FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Streaming Engine for Dedicated Accelerators

Vasco Costa Vieira



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. João Paulo de Castro Canas Ferreira

July 31, 2026

© Vasco Costa Vieira, 2026

Streaming Engine for Dedicated Accelerators

Vasco Costa Vieira

Mestrado em Engenharia Eletrotécnica e de Computadores

Approved in oral examination by the committee:

President: Prof. Hélio Mendes de Sousa Mendonça
Examiner: Prof. Gustavo Ribeiro da Costa Alves
Supervisor: Prof. João Paulo de Castro Canas Ferreira

July 31, 2026

Resumo

As arquiteturas do tipo *Coarse-Grained Reconfigurable Array* (CGRA) oferecem uma elevada eficiência computacional, no entanto, o seu débito de processamento é frequentemente limitado por *stalls* de memória durante acessos de alta latência e padrões complexos de travessia de dados multidimensionais. Para resolver este *bottleneck*, esta tese apresenta uma avaliação arquitetural de um paradigma de desacoplamento entre memória e computação, orientado por um *Streaming Engine* (SE) configurável em hardware. A arquitetura proposta isola a orquestração do acesso aos dados do substrato computacional, transferindo as rotinas de geração de endereços, o acesso à memória e os esquemas de reutilização local de dados entre dois fluxos de entrada e um fluxo de saída, para o SE.

A metodologia de avaliação foi implementada através de uma linha de simulação RTL automatizada em Verilog HDL, comparando diretamente uma topologia de referência convencional com acoplamento à memória contra a topologia com desacoplamento da memória proposta. O desempenho foi avaliado através de varrimentos sistemáticos de parâmetros em quatro aplicações computacionais representativas: Filtragem FIR, Multiplicação de Matrizes, Convolução 2D e Convolução 3D.

Os resultados experimentais demonstram que adotar esta ideologia desacoplada permite reduzir eficazmente as paragens dos elementos de processamento, diminuindo significativamente o tempo de execução total e maximizando a utilização das unidades de cálculo. Por fim, esta investigação fornece orientações fundamentais sobre os compromissos dos substratos reconfiguráveis desacoplados da memória e estabelece direções essenciais para a integração futura com controladores de memória físicos e plataformas FPGA.

Abstract

Coarse-Grained Reconfigurable Array (CGRA) architectures provide high computational efficiency, yet their achievable throughput is frequently constrained by memory stalls during high-latency accesses and complex multi-dimensional data traversals. To address this bottleneck, this thesis presents an architectural evaluation of a memory-computation decoupled paradigm driven by a hardware-configurable Streaming Engine (SE). The proposed architecture isolates data access orchestration from the compute substrate by offloading address generation routines, memory access, and localized data reuse schemes across dual input streams and a single output stream to the SE.

The evaluation methodology was implemented using an automated RTL simulation pipeline in Verilog HDL, directly comparing a conventional memory-coupled baseline topology against the proposed memory-decoupled topology. Performance was evaluated through systematic parameter sweeps across four representative computational workloads: FIR Filtering, Matrix Multiplication, 2D Convolution, and 3D Convolution.

The experimental findings demonstrate that adopting this decoupled ideology can effectively reduce processing element stalls, significantly decreasing the total execution runtime and maximizing the compute unit utilization. Ultimately, this research provides foundational insights into the trade-offs of memory-decoupled reconfigurable substrates and establishes key directions for future integration with physical memory controllers and FPGA platforms.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals[1] to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

The specific Sustainable Development Goals mentioned have the following names:

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG	Target	Contribution	Performance Indicators and Metrics
7	7.3	Decoupling memory access and address generation shortens application execution runtimes, lowering the energy consumed by active hardware units per workload.	Application speedup (s), execution runtime reduction (%), and energy per computed workload.
	7.b	Offloading multi-dimensional indexing and local cache hierarchies to a dedicated streaming unit reduces total hardware area in CGRAs, minimizing static power consumption.	Silicon area savings (resource utilization in LUTs, FFs, DSPs) and static power reduction.
9	9.4	Sustaining high continuous data rates reduces memory stalls, allowing compute units to remain active for a higher percentage of the overall execution time.	Processing element active utilization rate (%) and sustained operational throughput (GOPS).

[1] https://sdgs.un.org/goals

Acknowledgements

This work is financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project AIMaCoV, with reference 2023.14899.PEX (https://doi.org/10.54499/2023.14899.PEX).

I would like to extend my deepest appreciation to my supervisor, Professor João Paulo de Castro Canas Ferreira, for his guidance, advice, and steady support throughout the course of this work. His insights and feedback were instrumental in steering this project to completion and helping me navigate the technical challenges along the way.

My thanks also go to all the professors and staff at FEUP who guided me across my academic journey. The knowledge, problem-solving skills, and foundation I gained under their instruction were essential in enabling me to design and implement this work.

To my classmates and colleagues, thank you for the shared moments, useful technical discussions, and encouragement throughout our studies.

Finally, I am infinitely thankful to my family and friends. Their constant belief in me, patience, and support have been my main foundation. I could not have reached this milestone without them by my side, and this work would not have been possible without their presence in my life.

Vasco Costa Vieira

"To find greatness, you must become obsessed with your own potential."

David Goggins

Contents

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Objectives . . . 2
- 1.4 Contributions . . . 3
- 1.5 Document Outline . . . 3

2 Background **5**
- 2.1 Coarse-Grained Reconfigurable Architectures . . . 5
  - 2.1.1 Overview and Applications . . . 5
  - 2.1.2 Classification . . . 6
  - 2.1.3 ADRES Architecture . . . 6
  - 2.1.4 Data Streaming . . . 7
- 2.2 Memory System . . . 8
  - 2.2.1 Memory Hierarchy . . . 8
  - 2.2.2 Locality Principles . . . 9
  - 2.2.3 Cache Mapping Schemes . . . 9

3 State of the Art **11**
- 3.1 Data Streaming Unit . . . 11
- 3.2 UVE Streaming Engine . . . 14
- 3.3 Discussion . . . 15

4 Designed Architectures **18**
- 4.1 Streaming Engine . . . 18
  - 4.1.1 Architecture Overview . . . 18
  - 4.1.2 Sub-Module Description . . . 20
- 4.2 Memory-Coupled Computing Engine . . . 50
  - 4.2.1 Architecture Overview . . . 50
  - 4.2.2 Sub-Module Description . . . 52
- 4.3 Memory-Decoupled Computing Engine . . . 60
  - 4.3.1 Architecture Overview . . . 60
  - 4.3.2 Sub-Module Description . . . 62
- 4.4 Memory Controller . . . 65
  - 4.4.1 Module Description . . . 65
- 4.5 Memory Block . . . 69
  - 4.5.1 Module Description . . . 69

5 Benchmarks and Evaluation Metrics **72**
5.1 Benchmarks . . . 72
5.1.1 FIR Filter . . . 72
5.1.2 Matrix Multiplication . . . 73
5.1.3 2D Convolution . . . 74
5.1.4 3D Convolution . . . 75
5.2 Performance Evaluation Metrics . . . 76

6 Experimental Setup and Methodology **80**
6.1 Evaluated Architectures . . . 80
6.1.1 Baseline System Architecture . . . 80
6.1.2 Proposed System Architecture . . . 81
6.2 Experimental Setup . . . 81
6.3 Methodology . . . 82
6.4 Problem Formulation . . . 83
6.4.1 FIR Filter . . . 83
6.4.2 Matrix Multiplication . . . 84
6.4.3 2D Convolution . . . 84
6.4.4 3D Convolution . . . 84

7 Results and Discussion **86**
7.1 FIR Filter . . . 87
7.1.1 Problem Size Sweep . . . 87
7.1.2 Operating Frequency Sweep . . . 97
7.1.3 Memory Latency Sweep . . . 109
7.2 Matrix Multiplication . . . 120
7.2.1 Problem Size Sweep . . . 120
7.2.2 Operating Frequency Sweep . . . 131
7.2.3 Memory Latency Sweep . . . 143
7.3 2D Convolution . . . 154
7.3.1 Problem Size Sweep . . . 154
7.3.2 Operating Frequency Sweep . . . 163
7.3.3 Memory Latency Sweep . . . 175
7.4 3D Convolution . . . 186
7.4.1 Problem Size Sweep . . . 186
7.4.2 Operating Frequency Sweep . . . 193
7.4.3 Memory Latency Sweep . . . 206
7.5 Discussion . . . 217
7.5.1 FIR Filter . . . 217
7.5.2 Matrix Multiplication . . . 218
7.5.3 2D Convolution . . . 218
7.5.4 3D Convolution . . . 219
7.6 Summary . . . 220

8 Conclusion **222**
8.1 Achievements and Limitations . . . 222
8.2 Summary . . . 223
8.3 Future Work . . . 223

References **224**

List of Figures

2.1 ADRES - Architecture Template Overview (Adapted from [1]). 6

3.1 T-CGRA Accelerator - High-Level Overview (Adapted from [12]). 11
3.2 DSU - High-Level Architecture Overview (Adapted from [12]). 12
3.3 DSU - RE Architecture Overview (Adapted from [12]). 12
3.4 DSU - WE Architecture Overview (Adapted from [12]). 13
3.5 UVE SE - Integration Overview Within a CPU Core Architecture (Adapted from [13]). 14
3.6 UVE SE - Architecture High-Level Overview (Adapted from [13]). 15

4.1 SE - Architecture Overview. 19
4.2 SE - SCH Block Diagram. 21
4.3 SCH - FSM Diagram. 26
4.4 SCH - IS0 Load Branch Diagram. 27
4.5 SCH - IS1 Load Branch Diagram. 28
4.6 SCH - Write Branch Diagram. 28
4.7 SE - LDR Block Diagram. 30
4.8 LDR - FSM Diagram. 33
4.9 SE - RE Block Diagram. 35
4.10 RE - FSM Diagram. 37
4.11 SE - WE Block Diagram. 39
4.12 WE - FSM Diagram. 41
4.13 SE - AGU Block Diagram. 42
4.14 SE - SC Block Diagram. 44
4.15 SE - TT Block Diagram. 46
4.16 SE - SB Block Diagram. 48
4.17 MC-CE - Architecture Overview. 51
4.18 MC-CE - SCH Block Diagram. 53
4.19 SCH - FSM Diagram. 57
4.20 PE - Architecture Overview. 60
4.21 MD-CE - Architecture Overview. 61
4.22 MD-CE - FSM Block Diagram. 62
4.23 FSM - FSM Diagram. 64
4.24 MC Block Diagram. 66
4.25 MC - FSM Diagram. 68
4.26 MB Block Diagram. 70

6.1 Baseline Architecture - System Block Diagram. 80
6.2 Proposed Architecture - System Block Diagram. 81

7.1 Baseline Architecture - Simulation Time across Problem Size N. 88
7.2 Proposed Architecture - Simulation Time across Problem Size N. 89
7.3 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Problem Size N. 90
7.4 Baseline Architecture - Memory Communication Time across Problem Size N. . 91
7.5 Proposed Architecture - Memory Access Factor (MAF) across Problem Size N. . 92
7.6 Proposed Architecture - Memory Communication Time across Problem Size N. . 93
7.7 Proposed Architecture - SE to CE Throughput across Problem Size N. 94
7.8 Proposed Architecture - SE to CE Bandwidth Utilization across Problem Size N. 94
7.9 Proposed Architecture - CE Throughput across Problem Size N. 95
7.10 Proposed Architecture - CE Bandwidth Utilization across Problem Size N. 96
7.11 Proposed Architecture - CE Compute Utilization across Problem Size N. 96
7.12 Baseline Architecture - Simulation Time across Operating Frequency f. 97
7.13 Proposed Architecture - Simulation Time across Operating Frequency f. 98
7.14 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Operating Frequency f. 99
7.15 Baseline Architecture - Memory Communication Time across Operating Frequency f. 100
7.16 Proposed Architecture - Memory Communication Time across Operating Frequency f. 101
7.17 Baseline Architecture - MC to CE Maximum Theoretical Throughput across Operating Frequency f. 102
7.18 Baseline Architecture - MC to CE Throughput across Operating Frequency f. . . 102
7.19 Baseline Architecture - MC to CE Bandwidth Utilization across Operating Frequency f. 103
7.20 Proposed Architecture - SE to CE Maximum Theoretical Throughput across Operating Frequency f. 104
7.21 Proposed Architecture - SE to CE Throughput across Operating Frequency f. . . 104
7.22 Proposed Architecture - SE to CE Bandwidth Utilization across Operating Frequency f. 105
7.23 Baseline Architecture - CE Maximum Theoretical Throughput across Operating Frequency f. 106
7.24 Baseline Architecture - CE Throughput across Operating Frequency f. 106
7.25 Baseline Architecture - CE Bandwidth Utilization across Operating Frequency f. 107
7.26 Baseline Architecture - CE Compute Utilization across Operating Frequency f. . 107
7.27 Proposed Architecture - CE Maximum Theoretical Throughput across Operating Frequency f. 108
7.28 Proposed Architecture - CE Throughput across Operating Frequency f. 108
7.29 Proposed Architecture - CE Bandwidth Utilization across Operating Frequency f. 109
7.30 Proposed Architecture - CE Compute Utilization across Operating Frequency f. . 109
7.31 Baseline Architecture - Simulation Time across Memory Latency l. 110
7.32 Proposed Architecture - Simulation Time across Memory Latency l. 111
7.33 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Memory Latency l. 112
7.34 Baseline Architecture - Memory Communication Time across Memory Latency l. 113
7.35 Proposed Architecture - Memory Communication Time across Memory Latency l. 114
7.36 Baseline Architecture - MC to CE Throughput across Memory Latency l. 115
7.37 Baseline Architecture - MC to CE Bandwidth Utilization across Memory Latency l. 115

7.38 Proposed Architecture - SE to CE Throughput across Memory Latency l. 116
7.39 Proposed Architecture - SE to CE Bandwidth Utilization across Memory Latency l. 116
7.40 Baseline Architecture - CE Throughput across Memory Latency l. 117
7.41 Baseline Architecture - CE Bandwidth Utilization across Memory Latency l. . . . 117
7.42 Baseline Architecture - CE Compute Utilization across Memory Latency l. . . . 118
7.43 Proposed Architecture - CE Throughput across Memory Latency l. 118
7.44 Proposed Architecture - CE Bandwidth Utilization across Memory Latency l. . . 119
7.45 Proposed Architecture - CE Compute Utilization across Memory Latency l. . . . 119
7.46 Baseline Architecture - Simulation Time across Problem Size N. 120
7.47 Proposed Architecture - Simulation Time across Problem Size N. 122
7.48 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Problem Size N. 122
7.49 Baseline Architecture - Memory Access Factor (MAF) across Problem Size N. . 123
7.50 Baseline Architecture - Memory Communication Time across Problem Size N. . 124
7.51 Proposed Architecture - Memory Access Factor (MAF) across Problem Size N. . 124
7.52 Proposed Architecture - Memory Communication Time across Problem Size N. . 125
7.53 Baseline Architecture - MC to CE Throughput across Problem Size N. 126
7.54 Baseline Architecture - MC to CE Bandwidth Utilization across Problem Size N. 126
7.55 Proposed Architecture - SE to CE Throughput across Problem Size N. 127
7.56 Proposed Architecture - SE to CE Bandwidth Utilization across Problem Size N. 127
7.57 Baseline Architecture - CE Throughput across Problem Size N. 128
7.58 Baseline Architecture - CE Bandwidth Utilization across Problem Size N. 129
7.59 Baseline Architecture - CE Compute Utilization across Problem Size N. 129
7.60 Proposed Architecture - CE Throughput across Problem Size N. 130
7.61 Proposed Architecture - CE Bandwidth Utilization across Problem Size N. 130
7.62 Proposed Architecture - CE Compute Utilization across Problem Size N. 131
7.63 Baseline Architecture - Simulation Time across Operating Frequency f. 132
7.64 Proposed Architecture - Simulation Time across Operating Frequency f. 132
7.65 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Operating Frequency f. 133
7.66 Baseline Architecture - Memory Communication Time across Operating Frequency f. 134
7.67 Proposed Architecture - Memory Communication Time across Operating Frequency f. 135
7.68 Baseline Architecture - MC to CE Maximum Theoretical Throughput across Operating Frequency f. 136
7.69 Baseline Architecture - MC to CE Throughput across Operating Frequency f. . . 136
7.70 Baseline Architecture - MC to CE Bandwidth Utilization across Operating Frequency f. 137
7.71 Proposed Architecture - SE to CE Maximum Theoretical Throughput across Operating Frequency f. 138
7.72 Proposed Architecture - SE to CE Throughput across Operating Frequency f. . . 138
7.73 Proposed Architecture - SE to CE Bandwidth Utilization across Operating Frequency f. 139
7.74 Baseline Architecture - CE Maximum Theoretical Throughput across Operating Frequency f. 140
7.75 Baseline Architecture - CE Throughput across Operating Frequency f. 140
7.76 Baseline Architecture - CE Bandwidth Utilization across Operating Frequency f. 141

7.77 Baseline Architecture - CE Compute Utilization across Operating Frequency f. . 141
7.78 Proposed Architecture - CE Maximum Theoretical Throughput across Operating Frequency f. 142
7.79 Proposed Architecture - CE Throughput across Operating Frequency f. 142
7.80 Proposed Architecture - CE Bandwidth Utilization across Operating Frequency f. 143
7.81 Proposed Architecture - CE Compute Utilization across Operating Frequency f. . 143
7.82 Baseline Architecture - Simulation Time across Memory Latency l. 144
7.83 Proposed Architecture - Simulation Time across Memory Latency l. 145
7.84 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Memory Latency l. 146
7.85 Baseline Architecture - Memory Communication Time across Memory Latency l. 147
7.86 Proposed Architecture - Memory Communication Time across Memory Latency l. 148
7.87 Baseline Architecture - MC to CE Throughput across Memory Latency l. 149
7.88 Baseline Architecture - MC to CE Bandwidth Utilization across Memory Latency l. 149
7.89 Proposed Architecture - SE to CE Throughput across Memory Latency l. 150
7.90 Proposed Architecture - SE to CE Bandwidth Utilization across Memory Latency l. 150
7.91 Baseline Architecture - CE Throughput across Memory Latency l. 151
7.92 Baseline Architecture - CE Bandwidth Utilization across Memory Latency l. . . . 151
7.93 Baseline Architecture - CE Compute Utilization across Memory Latency l. . . . 152
7.94 Proposed Architecture - CE Throughput across Memory Latency l. 152
7.95 Proposed Architecture - CE Bandwidth Utilization across Memory Latency l. . . 153
7.96 Proposed Architecture - CE Compute Utilization across Memory Latency l. . . . 153
7.97 Baseline Architecture - Simulation Time across Problem Size N. 154
7.98 Proposed Architecture - Simulation Time across Problem Size N. 155
7.99 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Problem Size N. 156
7.100 Baseline Architecture - Memory Communication Time across Problem Size N. . 157
7.101 Proposed Architecture - Memory Access Factor (MAF) across Problem Size N. . 158
7.102 Proposed Architecture - Memory Communication Time across Problem Size N. . 159
7.103 Proposed Architecture - SE to CE Throughput across Problem Size N. 160
7.104 Proposed Architecture - SE to CE Bandwidth Utilization across Problem Size N. 160
7.105 Proposed Architecture - CE Throughput across Problem Size N. 161
7.106 Proposed Architecture - CE Bandwidth Utilization across Problem Size N. 162
7.107 Proposed Architecture - CE Compute Utilization across Problem Size N. 162
7.108 Baseline Architecture - Simulation Time across Operating Frequency f. 163
7.109 Proposed Architecture - Simulation Time across Operating Frequency f. 164
7.110 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Operating Frequency f. 165
7.111 Baseline Architecture - Memory Communication Time across Operating Frequency f. 166
7.112 Proposed Architecture - Memory Communication Time across Operating Frequency f. 167
7.113 Baseline Architecture - MC to CE Maximum Theoretical Throughput across Operating Frequency f. 168
7.114 Baseline Architecture - MC to CE Throughput across Operating Frequency f. . . 168
7.115 Baseline Architecture - MC to CE Bandwidth Utilization across Operating Frequency f. 169

7.116 Proposed Architecture - SE to CE Maximum Theoretical Throughput across Operating Frequency f. 170
7.117 Proposed Architecture - SE to CE Throughput across Operating Frequency f. . . 170
7.118 Proposed Architecture - SE to CE Bandwidth Utilization across Operating Frequency f. 171
7.119 Baseline Architecture - CE Maximum Theoretical Throughput across Operating Frequency f. 172
7.120 Baseline Architecture - CE Throughput across Operating Frequency f. 172
7.121 Baseline Architecture - CE Bandwidth Utilization across Operating Frequency f. 173
7.122 Baseline Architecture - CE Compute Utilization across Operating Frequency f. . 173
7.123 Proposed Architecture - CE Maximum Theoretical Throughput across Operating Frequency f. 174
7.124 Proposed Architecture - CE Throughput across Operating Frequency f. 174
7.125 Proposed Architecture - CE Bandwidth Utilization across Operating Frequency f. 175
7.126 Proposed Architecture - CE Compute Utilization across Operating Frequency f. . 175
7.127 Baseline Architecture - Simulation Time across Memory Latency l. 176
7.128 Proposed Architecture - Simulation Time across Memory Latency l. 177
7.129 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Memory Latency l. 178
7.130 Baseline Architecture - Memory Communication Time across Memory Latency l. 179
7.131 Proposed Architecture - Memory Communication Time across Memory Latency l. 180
7.132 Baseline Architecture - MC to CE Throughput across Memory Latency l. 181
7.133 Baseline Architecture - MC to CE Bandwidth Utilization across Memory Latency l. 181
7.134 Proposed Architecture - SE to CE Throughput across Memory Latency l. 182
7.135 Proposed Architecture - SE to CE Bandwidth Utilization across Memory Latency l. 182
7.136 Baseline Architecture - CE Throughput across Memory Latency l. 183
7.137 Baseline Architecture - CE Bandwidth Utilization across Memory Latency l. . . . 183
7.138 Baseline Architecture - CE Compute Utilization across Memory Latency l. . . . 184
7.139 Proposed Architecture - CE Throughput across Memory Latency l. 184
7.140 Proposed Architecture - CE Bandwidth Utilization across Memory Latency l. . . 185
7.141 Proposed Architecture - CE Compute Utilization across Memory Latency l. . . . 185
7.142 Baseline Architecture - Simulation Time across Problem Size N. 186
7.143 Proposed Architecture - Simulation Time across Problem Size N. 187
7.144 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Problem Size N. 188
7.145 Baseline Architecture - Memory Communication Time across Problem Size N. . 189
7.146 Proposed Architecture - Memory Access Factor (MAF) across Problem Size N. . 189
7.147 Proposed Architecture - Memory Communication Time across Problem Size N. . 190
7.148 Proposed Architecture - SE to CE Throughput across Problem Size N. 191
7.149 Proposed Architecture - SE to CE Bandwidth Utilization across Problem Size N. 191
7.150 Proposed Architecture - CE Throughput across Problem Size N. 192
7.151 Proposed Architecture - CE Bandwidth Utilization across Problem Size N. 192
7.152 Proposed Architecture - CE Compute Utilization across Problem Size N. 193
7.153 Baseline Architecture - Simulation Time across Operating Frequency f. 194
7.154 Proposed Architecture - Simulation Time across Operating Frequency f. 195
7.155 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Operating Frequency f. 196

7.156 Baseline Architecture - Memory Communication Time across Operating Frequency f. 197
7.157 Proposed Architecture - Memory Communication Time across Operating Frequency f. 198
7.158 Baseline Architecture - MC to CE Maximum Theoretical Throughput across Operating Frequency f. 198
7.159 Baseline Architecture - MC to CE Throughput across Operating Frequency f. . . 199
7.160 Baseline Architecture - MC to CE Bandwidth Utilization across Operating Frequency f. 199
7.161 Baseline Architecture - CE Maximum Theoretical Throughput across Operating Frequency f. 200
7.162 Baseline Architecture - CE Throughput across Operating Frequency f. 201
7.163 Baseline Architecture - CE Bandwidth Utilization across Operating Frequency f. 201
7.164 Baseline Architecture - CE Compute Utilization across Operating Frequency f. . 202
7.165 Proposed Architecture - SE to CE Maximum Theoretical Throughput across Operating Frequency f. 202
7.166 Proposed Architecture - SE to CE Throughput across Operating Frequency f. . . 203
7.167 Proposed Architecture - SE to CE Bandwidth Utilization across Operating Frequency f. 203
7.168 Proposed Architecture - CE Maximum Theoretical Throughput across Operating Frequency f. 204
7.169 Proposed Architecture - CE Throughput across Operating Frequency f. 205
7.170 Proposed Architecture - CE Bandwidth Utilization across Operating Frequency f. 205
7.171 Proposed Architecture - CE Compute Utilization across Operating Frequency f. . 206
7.172 Baseline Architecture - Simulation Time across Memory Latency l. 207
7.173 Proposed Architecture - Simulation Time across Memory Latency l. 208
7.174 Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Memory Latency l. 209
7.175 Baseline Architecture - Memory Communication Time across Memory Latency l. 210
7.176 Proposed Architecture - Memory Communication Time across Memory Latency l. 211
7.177 Baseline Architecture - MC to CE Throughput across Memory Latency l. 212
7.178 Baseline Architecture - MC to CE Bandwidth Utilization across Memory Latency l. 212
7.179 Proposed Architecture - SE to CE Throughput across Memory Latency l. 213
7.180 Proposed Architecture - SE to CE Bandwidth Utilization across Memory Latency l. 213
7.181 Baseline Architecture - CE Throughput across Memory Latency l. 214
7.182 Baseline Architecture - CE Bandwidth Utilization across Memory Latency l. . . . 214
7.183 Baseline Architecture - CE Compute Utilization across Memory Latency l. . . . 215
7.184 Proposed Architecture - CE Throughput across Memory Latency l. 215
7.185 Proposed Architecture - CE Bandwidth Utilization across Memory Latency l. . . 216
7.186 Proposed Architecture - CE Compute Utilization across Memory Latency l. . . . 216

List of Tables

3.1 Architectural Comparison of Streaming Units. 16

List of Acronyms

ADRES	Architecture for Dynamically Reconfigurable Embedded Systems
AG	Address Generator
AGI	Address Generation Instruction
AGU	Address Generation Unit
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
CB	Circular Buffer
CE	Compute Engine
CGRA	Coarse-Grained Reconfigurable Array
CGRA-ME	Coarse-Grained Reconfigurable Array Modeling and Exploration
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
DC	Data Collector
DD	Data Distributor
DR	Data Receiver
DRAM	Dynamic Random-Access Memory
DRF	Distributed Register File
DS	Data Sender
DSP	Digital Signal Processing
DSU	Data Streaming Unit
FF	Flip-Flop
FIFO	First-In, First-Out
FIR	Finite Impulse Response
FP	Floating-Point
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GPP	General-Purpose Processor
HDL	Hardware Description Language

II	Initiation Interval
IS0	Input Stream 0
IS1	Input Stream 1
ISA	Instruction Set Architecture
LDR	Loader
LRF	Local Register File
LUT	Look Up Table
MAF	Memory Access Factor
MB	Memory Block
MC	Memory Controller
MC-CE	Memory-Coupled Compute Engine
MD-CE	Memory-Decoupled Compute Engine
ML	Machine Learning
MMU	Memory Management Unit
NSEW	North, South, East, and West
OS	Output Stream
PE	Processing Element
PPA	Power, Performance, and Area
RAM	Random-Access Memory
RE	Read Engine
RF	Register File
RG	Register
RTL	Register-Transfer Level
SB	Stream Buffer
SC	Stream Cache
SCH	Scheduler
SE	Streaming Engine
SIMD	Single Instruction, Multiple Data
SRAM	Static Random-Access Memory
SSD	Solid State Drive
T-CGRA	Tightly-Coupled CGRA
TLB	Translation Lookaside Buffer
TT	Tag Table
UVE SE	Unlimited Vector Extension Streaming Engine
VLIW	Very Long Instruction Word
WE	Write Engine
XCU	Extended Compute Unit

Chapter 1

Introduction

1.1 Context and Motivation

Modern computing workloads in Digital Signal Processing (DSP), Computer Vision (CV), and Machine Learning (ML) are increasingly bounded by data movement rather than raw arithmetic capability. As algorithms scale across higher dimensional spaces — from 1D time-series filtering to 2D spatial image processing and 3D volumetric tensor operations — the complexity of generating non-linear memory access addresses grows exponentially.

Coarse-Grained Reconfigurable Arrays (CGRAs) have emerged as highly efficient computing substrates, offering hardware-level parallelism with software-like flexibility. However, traditional CGRA designs often tightly couple memory address generation and data control logic directly within the Processing Elements (PEs) or compute fabric. This coupling forces compute units to spend significant operational cycles executing pointer arithmetic and waiting on memory latency stalls rather than performing active computations. To break this fundamental throughput bottleneck, decoupling memory access and address calculation from core computation logic is essential.

1.2 Problem Statement

Core Problem: Traditional CGRAs suffer from severe data starvation and low compute unit utilization when scaling to large datasets due to tightly coupled memory access logic operating under high memory latencies.

While CGRAs efficiently process small datasets fitting within small local Random-Access Memorys (RAMs), large-scale dataset processing forces them to fetch data from main memory. Because CGRAs operate at high clock frequencies, even short memory access latencies translate into a high number of wasted clock cycles. This architectural coupling creates two critical bottlenecks that this thesis addresses:

1. **Memory Latency and Data Starvation:** When CGRAs negotiate external memory requests dynamically, latency gaps starve the PEs of data. This drastically lowers active hardware resource utilization and extends overall application execution runtimes.

2. **Hardware Area Overhead:** Embedding memory address generation, multi-nested loop indexing, and memory request control logic inside the compute fabric inflates silicon area and resource utilization (Look Up Tables (LUTs), Flip-Flops (FFs), DSP blocks), leading to higher static leakage power.

To reduce these bottlenecks, this thesis presents an exploratory architectural approach: **decoupling memory access, address generation and data streaming from the computing fabric**. By moving memory orchestration into a specialized Streaming Engine (SE) with localized data reuse capabilities, the CGRA compute fabric can be simplified to execute computations exclusively, mitigating memory latency stalls and improving system throughput.

1.3 Objectives

The primary objective of this thesis is to model, implement in Verilog Hardware Description Language (HDL), and evaluate a hardware-configurable SE that autonomously streams multi-dimensional data arrays directly to a CGRA-like compute substrate based on programmed memory access patterns. Rather than producing a market-ready hardware prototype, this work serves as an exploratory architectural study to evaluate how memory decoupling and localized data reuse can transform future CGRA designs.

To achieve this main goal, the following specific objectives are established:

- **Design and Model a Configurable SE:** Develop a Verilog-based SE capable of:
  - Dynamically orchestrating dual input data streams into a compute substrate, as well as managing a single output stream originating from the compute substrate.
  - Enabling independent configuration of specific memory access patterns for each individual stream (input and output), regardless of the data structure dimension (1D, 2D, or 3D).
  - Promoting localized data reuse through caching and buffering mechanisms to improve compute substrate throughput while mitigating processing starvation and external memory accesses.
- **Develop CGRA-Based Compute Substrates:** Design two distinct Compute Engine (CE) variants to enable a direct memory-coupling versus memory-decoupling comparative study:
  - Memory-Coupled Compute Engine (MC-CE) architecture.
  - Memory-Decoupled Compute Engine (MD-CE) architecture driven by the SE.
- **Model the External Memory Subsystem:** Implement a dedicated Memory Controller (MC) and Memory Block (MB) to accurately model external memory access delays and bus communication dynamics.

- **Execute Workload Benchmarking:** Conduct comparative evaluation experiments across both complete system architectures, described in section 6.1 using a selected set of multi-dimensional benchmark workloads (Finite Impulse Response (FIR) Filter, Matrix Multiplication, 2D Convolution, and 3D Convolution).
- **Perform System Performance Characterization:** Analyze and characterize the key performance metrics of both systems, including execution runtimes, speedup, active hardware resource utilization, and operational throughput.

1.4 Contributions

The main contributions of this research are summarized as follows:

- **Register-Transfer Level (RTL) Design of a Configurable SE:** Implementation of a Verilog-based, hardware-configurable SE capable of dynamically driving dual input streams and a single output stream across multi-dimensional memory access patterns.
- **Comparative Case Study on Memory Decoupling:** A quantitative evaluation comparing memory-coupled and memory-decoupled CGRA system architectures under scaling dataset sizes, demonstrating that decoupling memory access significantly reduces application runtimes, mitigates memory stall latency, and increases compute unit utilization.
- **Architectural Insights for CGRA Optimization:** Demonstration that offloading memory request orchestration and address generation to a specialized streaming unit provides a clear path toward simplifying CGRA compute fabrics, enabling silicon area footprint reductions and lower power consumption in future reconfigurable architectures.

1.5 Document Outline

The remainder of the document is organized as follows:

- **Chapter 2 (Background):** Provides fundamental context on CGRAs, covering domain classification, an architectural example, and data streaming bottlenecks. It also reviews memory and cache organization concepts, focusing on locality principles and addressing schemes.
- **Chapter 3 (State of the Art):** Reviews related work in streaming hardware architectures.
- **Chapter 4 (Designed Architectures):** Details the microarchitectural implementation of the proposed SE, the MC-CE, the MD-CE, and the external memory subsystem, composed by the MC and the MB.

- **Chapter 5 (Benchmarks and Evaluation Metrics):** Formulates the computational workloads (FIR Filter, Matrix Multiplication, 2D Convolution, and 3D Convolution) and defines the quantitative metrics used to assess performance.
- **Chapter 6 (Experimental Setup and Methodology):** Outlines the environment setup for the evaluation of the developed system architectures, and the methodology adopted in order to achieve the performance results for each architecture.
- **Chapter 7 (Results and Discussion):** Presents empirical simulation results and provides an in-depth performance analysis, evaluating comparative speedup, memory access efficiency, and bandwidth utilization across the defined benchmark suite.
- **Chapter 8 (Conclusion):** Summarizes the core achievements, addresses system limitations, and outlines directions for future work.

