutive windows advance regularly and reuse a portion of the buffered input data. In the analytical model used here, we approximate this regime as memory-bound: each window advance is dominated by the traffic required to read and write the output tile, scaled by the effective memory latency.

To model this data-movement bound, we upper-bound the per-window bus traffic by a term proportional to $2N^2 + 1$ samples, scaled by the effective memory latency and amortized by the bus width B_{width}. Intuitively, N^2 is the number of samples in a single $N \times N$ output tile that must be communicated with the write-side memory. The factor of 2 captures the common read–modify–write pattern of output tiles during accumulation (i.e., reading a tile or partial sums and writing back the updated values). If the architecture accumulates the entire output tile on chip and writes it only once, the write-side traffic approaches N^2 rather than $2N^2$. The expression used here is a conservative bound that matches read–modify–write implementations. Finally, the additional $+1$ cycle accounts for an extra hand-off cycle required to fetch the first batch of convolution results from the datapath into the memory interface.

A natural alternative would be to bound steady-state traffic using the IFMAP side, since advancing a $M \times M$ input tile by an N-wide step introduces up to MN new input samples (plus a similar hand-off term). In the fast-convolution family considered here, however, M is typically linked to N by $M = N + L - 1$, with small fixed kernels (most commonly $L = 3$). In this common case, $M = N + 2$, and the per-step IFMAP increment is

$$MN + 1 = (N + 2)N + 1 = N^2 + 2N + 1,$$

whereas the OFMAP-oriented bound is $2N^2 + 1$. The difference is

$$(2N^2 + 1) - (MN + 1) = (2N^2 + 1) - (N^2 + 2N + 1) = N^2 - 2N = N(N - 2).$$

Thus, for integer $N \geq 0$, the OFMAP bound is (i) strictly larger for $N \geq 3$, (ii) equal at $N = 2$, and (iii) smaller only for the degenerate $N = 1$ case (which is not a practically relevant

output-tile size for Winograd-style CNN accelerators). Therefore, for the tile sizes used in practice ($N \geq 2$), using $2N^2+1$ either matches or conservatively upper-bounds the IFMAP increment, while also directly reflecting the mandatory write-side communication of produced outputs (and, when applicable, output accumulation).

With (F_{map}/N) windows per dimension, there are $(F_{\text{map}}/N)^2$ window advances in steady streaming along the 2D traversal. Each window advance is bounded by the slower of two mechanisms: the memory-traffic term and the internal pipeline depth. This yields

$$f_{\text{steady}}(M, N, B_{\text{width}}, L_{\text{mem}}, F_{\text{ifmap}}) = \max\left(\frac{(2N^2+1)}{B_{\text{width}}} L_{\text{mem}}, \frac{K^2}{M_{\text{AC}}} + 2\right)\left(\frac{F_{\text{ifmap}}-2}{N}\right)^2, \quad \text{(D.4)}$$

where the factor $(F_{\text{ifmap}}-2)$ accounts for the output-side traversal implied by a stride-1, 3×3 kernel.

D.2 Instantiation for the IF 2D configuration

For the IF 2D configuration used in this work, we instantiate Equation (D.4) with $N=3$ and obtain

$$\tilde{f}_{\text{IF}}(F_{\text{ifmap}}) = C_{\text{in}}C_{\text{out}} \max\left(\frac{19}{B_{\text{width}}} L_{\text{mem}}, \frac{6^2}{M_{\text{AC}}} + 2\right)\left(\frac{F_{\text{ifmap}}-2}{3}\right)^2, \quad \text{(D.5)}$$

with transform parameters fixed to $M=5$, $N=3$, and transformed-kernel size $K=6$ (36 elementwise products per tile).

In the specific instance used for plotting, we set $C_{\text{in}} = C_{\text{out}} = 1$, $B_{\text{width}} = 1$, and $L_{\text{mem}} = 1$. Under this memory-dominated setting, most curves overlap once $K^2/M_{\text{AC}}+2$ drops below the memory term; only the smallest MAC counts remain compute-bound. Figure D.1 therefore shows a largely collapsed family for $M_{\text{AC}} \in \{1,2,3,4,6,9,18,36\}$ alongside the naïve baseline, completing the cycle-based comparison between the baseline and the fast-convolution designs.

D.3 Specializations: minimal Winograd and Toom–Cook-3

Two additional specializations of Equation (D.4) used in this chapter are the classical Winograd minimal filter and the Toom–Cook-3 accelerator.