Chapter 2

Background

This chapter establishes the theoretical and architectural foundations relative to this work. First, Section 2.1 explores Coarse-Grained Reconfigurable Architectures (CGRAs), covering their domain classification, the Architecture for Dynamically Reconfigurable Embedded Systems (ADRES) template that inspired the proposed Compute Engines (CEs), and the memory bottlenecks associated with data streaming. Next, Section 2.2 explores memory system fundamentals, covering multi-tiered memory hierarchies, the principles of program locality, and foundational cache mapping schemes. Together, these background concepts provide the essential context for the architectures presented in later chapters.

2.1 Coarse-Grained Reconfigurable Architectures

2.1.1 Overview and Applications

CGRAs represent a hybrid compute domain positioned between General-Purpose Processors (GPPs) and Application-Specific Integrated Circuits (ASICs). Unlike fine-grained reconfigurable fabrics like Field-Programmable Gate Arrays (FPGAs) that operate at the bit level, CGRAs utilize word-level functional units (typically 16-bit or 32-bit datapaths) connected through dynamic interconnect networks. This coarse granularity significantly reduces configuration memory overhead and interconnect latency while maintaining software programmability [1], [2], [3].

The primary objective of a CGRA is to maximize execution throughput and energy efficiency by offloading compute-intensive kernels, particularly nested loops and dataflow-heavy program regions. Execution typically alternates between two modes: a control-centric mode (e.g., Very Long Instruction Words (VLIWs) or scalar execution) for sequential control flow, and a data-centric mode where loop bodies are mapped spatially onto the array of PEs. By streaming data directly across PEs cycle-by-cycle, CGRAs exploit high instruction-level and loop-level parallelism without continuous main memory accesses. Consequently, CGRAs — and exploration frameworks like Coarse-Grained Reconfigurable Array Modeling and Exploration (CGRA-ME) 2.0 — are widely targeted toward data-intensive domains such as digital signal processing, real-time image processing, and embedded edge acceleration [1].

2.1.2 Classification

To systematically navigate the vast design space of reconfigurable computing, CGRAs are formally characterized by their reconfiguration granularity, operating spatially at the functional unit level or above and temporally at the region or loop-nest level or above [4]. Based on comprehensive architectural surveys, CGRAs can be classified across four primary fields:

- **Structure:** Evaluates the physical hardware composition, including the interconnect topology (such as 2D mesh grids, crossbars, or direct neighbor links), word-level datapath granularity (e.g., 16-bit or 32-bit), register file organization, and the local memory hierarchy (ranging from scratchpads and cell First-In, First-Outs (FIFOs) to shared caches) [4].
- **Control:** Defines how execution state and routing are managed — specifically whether operation scheduling, configuration context switching, network routing, and custom operation chaining are performed statically at compile-time or dynamically at run-time [4].
- **Integration:** Evaluates how the CGRA operates relative to a host processor, distinguishing between standalone architectures and tightly or loosely coupled accelerators, as well as whether physical resources are shared with the host [4].
- **Tooling:** Evaluates the available software tool support necessary to program the architecture, focusing primarily on compilers and place-and-route tools [4].

2.1.3 ADRES Architecture

A widely studied CGRA baseline is the ADRES. ADRES combines a VLIW execution engine with a 2D grid of reconfigurable PEs, enabling seamless transitions between scalar processing and spatial acceleration. Figure 2.1 illustrates the top-level array organization alongside the detailed internal datapath of an individual PE [1].

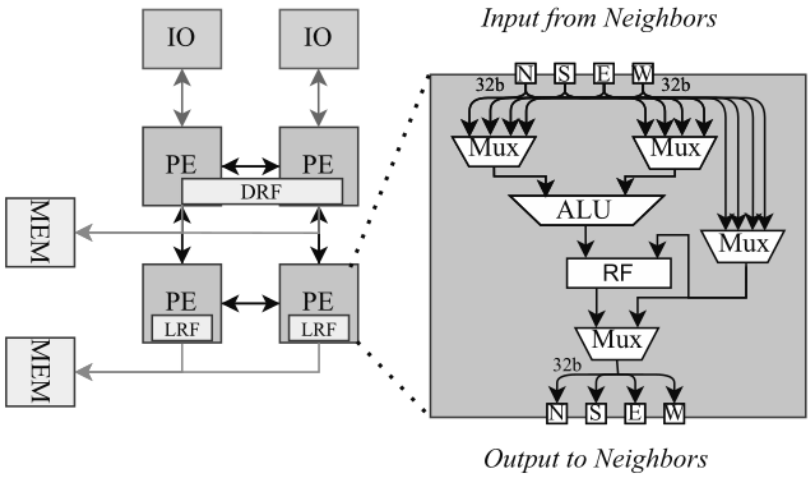


Figure 2.1: ADRES - Architecture Template Overview (Adapted from [1]).

At the array level, PEs are arranged in a grid interconnected by direct North, South, East, and West (NSEW) point-to-point routing channels [1]. Data management relies on a two-tier memory hierarchy:

- **Distributed Register File (DRF):** Positioned along the top row of PEs to serve as a shared storage interface for multi-issue VLIW instructions and global data distribution across neighboring units [1].
- **Local Register Files (LRFs):** Integrated within the individual PEs across the lower array rows to store intermediate values locally. This minimizes routing traffic across the global grid and reduces access latency [1].

While Figure 2.1 depicts a minimal 2×2 matrix as a simplified topological example, the ADRES template within the CGRA-ME framework is fully parametric. Depending on the target application's workload requirements, the fabric scales to larger configurations such as 4×4 or 6×6 arrays [1].

As detailed on the right side of Figure 2.1, each PE is built around a 32-bit compute datapath. Incoming data streams from cardinal neighbors (NSEW) are routed into two primary input multiplexers, which dynamically select operands for the Arithmetic Logic Unit (ALU) on every clock cycle. The computed ALU output can either be stored inside the internal Register File (RF) for multi-cycle local reuse or bypass the register to reduce transfer delay. Finally, an output multiplexer routes the calculated result to adjacent PEs via the cardinal boundary ports [1].

Ultimately, the study of the ADRES architecture served as an inspiration for developing the core structure of the CEs (MC-CE and MD-CE), which will be detailed later in this document.

2.1.4 Data Streaming

While CGRAs deliver massive computational throughput by mapping loop bodies spatially across arrays of PEs, their practical performance is strictly bounded by the memory subsystem's ability to supply operands continuously [5], [6]. As the breakdown of Dennard scaling and the slowdown of Moore's law push modern computing platforms toward a utilization wall, data movement and memory communication have emerged as the primary bottlenecks limiting system performance and energy efficiency [7]. Unlike general-purpose CPUs that exhibit irregular, instruction-driven, and scalar memory access patterns, dedicated hardware accelerators and spatial compute substrates generate distinct memory access streams characterized by four defining properties:

- **Sequentiality:** Memory access requests form long, contiguous, and highly predictable address streams [5].
- **Decoupled Load and Store Paths:** Read and write data movements operate as separate, non-interleaved streams rather than dynamically interleaved demand accesses [5].
- **Deterministic Request Timing:** Pipelined execution arrays issue memory transactions at constant cycle intervals during steady-state processing [5].

- **Page-Bounded Stream Sizes:** Memory access bursts typically occur in contiguous blocks bounded within physical memory page boundaries (e.g., up to 4 kB) [5].

Despite these highly predictable access patterns, traditional on-chip memory hierarchies and cache-coherence protocols remain optimized for general-purpose demand misses. Standard snooping and directory-based coherence protocols process memory transactions at the granularity of individual cache lines, requiring multi-phase handshakes, snoop broadcasts, and tag-array lookups for every line. When applied to continuous streaming workloads, this line-by-line protocol overhead induces severe tag-lookup congestion, bus arbitration delays, and excessive interconnect energy consumption [5].

Furthermore, in conventional CGRAs that lack dedicated memory streaming units, memory address calculations must be executed directly on the spatial array. This forces 20% to 80% of available PEs to execute Address Generation Instructions (AGIs) and multi-bank offset functions rather than actual arithmetic operations. This on-array address computation overhead severely inflates the loop Initiation Interval (II) and leads to significant PE starvation [6].

To overcome these memory bottlenecks, modern spatial architectures must shift towards an access-compute decoupled paradigm. By offloading address generation and memory access to autonomous, specialized hardware streaming logic, data can be streamed directly into the PE array, bypassing scalar instruction fetch and tag-lookup overheads while restoring the array's full capacity for pure arithmetic computation [5], [6].

2.2 Memory System

2.2.1 Memory Hierarchy

The core objective of a memory hierarchy is to provide a memory system with an average latency approaching that of its fastest component, a capacity bounded only by its largest storage tier, and a cost per bit aligned with its cheapest component. Because no single memory technology simultaneously offers sub-nanosecond access speed, infinite capacity, non-volatility, and low fabrication cost, modern architectures organize storage technologies into a multi-tiered hierarchy [8].

As illustrated across standard architectural models, memory structures are arranged in levels characterized by trade-offs in access speed, storage capacity, cost, and physical proximity to the processing core [8]:

- **Registers and On-Chip Caches (SRAM):** Positioned closest to the processor pipeline, Static Random-Access Memory (SRAM) arrays provide ultra-low access latencies and high bandwidth. However, due to high cell area overhead and leakage power constraints, these high-speed structures are severely restricted in total capacity [8].

- **Main Memory (DRAM):** Dynamic Random-Access Memory (DRAM) serves as the primary volatile operating store, balancing reasonable access times (tens of nanoseconds) with

high storage density and cost efficiency. Unlike SRAM, DRAM requires periodic refresh operations to retain charge in its capacitive storage cells [8].

- **Non-Volatile Secondary Storage (Flash/Disk):** Located at the base of the hierarchy, Solid State Drives (SSDs) and mechanical hard drives offer massive persistent capacity at minimal cost per bit, but introduce access latencies several orders of magnitude higher than DRAM [8], [9].

Managing these distinct operational tiers requires a holistic design approach, as physical circuit limits (e.g., interconnect delays and leakage current) continuously dictate system-level execution performance. By structuring memory in hierarchical levels, computer architectures hide the slow access times of secondary and main memory stores behind the high speed of small, hardware-managed cache structures [8].

2.2.2 Locality Principles

Memory hierarchies rely fundamentally on two forms of program locality to bridge the speed gap between high-speed processors and main memory [10]:

- **Temporal Locality:** The principle that memory addresses accessed recently have a high probability of being referenced again in the near future. Caches capitalize on temporal locality by retaining recently accessed data blocks for extended durations [10].
- **Spatial Locality:** The tendency of programs to access memory locations that are adjacent or physically close to recently referenced addresses close together in time. Caches exploit spatial locality by fetching multi-word contiguous blocks upon every cache miss [10].

For any fixed cache capacity, optimizing for both types of locality creates an inherent structural contradiction. Increasing the block size enhances spatial prefetching efficiency, but inversely reduces the total number of cache entries available. This reduction in entries shortens the lifetime of individual cache lines, resulting in higher conflict miss rates and thrashing for temporal data. Conversely, utilizing small block sizes maximizes the number of cache entries — extending the lifetime of data exhibiting temporal locality — at the expense of spatial prefetching benefits [10].

Conventional uniform cache designs enforce a single fixed block size across all accesses, failing to adapt to varying locality patterns across and within program executions. Advanced cache architectures resolve this trade-off by decoupling temporal and spatial mechanisms — for instance, using small-block direct-mapped arrays alongside large-block spatial buffers to selectively preserve high-temporal data while aggressively fetching contiguous spatial blocks [10].

2.2.3 Cache Mapping Schemes

Cache organization dictates how data blocks transferred from the main memory are positioned within the smaller, high-speed cache structure. When the processor initiates a memory request,

the hardware checks whether the requested address resides in the cache. To manage this lookup transparently and efficiently, memory systems rely on specific placement rules — known as mapping functions — to bridge main memory addresses with internal cache locations [11]. Modern architectures generally employ one of three foundational mapping techniques:

- **Direct Mapping:** In this straightforward scheme, every block from main memory is restricted to exactly one designated cache slot. The structure can be conceptualized as a table containing the actual cached data block, an address tag to uniquely identify the data, and a valid flag to signal if the entry contains active information. While direct mapping is hardware-efficient and fast due to its simple indexing, it suffers from severe contention when multiple active memory lines contend for the same cache line [11].
- **Fully Associative Mapping:** To eliminate the placement rigidities of direct mapping, fully associative cache structures permit any main memory block to be stored in any arbitrarily chosen location within the cache. Because a block has no predetermined index, the lookup process requires comparing the address tag across all cache entries simultaneously. While this scheme optimizes hit rates by maximizing storage flexibility, it incurs significant hardware overhead and lookup complexity [11].
- **Set-Associative Mapping:** Serving as an optimal balance between the direct and fully associative paradigms, set-associative mapping partitions the cache into sets of N lines (referred to as N-way set associativity). A given memory address is mapped to a specific set based on its index bits; however, within that assigned set, the block can occupy any of the N available slots. This flexible approach reduces conflict misses compared to direct mapping while constraining hardware lookup costs relative to fully associative designs [11].

In addition to standard instruction and data mapping, computer architectures often deploy specialized caches to streamline system operations. These include shared Level 3 (L3) caches designed across processor cores, dedicated Translation Lookaside Buffers (TLBs) for rapid virtual-to-physical address translation, and secondary hardware or flash-based disk caches aimed at offloading frequent storage I/O operations [11].

Chapter 3

State of the Art

In this chapter, we review the state-of-the-art by looking into two data streaming architectures: the Data Streaming Unit (DSU) and the Unlimited Vector Extension Streaming Engine (UVE SE). Following the individual overviews of the DSU and UVE SE architectures, we present a comparative discussion that highlights their structural differences, memory coupling models, and multi-dimensional indexing capabilities relative to the SE proposed in this work.

3.1 Data Streaming Unit

The Tightly-Coupled CGRA (T-CGRA) proposed by Abbaszadeh [12] comprises the DSU that acts as a hardware streaming bridge between the memory system and the spatial CGRA compute substrate. However, unlike streaming architectures that fully decouple address generation and memory traversal responsibilities from the compute substrate (like the developed SE), the DSU maintains tighter coupling with the processing array. While it streams data directly between system memory and the input/output FIFOs of individual CGRA cells to bypass traditional local buffers, address tracking and control often remain closely bound to the compute execution model [12].

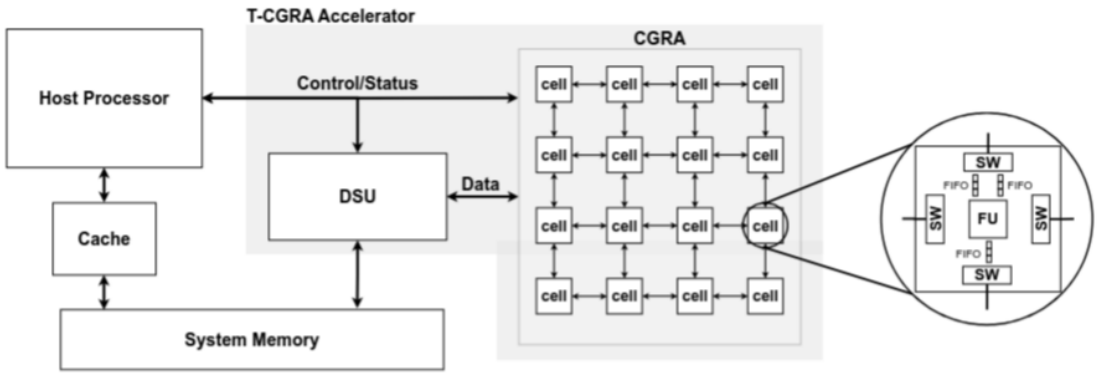


Figure 3.1: T-CGRA Accelerator - High-Level Overview (Adapted from [12]).

As shown in Figure 3.2, the DSU streams data at cache line granularity directly to and from the FIFOs of individual CGRA cells. To manage diverse access patterns without bottlenecking a

single central pipeline, the DSU structures its data paths into two primary functional engines: a **Read Engine (RE)** and a **Write Engine (WE)**, coordinated via a **Memory Arbiter**, as depicted below [12].

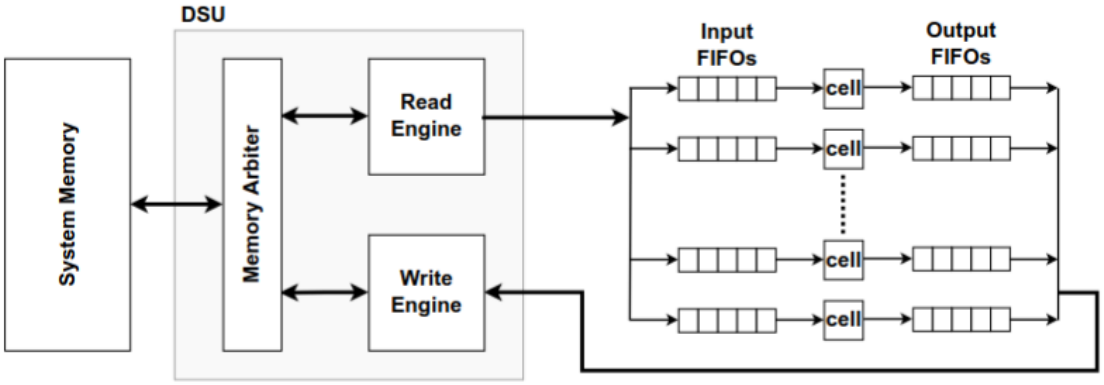


Figure 3.2: DSU - High-Level Architecture Overview (Adapted from [12]).

At the microarchitectural level, both the RE and WE house independent, hardware-configurable streaming slices that process access patterns in parallel using decoupled internal FIFOs, round-robin arbiters, and distribution networks [12]:

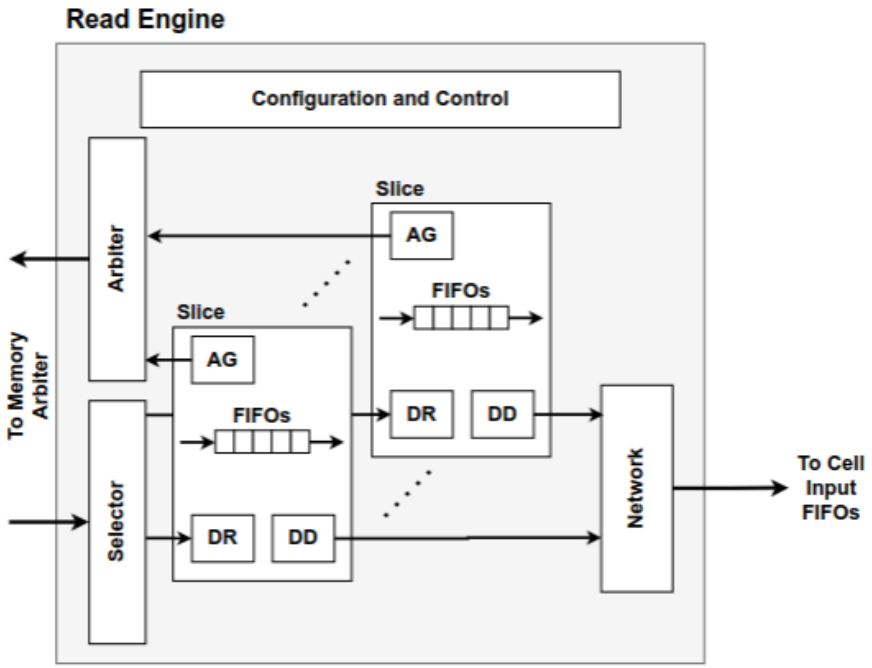


Figure 3.3: DSU - RE Architecture Overview (Adapted from [12]).

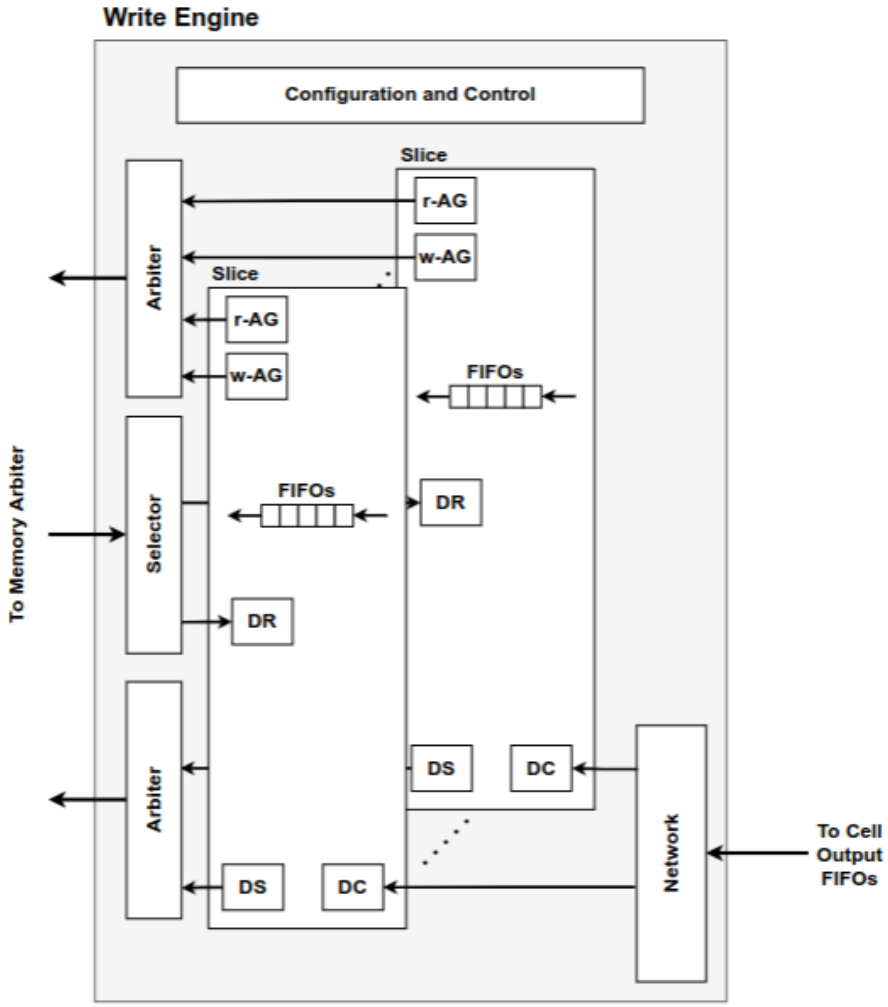


Figure 3.4: DSU - WE Architecture Overview (Adapted from [12]).

- **RE (Figure 3.3):** Houses multiple streaming slices dedicated to read access patterns. Each slice consists of an Address Generator (AG), Data Receiver (DR), and Data Distributor (DD) connected by internal FIFOs. The AG requests cache lines from memory, which are received by the DR and then forwarded to the DD, which routes the word elements through an output network to the target CGRA cells' input FIFOs [12].
- **WE (Figure 3.4):** Houses multiple streaming slices dedicated to write access patterns. Each slice contains a Read AG (r-AG), Write AG (w-AG), DR, Data Collector (DC), and Data Sender (DS). Whenever a write operation is being performed, the r-AG fetches the target cache line, which is then received by the DR and passed onto the DC, which collects the necessary words to update in that line, coming from CGRA store cells via the input network, and sends the updated cache lines back to memory via the DS [12].

Through this slice-based organization, the DSU natively supports a variety of memory access patterns, including single-strided, multiple-strided, data reuse, write-sharing, indirect sparse, and

dynamic index patterns generated at run-time by CGRA cells [12]. For regular and structured memory streams, address generation, stride increments, and pattern boundary tracking are managed completely in by the DSU's Address Generators (AGs). The CGRA Functional Units (FUs) are only involved in memory index calculations for the dynamic index access patterns whose addresses cannot be determined prior to run-time.

3.2 UVE Streaming Engine

The UVE SE, evaluated by Midões [13], approaches memory management directly at the Instruction Set Architecture (ISA) level. Rather than operating as an external system accelerator bridge, the UVE SE integrates directly into the execution pipeline of a core processor — such as a reduced instruction set computer (RISC-V) or ARM vector processor — to feed the Single Instruction, Multiple Data (SIMD) and Floating-Point (FP) register file [13].

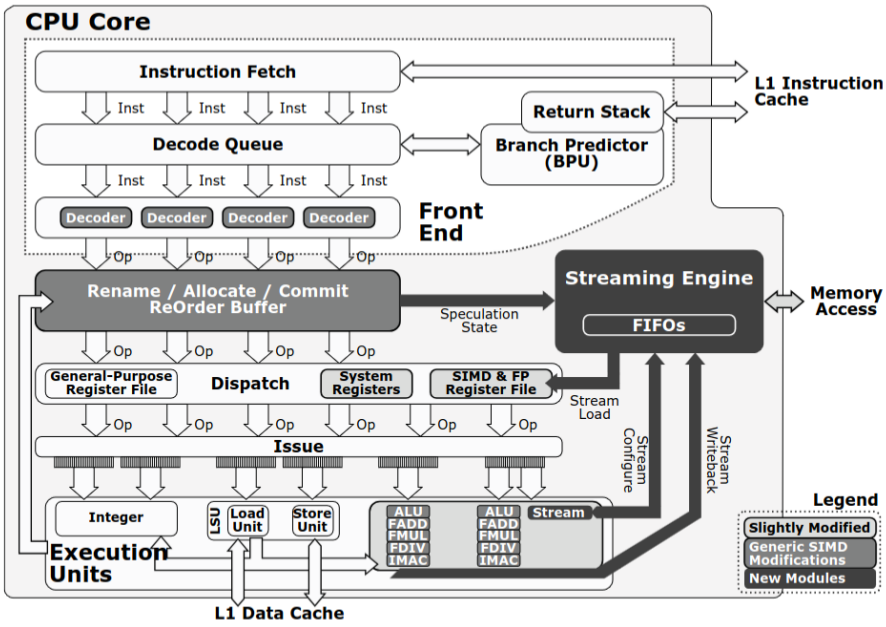


Figure 3.5: UVE SE - Integration Overview Within a CPU Core Architecture (Adapted from [13]).

As illustrated in Figure 3.5, the UVE SE attaches straight to the processor's front-end decode queue and issue stage, placing stream data directly into the core SIMD and FP register file through specialized internal FIFOs [13]. The UVE SE model replaces explicit, repetitive scalar or vector load and store instructions inside inner execution loops with software-configured *Stream Descriptors* [13].

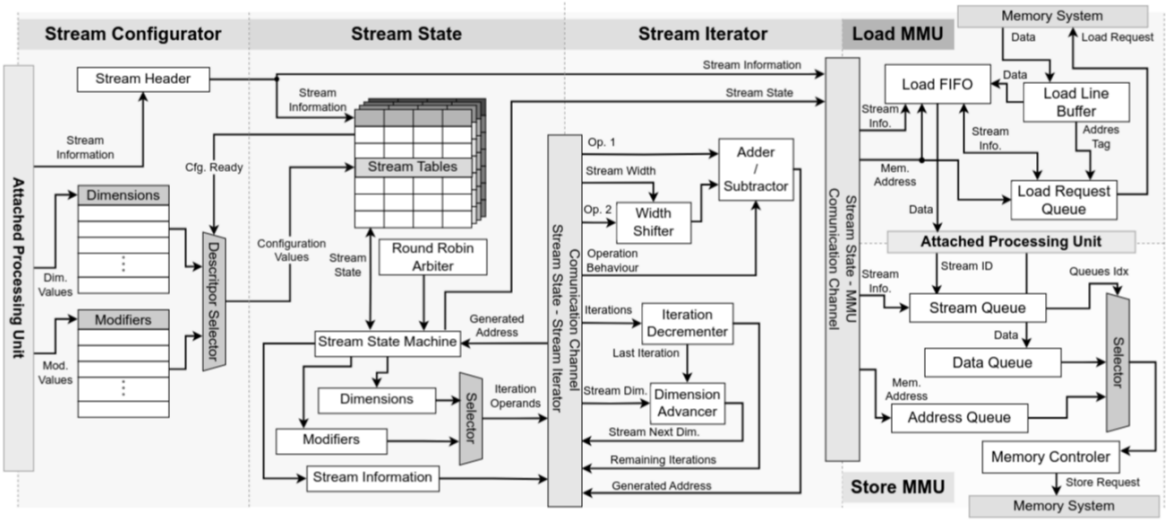


Figure 3.6: UVE SE - Architecture High-Level Overview (Adapted from [13]).

At a high level, the internal micro-architecture of the UVE SE (Figure 3.6) is organized into four cooperative functional modules that orchestrate stream manipulation and memory handling. The process begins at the **Stream Configurator**, which decodes pattern parameters dispatched by the processor pipeline and maps dimension strides, bounds, and state modifiers into active stream descriptors. These descriptors are passed to the **Stream State** module, which maintains active stream contexts across multi-dimensional stream tables and orchestrates round-robin arbitration among concurrent instruction streams. Address generation is carried out by the **Stream Iterator**, which utilizes dedicated Adders, Width Shifters, and a Dimension Advancer to step through array memory addresses in hardware. Finally, the **Load and Store Memory Management Units (MMUs)** interface directly with the memory hierarchy via dedicated Load/Store FIFOs and Line Buffers, hiding memory access latency and ensuring continuous data availability for the core execution units [13].

While the UVE SE effectively eliminates instruction-fetch overheads inside compute loops and hides pipeline latencies for vector processing, its multidimensional boundary traversal remains governed by software loop control and instruction-level stream register updates.

3.3 Discussion

To contextualize the design of the proposed SE, Table 3.1 summarizes the primary design differences between the UVE SE, the DSU, and the proposed SE across their architectural scope, target destination structures, and memory decoupling levels.

Table 3.1: Architectural Comparison of Streaming Units.

Streaming Unit	Architectural Scope	Target Structure	Memory Decoupling Level
UVE SE [13]	In-pipeline CPU execution core extension (integrated between the decode and issue stages).	CPU SIMD and FP register file.	**ISA-Level Decoupling:** Eliminates inner-loop explicit load/store and address instructions, but outer loop control remains governed by CPU pipeline stream-branch instructions.
DSU [12]	Memory streaming interface within a tightly-coupled CPU-CGRA accelerator (T-CGRA).	Spatial CGRA cell grid (input/output cell FIFOs).	**Full Decoupling:** Handles address generation and cache-line coalescing autonomously in hardware; offloads index calculation to cells only for dynamic data-dependent accesses.
Proposed SE	Independent hardware SE placed between the memory system and the compute substrate (MD-CE).	MD-CE PEs.	**Full Decoupling:** Offloads address generation and memory accesses entirely from the compute substrate, autonomously managing complex multidimensional access patterns.

Although all three streaming engines adopt a decoupled access-execute execution paradigm to hide memory access latency, they differ fundamentally in their integration domains, configuration models, target hardware structures, and memory decoupling levels.

Regarding architectural scope and configuration, the UVE SE [13] operates as an in-pipeline execution core extension integrated directly at the decode and issue stages of a CPU core. Its memory access patterns are pre-configured through ISA-level stream descriptors declared in the loop preamble. In contrast, the DSU [12] functions as an internal memory interface module within a tightly-coupled CPU-CGRA accelerator (T-CGRA). When offloading a loop, the host CPU issues a specialized instruction that delivers bundled configuration bitstreams to configure both the CGRA array and the DSU's internal streaming slices. Distinct from both approaches, the proposed SE is implemented as a standalone hardware streaming unit positioned between the memory hierarchy and the compute substrate (MD-CE), without CPU interaction. Instead of relying on instruction-bundled setup vectors, it is configured directly through dedicated hardware registers, enabling its internal address generation units to autonomously orchestrate complex memory access patterns structured up to 6 nested loops.

The target structures fed by each streaming engine highlight their distinct architectural purposes. The UVE SE delivers fetched memory data directly into the core's CPU SIMD and FP

register file, where stream-mapped registers supply operands directly to the processor's execution units. Conversely, the DSU and the proposed SE both feed dedicated compute substrates, though they interface with their target arrays differently. While the DSU connects to the spatial CGRA cell grid through input and output cell FIFOs at cache-line granularity, the proposed SE feeds the PEs directly through their input ports, transferring raw data streams without requiring intermediate FIFO queues.

Finally, the architectures exhibit key distinctions in their level of memory decoupling. The UVE SE provides ISA-level decoupling by eliminating explicit load, store, and pointer arithmetic instructions inside inner execution loops; however, higher-level program flow and multidimensional loop boundaries remain synchronized with CPU pipeline stream-branch instructions. The DSU achieves full hardware decoupling for affine, strided, multi-strided, data-reuse, and structured sparse access patterns through internal address generators, involving the CGRA cells only when evaluating dynamic, data-dependent index calculations that stream runtime offsets back to its index FIFOs. The proposed SE delivers complete hardware decoupling across all supported access patterns by offloading address generation, stride stepping, and multidimensional boundary tracking entirely from the compute substrate. As a result, the PEs in the MD-CE perform zero address arithmetic or loop control operations, functioning strictly as processing units.

Chapter 4

Designed Architectures

This chapter presents the detailed structural and functional design of the hardware architectures developed in this work. To establish a clear understanding of the overall processing pipeline, the system components are described in a modular, top-down hierarchy.

First, Section 4.1 details the SE, which manages data movement, cache synchronization, and execution scheduling across the input and output streams. Next, Sections 4.2 and 4.3 cover the specialized processing units, the MC-CE and the MD-CE, detailing their internal datapath organization and execution models. Following the compute units, Section 4.4 describes the MC, which arbitrates read and write requests and bridges communication between the execution engines and memory. Finally, Section 4.5 outlines the MB architecture, detailing the underlying storage hierarchy and data access interfaces.

4.1 Streaming Engine

4.1.1 Architecture Overview

The SE is responsible for managing three distinct data streams: Input Stream 0 (IS0), Input Stream 1 (IS1), and Output Stream (OS). The system architecture is logically partitioned into stream-specific processing units, a centralized control scheduler, shared memory execution engines, and memory interface components.

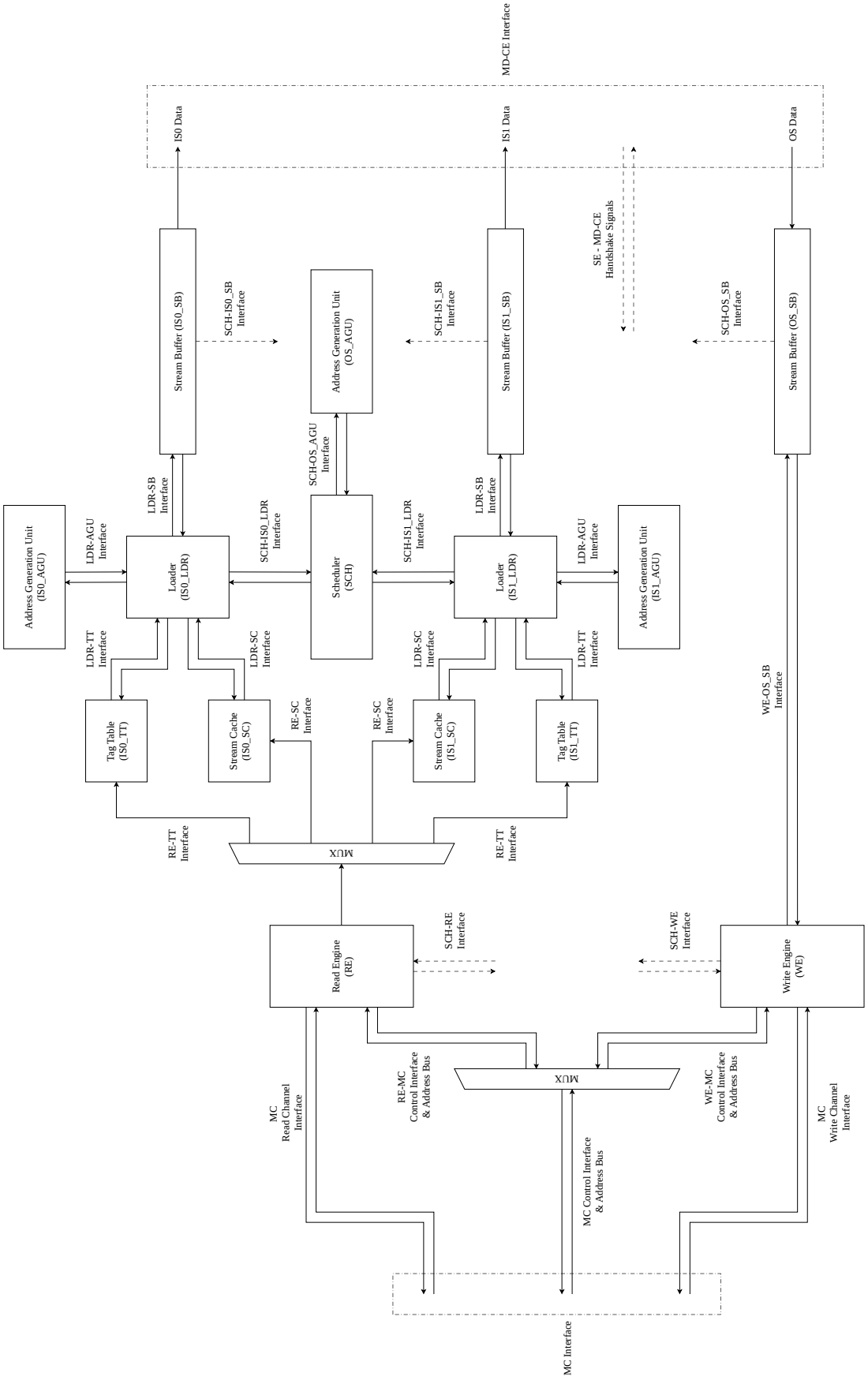


Figure 4.1: SE - Architecture Overview.

As shown in Figure 4.1, each stream is composed of a dedicated set of modules:

- **Input Streams (IS0 / IS1):** Each input stream block contains an Address Generation Unit (AGU), a Tag Table (TT), a Stream Cache (SC), a Loader (LDR), and a Stream Buffer (SB). Together, these components calculate data addresses, manage cache hits and tag lookups, store prefetched stream elements, and feed validated operands to the processing logic.
- **Output Stream (OS):** The output stream block consists of an AGU and a SB, which manage the buffering and target address generation for computed result data before writing back to memory.

Centralized coordination and memory interfacing are handled by the following core components:

- **Scheduler (SCH):** Serves as the main control unit, dynamically evaluating buffer occupancies, prioritizing stream execution tasks, resolving cache misses, and controlling pipeline multiplexers.
- **RE and WE:** Act as shared memory access masters. The RE performs cache line refills upon detecting misses, while the WE flushes buffered output data back to memory.

All inter-module, scheduler, engine, and memory controller interfaces depicted in Figure 4.1 are detailed in subsequent sections.

4.1.2 Sub-Module Description

4.1.2.1 Scheduler

The SCH is the central control unit responsible for orchestrating execution flow, task scheduling, and memory access arbitration across the system. Whenever stream processing is enabled, the SCH dynamically monitors the occupancy levels of the input and output SBs, evaluates a multi-tiered priority scheme, and dispatches execution commands to the LDRs, the memory communication engines (RE and WE), and the OS AGU. When one of the LDRs detect a cache miss during its loading process, the SCH pauses the corresponding load pipeline, instructs the RE to fetch the required cache line from main memory to update the internal cache infrastructure, and subsequently signals the LDR to resume execution. Simultaneously, the SCH prevents stream backpressure by monitoring the OS SB and triggering the WE to perform memory writebacks, allowing continuous, non-blocking execution throughout the streaming pipeline.

The structural characteristics are configured through four static parameters:

- `AW` (Address Width): Defines the bit-width of the address buses.
- `LEN` (Burst Length): Defines the total number of elements transferred during a burst operation.

- `SC_WDT` (SC Width): Defines the cache line entry width, used to determine the line offset bit-width.
- `SB_DPT` (SB Depth): Defines the maximum element capacity of the input and output SBs.

The SCH's block diagram is presented below.

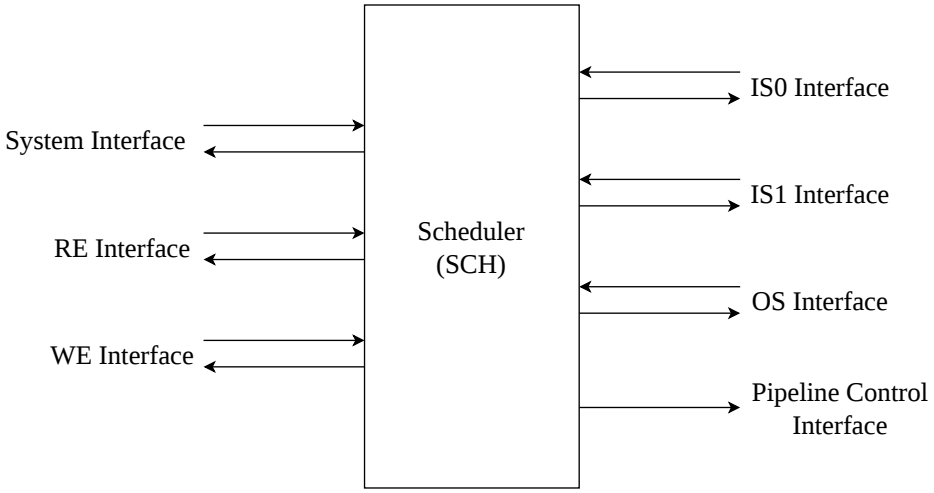


Figure 4.2: SE - SCH Block Diagram.

The physical ports are organized into seven primary functional interfaces.

The **System Interface** handles fundamental operational timing, global control, and status tracking for the SCH, partitioned into a Control & Timing Interface and a Status Interface.

The **IS0 Interface** establishes the control and status boundary for the IS0, partitioned into a Configuration Interface for stream properties, a SB Interface for occupancy tracking, and a LDR Interface driving execution and miss resolution with the IS0 LDR.

The **IS1 Interface** establishes the control and status boundary for IS1, similarly partitioned into a Configuration Interface, a SB Interface, and a LDR Interface driving execution and miss resolution with the IS1 LDR.

The **OS Interface** manages the control and status boundary for the output stream, partitioned into a Configuration Interface, a SB Interface to track writeback availability, and an AGU Interface driving request triggers to the OS AGU.

The **RE Interface** connects directly to the RE, where the SCH acts as a master to issue line-refill burst requests upon the occurance of cache misses.

The **WE Interface** connects directly to the WE, where the SCH acts as a master to issue single or burst writeback requests when flushing output data to main memory.

Finally, the **Pipeline Control Interface** drives internal multiplexer selection signals that dynamically route memory bus ownership, input stream refill targets, and cache address access paths across the system.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - **Control & Timing Interface:**
    - `clk` (input): Clock signal.
    - `rst` (input): Active-high asynchronous reset signal.
    - `en` (input): Enable signal initiating SCH evaluation.
    - `in_rdy` (output): Backpressure control signal indicating readiness of the OS SB to accept computation results.
    - `out_vld` (output): Valid signal indicating that both IS0 and IS1 SBs contain valid data for processing units.
  - **Status Interface:**
    - `bsy` (output): Status signal indicating the SCH is actively executing or evaluating tasks.
    - `dn` (output): Completion signal indicating all stream processing operations are finished.
- **IS0 Interface:**
  - **Configuration Interface:**
    - `is0_n[31:0]` (input): Total element count for the IS0.
    - `is0_sb_cfg` (input): SB configuration mode (0 = FIFO, 1 = CB).
  - **SB Interface:**
    - **Status Interface:**
      - `is0_sb_lvl[31:0]` (input): Current occupancy level of the IS0 SB.
      - `is0_sb_emp` (input): Empty status signal from the IS0 SB.
      - `is0_sb_fll` (input): Full status signal from the IS0 SB.
  - **LDR Interface:**
    - **Control Interface:**
      - `is0_ldr_req` (output): Request signal driven to the IS0 LDR to initiate a loading operation.
      - `is0_ldr_n[31:0]` (output): Length of the loading request for the IS0 LDR.
      - `is0_ldr_rsm` (output): Resume signal asserted to instruct the IS0 LDR to proceed following miss resolution.

    - ∗ **Address Bus:**
        - · `is0_ldr_adr[AW-1:0]` (input): Miss target memory address reported by the IS0 LDR.
    - ∗ **Status Interface:**
        - · `is0_ldr_miss` (input): Cache miss signal asserted by the IS0 LDR.
        - · `is0_ldr_ack` (input): Single-cycle completion acknowledgment signal driven by the IS0 LDR.

- **IS1 Interface:**
    - **Configuration Interface:**
        - ∗ `is1_n[31:0]` (input): Total element count for the IS1.
        - ∗ `is1_sb_cfg` (input): SB configuration mode (0 = FIFO, 1 = CB).
    - **SB Interface:**
        - ∗ **Status Interface:**
            - · `is1_sb_lvl[31:0]` (input): Current occupancy level of the IS1 SB.
            - · `is1_sb_emp` (input): Empty status signal from the IS1 SB.
            - · `is1_sb_fll` (input): Full status signal from the IS1 SB.
    - **LDR Interface:**
        - ∗ **Control Interface:**
            - · `is1_ldr_req` (output): Request signal driven to the IS1 LDR to initiate a loading operation.
            - · `is1_ldr_n[31:0]` (output): Length of the loading request for the IS1 LDR.
            - · `is1_ldr_rsm` (output): Resume signal asserted to instruct the IS1 LDR to proceed following miss resolution.
        - ∗ **Address Bus:**
            - · `is1_ldr_adr[AW-1:0]` (input): Miss target memory address reported by the IS1 LDR.
        - ∗ **Status Interface:**
            - · `is1_ldr_miss` (input): Cache miss signal asserted by the IS1 LDR.
            - · `is1_ldr_ack` (input): Single-cycle completion acknowledgment signal driven by the IS1 LDR.

- **OS Interface:**
    - **Configuration Interface:**
        - ∗ `os_n[31:0]` (input): Total element count for the OS.
        - ∗ `os_sb_cfg` (input): SB configuration mode (0 = FIFO, 1 = CB).

  - **SB Interface:**
    - **Status Interface:**
      - `os_sb_lvl[31:0]` (input): Current occupancy level of the OS SB.
      - `os_sb_emp` (input): Empty status signal from the OS SB.
      - `os_sb_fll` (input): Full status signal from the OS SB.
  - **AGU Interface:**
    - **Control Interface:**
      - `os_agu_req` (output): Request signal asserted to trigger the OS AGU to advance writeback memory addresses.
    - **Address Bus:**
      - `os_agu_adr[AW-1:0]` (input): Target memory writeback address generated by the OS AGU.

- **RE Interface:**
  - **Control Interface:**
    - `re_req` (output): Request signal asserted to initiate a refill operation.
    - `re_mod` (output): Transaction mode configuration (0 = Single, 1 = Burst).
  - **Address Bus:**
    - `re_adr[AW-1:0]` (output): Target base memory address driven to the RE.
  - **Status Interface:**
    - `re_ack` (input): Completion acknowledgment signal returned by the RE.

- **WE Interface:**
  - **Control Interface:**
    - `we_req` (output): Request signal asserted to initiate a writeback operation.
    - `we_mod` (output): Writeback transaction mode configuration (0 = Single, 1 = Burst).
  - **Address Bus:**
    - `we_adr[AW-1:0]` (output): Target base memory address driven to the WE.
  - **Status Interface:**
    - `we_ack` (input): Completion acknowledgment signal returned by the WE.

- **Pipeline Control Interface:**
  - `sel_egn` (output): Engine multiplexer select signal determining which memory engine (0 = RE, 1 = WE) gains access to the Control Interface of the MC to issue memory requests.

- `sel_is` (output): Input stream multiplexer select signal determining which input stream block (0 = IS0, 1 = IS1) receives write updates to its SC and TT from the RE during a cache line refill.
- `sel_acs` (output): Cache access multiplexer select signal determining for both input stream cache systems whether address bus control is granted to the stream's respective LDR (0 = LDR) or to the RE (1 = RE).

The overall operation of the SCH is managed by an internal 16-state Finite State Machine (FSM). The primary purpose of this state machine is to continuously evaluate SB conditions, dispatch loading tasks to LDRs, handle cache misses via the RE, and initiate memory writebacks via the WE. The defined operational states are:

- `IDL` (Idle): Default idle state awaiting activation via the `en` signal.
- `CHK` (Check): Evaluates priority conditions (`p0`–`p7`) and total stream completion to select the highest-priority operational task.
- `RQ_LD0` (IS0 Load Request): Configures the operation parameters and asserts the loading request to the IS0 LDR.
- `LD0` (IS0 Load Active): Monitors IS0 LDR execution until load completion or cache miss detection.
- `RQ_RD0` (IS0 Read Request): Asserts a line-refill burst request to the RE with aligned memory addresses to resolve an IS0 LDR miss.
- `RD0` (IS0 Read Active): Awaits RE completion while routing cache access paths to IS0.
- `RSM0` (IS0 Load Resume): Asserts the resume signal to the IS0 LDR to re-evaluate the access following miss resolution.
- `RQ_LD1` (IS1 Load Request): Configures the operation parameters and asserts the loading request to the IS1 LDR.
- `LD1` (IS1 Load Active): Monitors IS1 LDR execution until load completion or cache miss detection.
- `RQ_RD1` (IS1 Read Request): Asserts a line-refill burst request to the RE with aligned memory addresses to resolve an IS1 LDR miss.
- `RD1` (IS1 Read Active): Awaits RE completion while routing cache access paths to IS1.
- `RSM1` (IS1 Load Resume): Asserts the resume signal to the IS1 LDR to re-evaluate the access following miss resolution.
- `RQ_WR` (Write Request): Configures mode parameters, triggers the OS AGU, and issues a writeback request to the WE.

- `WR` (Write Active): Awaits WE completion while selecting WE access paths.
- `UPD` (Update AGU): Multi-cycle state advancing the OS AGU address pointer across burst lengths following writebacks.
- `DN` (Done): Asserts global stream completion signal (`dn=1`) upon processing all streams.

The SCH's FSM diagram is presented in Figure 4.3. To maintain modularity, the state machine branches from the `CHK` state into three dedicated sub-diagrams: the IS0 load branch (`LD0_OP`, Figure 4.4), the IS1 load branch (`LD1_OP`, Figure 4.5), and the output write branch (`WR_OP`, Figure 4.6). Each sub-FSM executes its respective sequence before returning to `CHK`.

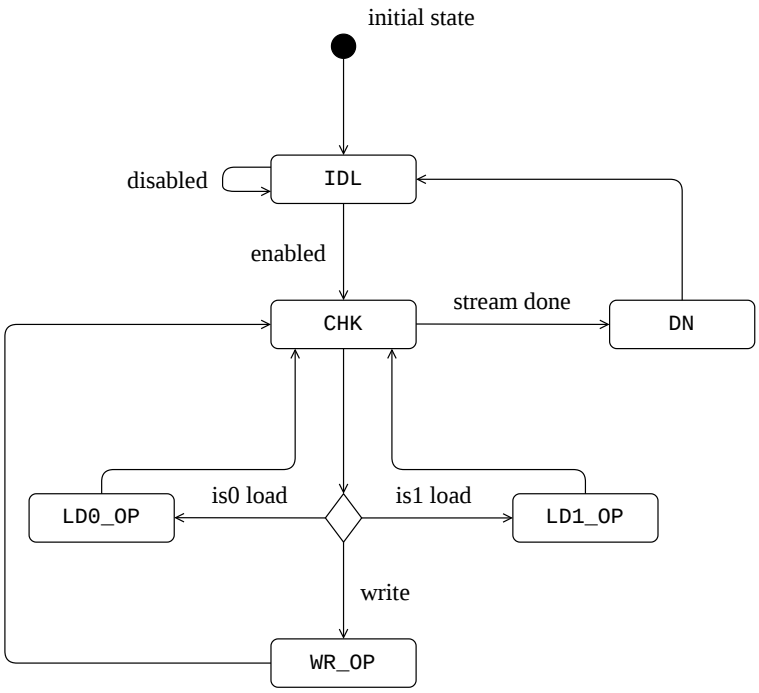


Figure 4.3: SCH - FSM Diagram.

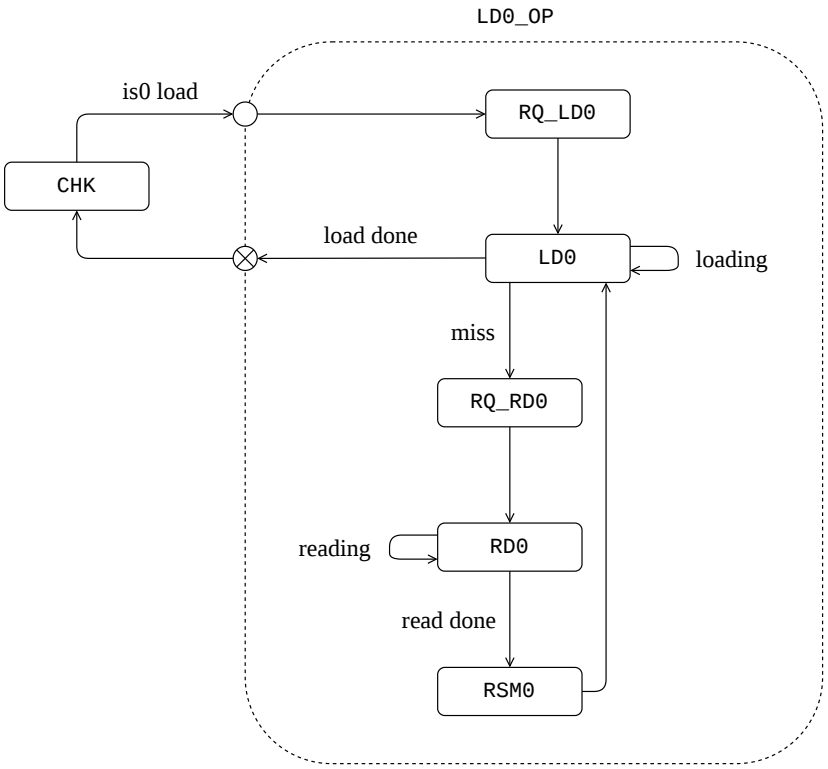


Figure 4.4: SCH - IS0 Load Branch Diagram.

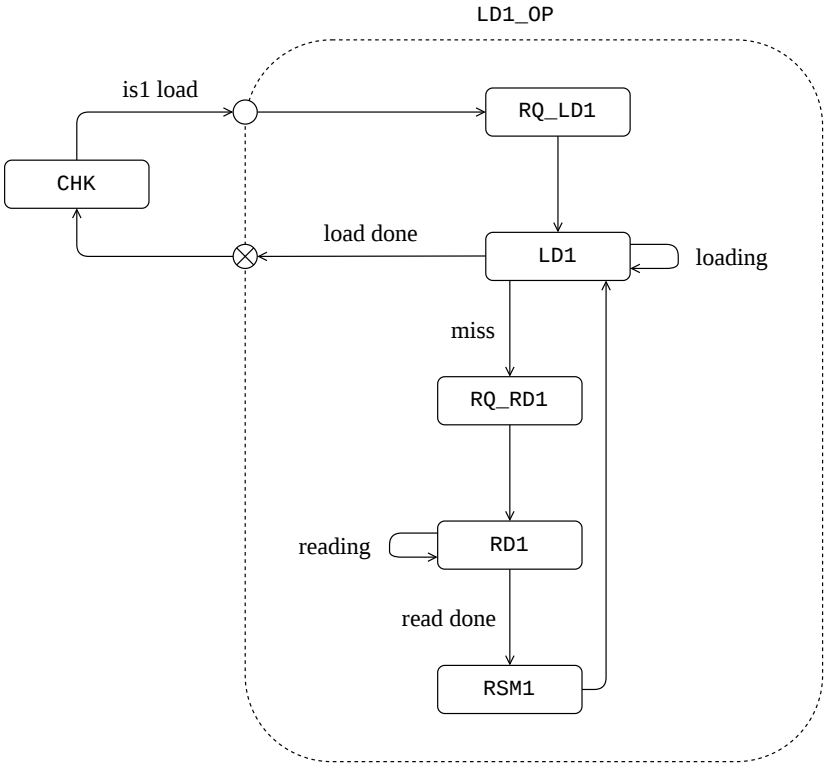


Figure 4.5: SCH - IS1 Load Branch Diagram.

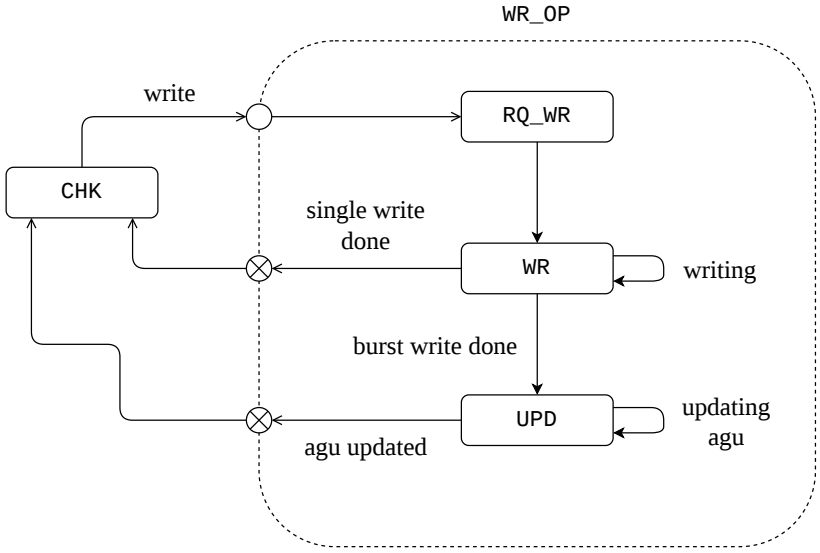


Figure 4.6: SCH - Write Branch Diagram.

The SCH rests in the IDL state until the enable signal (en) is asserted, whereupon it advances to the CHK state. In CHK, the SCH evaluates the overall completion condition (os_dn) alongside eight prioritized condition flags (p0–p7). If all output data processing has completed (os_dn=1), the state machine transitions to DN. Otherwise, it evaluates the priority conditions in strict order to select the next operational flow:

- **Priority 0 & 1 (p0, p1):** Mandatory fixed-size stream loading for IS0 or IS1, branching to RQ_LD0 or RQ_LD1. This priorities are asserted whenever the SBs are configured with the CB mode, where the loading proccess will only occur once, with all the necessary stream data being loaded.
- **Priority 2 (p2):** High-watermark writeback trigger, activated when the OS SB occupancy reaches or exceeds 60% capacity ($\geq 60\%$), branching to RQ_WR.
- **Priority 3 & 4 (p3, p4):** Low-watermark stream loading flags, triggered when IS0 or IS1 SBs occupancy drops to or below 30% capacity ($\leq 30\%$), branching to RQ_LD0 or RQ_LD1.
- **Priority 5 (p5):** Output buffer drain trigger, activated whenever the OS SB contains non-zero data, branching to RQ_WR.
- **Priority 6 & 7 (p6, p7):** Opportunistic loading, triggered whenever IS0 or IS1 is incomplete and its corresponding SB is not full, branching to RQ_LD0 or RQ_LD1.

When loading for IS0 is selected (RQ_LD0), the SCH calculates the requested length load size (is0_ldr_n) based on buffer configuration and remaining elements, asserts is0_ldr_req, and transitions to LD0. In LD0, the SCH monitors the IS0 LDR. If the IS0 LDR completes the loading request successfully (is0_ldr_ack=1), the SCH returns to CHK. If a cache miss occurs (is0_ldr_miss=1), the SCH captures the miss address (is0_ldr_adr) into an internal register and advances to RQ_RD0.

In RQ_RD0, the SCH asserts re_req=1, sets burst mode (re_mod=1), and drives the cache-line-aligned base address onto re_adr. It then moves to RD0, driving control signals (sel_egn=0, sel_is=0, sel_acs=1) to grant cache update access to the RE until the RE acknowledges completion (re_ack=1). The SCH then transitions to RSM0 to pulse is0_ldr_rsm=1, instructing the IS0 LDR to resume its execution, and returns to LD0. An identical execution model is followed for IS1 through states RQ_LD1, LD1, RQ_RD1, RD1, and RSM1.

When a writeback is initiated (RQ_WR), the SCH asserts os_agu_req=1 to fetch the target address from the OS AGU, asserts we_req=1, and configures writeback mode (we_mod) to burst mode if buffer occupancy contains at least LEN elements, or single mode otherwise. It then advances to WR, setting sel_egn=1 until the WE completes (we_ack=1). If a burst writeback was executed, the SCH enters UPD for LEN $-$ 2 cycles to step the OS AGU to the next address in line before returning to CHK.

Upon completing all stream transactions, the state machine enters DN, asserting the completion signal (dn=1), and returns to IDL to await subsequent streaming operations.

4.1.2.2 Loader

The LDR is a loading hardware unit responsible for transferring data elements into the SB, using addresses generated by the AGU to query the TT and retrieve valid data from the SC.

The structural characteristics are configured through two static parameters:

- `AW` (Address Width): Defines the bit-width of the address buses.
- `DW` (Data Width): Defines the bit-width of the data buses.

The LDR's block diagram is presented below.

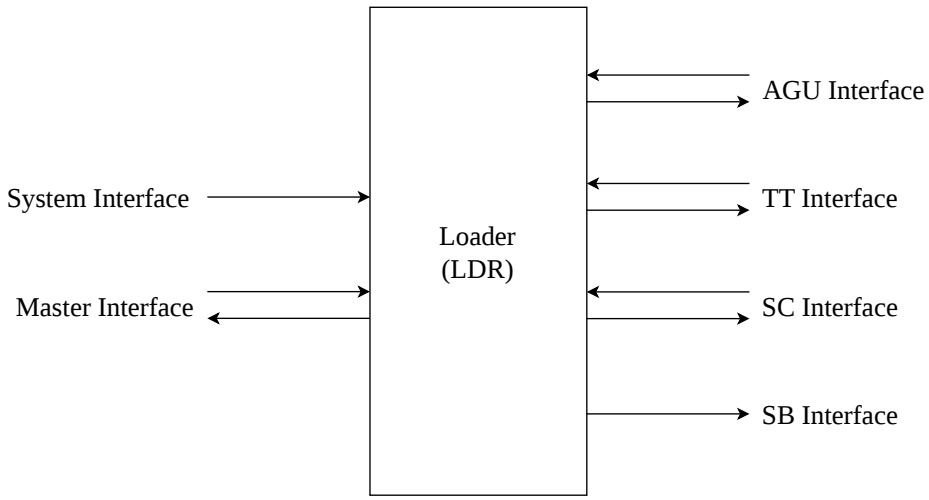


Figure 4.7: SE - LDR Block Diagram.

The physical ports are organized into six primary functional interfaces.

The **System Interface** handles fundamental operational timing and control, providing global clocking and reset distribution to the module.

The **Master Interface** establishes the upstream communication boundary where the LDR receives loading requests from the master, reports cache miss conditions, and asserts completion handshakes.

The **AGU Interface** connects the LDR to the AGU, where the LDR acts as the master to issue step-advance requests and sample calculated target memory addresses.

The **TT Interface** forms the query interface to the TT, with the LDR driving check commands and addresses to verify whether target locations reside in the cache.

The **SC Interface** establishes the read bus where the LDR acts as a master, driving read enable signals and addresses to fetch valid cached data elements.

Finally, the **SB Interface** forms the downstream write boundary, where the LDR masters push operations to write retrieved data elements into the SB.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - `clk` (input): Clock signal.
  - `rst` (input): Active-high asynchronous reset signal.
- **Master Interface:**
  - **Control Interface:**
    - `req` (input): Request signal asserted by a master to initiate a loading operation.
    - `n[31:0]` (input): Length of the loading request, driven by the master.
    - `rsm` (input): Resume signal asserted by the master once a cache miss is resolved.
  - **Address Bus:**
    - `adr[AW-1:0]` (output): Missing target address bus presented to the master for refill handling.
  - **Status Interface:**
    - `miss` (output): Cache miss signal asserted when a target address tag lookup fails.
    - `ack` (output): Single-cycle completion acknowledgment signal returned to the master.
- **AGU Interface:**
  - **Control Interface:**
    - `agu_req` (output): Advance request signal asserted to step internal AGU counters to the next address.
  - **Address Bus:**
    - `agu_adr[AW-1:0]` (input): Memory address bus continuously driven by the AGU.
- **TT Interface:**
  - **Control Interface:**
    - `tt_chk` (output): Tag check enable signal asserting a tag evaluation query.
  - **Common Address Bus:**
    - `tt_adr[AW-1:0]` (output): Address bus presented to the TT for tag matching.
  - **Status Interface:**
    - `tt_hit` (input): Status signal indicating a tag lookup hit.
    - `tt_miss` (input): Status signal indicating a tag lookup miss.
- **SC Interface:**

  - **Control Interface:**
    * `sc_rd` (output): Cache read enable signal asserted to fetch cached data.
  - **Common Address Bus:**
    * `sc_adr[AW-1:0]` (output): Target memory address bus driven to the SC.
  - **Read Channel Interface:**
    * `sc_rd_dt[DW-1:0]` (input): Read data bus returning cached data elements from the SC.

- **SB Interface:**
  - **Control Interface:**
    * `sb_psh` (output): Push enable signal driven to the SB.
  - **Input Channel Interface:**
    * `sb_din[DW-1:0]` (output): Data bus driven with the pushing data to the SB.

The execution of loading operations is managed by an internal 7-state FSM. The purpose of this state machine is to coordinate the loading requests from the upstream master by iteratively fetching n data elements and pushing them into the SB. The defined operational states are:

- `IDL` (Idle): Default state awaiting incoming load requests.
- `CHK` (Check): Queries the TT using the current address supplied by the AGU to evaluate cache hit or miss status.
- `REQ` (Request): Asserts a single-cycle miss notification and presents the missing address to the master to initiate a cache refill sequence.
- `HLD` (Hold): Pauses execution while awaiting the master to resolve the cache data allocation.
- `RD` (Read): Asserts a read request to the SC to fetch the validated data element.
- `PSH` (Push): Pushes the fetched data element directly into the SB and signals the AGU to advance to the next target address.
- `ACK` (Acknowledge): Asserts a single-cycle completion signal to notify the master that the n requested elements have been loaded.

The LDR's FSM diagram is presented in Figure 4.8.

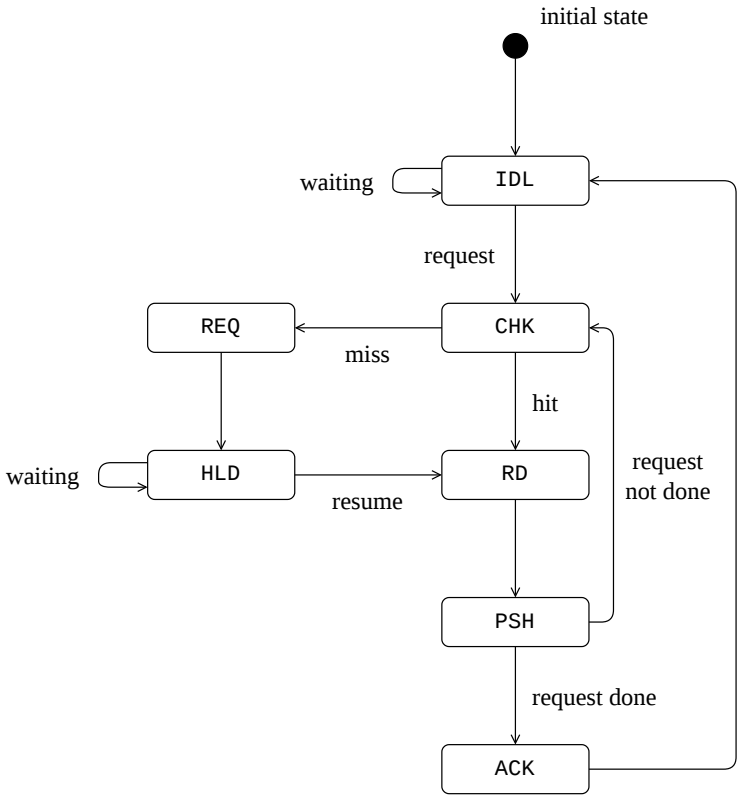


Figure 4.8: LDR - FSM Diagram.