For the Winograd minimal filter with $M=4$, $N=2$, and transformed-kernel size $K=4$ (16 elementwise products per tile), the steady-state model becomes

$$\tilde{f}_{\text{MinWin}}(F_{\text{ifmap}}) = C_{\text{in}}C_{\text{out}} \max\left(\frac{9}{B_{\text{width}}} L_{\text{mem}}, \frac{4^2}{M_{\text{AC}}} + 2\right)\left(\frac{F_{\text{ifmap}}-2}{2}\right)^2, \quad \text{(D.6)}$$

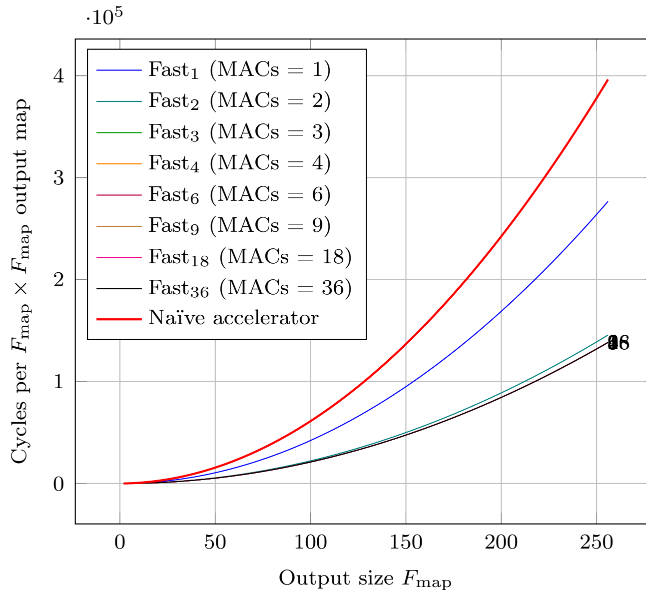


Figure D.1: Steady-state cycle models for the IF-based fast-convolution accelerator ($M = 5$, $N = 3$, $K = 6$) to produce an $F_{\text{map}} \times F_{\text{map}}$ output feature map, as a function of the output side F_{map}, for MAC counts $M_{\text{AC}} \in \{1, 2, 3, 4, 6, 9, 18, 36\}$, compared with the naïve-convolution baseline.

where $F_{\text{map}}/2$ corresponds to the number of 2×2 output tiles per dimension.

For a Toom–Cook-3 accelerator with $M = 5$, $N = 3$, and transformed-kernel size $K = 5$ (25 elementwise products per tile), the steady-state model is

$$\tilde{f}_{\text{TC3}}(F_{\text{ifmap}}) = C_{\text{in}} C_{\text{out}} \max\left(\frac{19}{B_{\text{width}}} L_{\text{mem}}, \frac{5^2}{M_{\text{AC}}} + 2\right) \left(\frac{F_{\text{ifmap}} - 2}{3}\right)^2, \tag{D.7}$$

which corresponds to 3×3 output tiles with a polynomially transformed 5×5 kernel.

Minimal Winograd (2D).

We instantiate Equation (D.6) for the classical minimal 2D configuration with $M = 4$, $N = 2$, $K = 4$, and plot $M_{\text{AC}} \in \{1, 2, 4, 8, 16\}$. Figure D.2 shows how the dominant term shifts from memory traffic to the datapath pipeline as M_{AC} increases.

Winograd with $K = 9$.

We next consider a Winograd configuration with $M = 6$, $N = 4$, filter length $L = 3$, and transformed-kernel size $K = 9$ (81 elementwise products per tile). This corresponds to a 2D scheme producing 4×4 output tiles with K^2 coefficients in the transformed kernel. In Equation (D.4) we therefore set $N = 4$ and plot $M_{\text{AC}} \in \{1, 3, 9, 27, 81\}$. Figure D.3 illustrates the transition between memory-limited and compute-limited regimes as the pipeline depth increases.