When the LDR is in the `IDL` state, it continuously monitors the incoming request signal (`req`), waiting for the master to initiate a loading sequence. The master assertion of `req` as a single-cycle pulse accompanied by the target length (`n`), instructs the LDR to load n elements into the SB. During this cycle, the LDR captures n and advances to the `CHK` state.

Upon entering the `CHK` state, the LDR checks whether the current element to load is present in the cache, by asserting the tag check signal (`tt_chk`) and presenting the target memory address generated by the AGU (`agu_adr`) to the TT.

If the lookup results in a hit (`tt_hit=1`), the loading sequence proceeds without stalling, and the state machine advances directly to the `RD` state to read the data element.

If the lookup results in a miss (`tt_miss=1`), the requested element is missing from the SC, causing a transition into the `REQ` state, where the LDR asserts a single-cycle miss notification (`miss=1`) and routes the missing address (`adr=agu_adr`) to the master. This requests the master to bring the missing element along with adjacent data into the cache. Immediately following this assertion, the LDR enters the `HLD` state, pausing its execution until the master resolves the cache data allocation and asserts the resume signal (`rsm=1`). Once resumed, the LDR advances to the `RD` state to fetch the missed element.

In the RD state, the LDR issues a read command to the SC (sc_rd) using the address provided by the AGU (sc_adr=agu_adr), retrieving the target data before moving to the PSH state.

In the PSH state, the LDR pushes the registered data element into the SB by asserting sb_psh and driving the data into the data bus (sb_din). At the same time, the LDR asserts agu_req to instruct the AGU to step its calculation logic to the next address.

While in the PSH state, the LDR checks whether the entire requested sequence has been loaded:

- If *n* elements have been pushed into the SB, the state machine advances to the ACK state.
- If more elements remain to be loaded, the state machine loops back to the CHK state to verify the TT for the next address in the sequence.

In the ACK state, the LDR asserts a single-cycle completion acknowledgment (ack=1) to inform the master that the request was completed, and then returns to IDL to await subsequent load commands.

4.1.2.3 Read Engine

The RE is a memory communication module, responsible for executing all read-side operations with the MC. Whenever data must be fetched from main memory, the RE assumes control, manages the external request handshake, and transfers the incoming data directly into the internal cache memory system. Driven by requests from the SCH, it handles the protocol requirements of the MC while simultaneously updating the internal cache infrastructure — writing the retrieved data into the SC and updating the corresponding entry in the TT. Although the RE hardware supports both single and burst transactions, the SCH exclusively issues burst read requests of a fixed length (LEN = 8) to refill a complete line of the SC.

The structural characteristics are configured through three static parameters:

- AW (Address Width): Defines the bit-width of the address buses.
- DW (Data Width): Defines the bit-width of the data buses.
- LEN (Burst Length): Defines the total number of elements retrieved during a burst request.

The RE's block diagram is presented below.

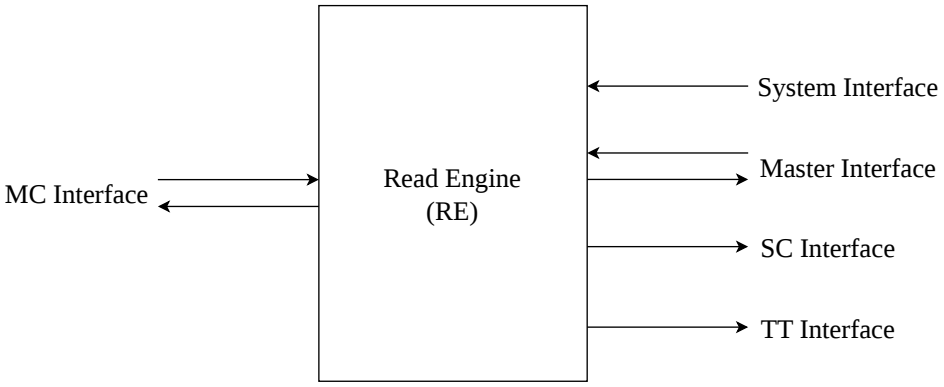


Figure 4.9: SE - RE Block Diagram.

The physical ports are organized into five primary functional interfaces.

The **System Interface** handles fundamental operational timing and control, providing global clocking and reset distribution to the module.

The **Master Interface** establishes the upstream communication boundary where the RE acts in a slave role to receive read request commands and base addresses from the requesting master and return the completion signal.

The **MC Interface** connects directly to the external MC, where the RE acts in a master role to issue read transaction commands and stream in returning memory data.

The **SC Interface** establishes the downstream write bus where the RE acts as a master, driving write enables, calculated target addresses, and payload data to populate the SC.

Finally, the **TT Interface** forms the update boundary to the TT, where the RE acts as a master driving write enables and target addresses to validate directory entries.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - `clk` (input): Clock signal.
  - `rst` (input): Active-high asynchronous reset signal.
- **Master Interface:**
  - **Control Interface:**
    - `req` (input): Request signal asserted by the upstream master to initiate a read operation.
    - `mod` (input): Transaction mode configuration (0 = Single, 1 = Burst).
  - **Address Bus:**
    - `adr[AW-1:0]` (input): Target base memory address bus driven by the master.

  - **Status Interface:**
    - `ack` (output): Single-cycle completion acknowledgment signal returned to the master.

- **MC Interface:**
  - **Control Interface:**
    - `mc_req` (output): Request signal asserted by the RE to initiate a transaction.
    - `mc_typ` (output): Operation type selection (0 = Read, 1 = Write).
    - `mc_mod` (output): Transaction mode configuration (0 = Single, 1 = Burst).
  - **Common Address Bus:**
    - `mc_adr[AW-1:0]` (output): Target memory base address bus driven to the MC.
  - **Status Interface:**
    - `mc_ack` (input): Completion acknowledgment signal returned by the MC.
  - **Read Channel Interface:**
    - `in_rdy` (output): Backpressure control signal driven by the RE to accept read data.
    - `in_vld` (input): Valid signal driven by the MC indicating read data availability.
    - `mc_rd_dt[DW-1:0]` (input): Read data bus driven by the MC.

- **SC Interface:**
  - **Control Interface:**
    - `sc_wr` (output): Active-high write enable signal asserted to write data into the SC.
  - **Common Address Bus:**
    - `sc_adr[AW-1:0]` (output): Target memory address bus driven to the SC.
  - **Read Channel Interface:**
    - `sc_wr_dt[DW-1:0]` (output): Write data bus driving data into the SC.

- **TT Interface:**
  - **Control Interface:**
    - `tt_wr` (output): Active-high write enable signal asserted to validate directory entries.
  - **Common Address Bus:**
    - `tt_adr[AW-1:0]` (output): Target memory address bus driven to the TT.

The execution of read operations is managed by an internal 5-state FSM. The defined operational states are:

- `IDL` (Idle): Default state awaiting incoming read requests.
- `REQ` (Request): Asserts a read transaction request to the MC with the captured target address and transfer mode.
- `RD` (Read): Asserts readiness to accept returning memory data elements and registers them internally.
- `OUT` (Output): Sequentially writes the registered data elements into the SC while simultaneously updating the TT entries.
- `ACK` (Acknowledge): Asserts a single-cycle completion signal to notify the master that the request has been completed.

The RE's FSM diagram is presented in Figure 4.10.

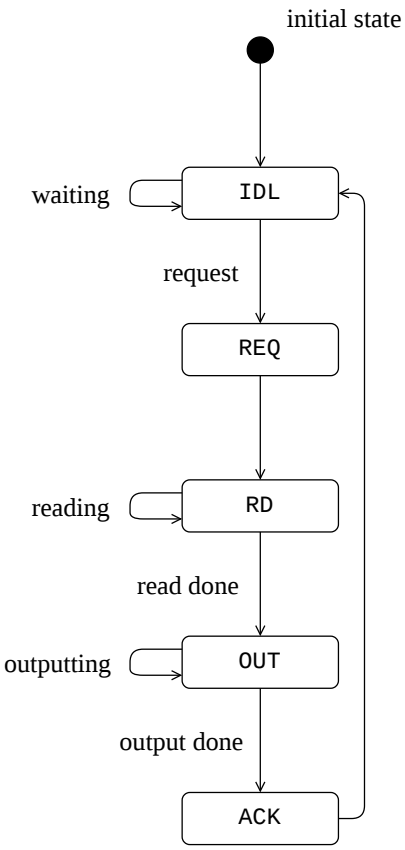


Figure 4.10: RE - FSM Diagram.

The RE rests in the `IDL` state until the master initiates a read operation by pulsing the `req` signal alongside the target address (`adr`) and mode selection (`mod`). Upon receiving this request,

the RE captures the address and mode into internal registers and transitions to the REQ state. The captured mode determines the target transfer length (n), setting it to LEN elements during burst mode or a single element during single mode.

In the REQ state, the RE issues a transaction request to the MC by asserting mc_req, driving the target address onto mc_adr, and setting the transfer mode on mc_mod. On the following clock cycle, the state machine moves directly to the RD state.

While in the RD state, the RE asserts in_rdy to indicate its readiness to receive incoming data. Whenever valid returning data is presented (in_vld=1), a successful transfer cycle occurs, and the incoming element (mc_rd_dt) is captured into an internal register array. Once all n expected data elements have been received and registered, the MC asserts its acknowledgment signal (mc_ack=1), and the state machine advances to the OUT state.

In the OUT state, the RE flushes the stored data elements into the memory subsystem. For each element, it drives the calculated address onto both sc_adr and tt_adr, presents the registered payload on sc_wr_dt, and simultaneously asserts write enables to both the SC (sc_wr) and the TT (tt_wr).

While in the OUT state, the RE evaluates the progress across the written elements:

- If all n elements have been written to the SC and the TT updated, the state machine advances to the ACK state.
- Otherwise, it steps to the next offset and remains in the OUT state to process the next fetched data.

In the ACK state, the RE asserts the acknowledgment signal (ack=1) to inform the master that the SC and TT are updated and ready for streaming operations, and then returns to IDL to await subsequent requests.

4.1.2.4 Write Engine

The WE is a memory communication module, responsible for executing all write-side operations with the MC. Whenever processed stream results must be written back to main memory, the WE assumes control, manages the external request handshake, and streams the data from the internal output storage buffer into external memory. Driven by requests from the SCH, it pops data elements from the OS SB, registers them internally, and handles the protocol requirements of the MC to execute the memory write transactions.

The structural characteristics are configured through three static parameters:

- AW (Address Width): Defines the bit-width of the address buses.
- DW (Data Width): Defines the bit-width of the data buses.
- LEN (Burst Length): Defines the total number of elements transferred during a burst write request.

The WE's block diagram is presented below.

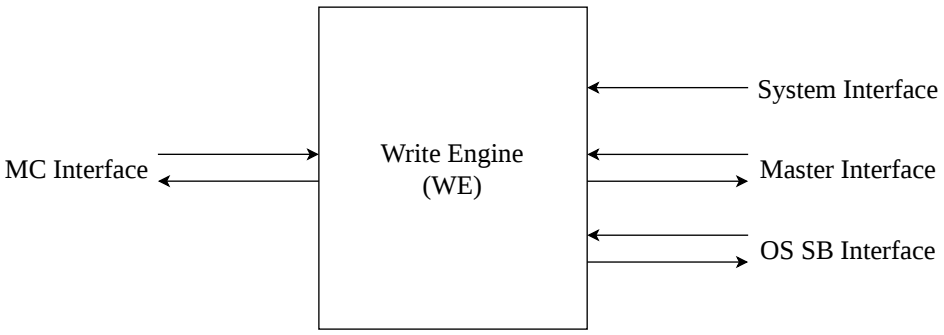


Figure 4.11: SE - WE Block Diagram.

The physical ports are organized into four primary functional interfaces.

The **System Interface** handles fundamental operational timing and control, providing global clocking and reset distribution to the module.

The **Master Interface** establishes the upstream communication boundary where the WE acts in a slave role to receive write request commands and base addresses from the requesting master and return the completion signal.

The **MC Interface** connects directly to the external MC, where the WE acts in a master role to issue write transaction commands and stream output data.

Finally, the **OS SB Interface** establishes the upstream data pull interface where the WE acts as a master to pop stored output data from the OS SB.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - `clk` (input): Clock signal.
  - `rst` (input): Active-high asynchronous reset signal.
- **Master Interface:**
  - **Control Interface:**
    - `req` (input): Request signal asserted by the upstream master to initiate a write operation.
    - `mod` (input): Transaction mode configuration (0 = Single, 1 = Burst).
  - **Address Bus:**
    - `adr[AW-1:0]` (input): Target base memory address bus driven by the master.
  - **Status Interface:**
    - `ack` (output): Single-cycle completion acknowledgment signal returned to the master.

- **MC Interface:**
  - **Control Interface:**
    - `mc_req` (output): Request signal asserted by the WE to initiate a transaction.
    - `mc_typ` (output): Operation type selection (0 = Read, 1 = Write).
    - `mc_mod` (output): Transaction mode configuration (0 = Single, 1 = Burst).
  - **Common Address Bus:**
    - `mc_adr[AW-1:0]` (output): Target memory base address bus driven to the MC.
  - **Status Interface:**
    - `mc_ack` (input): Completion acknowledgment signal returned by the MC.
  - **Write Channel Interface:**
    - `out_rdy` (input): Backpressure control signal driven by the MC indicating readiness to accept write data.
    - `out_vld` (output): Valid signal driven by the WE when write data is valid.
    - `mc_wr_dt[DW-1:0]` (output): Write data bus driven to the MC.
- **OS SB Interface:**
  - **Control Interface:**
    - `os_sb_pop` (output): Active-high pop enable signal driven to fetch data from the OS SB.
  - **Output Channel Interface:**
    - `os_sb_dout[DW-1:0]` (input): Data bus returning from the OS SB.

The execution of write operations is managed by an internal 5-state FSM. The defined operational states are:

- `IDL` (Idle): Default state awaiting incoming write requests.
- `IN` (Input): Pops output data elements sequentially from the OS SB and captures them into an internal register array.
- `REQ` (Request): Asserts a write transaction request to the MC with the captured target address and transfer mode.
- `WR` (Write): Drives registered data elements sequentially to the MC, observing handshakes until the write transaction completes.
- `ACK` (Acknowledge): Asserts a single-cycle completion signal to notify the master that the write operation has been completed.

The WE's FSM diagram is presented in Figure 4.12.

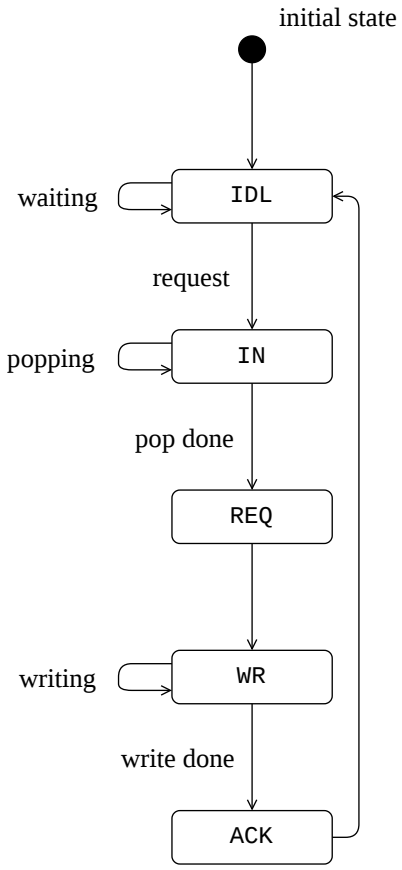


Figure 4.12: WE - FSM Diagram.

The WE rests in the `IDL` state until the master initiates a write operation by pulsing the `req` signal alongside the target address (`adr`) and mode selection (`mod`). Upon receiving this request, the WE captures the address and mode into internal registers and transitions to the `IN` state. The captured mode determines the target transfer length (n), setting it to `LEN` elements during burst mode or a single element during single mode.

In the `IN` state, the WE pops data from the OS SB by asserting `os_sb_pop`. On each clock cycle in this state, the incoming data (`os_sb_dout`) is captured into an internal register array. Once n elements have been collected and registered, the state machine advances to the `REQ` state.

In the `REQ` state, the WE issues a transaction request to the MC by asserting `mc_req`, driving the operation type to write (`mc_typ=1`), setting the transfer mode on `mc_mod`, and outputting the target address on `mc_adr`. On the following clock cycle, the state machine advances directly to the `WR` state.

While in the WR state, the WE presents the registered data element onto the mc_wr_dt bus. Whenever the MC signals readiness (out_rdy=1), the WE asserts out_vld=1 to execute a successful transfer cycle. The WE remains in the WR state, streaming subsequent elements on each valid handshake, until the MC acknowledges transaction completion (mc_ack=1), whereupon the state machine advances to the ACK state.

In the ACK state, the WE asserts the acknowledgment signal (ack=1) to notify the master that the write operation to external memory is complete, and then returns to IDL to await subsequent requests.

4.1.2.5 Address Generation Unit

The AGU is an autonomous hardware calculation engine designed to generate multi-dimensional memory address sequences across diverse access patterns. To support complex nested iterations, its underlying architecture consists of a 6-level synchronized loop counter cascade (lp$i$_cnt, where $i \in \{0,\ldots,5\}$), completion and increment propagation logic, and a multi-operand combinational address adder network.

The architectural characteristics are defined through a single static parameter:

- AW (Address Width): Defines the bit-width of the address bus.

The AGU's block diagram is presented below.

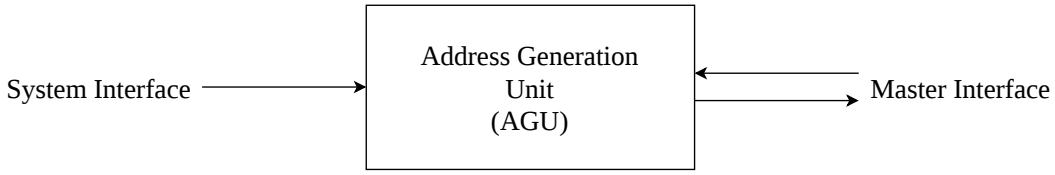


Figure 4.13: SE - AGU Block Diagram.

The physical ports are organized into two primary functional interfaces.

The **System Interface** handles fundamental operational timing, reset control, and the fixed loop configuration parameters — including the base address, loop enables, stride offsets, and iteration count limits for all six loop levels.

The **Master Interface** establishes the operational boundary with the master, where the AGU acts in a slave role to receive step advance requests and present the calculated next address in line.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - **Control & Timing Interface:**
    - clk (input): Clock signal.
    - rst (input): Active-high asynchronous reset signal.

  - **Configuration Interface:**
    * `ba[AW-1:0]` (input): Base address defining the starting memory location.
    * `lpi_en` (input): Enable signal for loop level *i*.
    * `lpi_str[31:0]` (input): Stride offset for loop level *i*, defining address displacement per iteration step.
    * `lpi_n[31:0]` (input): Iteration limit defining the total step count for loop level *i*.

- **Master Interface:**
  - **Control Interface:**
    * `req` (input): Advance request signal asserted by the master to step the internal loop counters to the next address in line.
  - **Address Bus:**
    * `adr[AW-1:0]` (output): Target memory address bus continuously presenting the current memory location in the sequence.

Operational behavior begins with a static configuration phase before execution, where the base address (`ba`) and per-loop parameters (`lpi_en`, `lpi_str`, `lpi_n`) are established to define the desired traversal trajectory. Once configured, the AGU continuously drives the target address on `adr` as a linear combination of the base address and the active loop contributions:

$$\texttt{adr} = \texttt{ba} + \sum_{i=0}^{5} (\texttt{lp}i\texttt{_cnt} \times \texttt{lp}i\texttt{_str}) \tag{4.1}$$

Because this evaluation is entirely combinational, the next required memory location is always present and stable at the output. When the master asserts `req`, the internal counter cascade advances according to its configured iteration limits, seamlessly updating `adr` to present the next address in the sequence.

4.1.2.6 Stream Cache

The SC is a specialized local storage designed to hold contiguous data elements fetched from main memory during burst read transactions.

Unlike traditional set-associative caches, this structure functions as a direct element-mapped cache, where every addressable location corresponds strictly to a single data element of width `DW`.

The primary objective of the SC is to exploit spatial locality. When a burst transaction fills an entire cache line aligned to a base address, subsequent requests for adjacent data elements within that same line can be served immediately from this memory. By satisfying these accesses locally, the system avoids triggering the SCH and RE to initiate external memory transactions, eliminating the latency overhead of external burst operations while freeing up the MC to handle concurrent write requests.

Structurally, the SC is organized as a matrix of `DPT` × `WDT` elements, visualized as a vertical stack of `DPT` cache lines, where each line contains `WDT` contiguous data elements. Each discrete location holds a single data element of width `DW`.

The architectural characteristics are configured through four static parameters:

- `AW` (Address Width): Defines the bit-width of the address bus.
- `DW` (Data Width): Defines the bit-width of an individual cached data element.
- `DPT` (Cache Depth): Defines the total number of cache lines.
- `WDT` (Cache Width): Defines the number of data elements stored per cache line.

The SC's block diagram is presented below.

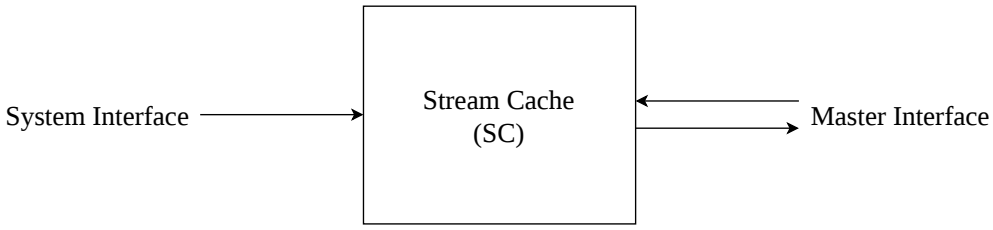


Figure 4.14: SE - SC Block Diagram.

The physical ports are organized into two primary functional interfaces.

The **System Interface** handles fundamental operational timing and control, providing global clocking and reset distribution to the module.

The **Master Interface** establishes the data transfer boundary, where the SC acts in a slave role to receive target addresses, accept incoming burst write payloads, and deliver cached data directly to the requesting master.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - `clk` (input): Clock signal.
  - `rst` (input): Active-high asynchronous reset signal.
- **Master Interface:**
  - **Control Interface:**
    - `rd` (input): Read enable signal driven by the master, asserting a cache read operation.
    - `wr` (input): Write enable signal driven by the master, asserting a cache write operation.

- **Common Address Bus:**
  - `adr[AW-1:0]` (input): Target memory address bus driven by the master.
- **Read Channel Interface:**
  - `rd_dt[DW-1:0]` (output): Read data bus driving the addressed cached element to the processing master.
- **Write Channel Interface:**
  - `wr_dt[DW-1:0]` (input): Write data bus driving incoming burst data into the cache storage array.

The address decoding is performed combinationally by bit-slicing the incoming target address using logarithmic sub-field extractions:

$$\texttt{OW} = \log_2(\texttt{WDT}), \quad \texttt{IW} = \log_2(\texttt{DPT}) \tag{4.2}$$

where `OW` represents the offset bit-width and `IW` represents the index bit-width. The lowest `OW` bits (`adr[OW-1:0]`, extracted as `ofs`) select the element offset within a single cache line, while the subsequent `IW` bits (`adr[OW+IW-1:OW]`, extracted as `idx`) determine the target line index within the cache stack.

Memory transactions are governed by distinct timing domains for writes and reads. Write operations are edge-sensitive; post-reset, an asserted `wr` signal synchronizes the commit of the `wr_dt` payload to the array location determined by the decoded index and offset values. Conversely, read operations are strictly level-sensitive and combinational, optimized for instantaneous data retrieval. An active `rd` signal continuously routes the addressed cell's contents to the read data bus, whereas a low `rd` state defaults the read data bus to zero.

4.1.2.7 Tag Table

The TT is a specialized control tracking unit responsible for maintaining cache coherence and determining data residency.

Unlike traditional associative cache tag structures that track entries strictly per cache line, this module functions as an element-level validation array, where it stores a tag and a valid bit for every discrete location of the SC.

The primary objective of the TT is to perform high-speed, combinational hit/miss evaluation, where by address tag comparison and valid bit verification, the module provides status information regarding the data presence and validity across the SC.

Structurally, the module maintains two internal 2D register arrays with cache matching dimensions, where one is responsible for storing the address tags and the other the valid bits.

The architectural characteristics are configured through three static parameters:

- `AW` (Address Width): Defines the bit-width of the address bus.

- `DPT` (Cache Depth): Defines the total number of cache lines.
- `WDT` (Cache Width): Defines the number of data elements stored per cache line.

The TT's block diagram is presented below.

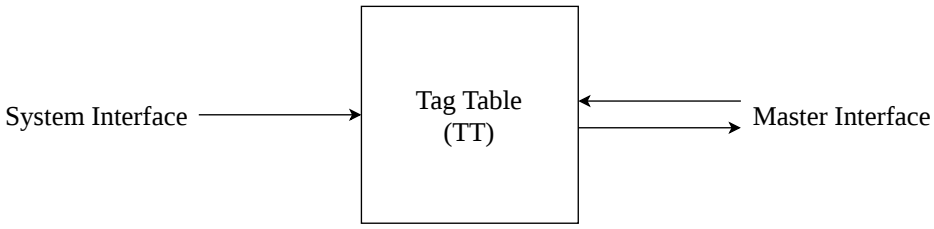


Figure 4.15: SE - TT Block Diagram.

The physical ports are organized into two primary functional interfaces.

The **System Interface** handles fundamental operational timing and control, providing global clocking and reset distribution to the module.

The **Master Interface** establishes a communication boundary where the TT acts in a slave role to receive update and check commands, incoming address streams, and return status results to the attached processing master.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - `clk` (input): Clock signal.
  - `rst` (input): Active-high asynchronous reset signal.
- **Master Interface:**
  - **Control Interface:**
    - `chk` (input): Check enable signal asserted by the master to initiate tag lookup evaluation.
    - `wr` (input): Write enable signal asserted by the master to commit new tag and valid entries.
  - **Common Address Bus:**
    - `adr[AW-1:0]` (input): Address bus driven by the accessing master.
  - **Status Interface:**
    - `hit` (output): Active-high status signal indicating a cache hit.
    - `miss` (output): Active-high status signal indicating a cache miss.

The address decoding is performed combinationally by bit-slicing the incoming target address using logarithmic and sub-field arithmetic extractions:

$$\texttt{OW} = \log_2(\texttt{WDT}), \quad \texttt{IW} = \log_2(\texttt{DPT}), \quad \texttt{TW} = \texttt{AW} - \texttt{IW} - \texttt{OW} \tag{4.3}$$

where `OW` represents the offset bit-width, `IW` represents the index bit-width, and `TW` represents the tag bit-width. The target address is partitioned into three distinct sub-fields: the lowest `OW` bits (`adr[OW-1:0]`, extracted as `ofs`) select the target element offset within a cache line, the adjacent `IW` bits (`adr[OW+IW-1:OW]`, extracted as `idx`) identify the specific line index, and the remaining upper `TW` bits (`adr[AW-1:OW+IW]`, extracted as `tag`) isolate the data element-specific tag.

Memory operations follow distinct timing models for updates and status evaluations. While write transactions execute synchronously, status check operations execute combinationally for zero-latency data residency detection and validation.

Upon system reset, all stored tags and validity bits are cleared to zero, while during normal operation, asserting `wr` on a rising clock edge commits the extracted `tag` into the array entry mapped by `idx` and `ofs`, simultaneously setting its corresponding valid bit to high.

Asserting `chk` compares the extracted `tag` against the stored tag at the (`idx`, `ofs`) table position: if the tags match and the valid bit is high, `hit` is asserted while `miss` remains low; if a mismatch occurs or the valid bit is low, `miss` is asserted instead. Deasserting `chk` safely clears both output flags to zero.

4.1.2.8 Stream Buffer

The SB is a specialized data forwarding unit designed to decouple data fetching from downstream execution. It acts as an elastic staging buffer that temporarily stores data elements fetched by the LDR, supplying a low-latency stream of operands directly to the processing pipeline.

Rather than being fixed to a single architecture, the module operates as a configurable forwarding structure that can be dynamically set to function either as a standard FIFO queue or as a CB.

The primary objectives of the SB are to absorb throughput mismatches between memory loading operations and pipeline consumption, and to support data reuse. By servicing repeated operand requests internally without triggering additional LDR transactions, the module minimizes downstream execution starvation and optimizes memory bandwidth usage.

Structurally, the SB maintains a 1D array of `DPT` storage cells, where each cell holds a single data element of width `DW`.

The architectural characteristics are configured through two static parameters:

- `DW` (Data Width): Defines the bit-width of an individual data element in the buffer.
- `DPT` (Buffer Capacity): Defines the maximum number of storable data elements in the buffer.

The SB's block diagram is presented below.

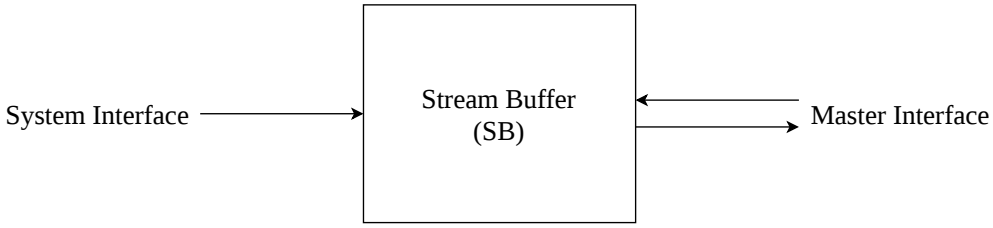


Figure 4.16: SE - SB Block Diagram.

The physical ports are organized into two primary functional interfaces.

The **System Interface** handles fundamental operational timing and control, providing global clocking and reset distribution to the module.

The **Master Interface** establishes the dataflow boundary where upstream and downstream units interact with the buffer. It acts as a slave receiver to the Loader to ingest incoming fetched data streams, while simultaneously operating as a data server to forward queued elements to the downstream processing pipeline.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - **Control & Timing Interface:**
    - `clk` (input): Clock signal.
    - `rst` (input): Active-high asynchronous reset signal.
  - **Configuration Interface:**
    - `cfg` (input): Operation mode configuration signal (0 = FIFO, 1 = CB).
    - `lmt[31:0]` (input): Programmable limit defining the active operational boundary of the buffer.
- **Master Interface:**
  - **Control Interface:**
    - `psh` (input): Push enable signal asserted by the master to perform a push operation.
    - `pop` (input): Pop enable signal asserted by the master to perform a pop operation.
  - **Input Channel Interface:**
    - `din[DW-1:0]` (input): Write data bus driven by the master, conveying incoming data elements.
  - **Output Channel Interface:**

    * `dout[DW-1:0]` (output): Read data bus continuously driving the element at the current read pointer position.

  - **Status Interface:**
    * `lvl[31:0]` (output): Current occupancy level indicating stored element quantity.
    * `emp` (output): Active-high status signal indicating the buffer is empty.
    * `fll` (output): Active-high status signal indicating the buffer is full.

The pointer bit-width (`PW`) required to address individual storage locations within the buffer is determined logarithmically based on the buffer's capacity:

$$\texttt{PW} = \log_2(\texttt{DPT}) \tag{4.4}$$

The tracking of the next position to write or read is maintained by internal write and read pointers. The write pointer indicates the destination entry for an incoming push operation, while the read pointer targets the current entry available for a pop operation. Both pointers advance sequentially through the buffer memory as operations occur and automatically wrap back to zero once they reach the designated boundary limit (`lmt`), ensuring continuous pointer rotation.

During a push transaction, an incoming data element on `din` is written into the location designated by the write pointer, after which the write pointer advances to the next position and the occupancy counter increments. During a pop transaction, the data element currently indexed by the read pointer is forwarded out on `dout`, after which the read pointer advances and the occupancy counter decrements. If push and pop operations occur simultaneously, the occupancy level remains unchanged.

The behavioral distinction between the two configuration modes lies in how the buffer manages occupancy and data retention:

- **FIFO Mode:** Operates as a standard elastic queue where data elements are consumed upon being read. The level counter dynamically tracks the net difference between pushed and popped elements, asserting `emp` when empty and `fll` when full capacity is reached.
- **CB Mode:** Designed for data reuse. When the buffer is configured in circular mode, the data is initially loaded up to the configured boundary limit (`lmt`). Once the write pointer completes a full pass across this limit, an internal circular completion state is set. This locks the occupancy counter at the configured limit while allowing the read and write pointers to continue cycling, enabling continuous operand recycling without losing buffer state synchronization.

4.2 Memory-Coupled Computing Engine

4.2.1 Architecture Overview

The MC-CE serves as the baseline comparison model designed to evaluate the performance acceleration obtained by the memory-decoupled approach with the SE. Unlike stream-decoupled architectures, the MC-CE tightly orchestrates execution by directly issuing memory access requests to fetch individual data elements required for each computation phase from physical memory and writing output results back to memory once all necessary compute stages finalize.

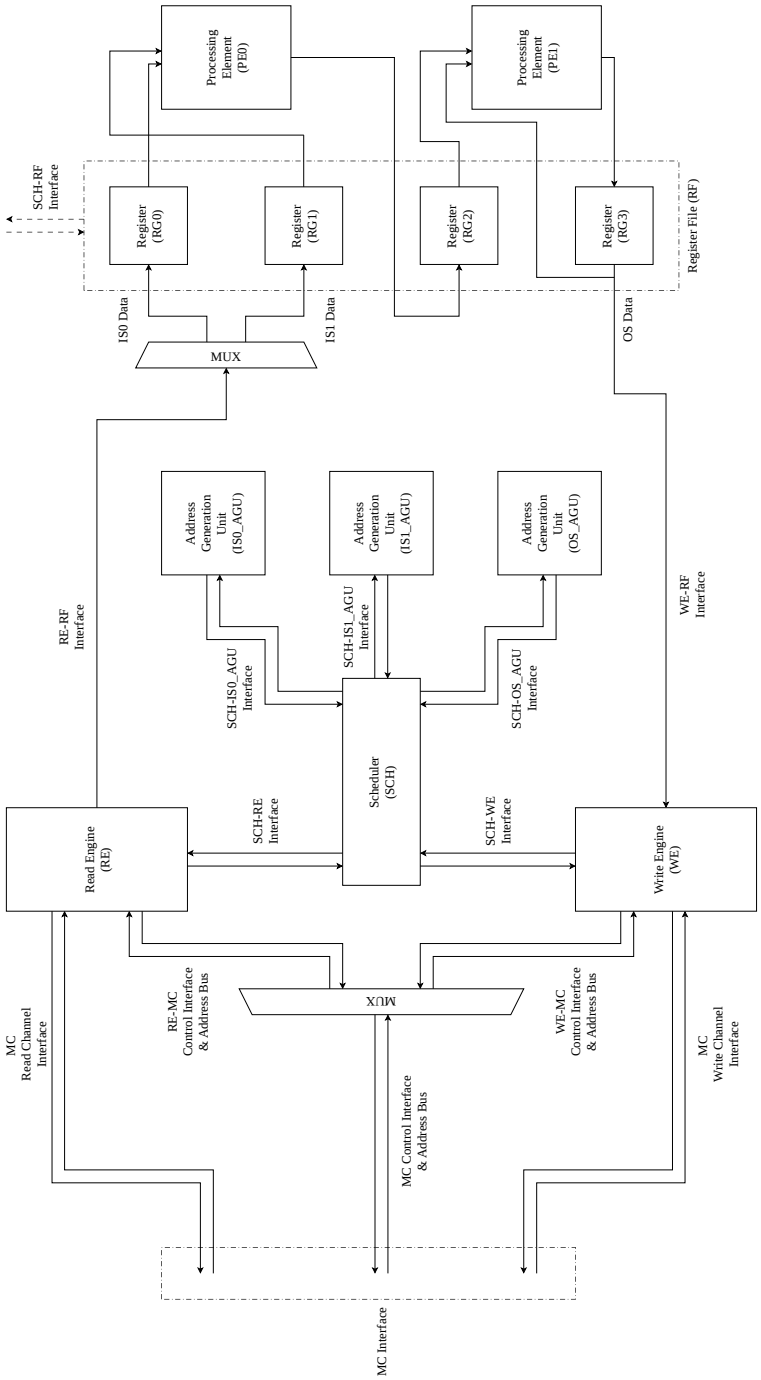


Figure 4.17: MC-CE - Architecture Overview.

As illustrated in Figure 4.17, the structural architecture of the MC-CE consists of several main hardware blocks:

- **SCH:** Acts as the central control state machine managing memory request sequences, inter-module control handshaking, address generation triggers, and RF control signals.

- **AGUs (IS0 AGU, IS0 AGU and OS AGU):** Dedicated hardware units that calculate target memory addresses for IS0, IS1, and the OS based on multi-nested loop configurations.
- **RE and WE:** Memory communication units interfacing with the downstream MC to perform physical data fetch and store transactions.
- **RF:** Contains internal Registers (RGs) (RG0 - RG3) that store retrieved input data and buffer intermediate compute pipeline transitions.
- **PEs (PE0 and PE1):** Execution units configured to perform arithmetic pipeline operations across data stored within the RF.

4.2.2 Sub-Module Description

4.2.2.1 Scheduler

The SCH is the central control unit responsible for orchestrating the overall execution sequence, memory transaction issuance, address generation synchronization, and pipeline control within the MC-CE. Driven by a dedicated state machine, the SCH directly coordinates input element fetching via the RE for both input streams (IS0 and IS1), enables PE calculation stages, triggers output write-backs via the WE, and manages register resetting between computation phases.

The structural characteristics are configured through one static parameter:

- `AW` (Address Width): Defines the bit-width of target address buses driven to the AGUs, the RE, and the WE.

The SCH's block diagram is presented below.

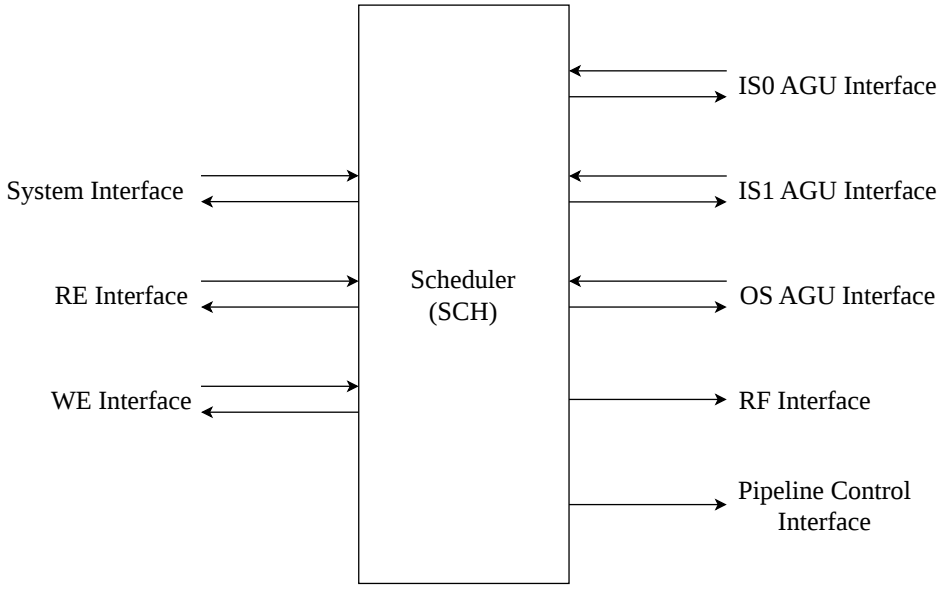


Figure 4.18: MC-CE - SCH Block Diagram.

The physical ports are organized into eight primary functional interfaces.

The **System Interface** handles fundamental operational timing, activation, configuration and status tracking for the CE, partitioned into a Control & Timing Interface, Configuration Interface and a Status Interface.

The **IS0 AGU Interface** connects directly to the IS0 AGU, issuing request triggers to fetch target physical addresses for reads.

The **IS1 AGU Interface** connects directly to the IS1 AGU, issuing request triggers to fetch target physical addresses for reads.

The **OS AGU Interface** connects directly to the OS AGU, issuing request triggers to fetch target physical addresses for writes.

The **RE Interface** connects directly to the RE, where the SCH acts as a master issuing memory access requests to fetch single data elements from main memory.

The **WE Interface** connects directly to the WE, where the SCH acts as a master issuing writeback requests to store calculated results to main memory.

The **RF Interface** manages load enabling and reset cycles for individual internal registers within the RF.

Finally, the **Pipeline Control Interface** provides active multiplexer selection signals that route memory engine control and stream targeting.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**

  - **Control & Timing Interface:**
    - `clk` (input): Clock signal.
    - `rst` (input): Active-high asynchronous reset signal.
    - `en` (input): Enable signal initiating SCH execution.
  - **Configuration Interface:**
    - `nt[31:0]` (input): Number of input transfers per compute phase/output.
    - `no[31:0]` (input): Number of output transfers/outputs.
  - **Status Interface:**
    - `bsy` (output): Status signal indicating the engine is actively executing.
    - `dn` (output): Completion pulse signal indicating processing operations have completed.

- **IS0 AGU Interface:**
  - **Control Interface:**
    - `is0_agu_req` (output): Request trigger signal to the IS0 AGU.
  - **Address Bus:**
    - `is0_agu_adr[AW-1:0]` (input): Target memory address calculated by the IS0 AGU.

- **IS1 AGU Interface:**
  - **Control Interface:**
    - `is1_agu_req` (output): Request trigger signal to the IS1 AGU.
  - **Address Bus:**
    - `is1_agu_adr[AW-1:0]` (input): Target memory address calculated by the IS1 AGU.

- **OS AGU Interface:**
  - **Control Interface:**
    - `os_agu_req` (output): Request trigger signal to the OS AGU.
  - **Address Bus:**
    - `os_agu_adr[AW-1:0]` (input): Target memory address calculated by the OS AGU.

- **RE Interface:**
  - **Control Interface:**
    - `re_req` (output): Request signal asserted to initiate a read transaction.

        * `re_mod` (output): Read access mode configuration (0 = Single, 1 = Burst).
    - **Address Bus:**
        * `re_adr[AW-1:0]` (output): Target base memory address driven to the RE.
    - **Status Interface:**
        * `re_ack` (input): Completion acknowledgment signal returned by the RE.

- **WE Interface:**
    - **Control Interface:**
        * `we_req` (output): Request signal asserted to initiate a write transaction.
        * `we_mod` (output): Write access mode configuration (0 = Single, 1 = Burst).
    - **Address Bus:**
        * `we_adr[AW-1:0]` (output): Target base memory address driven to the WE.
    - **Status Interface:**
        * `we_ack` (input): Completion acknowledgment signal returned by the WE.

- **RF Interface:**
    - **RG0 Interface:**
        * **Control Interface:**
            · `rf_rg0_rst` (output): Reset control signal for RG0.
    - **RG1 Interface:**
        * **Control Interface:**
            · `rf_rg1_rst` (output): Reset control signal for RG1.
    - **RG2 Interface:**
        * **Control Interface:**
            · `rf_rg2_rst` (output): Reset control signal for RG2.
            · `rf_rg2_en` (output): Enable control signal for RG2.
    - **RG3 Interface:**
        * **Control Interface:**
            · `rf_rg3_rst` (output): Reset control signal for RG3.
            · `rf_rg3_en` (output): Enable control signal for RG3.

- **Pipeline Control Interface:**
    - `sel_egn` (output): Engine multiplexer select signal determining which engine gets access to the MC's Control Interface and address bus to issue its memory request (0 = RE, 1 = WE).

  - `sel_is` (output): Input stream select signal determining current stream targeting (0 = IS0, 1 = IS1).

The overall operation of the SCH is managed by an internal 11-state FSM. The primary purpose of this state machine is to coordinate data fetching across input streams, orchestrate compute pipeline progression, trigger memory writebacks, and reset state registers. The defined operational states are:

- `IDL` (Idle): Default state awaiting activation via the `en` signal.
- `REQ_RD0` (IS0 Read Request): Triggers the IS0 AGU for address calculation and issues a single-element read request to the RE.
- `RD0` (IS0 Read Active): Grants bus access to the RE (`sel_egn=0`, `sel_is=0`) and awaits read acknowledgment.
- `REQ_RD1` (IS1 Read Request): Triggers the IS1 AGU for address calculation and issues a single-element read request to the RE.
- `RD1` (IS1 Read Active): Grants bus access to the RE (`sel_egn=0`, `sel_is=1`) and awaits read acknowledgment.
- `CMP` (Compute Step): Enables pipeline registers (RG2 and RG3) to advance computational results across active elements.
- `DRN` (Drain Stage): Multi-cycle latency stage waiting for remaining pipeline operation cycles to settle.
- `REQ_WR` (Write Request): Triggers the OS AGU for address calculation and issues a write-back request to the WE.
- `WR` (Write Active): Grants bus access to the WE (`sel_egn=1`) and awaits memory write-back completion.
- `RST` (Reset Stage): Asserts reset signals across all RF registers (RG0 - RG3) to prepare for the next operational loop.
- `DN` (Done): Asserts global completion signal (`dn=1`) upon finishing all output iterations.

The SCH's FSM diagram is presented in Figure 4.19.

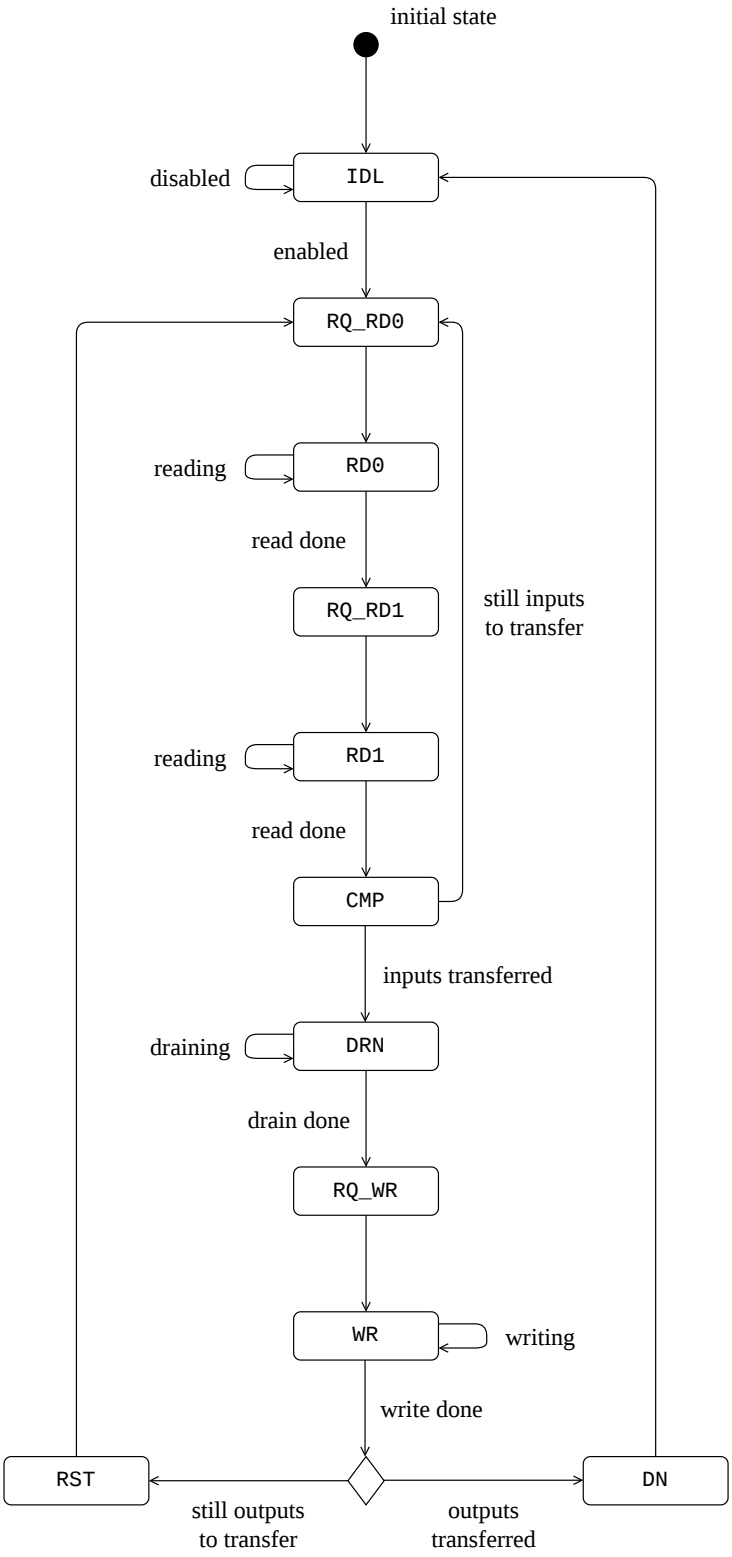


Figure 4.19: SCH - FSM Diagram.

The SCH rests in the IDL state until the enable signal (en) is asserted, whereupon it advances to REQ_RD0. In REQ_RD0, the SCH triggers the IS0 AGU via is0_agu_req=1, issues a single-element read request (re_req=1, re_mod=0) with address re_adr = is0_agu_adr, and transitions to RD0. In RD0, it routes data control paths to IS0 (sel_egn=0, sel_is=0) until the RE completes (re_ack=1), transitioning to REQ_RD1.

In REQ_RD1, the SCH similarly triggers the IS1 AGU (is1_agu_req=1) and requests the IS1 element from physical memory via the RE. Upon transitioning to RD1, it sets sel_is=1 until re_ack=1 is received.

Once both input operands are retrieved, the state machine enters CMP, asserting register write enables (rf_rg2_en=1, rf_rg3_en=1) to compute intermediate pipeline results. If the input transfer counter has not reached the total input count bound (nt-1), the state machine increments the counter and loops back to REQ_RD0 to fetch the next input pair.

When all required input elements for a compute stage are processed, the SCH enters DRN for a fixed number of cycles to drain remaining internal pipeline transitions. It then transitions to REQ_WR, triggering the OS AGU (os_agu_req=1) and initiating a single-element writeback request to the WE (we_req=1, we_mod=0, we_adr = os_agu_adr). In WR, the SCH sets sel_egn=1 until writeback completes (we_ack=1).

If the output counter reaches the final target bound (no-1), the SCH transitions directly to DN, pulsing dn=1 and returning to IDL. Otherwise, it enters RST to reset all internal registers of the register file before returning to REQ_RD0 to execute the next output iteration phase.

4.2.2.2 Read Engine

The RE within the MC-CE performs the same underlying operation as the RE described in the SE architecture, handling memory access handshaking, address latching, and memory response tracking across single and burst transfer modes.

The primary architectural difference resides in its local target interface: rather than forwarding retrieved memory data to the SC or updating tracking status flags in the TT, the RE directly drives the internal RF.

- **RF Interface:** Drives write-enable signals (rf_rg_en) and raw data buses (rf_rg_in) directly into target registers within the RF (RG0 and RG1).

4.2.2.3 Write Engine

The WE within the MC-CE performs the same underlying operation as the WE described in the SE architecture, handling memory access handshaking, address latching, and physical write transfers across single and burst transfer modes.

The primary architectural difference resides in its local source interface: rather than retrieving output data from the OS SB, the WE directly samples data from the internal RF.

- **RF Interface:** Receives output data lines (`rf_rg_out`) directly from target registers within the RF (RG3) to register write data prior to issuing physical memory transactions.

4.2.2.4 Processing Element

The PE acts as the core execution unit responsible for performing reconfigurable arithmetic and logical operations. It consists of input-selection multiplexing logic paired with an Extended Compute Unit (XCU) execution core.

Each PE encapsulates two primary functional stages:

- **Operand Selection Multiplexers:** Driven by selection control signals (`sel_a` and `sel_b`), these multiplexers routes data operands to internal execution buses `a` and `b`. Inputs can be dynamically selected from external input data streams (`in0–in4`) or directly from internal RF feedback lines (`rf_to_ma` and `rf_to_mb`).
- **XCU:** Serves as the combinational ALU executing `DW`-bit operations configured by an operation select line (`op`). Supported operations include basic arithmetic addition and subtraction, signed multiplication, bitwise logical operations, and arithmetic and logical shift operations.

The resulting computed output is continuously driven to the `out` bus, making it immediately available to be routed back into the RF or forwarded to downstream compute stages. The internal architecture of the PE is presented in the Figure 4.20.

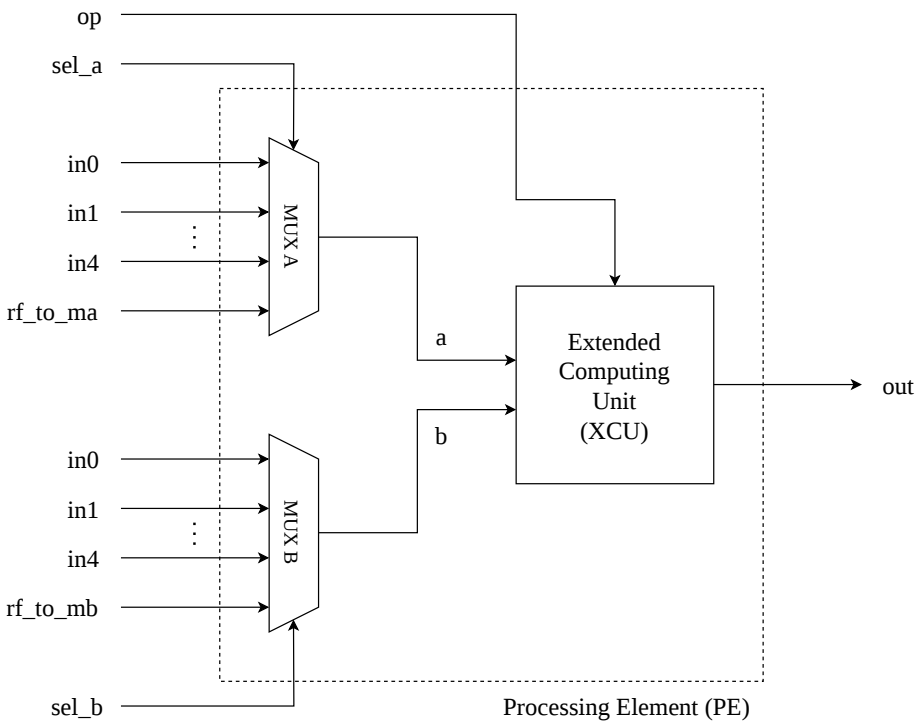


Figure 4.20: PE - Architecture Overview.

4.2.2.5 Register File

The RF serves as local storage for holding input operands, intermediate calculation values, and final execution outputs across compute cycles.

It is instantiated as a collection of four independent, parameterized register modules (RG0 - RG3). Each register operates synchronously on the clock edge, featuring dedicated write-enable (`rgX_en`) and asynchronous reset controls (`rgX_rst`) managed by the SCH. This modular structure allows independent sampling, buffering, and driving of data streams (`rgX_in` and `rgX_out`) between the PEs and the memory interfacing modules like the RE and WE.

4.3 Memory-Decoupled Computing Engine

4.3.1 Architecture Overview

The MD-CE is a specialized processing unit designed to perform high-throughput compute operations decoupled from memory access latencies. Driven by a dedicated control state machine, it accepts dual stream inputs (IS0 and IS1), processes data through a multi-stage PE pipeline, and buffers results into an internal RF before transferring the final output to the OS.

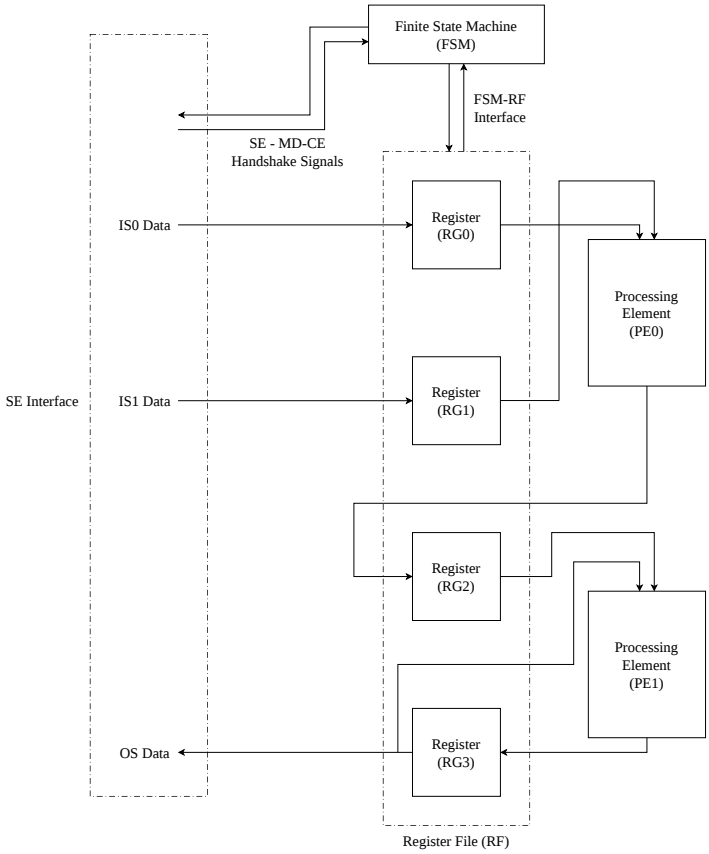


Figure 4.21: MD-CE - Architecture Overview.

As illustrated in Figure 4.21, the structural architecture of the MD-CE consists of three main hardware blocks:

- **FSM:** Manages global timing, stream handshake protocols with the SE, loop iteration counters, and control signals (enables and resets) for individual registers in the RF.
- **RF:** Contains four internal registers (RG0 - RG3) that store intermediate operands and stage pipeline data transitions between compute stages.
- **PEs (PE0 and PE1):** Execution units configured to perform arithmetic pipeline operations across data stored within the RF.

Communication with the SE is synchronized via dedicated handshake signals (`in_vld`, `in_rdy`, `out_vld`, and `out_rdy`). Since the PE and the RF architectures are the same as the ones described in the MC-CE architecture, then it's only presented the details of the FSM.

4.3.2 Sub-Module Description

4.3.2.1 Finite State Machine

The FSM serves as the primary control unit of the MD-CE, orchestrating data ingestion handshakes, execution pipeline register latching, pipeline draining, and output stream flow.

The block diagram of the FSM is presented below.



Figure 4.22: MD-CE - FSM Block Diagram.

The physical ports are organized into three primary functional interfaces.

The **System Interface** handles fundamental operational timing, configuration, global control, and status tracking for the state machine, providing clocking, synchronous reset, activation control, and execution status signals.

Finally, the **RF Interface** forms the internal control path where the FSM dynamically drives reset and enable strobes to manage pipeline stage transitions across the RF registers.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - **Control & Timing Interface:**
    - `clk` (input): Clock signal.
    - `rst` (input): Active-high asynchronous reset signal.
    - `en` (input): Enable signal.
    - `in_rdy` (output): Backpressure control signal driven high when ready to accept input data.
    - `in_vld` (input): Valid signal asserted by the SE when valid input data is available.
    - `out_rdy` (input): Backpressure control signal driven by the SE accepting output data.
    - `out_vld` (output): Valid signal asserted high when valid output data is ready.
  - **Configuration Interface:**
    - `nt[31:0]` (input): Total input transfers per output computation to perform.
    - `no[31:0]` (input): Total number of output transfers to perform.
  - **Status Interface:**
    - `bsy` (output): Status signal indicating the CE is active.

        * `dn` (output): Single-cycle completion signal indicating compute operation termination.

- **RF Interface:**
  - **RG0 Interface:**
    * **Control Interface:**
      · `rf_rg0_rst` (output): Synchronous reset signal driven to RG0.
      · `rf_rg0_en` (output): Write-enable control signal driven to RG0.
  - **RG1 Interface:**
    * **Control Interface:**
      · `rf_rg1_rst` (output): Synchronous reset signal driven to RG1.
      · `rf_rg1_en` (output): Write-enable control signal driven to RG1.
  - **RG2 Interface:**
    * **Control Interface:**
      · `rf_rg2_rst` (output): Synchronous reset signal driven to RG2.
      · `rf_rg2_en` (output): Write-enable control signal driven to RG2.
  - **RG3 Interface:**
    * **Control Interface:**
      · `rf_rg3_rst` (output): Synchronous reset signal driven to RG3.
      · `rf_rg3_en` (output): Write-enable control signal driven to RG3.

The execution of compute operations is managed by an internal 6-state FSM. The defined operational states are:

- `IDL` (Idle): Default state awaiting execution initiation via the global enable signal.
- `IN` (Data Ingestion): Ingests dual input data streams and latches intermediate execution results into pipeline registers.
- `DRN` (Pipeline Drain): Holds pipeline register enables active for fixed clock cycles to flush multi-stage arithmetic latency paths.
- `OUT` (Output): Transmits computed stream results to the output stream interface using standard valid/ready handshaking.
- `RST` (Pipeline Reset): Asserts asynchronous resets across internal pipeline registers to prepare for subsequent block transfers.
- `DN` (Done): Asserts global completion signal signaling task termination.

The FSM's FSM diagram is presented in Figure 4.23.

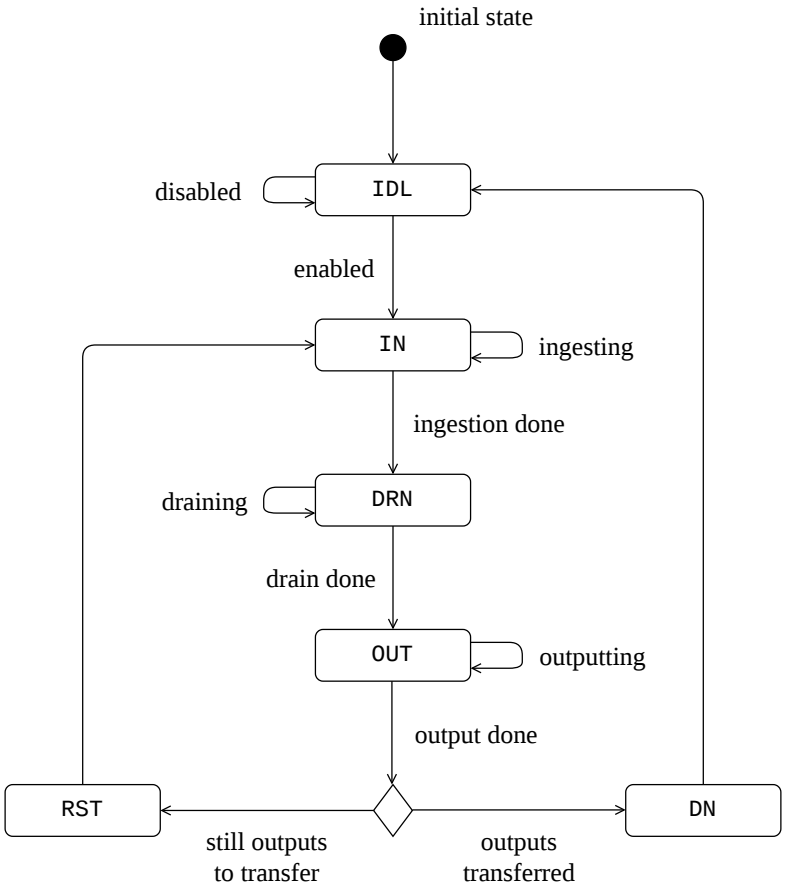


Figure 4.23: FSM - FSM Diagram.

When the CE is in the `IDL` state, it continuously monitors the enable signal (`en`). Initiating execution requires the system to assert `en=1`, which causes the FSM to transition directly to the `IN` state.

During data ingestion in the `IN` state, the engine asserts the input ready signal (`in_rdy=1`) to notify the SE of its readiness to receive incoming stream operands. Data transfers occur whenever a valid handshake takes place (`in_xfr = in_vld & in_rdy`). On every successful handshake cycle, the FSM asserts write-enable controls to capture active PE computation outputs into pipeline registers `RG2` and `RG3` (`rf_rg2_en=1` and `rf_rg3_en=1`). An internal input transfer counter tracks the number of input transfers, and once the counter reaches $\texttt{nt} - 1$, the counter resets and the FSM advances to the `DRN` state.

In the `DRN` state, the FSM holds register enables (`rf_rg2_en=1` and `rf_rg3_en=1`) high while tracking an internal clock cycle counter. This holds the datapath active for a designated cycle latency to allow multi-stage arithmetic operations within the PE blocks to completely finalize and

flush intermediate results into the pipeline registers. Upon expiration of the cycle counter, the FSM transitions to the `OUT` state.

In the `OUT` state, the FSM drives the output valid signal (`out_vld=1`), exposing the computed result from register RG3 onto the output stream bus. Data elements are transferred upon successful handshake cycles (`out_xfr = out_vld & out_rdy`), which increments an internal output transfers counter. Once the counter reaches `no` $-$ 1, the FSM advances to the `DN` state; otherwise, a successful transfer advances the FSM to the `RST` state.

In the `RST` state, the FSM asserts the asynchronous active-high reset signals across all internal registers in the RF. This flushes all stored intermediate register state values in a single clock cycle before automatically transitioning back to the `IN` state to ingest the next set of stream data elements.

Upon transferring the final output element, the state machine enters the `DN` state, pulsing the global completion signal (`dn=1`) high for a single clock cycle before returning to `IDL` to await subsequent operational triggers.

4.4 Memory Controller

4.4.1 Module Description

The MC serves as an intermediate abstraction layer between physical memory storage (such as the MB) and any hardware master attempting to access memory. Its primary purpose is to simplify memory interactions by translating high-level single or burst commands into physical memory access sequences, while absorbing fixed operational latencies.

The structural characteristics are configured through four static parameters:

- `AW` (Address Width): Defines the bit-width of the address buses.
- `DW` (Data Width): Defines the bit-width of a single data element on both memory access and physical memory data buses.
- `LEN` (Burst Length): Defines the total number of data elements transferred during a burst transaction.
- `LAT` (Access Latency): Defines the fixed delay duration (in clock cycles) used to model physical memory access time.

The MC's block diagram is presented below.

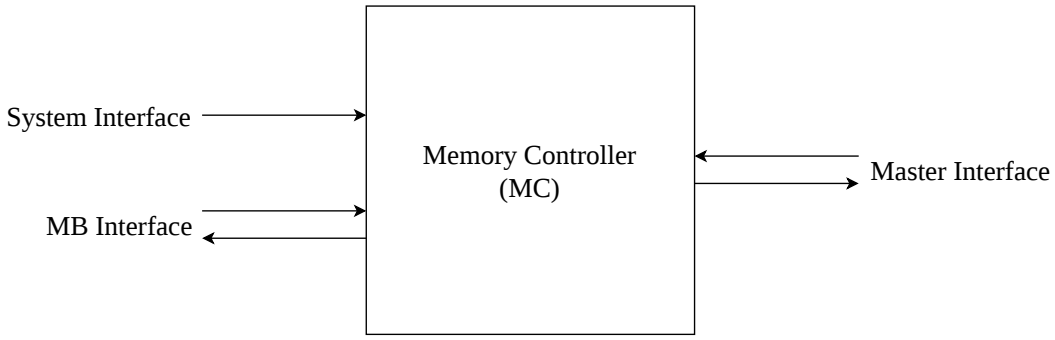


Figure 4.24: MC Block Diagram.