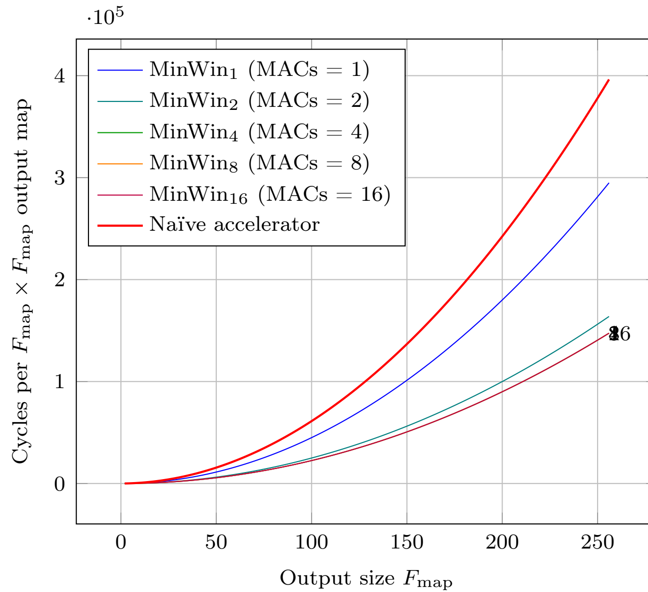


Figure D.2: Steady-state cycle models for a Winograd minimal-filter accelerator ($M = 4$, $N = 2$, $K = 4$) to produce an $F_{\text{map}} \times F_{\text{map}}$ output feature map, as a function of the output side F_{map}, for MAC counts $M_{\text{AC}} \in \{1, 2, 4, 8, 16\}$, compared with the naïve-convolution baseline.

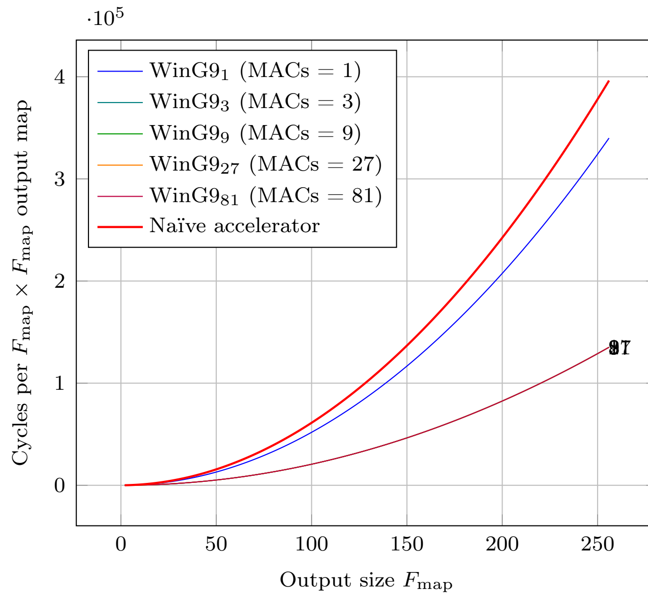


Figure D.3: Steady-state cycle models for a Winograd convolution with $M = 6$, $N = 4$, filter length $L = 3$, and transformed-kernel size $K = 9$ to produce an $F_{\text{map}} \times F_{\text{map}}$ output feature map, as a function of the output side F_{map}, for MAC counts $M_{\text{AC}} \in \{1, 3, 9, 27, 81\}$, compared with the naïve-convolution baseline.

Toom–Cook-3 (2D).

Finally, we apply Equation (D.7) to a Toom–Cook-3 2D accelerator with $M = 5$, $N = 3$, and $K = 5$, plotting $M_{\text{AC}} \in \{1, 5, 25\}$. Figure D.4 shows how the datapath term becomes dominant for deeper pipelines, while shallow pipelines remain memory-bound.

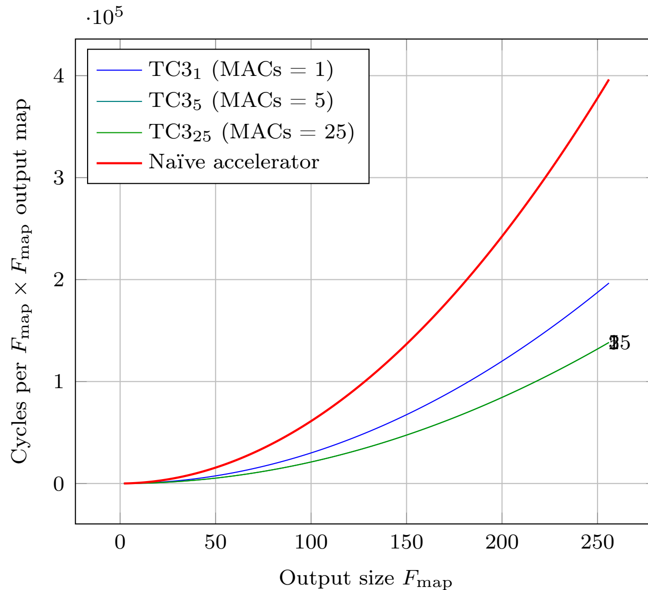


Figure D.4: Steady-state cycle models for a Toom–Cook-3 accelerator ($M = 5$, $N = 3$, $K = 5$) to produce an $F_{\text{map}} \times F_{\text{map}}$ output feature map, as a function of the output side F_{map}, for MAC counts $M_{\text{AC}} \in \{1, 5, 25\}$, compared with the naïve-convolution baseline.