The physical ports are organized into three primary functional interfaces.

The **System Interface** handles fundamental operational timing and control, providing global clocking and reset distribution to the module.

The **Master Interface** establishes the upstream communication boundary where the controller acts in a slave role to receive access commands, incoming write streams, and dispatch read results to an attached processing master.

Finally, the **MB Interface** forms the downstream physical bus where the MC acts in a master role to directly drive address and control signals to memory and manage low-level data transfers.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - `clk` (input): Clock signal.
  - `rst` (input): Active-high asynchronous reset signal.
- **Master Interface:**
  - **Control Interface:**
    - `req` (input): Request signal asserted by a master to initiate a transaction.
    - `typ` (input): Operation type selection (0 = Read, 1 = Write).
    - `mod` (input): Transaction mode configuration (0 = Single, 1 = Burst).
  - **Common Address Bus:**
    - `adr[AW-1:0]` (input): Target memory base address bus driven by the master.
  - **Status Interface:**
    - `ack` (output): Single-cycle completion acknowledgment signal returned to the master.
    - `bsy` (output): Status flag indicating the controller is actively servicing a transaction.

  - **Read Channel Interface:**
    * `out_rdy` (input): Backpressure control signal driven by the master accepting the read data.
    * `out_vld` (output): Valid signal driven by the controller when read data is available.
    * `rd_dt[DW-1:0]` (output): Read data bus driven by the controller.
  - **Write Channel Interface:**
    * `in_rdy` (output): Backpressure control signal driven by the controller to accept write data.
    * `in_vld` (input): Valid signal driven by the master indicating write data availability.
    * `wr_dt[DW-1:0]` (input): Write data bus driven by the master.

- **MB Interface:**
  - **Control Interface:**
    * `rd` (output): Physical memory read enable signal.
    * `wr` (output): Physical memory write enable signal.
  - **Common Address Bus:**
    * `mb_adr[AW-1:0]` (output): Physical target address bus routed to physical memory.
  - **Read Channel Interface:**
    * `mb_rd_dt[DW-1:0]` (input): Physical read data bus returning from physical memory.
  - **Write Channel Interface:**
    * `mb_wr_dt[DW-1:0]` (output): Physical write data bus driven to physical memory.

The execution of memory operations is managed by an internal 6-state FSM. The defined operational states are:

- `IDL` (Idle): Default state awaiting incoming memory transaction requests.
- `RD` (Read): Reads data elements directly from physical memory into the internal read register array.
- `IN` (Input): Ingests incoming write payload elements from the write channel interface into the internal write register array.
- `HLD` (Hold): Pauses execution for `LAT` clock cycles to model physical memory access latency.

- `OUT` (Output): Drives fetched read payload elements from the internal read register array out through the read channel interface to the requesting master.
- `WR` (Write): Flushes stored write payload elements from the internal write register array out to physical memory.

The MC's FSM diagram is presented in Figure 4.25.

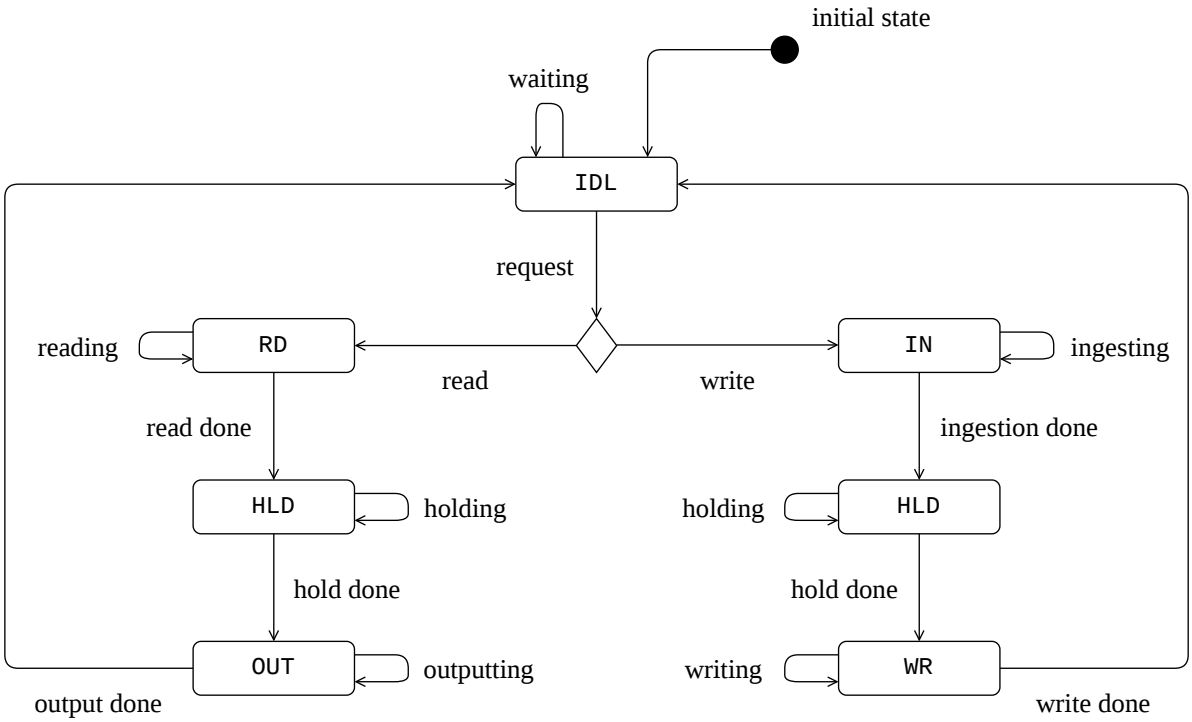


Figure 4.25: MC - FSM Diagram.

When the MC is in the `IDL` state, it continuously monitors the incoming request signal (`req`). Initiating a transaction requires the master to assert `req` as a single-cycle pulse accompanied by a payload of request parameters containing the operation type (`typ`), transaction mode (`mod`), and target base address (`adr`). During this single active clock cycle, the MC captures and registers this transaction package internally. Outside of this single-cycle request assertion, the incoming control signals remain in their default unasserted states.

Depending on the requested operation type, a read request (`typ=0`) causes an immediate transition to the `RD` state, whereas a write request (`typ=1`) causes a transition to the `IN` state to begin gathering incoming data.

During a read transaction in the `RD` state, the MC asserts the physical memory read enable signal (`rd`) and sequentially increments the target physical address (`mb_adr`) starting from the registered base address. As memory returns data on `mb_rd_dt`, the MC stores each element

into its internal read register array. The total number of elements transferred (n) depends on the transaction mode captured during the request: single mode (mod=0) fetches $n = 1$ element of width DW, while burst mode (mod=1) fetches $n =$ LEN elements of width DW. Once all required elements are fetched, the FSM advances to the HLD state.

During a write transaction in the IN state, the MC asserts its input ready signal (in_rdy) to accept data arriving on the write data bus (wr_dt) from the requesting master. Data transfers occur whenever both valid and ready signals are high (in_vld and in_rdy), capturing n elements ($n = 1$ for single mode, or $n =$ LEN for burst mode) into the internal write register array. Once the full set of elements has been ingested, the FSM advances to the HLD state.

In the HLD state, the FSM holds execution for LAT clock cycles to simulate physical memory access delay. Upon expiration of this latency counter, read transactions branch to the OUT state, while write transactions advance to the WR state.

In the OUT state, the MC transmits the stored read elements from its internal read register array to the master over the read data bus (rd_dt) using standard valid/ready handshaking (out_vld and out_rdy). In the WR state, the MC asserts the physical memory write enable signal (wr) and sequentially writes the stored data elements from its internal write register array out to physical memory (mb_wr_dt). Upon transferring the final element of either a read or write sequence, the MC asserts a single-cycle completion acknowledgment signal (ack) and returns to the IDL state, ready to service subsequent requests.

4.5 Memory Block

4.5.1 Module Description

The MB serves as the primary physical storage within the system. Unlike standard byte-addressable memories where each address points to an individual byte, this module is designed as an element-addressable storage unit. Structurally, it can be visualized as a vertical stack of DPT data elements, where each entry stores a full data element of width DW.

In this scheme, a single physical address maps directly to exactly one complete data element. The memory stack spans contiguous addresses starting vertically from a base low address (LA) up to a high address (HA), where incrementing an address by 1 selects the next consecutive DW-wide element in the vertical stack.

The structural characteristics, memory capacity, and physical addressing boundaries are configured through four static parameters:

- AW (Address Width): Defines the bit-width of the address bus (adr).
- DW (Data Width): Defines the bit-width of a single data element on both read and write data buses.
- DPT (Memory Depth): Defines the total number of storage locations available in the internal memory array.

- `LA` (Low Address): Specifies the starting base address of the physical address space assigned to this MB.

Based on these parameters, the module internally computes a local parameter `HA` (High Address), defined as

$$\texttt{HA} = \texttt{LA} + \texttt{DPT} - 1 \tag{4.5}$$

allowing the establishment of the address range $[\texttt{LA}, \texttt{HA}]$. Incoming requests are validated by checking if the target address falls within this window, where the address's validity is stored in an internal register, `adr_vld`. If valid, the physical address is mapped to a zero-based internal array index (`idx`) using index translation,

$$\texttt{idx} = \texttt{adr} - \texttt{LA} \tag{4.6}$$

The MB's block diagram is presented below.

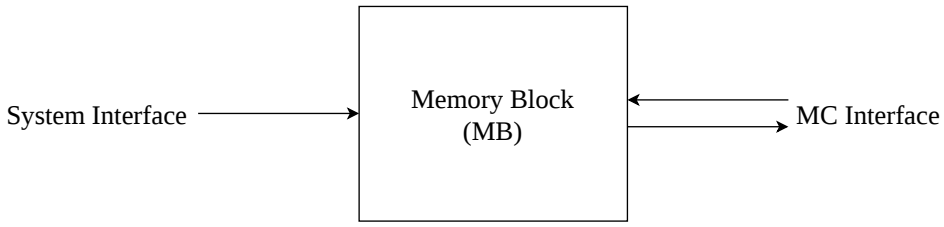


Figure 4.26: MB Block Diagram.

The physical ports are categorized into two primary functional interfaces.

The **System Interface** handles fundamental operational timing and control, supplying global clocking and reset distribution.

The **MC Interface** forms the physical upstream connection where the MB operates strictly in a slave role, receiving target physical addresses, control commands (read/write enables), and incoming write data from the MC, while returning stored payload data back to the controller.

The complete port breakdown across these interfaces is structured as follows:

- **System Interface:**
  - `clk` (input): Clock signal.
  - `rst` (input): Active-high asynchronous reset signal.
- **MC Interface:**
  - **Control Interface:**
    - `rd` (input): Physical memory read enable signal.

    * `wr` (input): Physical memory write enable signal.

  - **Common Address Bus:**
    * `adr[AW-1:0]` (input): Physical address bus driven by the MC.
  - **Read Channel Interface:**
    * `rd_dt[DW-1:0]` (output): Physical read data bus returning to the MC.
  - **Write Channel Interface:**
    * `wr_dt[DW-1:0]` (input): Physical write data bus driven by the MC.

The operational behavior of the MB is governed by synchronous write logic and combinational read/hit evaluation logic. Upon an active reset (`rst=1`), every entry in the internal storage array (`mem`) is cleared to zero across all `DPT` locations.

During normal operation, write accesses occur synchronously on the rising edge of the system clock. When the write enable signal is asserted (`wr=1`) and the address is validated (`adr_vld=1`), the data element on `wr_dt` is stored into the target array location `mem[idx]`.

Read operations, address decoding, and hit status evaluations are handled combinationally. When the read enable signal is active (`rd=1`) with a valid address, the data element stored at `mem[idx]` is driven immediately onto the read data bus `rd_dt`. If the read enable signal is deasserted or the target address falls outside the configured boundary (`adr_vld=0`), `rd_dt` is driven to zero. Additionally, an internal status signal (`hit`) evaluates to high whenever a read or write operation targets an address within the valid range $[\texttt{LA}, \texttt{HA}]$.

Chapter 5

Benchmarks and Evaluation Metrics

This chapter presents the workload benchmarks selected to evaluate the streaming acceleration capabilities of the proposed architecture and details the mathematical framework used for hardware performance quantification.

Section 5.1 describes the algorithmic structures of the four chosen benchmarks: FIR Filter, Matrix Multiplication, 2D Convolution, and 3D Convolution. Section 5.2 defines the evaluation metrics used to assess the performance of both CEs, in order to establish the performance relationships between the baseline and proposed architectures.

5.1 Benchmarks

5.1.1 FIR Filter

The FIR filter is a fundamental operator in DSP, representing a linear time-invariant system used for spectral filtering, signal smoothing, and feature extraction. The filtering operation computes a discrete output signal sequence y by continuously evaluating a weighted inner product between a static filter vector h and an input signal array x.

A single output sample $y[n]$ at discrete time step n is mathematically defined as:

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k] \tag{5.1}$$

where:

- x is the 1D input signal array of size N_S.
- h is the static 1D filter vector of size N.
- y is the 1D output signal array of size N_S.
- N_S is the signal length of the input array x and output array y.
- N is the filter length (number of coefficient taps) of the filter vector h.

- k is the local index traversing the filter taps ($0 \leq k \leq N-1$).

Processing the complete filtering workload requires coordinating three distinct data streams:

- **Input Signal Stream (S_x):** Stream of the data samples from the input signal array x.
- **Filter Stream (S_h):** Stream of the coefficient taps from the filter vector h.
- **Output Signal Stream (S_y):** Stream of the generated samples for the output signal y.

The memory access pattern of S_x corresponds to a *1D sliding window* pattern. To evaluate a single output sample $y[n]$, a contiguous window of N input samples, defined by the set $\{x[n], x[n-1], \ldots, x[n-(N-1)]\}$, is retrieved from memory. As execution advances from $y[n]$ to $y[n+1]$, the window slides along the 1D address space by a unit stride of 1 element. Across successive time steps, $N-1$ elements are retained and reused within the window, creating significant spatial and temporal locality.

The memory access pattern of S_h corresponds to a *1D linear* pattern. For every calculated output sample $y[n]$, the entire coefficient vector $h = \{h[0], h[1], \ldots, h[N-1]\}$ is read sequentially from index 0 to $N-1$ with a unit stride. Once the N-th tap is processed, the memory pointer resets to the base address of h to prepare for the next output calculation. Because the coefficient values remain constant throughout the execution lifetime, S_h exhibits complete temporal data reuse across all N_S output samples.

The memory access pattern of S_y corresponds to a *1D linear* pattern. As each scalar output sample $y[n]$ is computed, it is written directly to main memory at contiguous address locations with a unit stride, sequentially filling the output address space until all N_S output samples are written.

5.1.2 Matrix Multiplication

Matrix Multiplication is a foundational linear algebra algorithm with 2D memory access requirements, extensively utilized in scientific computing, transformations, and deep learning layers. The computation evaluates the row-by-column inner product between two 2D matrices to produce a 2D result matrix.

A single output matrix element $\mathbf{C}[i, j]$ at row i and column j is mathematically defined as:

$$\mathbf{C}[i, j] = \sum_{k=0}^{K-1} \mathbf{A}[i, k] \cdot \mathbf{B}[k, j] \tag{5.2}$$

where:

- $\mathbf{A}$ is the primary 2D input matrix of size $M \times K$.
- $\mathbf{B}$ is the secondary 2D input matrix of size $K \times N$.
- $\mathbf{C}$ is the 2D output matrix of size $M \times N$.
- M, N, K represent the outer and inner matrix dimensions.

- i, j, k are iteration indices traversing dimensions M, N, and K ($0 \leq i \leq M-1, 0 \leq j \leq N-1, 0 \leq k \leq K-1$).

Processing the complete matrix product workload requires coordinating three distinct data streams:

- **Matrix A Stream (S_a):** Stream of data elements from the input matrix **A**.
- **Matrix B Stream (S_b):** Stream of data elements from the input matrix **B**.
- **Matrix C Stream (S_c):** Stream of the computed elements for the output matrix **C**.

The memory access pattern of S_a corresponds to a *2D strided* pattern. While elements within a fixed row i are read with a unit stride as k increments, advancing across rows (from row i to row $i+1$) introduces a row stride equal to the matrix width K. Each row i is sequentially re-read N times to compute all target output columns $j \in [0, N-1]$, exhibiting substantial temporal data reuse across the iteration domain.

The memory access pattern of S_b corresponds to a *2D strided* pattern. Computing an output element requires traversing column j as index k advances from 0 to $K-1$. In standard row-major memory storage, stepping vertically between adjacent elements within column j imposes a non-unit column stride equal to the matrix row length N. Once column j is completely traversed, the access pointer advances horizontally to column $j+1$ with a unit offset, repeating this strided column traversal M times over the execution lifetime.

The memory access pattern of S_c corresponds to a *2D linear* pattern. Output elements $\mathbf{C}[i, j]$ are written sequentially in row-major order with a unit stride as j increments across columns. When a row terminates ($j = N-1$), the write pointer advances seamlessly to the start of the next row ($i = i+1$), filling the $M \times N$ output address space.

5.1.3 2D Convolution

The 2D Convolution kernel extends spatial filtering across two dimensions, serving as a core computational workload in image processing, CV, and Convolutional Neural Networks (CNNs). The operation computes an output feature map by sliding a 2D filter mask across two orthogonal axes of an input matrix.

A single output sample $\mathbf{Y}[i, j]$ at spatial coordinate $[i, j]$ is mathematically defined as:

$$\mathbf{Y}[i, j] = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} \mathbf{H}[m, n] \cdot \mathbf{X}[i+m, j+n] \tag{5.3}$$

where:

- $\mathbf{X}$ is the 2D input map of size $H \times W$.
- $\mathbf{H}$ is the static 2D filter kernel of size $K \times K$.
- $\mathbf{Y}$ is the 2D output map of size $(H-K+1) \times (W-K+1)$.

- H, W are the height and width of the input feature map $\mathbf{X}$, respectively.
- K is the spatial dimension (height and width) of the square filter kernel $\mathbf{H}$.
- m, n are local indices traversing the filter kernel rows and columns ($0 \leq m, n \leq K-1$).

Processing the complete 2D convolution workload requires coordinating three distinct data streams:

- **Input Map Stream (S_x):** Stream of data samples from the 2D input array $\mathbf{X}$.
- **Filter Mask Stream (S_h):** Stream of coefficient weights from the 2D kernel matrix $\mathbf{H}$.
- **Output Map Stream (S_y):** Stream of computed results for the 2D output array $\mathbf{Y}$.

The memory access pattern of S_x corresponds to a *2D sliding window* pattern. Evaluating an output point $\mathbf{Y}[i, j]$ requires retrieving a $K \times K$ sub-grid of input elements. Horizontal window sliding along columns proceeds with a unit stride. When the horizontal limit ends at the end of a row ($j = W - K$), the window steps vertically down to row $i+1$, incurring a row-pitch stride equal to the input width W. Across overlapping rows and columns of adjacent windows, significant 2D spatial data reuse occurs.

The memory access pattern of S_h corresponds to a *2D linear* pattern. Within the inner summation loops, the entire $K \times K$ filter matrix is read sequentially row-by-row with a unit stride. Because the filter mask remains static throughout execution, S_h exhibits complete temporal data reuse as it is repeatedly re-read across all spatial target coordinates $[i, j]$ of the output map.

The memory access pattern of S_y corresponds to a *2D linear* pattern. Computed scalar values $\mathbf{Y}[i, j]$ are written sequentially in row-major order with a unit stride, moving left-to-right across columns and top-to-bottom across rows until the entire 2D output grid is written.

5.1.4 3D Convolution

The 3D Convolution kernel processes volumetric data structures across three orthogonal spatial dimensions. It represents a primary operator in 3D medical imaging, video analysis, and spatial physical simulations. The filtering operation computes a 3D output tensor by traversing a volumetric filter cube through an input volume.

A single output voxel $\mathbf{Y}[d, i, j]$ at depth d, row i, and column j is mathematically defined as:

$$\mathbf{Y}[d,i,j] = \sum_{p=0}^{K-1} \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} \mathbf{H}[p,m,n] \cdot \mathbf{X}[d+p, i+m, j+n] \tag{5.4}$$

where:

- $\mathbf{X}$ is the 3D input tensor of size $D \times H \times W$.
- $\mathbf{H}$ is the static 3D filter cube of size $K \times K \times K$.
- $\mathbf{Y}$ is the 3D output tensor of size $(D-K+1) \times (H-K+1) \times (W-K+1)$.

- D, H, W represent the depth, height, and width of the input tensor $\mathbf{X}$, respectively.
- K is the spatial dimension (depth, height, and width) of the cubic filter $\mathbf{H}$.
- p, m, n are local indices traversing the filter depth, rows, and columns ($0 \leq p, m, n \leq K-1$).

Processing the complete 3D volumetric filtering workload requires coordinating three distinct data streams:

- **Input Tensor Stream (S_x):** Stream of voxels from the 3D input tensor $\mathbf{X}$.
- **Filter Cube Stream (S_h):** Stream of coefficient weights from the 3D filter cube $\mathbf{H}$.
- **Output Tensor Stream (S_y):** Stream of generated voxels for the 3D output tensor $\mathbf{Y}$.

The memory access pattern of S_x corresponds to a *3D sliding window* pattern. Accessing the required $K \times K \times K$ sub-volume combines three distinct memory steps: a unit stride along contiguous horizontal row elements (n), a row-pitch stride (W) when advancing vertical rows (m), and a slice-pitch stride ($H \times W$) when stepping across depth slices (p). Execution tracks two boundary conditions: detecting when the horizontal limit ends at the end of a row ($j = W - K$), and detecting when the vertical limit ends at the bottom of a 2D frame ($i = H - K$) before advancing to the next depth slice. Across overlapping sub-volumes, substantial 3D spatial data reuse occurs.

The memory access pattern of S_h corresponds to a *3D linear* pattern. For each generated output voxel, the entire $K \times K \times K$ coefficient volume is read sequentially from start to end with a unit stride. Because the filter cube remains static throughout execution, S_h exhibits complete temporal data reuse as it is repeatedly re-read across all spatial target coordinates $[d, i, j]$ of the 3D output tensor.

The memory access pattern of S_y corresponds to a *3D linear* pattern. Calculated output voxels $\mathbf{Y}[d, i, j]$ are written contiguously in memory with a unit stride, traversing row-by-row and slice-by-slice until the entire 3D output address space is completely written.

5.2 Performance Evaluation Metrics

To evaluate and compare the execution behavior across the target hardware architectures with the results presented in Chapter 7, a unified set of quantitative performance metrics is mathematically defined. For the simulation time and speedup analysis, the metrics quantify overall execution duration and the resulting relative performance gain achieved by the proposed architecture over the baseline design:

- **Simulation Time (t_{sim}):** Defines the total elapsed time from execution initiation to stream completion:

$$t_{\text{sim,us}} = \frac{n_{\text{clk,sim}} \cdot T_{\text{clk}}}{10^3} \quad [\mu\text{s}] \tag{5.5}$$

  where:

- $n_{\text{clk,sim}}$ is the total number of elapsed clock cycles during simulation [cycles].
- T_{clk} is the system clock period [ns].
- 10^3 is the unit scaling factor converting nanoseconds to microseconds (ns/μs).

- **Speedup (s):** Quantifies the relative execution performance gain achieved by the proposed architecture over the baseline architecture:

$$s = \frac{t_{\text{sim,bsl}}}{t_{\text{sim,prp}}} \tag{5.6}$$

where:

- $t_{\text{sim,bsl}}$ is the total simulation time measured for the baseline architecture [μs].
- $t_{\text{sim,prp}}$ is the total simulation time measured for the proposed architecture [μs].

Regarding the memory access analysis, the metrics characterize external memory communication delays and the operational fetch efficiency of the data transfers:

- **Memory Communication Time (t_{mem}):** Quantifies the cumulative time spent communicating with the external memory system across baseline and proposed configurations:

$$t_{\text{mem,cyc}} = (n_{\text{sg,rd}} + n_{\text{sg,wr}}) \cdot l_{\text{sg}} + (n_{\text{br,rd}} + n_{\text{br,wr}}) \cdot l_{\text{br}} \quad [\text{cycles}] \tag{5.7}$$

$$t_{\text{mem,us}} = \frac{t_{\text{mem,cyc}} \cdot T_{\text{clk}}}{10^3} \quad [\mu\text{s}] \tag{5.8}$$

where:

- $n_{\text{sg,rd}}, n_{\text{sg,wr}}$ are the total number of single read and write requests issued to the MC.
- $n_{\text{br,rd}}, n_{\text{br,wr}}$ are the total number of burst read and write requests issued to the MC.
- l_{sg} is the established fixed latency for single memory access transactions [cycles].
- l_{br} is the established fixed latency for burst memory access transactions [cycles].
- $t_{\text{mem,cyc}}$ is the memory communication time in clock cycles [cycles].
- T_{clk} is the system clock period [ns].
- 10^3 is the unit scaling factor converting nanoseconds to microseconds (ns/μs).

- **Memory Access Factor (MAF):** Quantifies external memory fetch efficiency and bus traffic overhead as the average number of memory elements transferred per generated output element:

$$\text{MAF} = \frac{(n_{\text{sg,rd}} \cdot 1) + (n_{\text{br,rd}} \cdot \text{LEN}) + (n_{\text{sg,wr}} \cdot 1) + (n_{\text{br,wr}} \cdot \text{LEN})}{n_{\text{out}}} \tag{5.9}$$

where:

- $n_{\text{sg,rd}}, n_{\text{sg,wr}}$ are the total number of single read and write requests issued to the MC.

- $n_{\text{br,rd}}, n_{\text{br,wr}}$ are the total number of burst read and write requests issued to the MC.
- LEN is the configured operational burst length.
- n_{out} is the total number of processed output elements.

This metric reflects data reusability during processing; a lower MAF directly indicates higher data reuse within the local buffers, avoiding redundant external memory fetches per computed output element.

Regarding the throughput and bandwidth utilization analysis, generalized formulas are established to assess data transfer efficiency across target input and output interfaces, alongside the CE compute utilization:

- **Maximum Theoretical Throughput (tp_{max}):** Establishes the physical upper bound of data throughput across a target data channel operating at every clock cycle:

$$\text{tp}_{\text{max}} = \frac{W_{\text{bus}}}{T_{\text{clk}} \cdot 10^{-3}} \quad [\text{MB/s}] \tag{5.10}$$

where:

- W_{bus} is the width of the physical data channel [bytes].
- T_{clk} is the system clock period [ns].
- 10^{-3} is the scaling conversion factor converting bytes/ns to megabytes per second (MB/s).

- **Throughput (tp):** Represents the effective operational data rate measured across a target communication channel:

$$\text{tp} = \frac{n_{\text{xfr}} \cdot W_{\text{bus}}}{t_{\text{act,us}}} \quad [\text{MB/s}] \tag{5.11}$$

where:

- n_{xfr} is the total number of data transfers executed across the targeted channel.
- W_{bus} is the width of the physical data channel [bytes].
- $t_{\text{act,us}}$ is the measured active time of the channel [μs].

- **Bandwidth Utilization (u_{bw}):** Quantifies the relative efficiency of data transfers relative to the channel's capacity:

$$u_{\text{bw}} = \left(\frac{\text{tp}}{\text{tp}_{\text{max}}} \right) \cdot 100 \quad [\%] \tag{5.12}$$

where:

- tp is the measured operational throughput across the targeted channel [MB/s].
- tp_{max} is the corresponding maximum theoretical channel throughput [MB/s].

- **CE Compute Utilization ($u_{\text{ce,cmp}}$):** Measures the proportion of active CE time spent performing actual arithmetic operations:

$$u_{\text{ce,cmp}} = \left(\frac{n_{\text{clk,ce,cmp}}}{n_{\text{clk,ce,act}}} \right) \cdot 100 \quad [\%] \tag{5.13}$$

  where:

  - $n_{\text{clk,ce,cmp}}$ is the clock cycles spent actively executing arithmetic computations [cycles].
  - $n_{\text{clk,ce,act}}$ is the total active duration of the CE in clock cycles [cycles].

Chapter 6

Experimental Setup and Methodology

To evaluate the architectural trade-offs and performance gains introduced by decoupling memory access from computation, a comprehensive simulation framework was constructed. This chapter details the structural differences between the evaluated system architectures, the hardware-level streaming configuration mechanisms, the automated end-to-end evaluation methodology, and the formal problem formulations for each benchmark.

6.1 Evaluated Architectures

To quantify the operational benefits of a memory-computation decoupled paradigm, two distinct system hardware topologies were implemented and simulated under strictly identical physical constraints.

6.1.1 Baseline System Architecture

The baseline hardware environment represents a conventional **Memory-Coupled** system model. As illustrated in Figure 6.1, this architecture comprises a MB, a central MC, and a MC-CE.

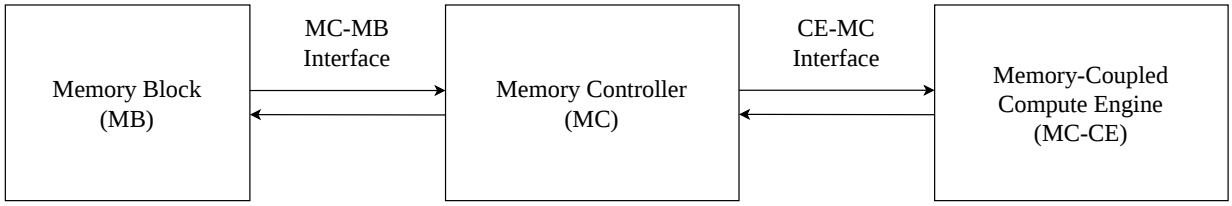


Figure 6.1: Baseline Architecture - System Block Diagram.

In this tightly coupled configuration, the CE directly manages its own external data fetch sequences and memory transactions via the CE-MC interface. Consequently, address generation, memory request stalling, and raw data alignment logic are embedded within or tightly bound to the computational execution units. This structural dependency forces the PEs to stall during high-latency external memory accesses, directly coupling computational throughput to memory latency.

6.1.2 Proposed System Architecture

The proposed environment introduces a **Memory-Computation Decoupled** paradigm. As presented in Figure 6.2, the system architecture decouples data access orchestration from execution by inserting a dedicated SE between the MC and the MD-CE.

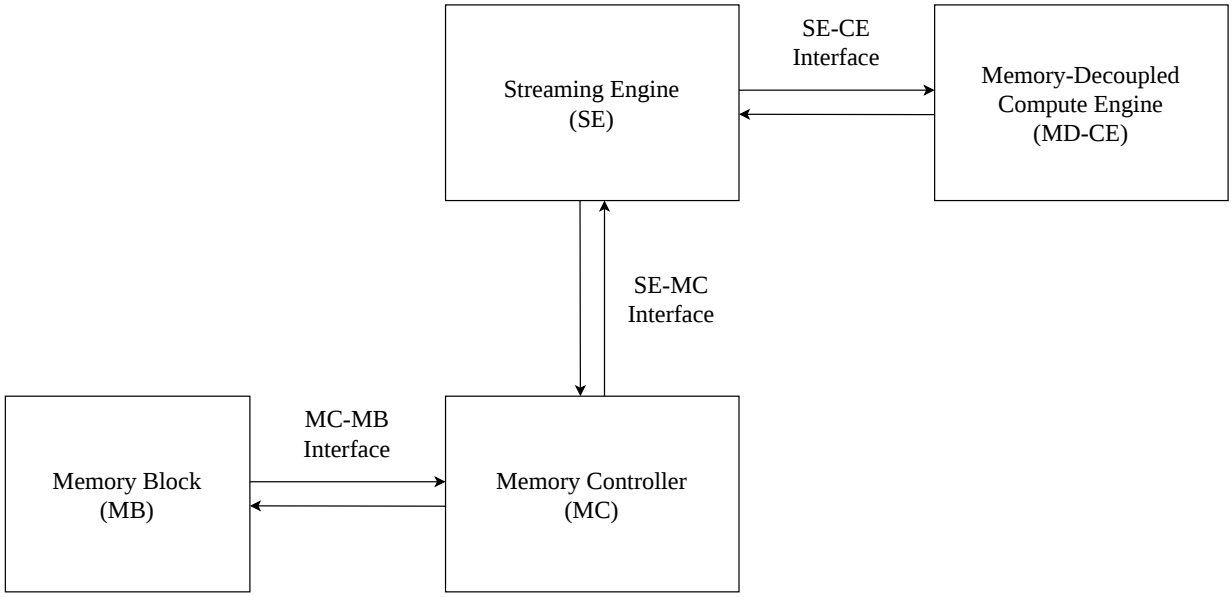


Figure 6.2: Proposed Architecture - System Block Diagram.

Under this architecture:

- The **SE** serves as the data provider. It interfaces directly with the MC via the SE-MC interface, issuing optimized burst memory requests, managing internal address generation patterns, and controlling input/output streams.
- The **MD-CE** is completely relieved of memory access overhead. It interfaces exclusively with the SE via the high-throughput SE-CE interface, receiving pre-fetched data streams and returning computed output streams without experiencing high memory latency stalls.

By isolating execution logic from memory access, the proposed architecture aims to maximize compute element utilization, optimize bus bandwidth, and reduce total simulation runtime across complex multi-dimensional workloads like the ones described in Section 5.1.

6.2 Experimental Setup

To ensure a rigorous and unbiased comparison between the baseline and proposed architectures, all hardware RTL modules were evaluated within a unified simulation environment. Both systems operated under strictly aligned physical parameters, bus configurations, and memory capacities

across all test runs. Note that for specific evaluation scenarios presented in Chapter 7 certain parameters were intentionally varied while keeping all other settings at these default values.

The system-level physical parameters were parameterized as followed:

- **System Clock Frequency ($f_{\mathbf{clk}}$):** Configured to 100 MHz ($T_{\text{clk}} = 10$ ns).
- **Memory Access Latency ($l_{\mathbf{ns}}$):** Fixed modeling memory latency of 15 ns ($\text{l}_{\text{cyc}} = 2$ cycles).
- **Burst Length (LEN):** MC burst transaction length configured to 8 data transfers.
- **Address Width (AW):** Address buses and addresses in general feature a bit-width of AW = 32 bits.
- **Data Width (DW):** Data buses and individual data elements operate on a bit-width of DW = 64 bits.
- **MB Capacity ($C_{\mathbf{MB}}$):** For both architectures, the MB defines a structured memory layout starting at base address LA = 32'h0000_0000. Memory space is allocated uniformly with a per-stream depth of MA = 16384 addresses. Across three distinct stream regions (two input streams and one output stream), the total depth reaches:

$$\text{MB}_{\text{DPT}} = 3 \cdot \text{MA} = 3 \cdot 16384 = 49152 \quad [\text{addresses}] \tag{6.1}$$

  yielding a total MB storage capacity calculated as:

$$C_{\text{MB}} = \frac{\text{MB}_{\text{DPT}} \cdot \text{DW}}{8 \cdot 10^3} = \frac{49152 \cdot 64}{8 \cdot 10^3} = 393.216 \quad [\text{KB}] \tag{6.2}$$

In the proposed architecture, the SE incorporates internal memory structures to handle local caching and data reuse to avoid stalling the CE. The dimensions and capacities for these internal hardware structures are configured as follows:

- **SB Capacity ($C_{\mathbf{SB}}$):** The SB's depth is defined as $\text{SB}_{\text{DPT}} = 1024$ positions, giving a total capacity C_{SB} of:

$$C_{\text{SB}} = \frac{\text{SB}_{\text{DPT}} \cdot \text{DW}}{8 \cdot 10^3} = \frac{1024 \cdot 64}{8 \cdot 10^3} = 8.192 \quad [\text{KB}] \tag{6.3}$$

- **SC Capacity ($C_{\mathbf{SC}}$):** The SC's depth is parameterized as $\text{SC}_{\text{DPT}} = 64$ cache lines, and the width as $\text{SC}_{\text{WDT}} = 8$ positions per line, giving a total capacity C_{SC} of:

$$C_{\text{SC}} = \frac{\text{SC}_{\text{DPT}} \cdot \text{SC}_{\text{WDT}} \cdot \text{DW}}{8 \cdot 10^3} = \frac{64 \cdot 8 \cdot 64}{8 \cdot 10^3} = 4.096 \quad [\text{KB}] \tag{6.4}$$

6.3 Methodology

To obtain the results across all target configurations, a Makefile-driven automated pipeline was implemented. Executing a single Makefile command automates the simulation flow across the defined parameter sweeps, comprising three key steps per run:

1. **Input Dataset Generation:** A custom Python script is ran, generating the deterministic input datasets and writing them to raw hexadecimal `.txt` files formatted for testbench memory preloading.
2. **Golden Reference Calculation:** A secondary Python script is executed, processing the generated input data files to compute the hexadecimal reference output values. These outputs are written into a `.txt` file, and serve as the ground truth for functional verification.
3. **Simulation and Metric Logging:** Executes the RTL simulation with the parsed sweep parameters, where the testbench computes and exports the determined performance metrics directly into a dedicated `.log` file.

Once all required simulation runs are completed across the sweep space, a post-processing Python script extracts the metrics from the generated `.log` files and produces the final plots for performance analysis.

6.4 Problem Formulation

Unconstrained design spaces involving multi-variable filter dimensions and arbitrary matrix shapes introduce unnecessary complexity into architectural evaluation. To isolate the efficiency of memory access handling and stream orchestration, each computational benchmark is constrained to a controlled design space: kernel and coefficient dimensions are held fixed, while primary input dataset sizes are subjected to a systematic parameter sweep (N). The swept parameter establishes the underlying problem dimension — referred throughout the remainder of the document as the problem size N — defining the scale of the executed workload across all evaluation scenarios.

6.4.1 FIR Filter

For the FIR filtering application, the filter impulse response size (number of filter taps/coefficients, N_T) is fixed, while the input signal length (N_S) scales with N.

- **Fixed Parameter:** Filter taps/kernel length $N_T = 16$.
- **Swept Parameter (N):** Total number of input time-domain signal samples ($N_S = N$).
- **SB Mode Configurations:**
  - **IS0 (Input Signal):** Configured as standard streaming FIFO mode.
  - **IS1 (Filter Coefficients):** Configured as CB mode, to cache the N_T coefficients once and reuse them cyclically across all output samples.
  - **OS (Output Signal):** Configured as standard streaming FIFO mode.
- **Problem Definition:** Evaluates sequential stream processing efficiency over 1D data arrays of scaling size N. The SE streams the coefficient array into the CB once and streams the input signal continuously through a 2-level nested loop architecture.

6.4.2 Matrix Multiplication

To maintain a uniform scaling behavior during dense 2D matrix multiplication ($A \times B = C$), the input matrices are constrained to be square ($N \times N$).

- **Fixed Parameter:** None (operates on full dense matrix layouts).
- **Swept Parameter (N):** Row and column dimension length N, where $A, B, C \in R^{N \times N}$.
- **SB Mode Configurations:**
  - **IS0 (Matrix A):** Configured as standard streaming FIFO mode.
  - **IS1 (Matrix B):** Configured as standard streaming FIFO mode.
  - **OS (Matrix C):** Configured as standard streaming FIFO mode.
- **Problem Definition:** The total number of elements per matrix scales quadratically as N^2. This setup tests 3-level nested strided loops across both input streams to handle row/column data re-use and strided memory layouts without internal circular buffer caching.

6.4.3 2D Convolution

The 2D spatial convolution applies a 2D filter/kernel over an input map. Both the input map and convolution kernel are defined as square matrices.

- **Fixed Parameter:** Convolution kernel dimensions ($K_R \times K_C = 3 \times 3$).
- **Swept Parameter (N):** Spatial height and width of the square input map ($I_R \times I_C = N \times N$).
- **SB Mode Configurations:**
  - **IS0 (Input Map):** Configured as standard streaming FIFO mode.
  - **IS1 (2D Kernel):** Configured as CB mode, to pre-load the 3×3 kernel weights once and reuse them cyclically across all output pixels.
  - **OS (Output Map):** Configured as standard streaming FIFO mode.
- **Problem Definition:** The input data size scales quadratically as N^2. This benchmark evaluates 4-level nested strided loops within the SE for the IS0, managing horizontal and vertical boundary traversals while continuously re-reading static kernel weights from the CB for the IS1.

6.4.4 3D Convolution

The 3D convolution application extends filtering to volumetric spatial tensors. The filter kernel is configured as a fixed-size cubic tensor, while the input volume scales isotropically.

- **Fixed Parameter:** Filter kernel dimensions ($K_R \times K_C \times K_D = 3 \times 3 \times 3$).

- **Swept Parameter (N):** Isotropic input volume edge length ($I_R \times I_C \times I_D = N \times N \times N$).
- **SB Mode Configurations:**
  - **IS0 (Input Volume):** Configured as standard streaming FIFO mode.
  - **IS1 (3D Kernel):** Configured as CB mode, to cache the $3 \times 3 \times 3$ cubic kernel coefficients once and reuse them cyclically across all 3D output voxels.
  - **OS (Output Volume):** Configured as standard streaming FIFO mode.
- **Problem Definition:** The total dataset volume scales cubically as N^3. This benchmark utilizes 6-level nested strided loops in the SE for IS0 to traverse 3D memory strides, testing hardware limit detection across horizontal, vertical, and depth boundaries while streaming kernel weights continuously from the CB for IS1.

Chapter 7

Results and Discussion

This chapter presents the performance evaluation and experimental results obtained from RTL simulations of the baseline architecture and the proposed architecture. To demonstrate scalability, memory handling efficiency, and data reuse gains across expanding spatial data patterns, the evaluation covers four distinct computational benchmarks.

The chapter is organized by benchmark section. Within each benchmark, system performance is systematically evaluated across three primary experimental parameter sweeps:

1. **Problem Size Sweep (N):** Evaluates algorithmic scalability and memory bandwidth efficiency across a defined dataset of workload sizes N. For all benchmarks, this sweep operates under a nominal fixed clock frequency of 100 MHz and a fixed baseline memory latency of 15 ns.

2. **Operating Frequency Sweep (f):** Evaluates clock-rate throughput scaling and pipeline timing margins by varying the system clock frequency across a defined dataset of frequency values. To isolate clock scaling, the Problem Size N is held fixed for each benchmark. The fixed value of N is selected by analyzing the CE throughput curves from the Problem Size Sweep and identifying a steady-state region where throughput stabilizes and remains approximately constant for both architectures. The memory latency is maintained at 15 ns.

3. **Memory Latency Sweep (l):** Stress-tests the latency-hiding capability of local stream buffers by simulating external memory response delays across a defined dataset of latency values. For each benchmark, the operating frequency is held fixed at 100 MHz, and the Problem Size N is fixed to the same steady-state operating point established in the frequency analysis.

To maintain a uniform analytical framework across all parameter sweeps, each evaluation is structured around three core analytical dimensions:

- **Simulation Time and Speedup Analysis:** Analyzes the simulation execution times across both architectures and measures the speedup gain achieved by the proposed architecture relative to the baseline architecture.

- **Memory Access Analysis:** Analyzes the MAFs and memory communication times across both architectures to clarify and compare their memory access dynamics.
- **Throughput and Bandwidth Utilization Analysis:** Compares CE throughput, bandwidth utilization, and compute utilization across both architectures. Additionally, it evaluates the SE-to-CE data transfer against the baseline MC-to-CE link — comparing both throughput and bandwidth utilization — to highlight the efficiency gains in data feeding to the CE.

The chapter concludes in Section 7.5 with a comparative discussion summarizing the key architectural trade-offs and efficiency gains across all evaluated benchmarks.

7.1 FIR Filter

7.1.1 Problem Size Sweep

To evaluate the execution performance of the FIR Filter benchmark, simulations were conducted across a range of problem sizes. As described in Section 6.4.1, the swept parameter N represents the total number of input signal samples ($N_S = N$), which was varied from $N = 1000$ to $N = 16000$ stream elements in steps of 1000 (where N is expressed in units of 10^3 samples in the fitted regression models).

7.1.1.1 Simulation Time and Speedup Analysis

In the baseline architecture, the MC-CE directly manages memory transactions with the memory system. For every computation phase, the MC-CE issues individual single-read operations to fetch each required operand from external memory, and once the output value is calculated, a single-write request is issued. Because memory accesses are directly bound to the execution pipeline without caching mechanisms or data reuse techniques, the baseline architecture experiences significant memory transfer latency overheads, resulting in high overall simulation times ($t_{\text{sim,bsl}}$) as depicted in Figure 7.1.

Mathematically, the simulation time for the baseline implementation follows a strictly linear relationship with respect to the scaled Problem Size N, modeled by the linear regression function:

$$t_{\text{sim,bsl}}(N) = 2820.000 \cdot N + 0.030 \quad [\mu\text{s}] \tag{7.1}$$

The predominant linear coefficient ($2820.000\,\mu$s per 10^3 elements) directly reflects the linear $O(N \cdot N_T)$ algorithmic complexity of processing N signal samples with $N_T = 16$ fixed taps. The negligible constant offset ($0.030\,\mu$s) confirms that setup overheads are minimal compared to the memory-bound execution loops.

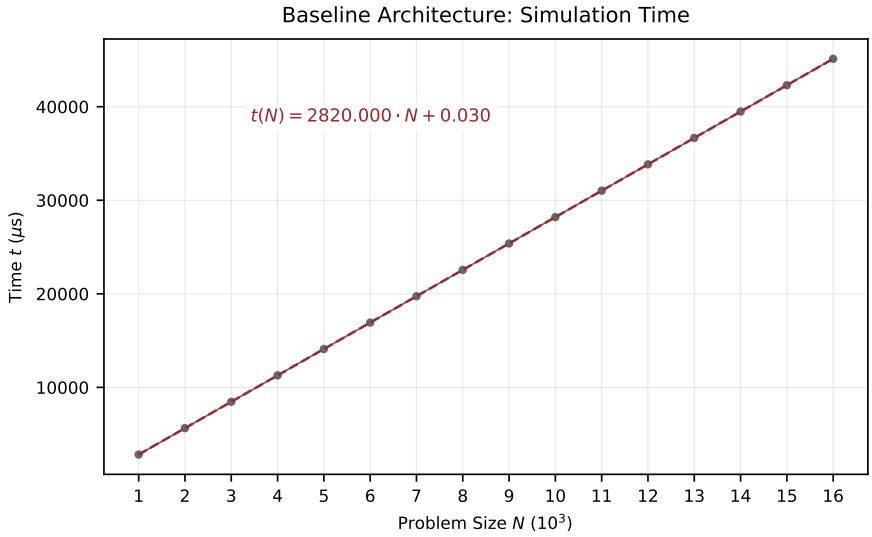


Figure 7.1: Baseline Architecture - Simulation Time across Problem Size N.

In contrast, the proposed architecture introduces a SE positioned between external memory and the MD-CE, decoupling memory access and address generation logic from the core computation substrate. Beyond offloading memory management from the MD-CE, the SE leverages specialized forwarding mechanisms designed for high data reuse. Specifically, the IS1 SB is configured in CB mode, caching the $N_T = 16$ filter coefficients locally upon initial loading and feeding subsequent sliding-window operations without issuing redundant external memory requests. By insulating the MD-CE from main memory access delays and supplying data directly through local stream buffers, the computing engine operates with significantly reduced memory stalling.

As illustrated in Figure 7.2, the simulation time of the proposed architecture ($t_{\text{sim,prp}}$) remains substantially lower across all problem sizes while maintaining a linear response:

$$t_{\text{sim,prp}}(N) = 595.000 \cdot N + 1.800 \quad [\mu\text{s}] \tag{7.2}$$

Comparing the scaling coefficients, the proposed architecture drastically reduces the per-sample execution cost from 2820.000 μs down to 595.000 μs per 10^3 elements. The small constant offset of 1.800 μs accounts for the initial setup phase wherein the SE populates its internal stream buffers and loads the circular coefficient array before continuous pipeline execution commences.

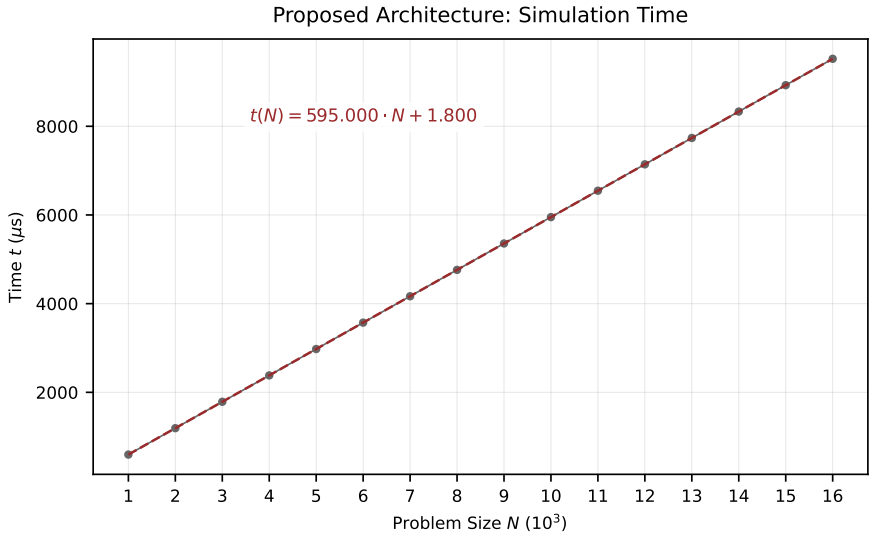


Figure 7.2: Proposed Architecture - Simulation Time across Problem Size N.

The resulting relative speedup $s(N) = \frac{t_{\text{sim,bsl}}(N)}{t_{\text{sim,prp}}(N)}$ achieved by the proposed architecture over the baseline architecture is shown in Figure 7.3. The empirical speedup curve follows an inverse model matching the plot regression:

$$s(N) = 4.739 - \frac{0.014}{N} \tag{7.3}$$

This inverse relationship directly matches the analytical ratio of two linear equations with constant offsets ($\frac{a_1 N + b_1}{a_2 N + b_2}$). For small problem sizes ($N = 1$), the constant setup overhead ($1.800\,\mu$s) of the SE slightly dilutes efficiency, resulting in an initial speedup of 4.725. However, as $N \to \infty$, the impact of fixed initialization overheads vanishes ($\frac{0.014}{N} \to 0$), causing the speedup factor to asymptotically approach a horizontal limit of $s_{\max} \approx 4.739$, which represents the ratio of execution slopes ($\frac{2820.000}{595.000} \approx 4.739$).

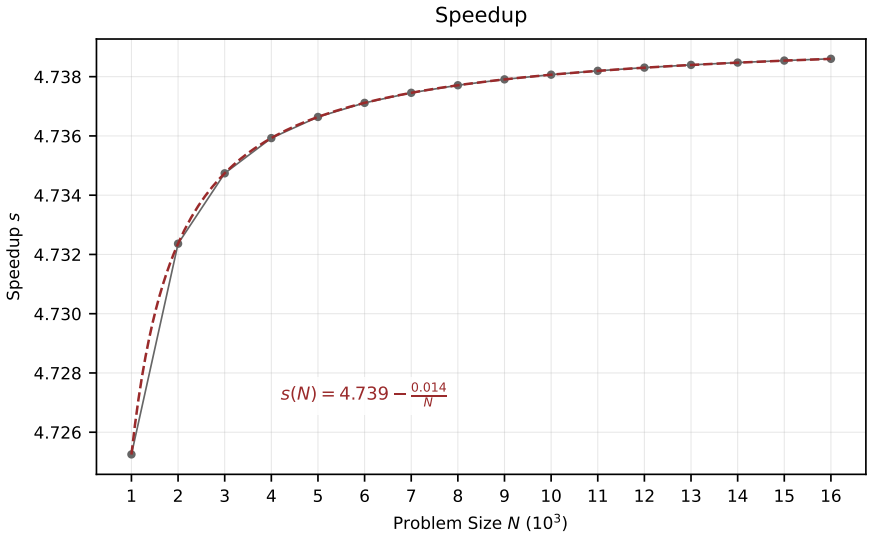


Figure 7.3: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Problem Size N.

7.1.1.2 Memory Access Analysis

The overall execution efficiency and system throughput are heavily dictated by memory access patterns and external communication overheads. In the baseline architecture, the MC-CE interacts directly with external memory through the MC. Lacking dedicated hardware caching or streaming buffers, the MC-CE relies on individual single-read requests for each input operand and single-write requests for every output sample computed. For a 16-tap FIR filter, computing a single output sample requires fetching 16 input signal samples and 16 filter coefficients, plus writing 1 result back to memory. Consequently, 33 discrete single-access memory transactions are required per output element, maintaining a constant MAF ($\text{MAF}_{\text{bsl}} = 33.0$) across all problem sizes N.

Without burst transfer mechanisms, each transaction incurs full bus handshake latency, causing the baseline memory communication time ($t_{\text{mem,bsl}}$) to scale linearly with problem size N, as illustrated in Figure 7.4:

$$t_{\text{mem,bsl}}(N) \approx 1320.000 \cdot N \quad [\mu\text{s}] \tag{7.4}$$

The linear slope of $1320.000\,\mu$s per 10^3 elements reflects the severe cumulative overhead of unbuffered, single-request external memory accesses.

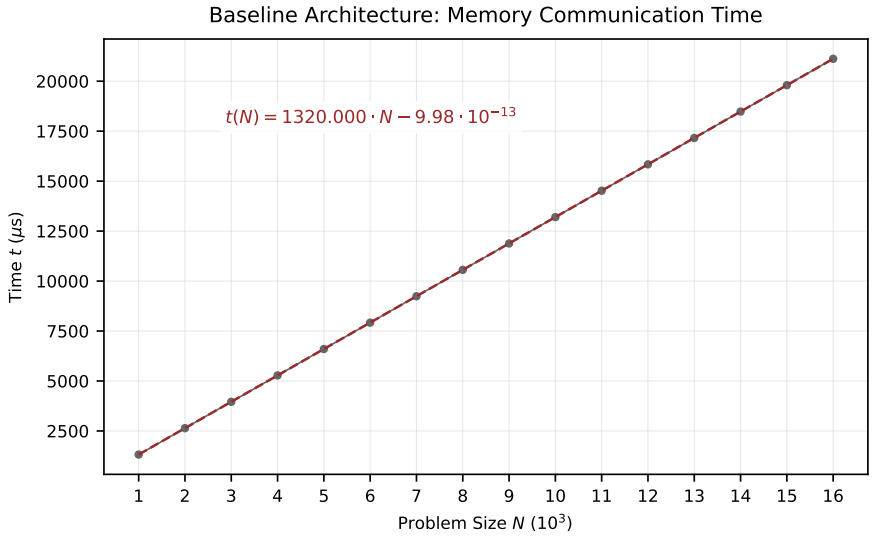


Figure 7.4: Baseline Architecture - Memory Communication Time across Problem Size N.

Conversely, the proposed architecture leverages the SE to decouple compute operations from external memory latency through hardware-managed data reuse and burst transfers. The $N_T = 16$ filter coefficients are pre-loaded into the IS1 SB operating in CB mode. Once cached in local stream buffers, redundant external coefficient fetches are eliminated, requiring the system to only stream new input signal samples and store generated output samples.

As depicted in Figure 7.5, the Memory Access Factor (MAF_{prp}) of the proposed architecture follows an inverse model across dataset length:

$$\text{MAF}_{\text{prp}}(N) = \frac{0.032}{N} + 2.000 \tag{7.5}$$

For small input lengths, the initial setup phase of populating the IS1 SB with 16 coefficients slightly elevates the ratio. However, as N increases, this fixed setup overhead is fully amortized ($\frac{0.032}{N} \to 0$), causing MAF_{prp} to asymptotically approach a steady-state limit of 2.000 (1 input read and 1 output write per computed sample).

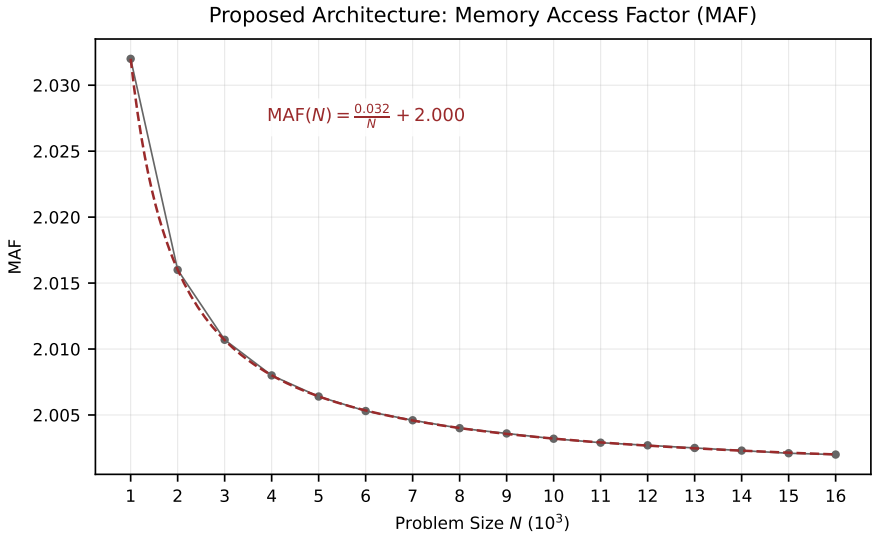


Figure 7.5: Proposed Architecture - Memory Access Factor (MAF) across Problem Size N.

Furthermore, replacing single-access transfers with high-efficiency burst operations significantly maximizes bus throughput. As shown in Figure 7.6, the memory communication time ($t_{\text{mem,prp}}$) scales linearly but at a drastically lower rate:

$$t_{\text{mem,prp}}(N) = 45.000 \cdot N + 0.720 \quad [\mu\text{s}] \tag{7.6}$$

The initial offset ($0.720\,\mu$s) captures the burst pre-loading of the coefficient array, while the slope drops from $1320.000\,\mu$s down to $45.000\,\mu$s per 10^3 elements. At $N = 16000$, $t_{\text{mem,prp}}$ reaches only $720.720\,\mu$s compared to $21120.000\,\mu$s in the baseline, demonstrating a $\sim 29.3\times$ reduction in total memory communication overhead.

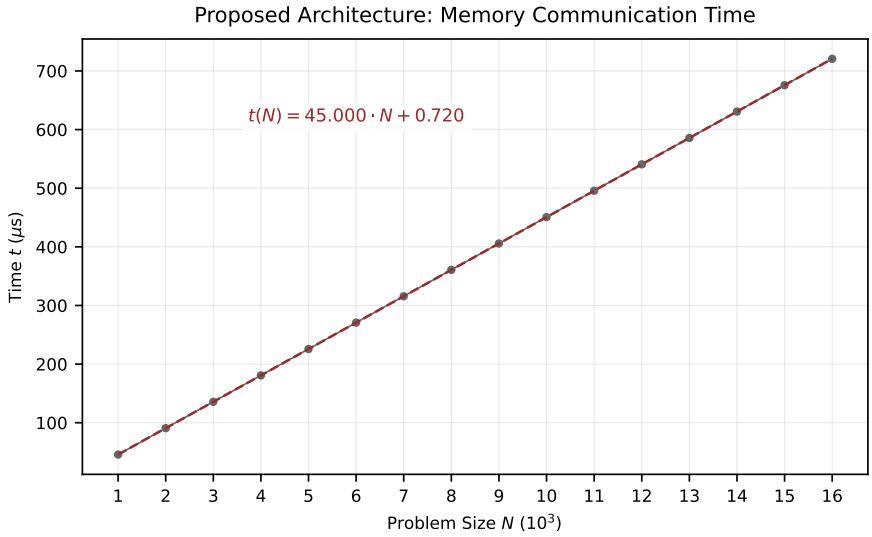


Figure 7.6: Proposed Architecture - Memory Communication Time across Problem Size N.

7.1.1.3 Throughput and Bandwidth Utilization Analysis

To further evaluate system performance, we analyze the CE throughput, bus bandwidth utilization, and internal execution efficiency across both architectures relative to their theoretical hardware limits.

In the baseline architecture, the direct coupling of memory accesses to the compute pipeline without caching or buffering mechanisms severely bottlenecks execution. As analyzed in Section 7.1.1.2, the high MAF_{bsl} (33.0) and single-transaction memory access model force the system to spend a significant portion of its runtime stalling for data. On the MC-to-CE interface — which has a maximum theoretical throughput $\text{tp}_{\text{max,mc-ce}} = 800.000$ MB/s — the MC-to-CE throughput ($\text{tp}_{\text{mc-ce}}$) remains flat at 90.780 MB/s, yielding a low bandwidth utilization ($u_{\text{bw,mc-ce}}$) of 11.348%. Because the CE constantly stalls during these unbuffered handshakes, its effective operational data rate ($\text{tp}_{\text{ce,bsl}}$) reaches a constant 2.837 MB/s across all problem sizes N.

This severe memory stalling directly impacts overall hardware utilization. Relative to the maximum theoretical throughput ($\text{tp}_{\text{max,ce,bsl}}$) of 800.000 MB/s, the baseline architecture maintains a constant CE bandwidth utilization ($u_{\text{bw,ce,bsl}}$) of just 0.355%. Furthermore, because the internal arithmetic units sit idle during memory transfers, the baseline CE compute utilization ($u_{\text{ce,cmp,bsl}}$) remains constant at 6.028% across all values of N.

Conversely, the proposed architecture leverages the SE to decouple memory transfers from execution. Internally, the SE streams pre-buffered operand data directly to the CE over a dedicated high-bandwidth local interface rated for a maximum theoretical throughput $\text{tp}_{\text{max,se-ce}} = 1600.000$ MB/s. As presented in Figure 7.7, the internal SE-to-CE throughput ($\text{tp}_{\text{se-ce}}$) follows a bounded inverse growth model:

$$\text{tp}_{\text{se-ce}}(N) = 430.252 - \frac{1.269}{N} \quad [\text{MB/s}] \tag{7.7}$$

Rising from 428.983 MB/s at $N = 1000$ to an asymptotic steady state of 430.252 MB/s as $N \to \infty$.

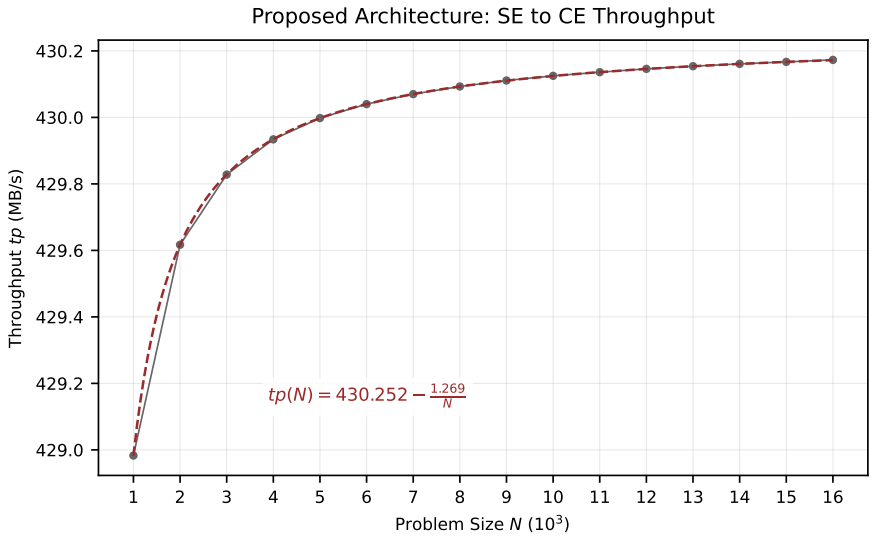


Figure 7.7: Proposed Architecture - SE to CE Throughput across Problem Size N.

Correspondingly, as illustrated in Figure 7.8, the local SE-to-CE bandwidth utilization ($u_{\text{bw,se-ce}}$) scales according to:

$$u_{\text{bw,se-ce}}(N) = 26.891 - \frac{0.080}{N} \quad [\%] \tag{7.8}$$

Scaling from 26.811% at $N = 1000$ and stabilizing near 26.891% for large dataset sizes.

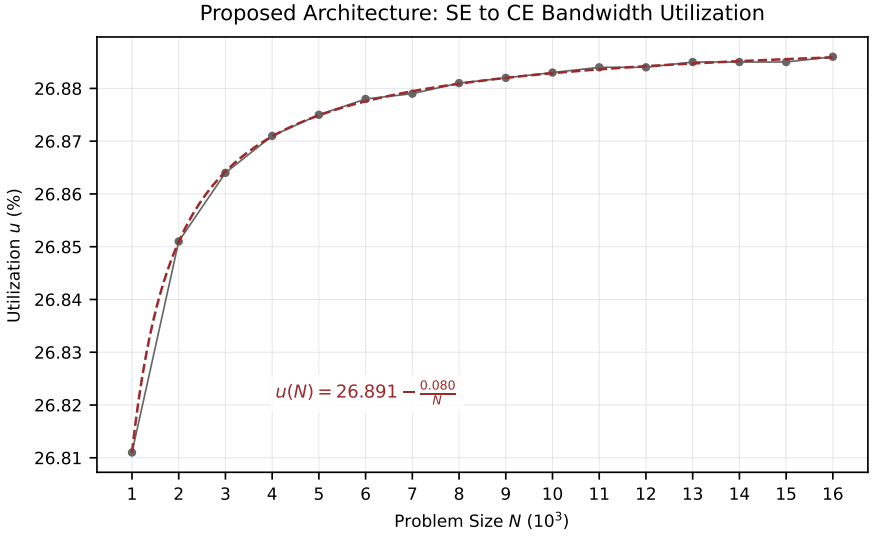


Figure 7.8: Proposed Architecture - SE to CE Bandwidth Utilization across Problem Size N.

Owing to burst-mode transfers and the reduction of the memory access factor down to $\text{MAF}_{\text{prp}} \approx 2.0$, external memory communication time is drastically reduced. As demonstrated in Figure 7.9,

the effective external CE throughput ($\mathrm{tp}_{\mathrm{ce,prp}}$) follows an inverse relationship relative to N:

$$\mathrm{tp}_{\mathrm{ce,prp}}(N) = \frac{0.632}{N} + 13.442 \quad [\mathrm{MB/s}] \tag{7.9}$$

At small N, initialization overheads yield a peak effective rate of 14.078 MB/s before stabilizing to a steady-state throughput of 13.442 MB/s. Compared to the baseline's 2.837 MB/s, this represents an effective throughput speedup of 4.74×.