D.4 Compute-only upper bound for the convolutional core

This section derives the compute-only upper bound used in Chapter 7. The analysis assumes instantaneous memory access: input samples and transformed coefficients are always available, and result writes incur no latency. Under these conditions, the cycle count depends only on the internal datapath capacity and on the scheduling of work in time.

Spatial tiling.

For stride-1 3×3 convolution, the output side is $F_{\text{map}} = F_{\text{ifmap}} - 2$. Partitioning the output plane into $N \times N$ tiles yields

$$T_{\text{1D}} = \left\lceil \frac{F_{\text{map}}}{N} \right\rceil, \qquad T = T_{\text{1D}}^2,$$

where T is the number of tiles processed per output feature map.

Compute-only model for fast schemes (IF/WM/TC3).

In fast schemes operating in the transformed domain (Winograd/Toom–Cook variants), the dominant per-tile cost is the Hadamard-style stage, which requires K^2 scalar products per $(C_{\text{in}}, C_{\text{out}})$ pair, where $K \times K$ is the transformed-tile size. With M_{AC} parallel multipliers,

the number of rounds needed to cover these products is

$$\text{state} = \left\lceil \frac{K^2}{M_{\text{AC}}} \right\rceil.$$

Adding a fixed pipeline overhead (here +2) to account for transform/reconstruction and control, the compute-only cost per tile is

$$C_{\text{tile}}^{\text{fast}} = \text{state} + 2.$$

Thus, the total compute-only cycles to process all tiles and channel pairs is

$$C_{\text{total}}^{\text{fast}} = C_{\text{in}}\, C_{\text{out}} \left\lceil \frac{F_{\text{map}}}{N} \right\rceil^2 \left(\left\lceil \frac{K^2}{M_{\text{AC}}} \right\rceil + 2 \right).$$

Point restriction ($M_{\text{AC}} \mid K^2$). To reflect exact microarchitectural scheduling (no fractional rounds and no partially idle cycles), we evaluate only configurations in which

$$M_{\text{AC}} \in \{d \in \mathbb{N} : d \mid K^2\},$$

i.e., only divisors of K^2. This makes K^2/M_{AC} an integer and represents a uniform partitioning of the transformed-domain product vector.

Compute-only model for the naïve convolution.

As a reference, the naïve baseline fixes the spatial tile size at $N = 3$. With $F_{\text{map}} = F_{\text{ifmap}} - 2$, the number of tiles is

$$T_{\text{1D}}^{\text{naive}} = \left\lceil \frac{F_{\text{map}}}{3} \right\rceil, \qquad T^{\text{naive}} = \left(T_{\text{1D}}^{\text{naive}}\right)^2.$$

In the adopted baseline, the per-tile cost is constant:

$$C_{\text{tile}}^{\text{naive}} = 27 + \text{overhead}.$$

Therefore, the compute-only total is

$$C_{\text{total}}^{\text{naive}} = C_{\text{in}}\, C_{\text{out}} \left\lceil \frac{F_{\text{map}}}{3} \right\rceil^2 (27 + \text{overhead}).$$

Interpreting the plot.

Figure 7.1 plots C_{total} against M_{AC} in the compute-only regime. The Naïve curve is horizontal because its per-tile cost is constant and independent of M_{AC}. The fast schemes (IF/WM/TC3) appear as discrete points because they are evaluated only at values of M_{AC} that divide K^2. As M_{AC} increases, total cycles decrease approximately with $1/M_{\text{AC}}$ while $\lceil K^2/M_{\text{AC}} \rceil$

shrinks; beyond a certain point, gains become marginal due to the fixed +2 overhead and the quantization imposed by the ceiling and the discrete divisor set. Because memory is assumed instantaneous, this saturation reflects internal datapath and pipeline limitations rather than communication bottlenecks, establishing the compute-bound upper performance limit for the convolutional core.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br