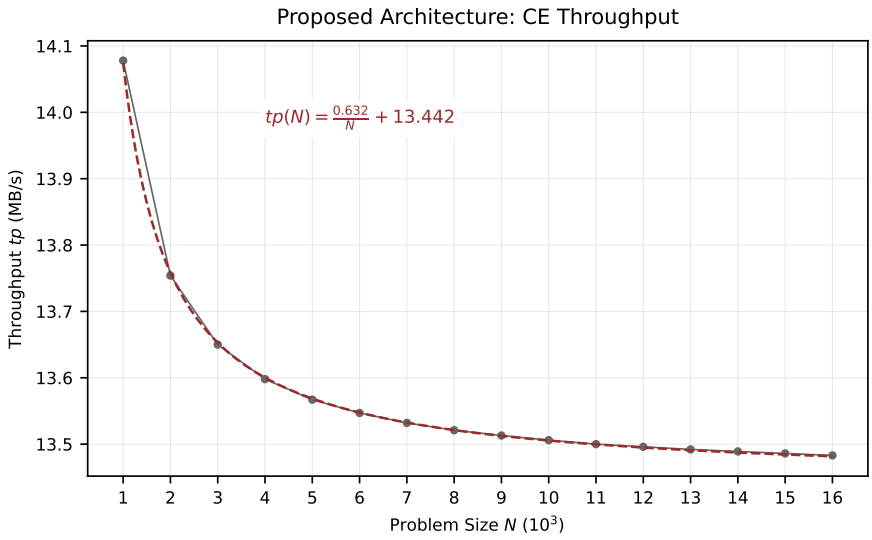


Figure 7.9: Proposed Architecture - CE Throughput across Problem Size N.

Figure 7.10 highlights the overall CE bandwidth utilization ($u_{\mathrm{bw,ce,prp}}$) against the maximum theoretical channel capacity $\mathrm{tp}_{\mathrm{max,ce,prp}} = 800.000$ MB/s:

$$u_{\mathrm{bw,ce,prp}}(N) = \frac{0.079}{N} + 1.680 \quad [\%] \tag{7.10}$$

The utilization settles to a steady value of 1.680%. While external memory bus bandwidth utilization remains modest — due to the minimal requirement of fetching 1 input sample and storing 1 output sample per computation step — the effective processing throughput achieved per transaction is significantly higher than the baseline.

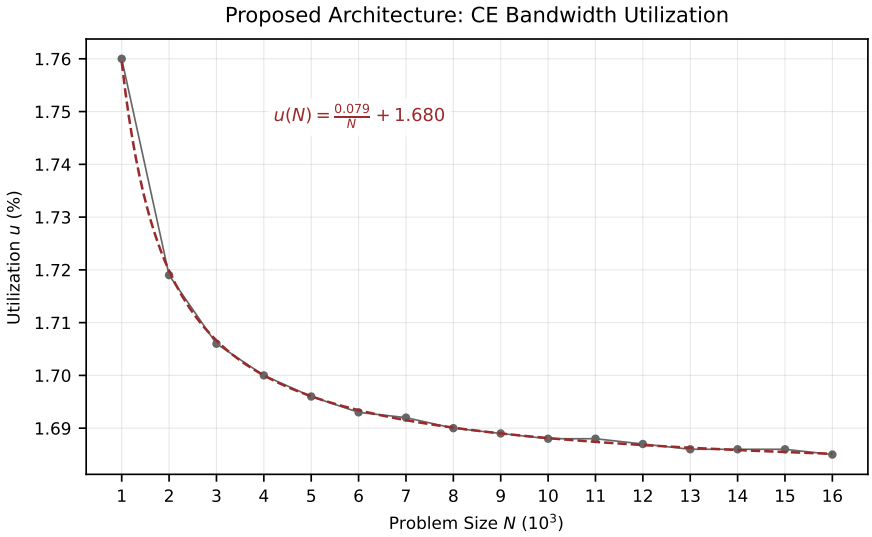


Figure 7.10: Proposed Architecture - CE Bandwidth Utilization across Problem Size N.

Finally, this continuous local data delivery enables the processing engine to execute with minimal stalling. As depicted in Figure 7.11, the CE compute utilization ($u_{\text{ce,cmp,prp}}$) scales as:

$$u_{\text{ce,cmp,prp}}(N) = \frac{1.343}{N} + 28.565 \quad [\%] \tag{7.11}$$

Starting at 29.908% for $N = 1000$, $u_{\text{ce,cmp,prp}}$ settles to an asymptotic steady-state efficiency of 28.565% as N increases. This yields a 4.74× increase in compute utilization over the baseline's constant 6.028%.

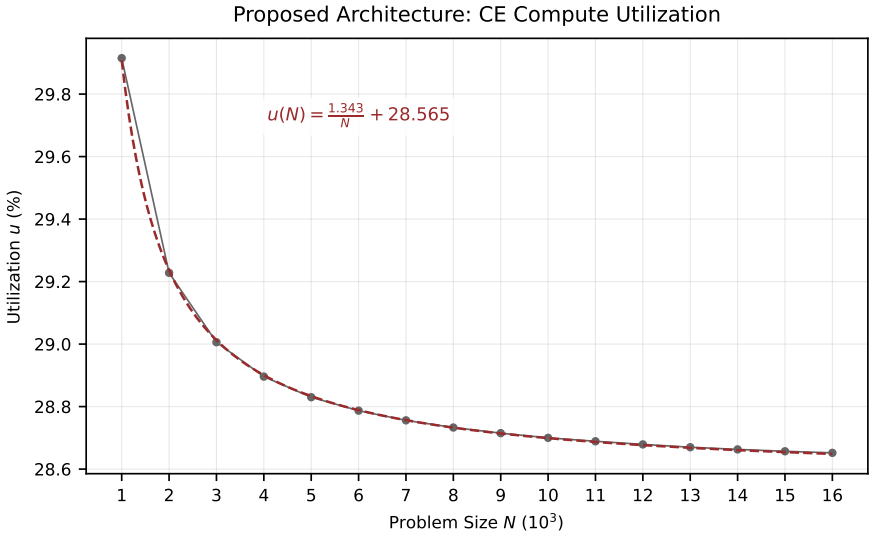


Figure 7.11: Proposed Architecture - CE Compute Utilization across Problem Size N.

7.1.2 Operating Frequency Sweep

To evaluate the scalability and performance sensitivity of both architectures to clock frequency variations, an operating frequency sweep is conducted across $f \in [100, 250]$ MHz. For all frequency sweep evaluations, the problem size is fixed at $N = 10000$. This dataset size was specifically selected because, as established in Section 7.1.1.3, $N = 10000$ represents a operating regime where the CE throughput (tp_{ce}) has stabilized and remains approximately constant for both the Baseline and proposed architectures, thereby eliminating dataset-scaling transients from the frequency response analysis.

7.1.2.1 Simulation Time and Speedup Analysis

The relationship between operating frequency and execution time highlights fundamental differences in pipeline clocking efficiency between the two system designs.

In the baseline architecture, execution time decreases non-linearly with increasing clock frequency. As illustrated in Figure 7.12, the simulation time $t_{\text{sim,bsl}}$ follows an inverse relationship with operating frequency f:

$$t_{\text{sim,bsl}}(f) = \frac{2.39 \cdot 10^6}{f} + 3768.207 \quad [\mu\text{s}] \tag{7.12}$$

As frequency increases from 100 MHz to 250 MHz, execution time drops from 28212.000μs down to 13912.000μs. However, due to fixed memory access handshakes and unbuffered bus latency penalties that remain unscaled by clock frequency, the baseline architecture exhibits diminishing returns at higher operating frequencies.

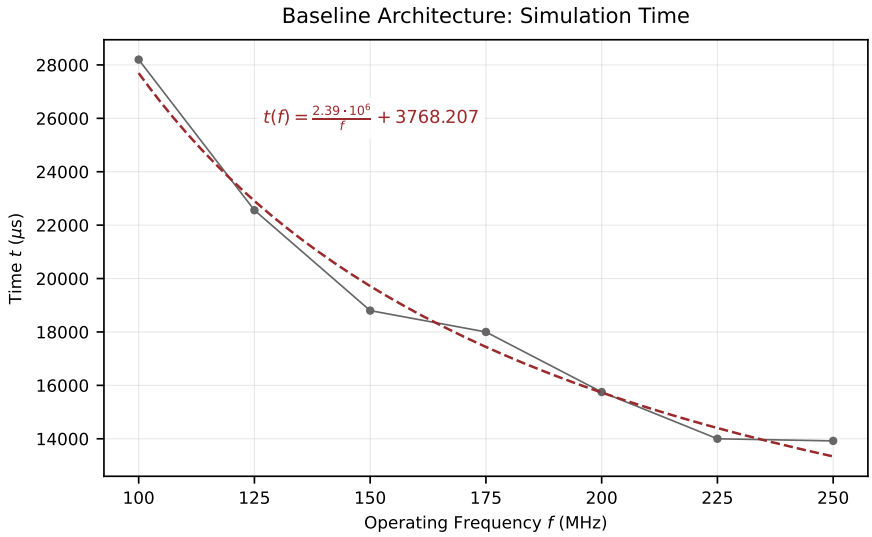


Figure 7.12: Baseline Architecture - Simulation Time across Operating Frequency f.

Conversely, the proposed architecture also exhibits an inverse relationship with frequency, but with drastically reduced overheads. As shown in Figure 7.13, the simulation time $t_{\text{sim,prp}}$ scales as:

$$t_{\text{sim,prp}}(f) = \frac{5.92 \cdot 10^5}{f} + 28.592 \quad [\mu\text{s}] \tag{7.13}$$

As f increases from 100 MHz to 250 MHz, execution time decreases from 5948.592μs to 2396.592μs. The offset constant is reduced to just 28.592μs, demonstrating that memory stalling overheads are substantially eliminated.

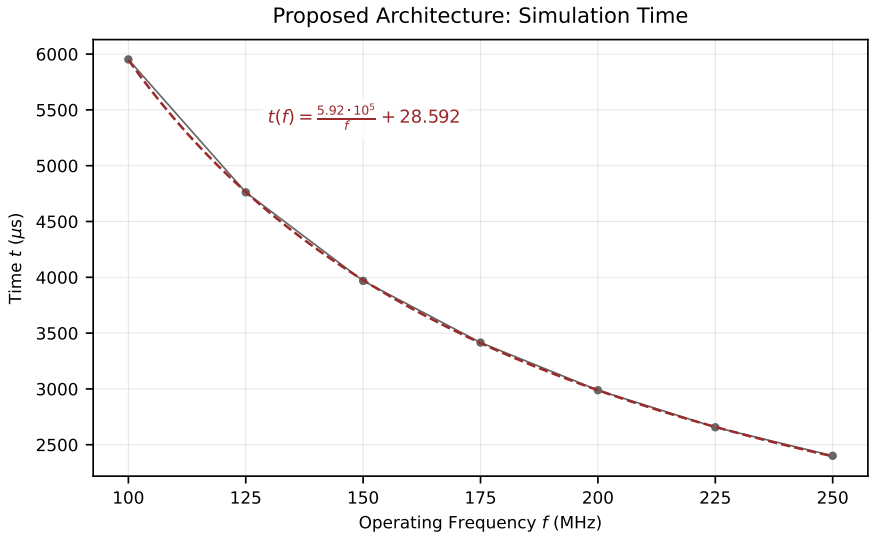


Figure 7.13: Proposed Architecture - Simulation Time across Operating Frequency f.

The resulting speedup $s(f)$ achieved by the proposed architecture over the baseline architecture across operating frequencies is depicted in Figure 7.14. Overall, the speedup grows from 4.74× at 100 MHz up to 5.80× at 250 MHz. The step-like jumps in the speedup curve occur because external memory latency is fixed in physical time, forcing the MC latency to scale in discrete clock cycles. As frequency increases, the cycle count remains constant until a frequency threshold is crossed, at which point the memory response time requires an additional clock cycle. Because these discrete latency penalties degrade the baseline architecture's efficiency more severely than the proposed design, each cycle threshold crossing produces a step jump in the speedup curve.

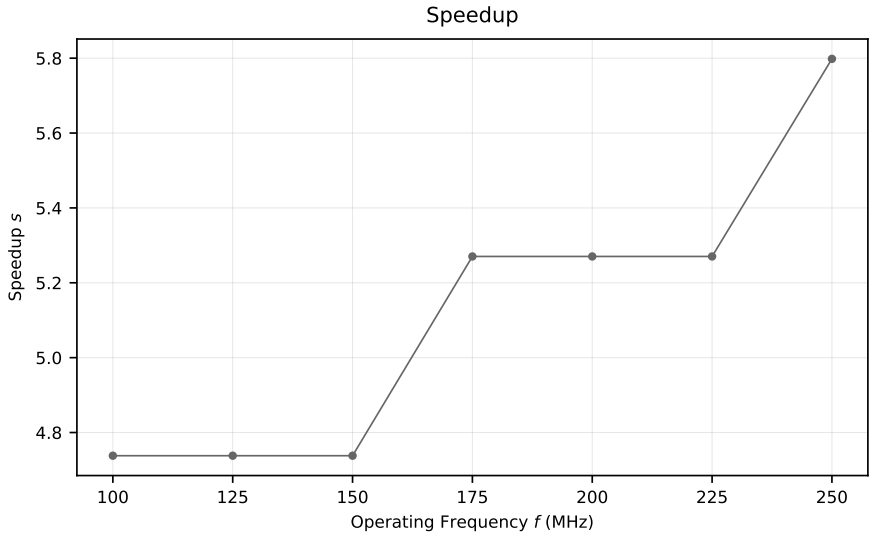


Figure 7.14: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Operating Frequency f.

7.1.2.2 Memory Access Analysis

To isolate the memory interface efficiency from computation across operating frequencies, we evaluate both the MAF and total memory communication time (t_{mem}) at a fixed problem size $N = 10000$.

In the baseline architecture, the memory access factor remains strictly invariant across all operating frequencies at $\text{MAF}_{\text{bsl}} = 33.000$. Because the baseline system performs unbuffered single-data memory transfers for every filter tap coefficient and sample fetch, frequency scaling does not alter the structural number of memory accesses required per computation.

However, as shown in Figure 7.15, the baseline memory communication time ($t_{\text{mem,bsl}}$) exhibits non-monotonic behavior across frequency increments. While higher frequencies generally reduce execution time, discrete upward jumps occur between 150 MHz $\rightarrow$ 175 MHz and 225 MHz $\rightarrow$ 250 MHz. This behavior stems directly from the fixed physical response delay of the memory system. Because the latency must round up to discrete integer clock cycles, operating at higher frequencies reduces the clock period T_{clk}, crossing threshold points where the MC must wait an additional full clock cycle per access. Because MAF_{bsl} is extremely high (33.000), these accumulated single-cycle penalty steps cause notable spikes in total communication delay.

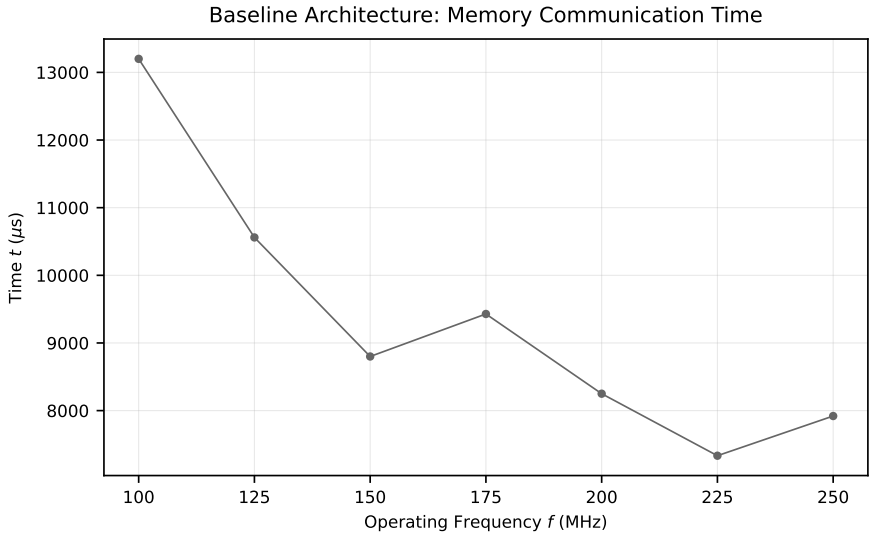


Figure 7.15: Baseline Architecture - Memory Communication Time across Operating Frequency f.

Conversely, the proposed architecture maintains a constant $\text{MAF}_{\text{prp}} = 2.003$ across all frequencies, demonstrating solid data reuse due to the SE's buffering mechanisms.

As illustrated in Figure 7.16, the memory communication time of the proposed architecture ($t_{\text{mem,prp}}$) follows an inverse relationship with operating frequency:

$$t_{\text{mem,prp}}(f) = \frac{4.18 \cdot 10^4}{f} + 28.593 \quad [\mu\text{s}] \tag{7.14}$$

Because burst-mode transfers and specialized buffering drastically minimize total memory transactions (MAF ≈ 2), integer clock cycle quantization effects on external latency are completely absorbed, allowing memory communication time to scale smoothly from 450μs down to 200μs as f increases from 100 MHz to 250 MHz.

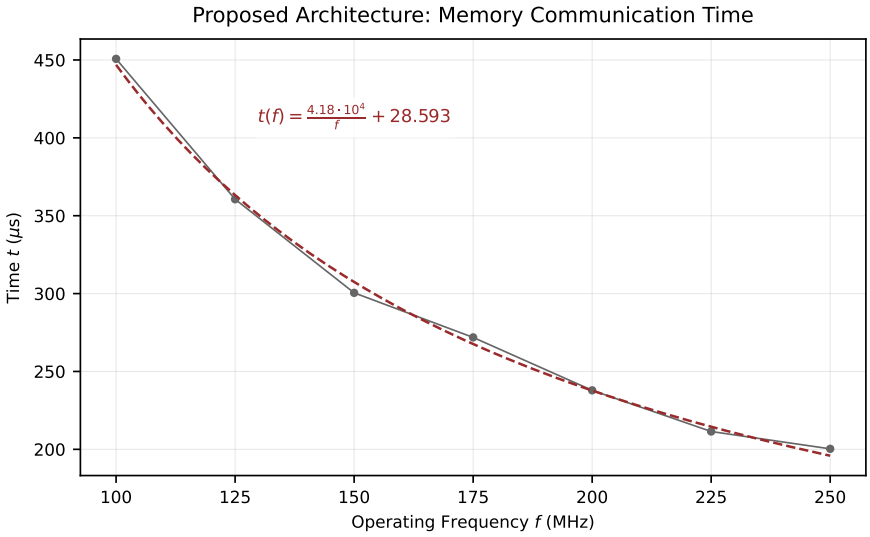


Figure 7.16: Proposed Architecture - Memory Communication Time across Operating Frequency f.

7.1.2.3 Throughput and Bandwidth Utilization Analysis

To evaluate memory bus efficiency and processing engine performance across operating frequencies, we analyze the throughput (tp) and utilization (u) metrics for both architectures at a fixed problem size $N = 10000$.

For the baseline architecture, the maximum theoretical MC-to-CE throughput ($\text{tp}_{\text{max,mc-ce}}$) scales linearly as a function of the operating frequency:

$$\text{tp}_{\text{max,mc-ce}}(f) \approx 8.000 \cdot f \tag{7.15}$$

reaching 2000 MB/s at 250 MHz. However, actual MC-to-CE throughput follows:

$$\text{tp}_{\text{mc-ce}}(f) = 0.635 \cdot f + 33.435 \tag{7.16}$$

affected by discrete clock-cycle quantization stalls between 150 MHz $\rightarrow$ 175 MHz and 225 MHz $\rightarrow$ 250 MHz. In regions where latency cycle requirements remain constant, both tp_{max} and measured throughput increase proportionally with frequency. Conversely, crossing latency threshold boundaries introduces additional stall cycles, causing measured throughput to deviate downward relative to the capacity. Consequently, the MC-to-CE bandwidth utilization ($u_{\text{bw,mc-ce}}$) exhibits step-down drops from 11.4% to 9.2% across the frequency range.

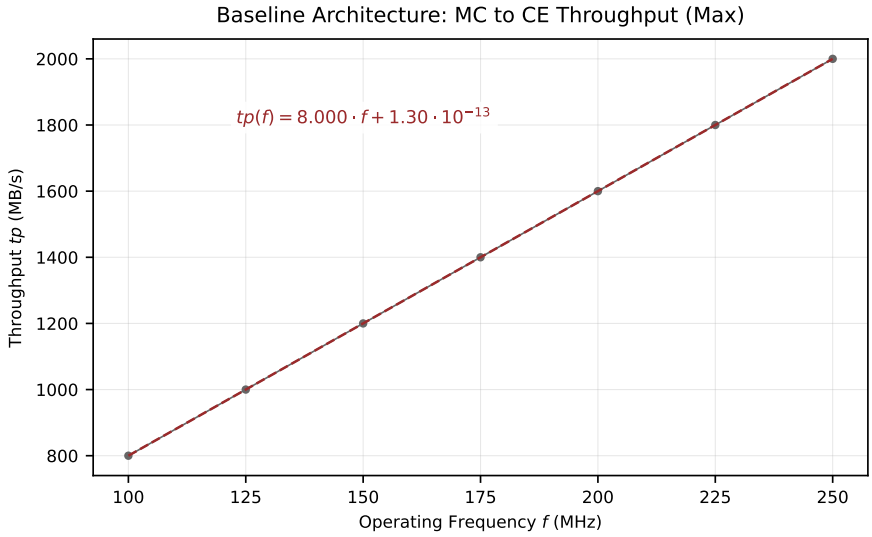


Figure 7.17: Baseline Architecture - MC to CE Maximum Theoretical Throughput across Operating Frequency f.

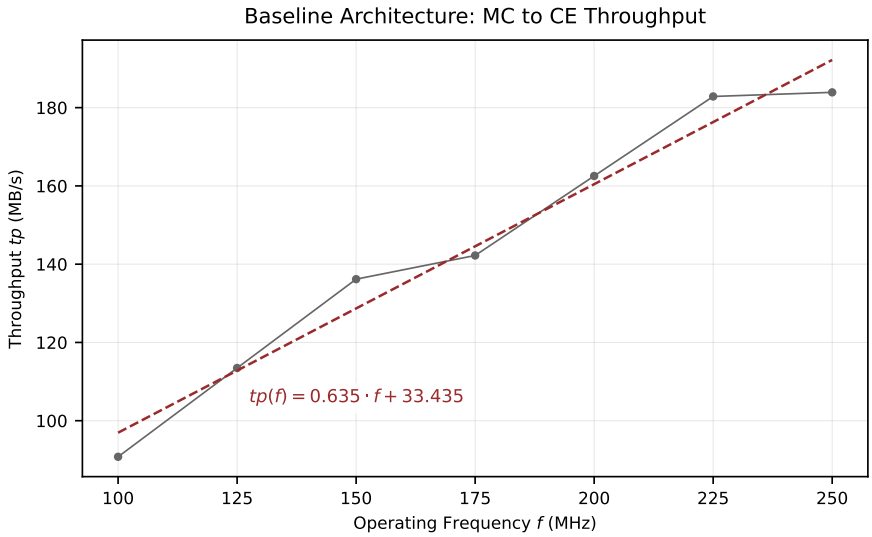


Figure 7.18: Baseline Architecture - MC to CE Throughput across Operating Frequency f.

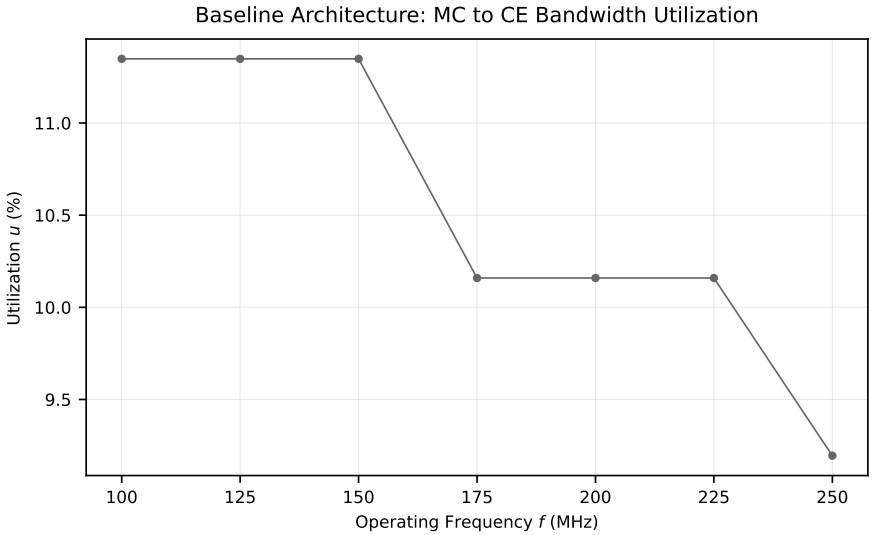


Figure 7.19: Baseline Architecture - MC to CE Bandwidth Utilization across Operating Frequency f.

For the proposed architecture, the SE-to-CE interface has a wider bus, yielding a theoretical maximum throughput of:

$$\mathrm{tp}_{\mathrm{max,se\text{-}ce}}(f) \approx 16.000 \cdot f \tag{7.17}$$

and an effective throughput of:

$$\mathrm{tp}_{\mathrm{se\text{-}ce}}(f) = 4.246 \cdot f + 6.831 \tag{7.18}$$

Bandwidth utilization $u_{\mathrm{bw,se\text{-}ce}}$ scales near 26.8%, representing a $\approx 2.5\times$ efficiency improvement over the MC-to-CE interface.

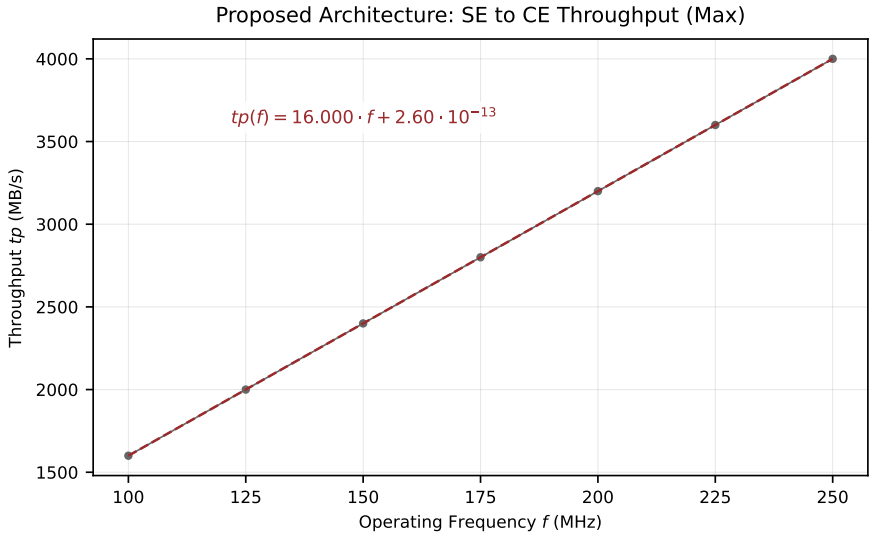


Figure 7.20: Proposed Architecture - SE to CE Maximum Theoretical Throughput across Operating Frequency f.

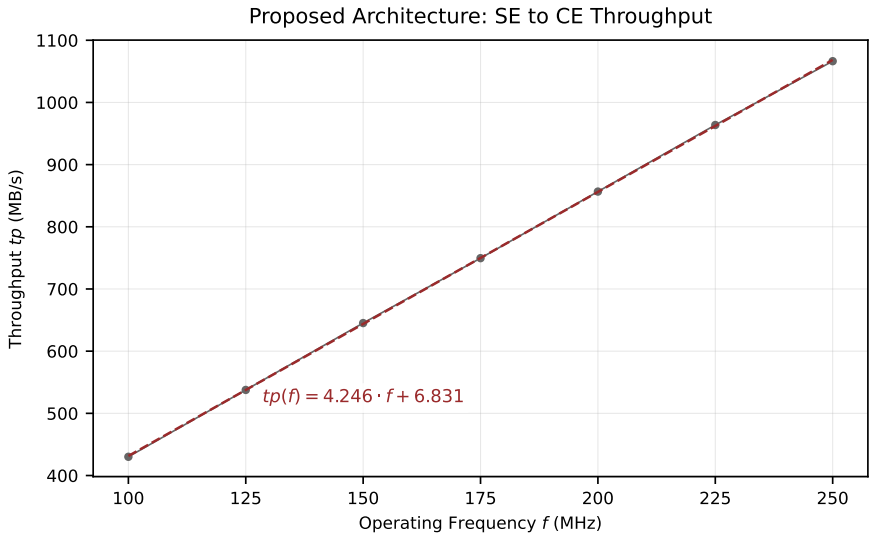


Figure 7.21: Proposed Architecture - SE to CE Throughput across Operating Frequency f.

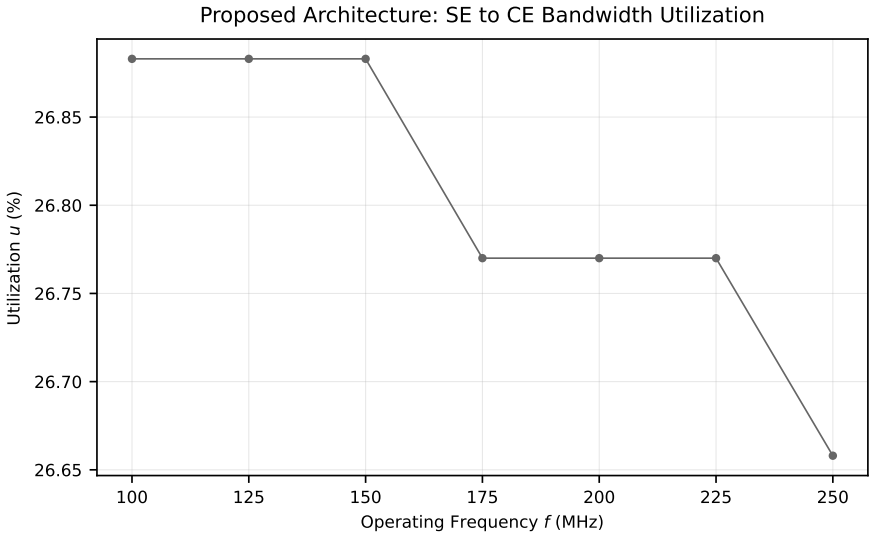


Figure 7.22: Proposed Architecture - SE to CE Bandwidth Utilization across Operating Frequency f.

The maximum CE theoretical throughput ($\mathrm{tp}_{\mathrm{max,ce}}$) scales identically in both architectures:

$$\mathrm{tp}_{\mathrm{max,ce}}(f) \approx 8.000 \cdot f \tag{7.19}$$

However, the baseline CE throughput ($\mathrm{tp}_{\mathrm{ce,bsl}}$) achieves only:

$$\mathrm{tp}_{\mathrm{ce,bsl}}(f) = 0.020 \cdot f + 1.045 \tag{7.20}$$

due to continuous memory stalls, leading to a low compute utilization ($u_{\mathrm{ce,cmp,bsl}}$) between 5.0% and 6.0%.

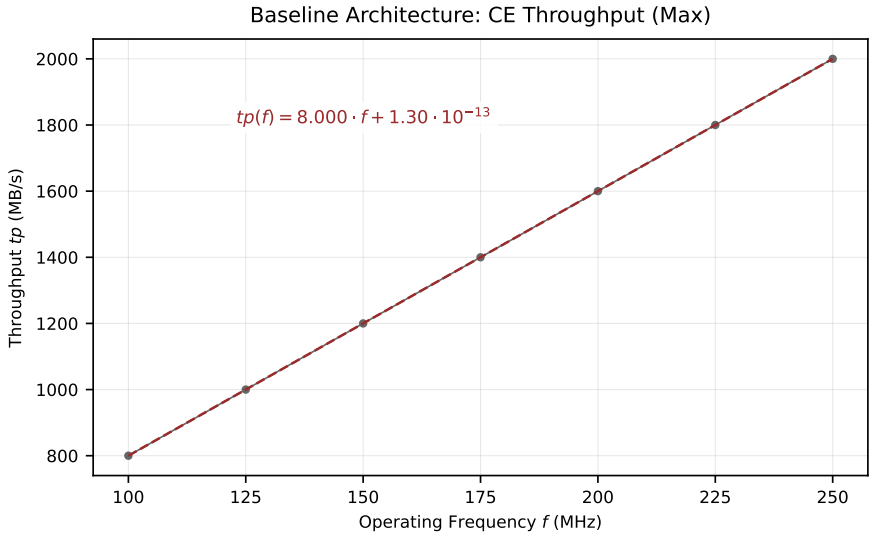


Figure 7.23: Baseline Architecture - CE Maximum Theoretical Throughput across Operating Frequency f.

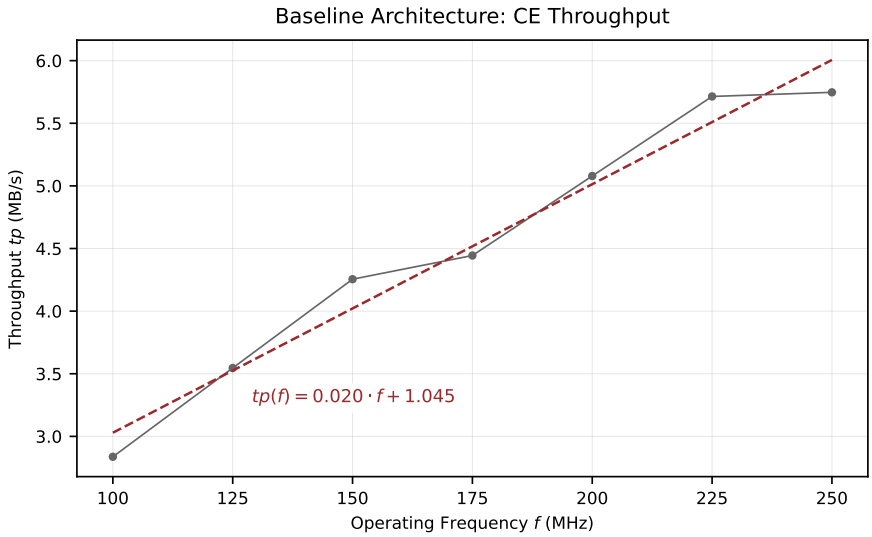


Figure 7.24: Baseline Architecture - CE Throughput across Operating Frequency f.

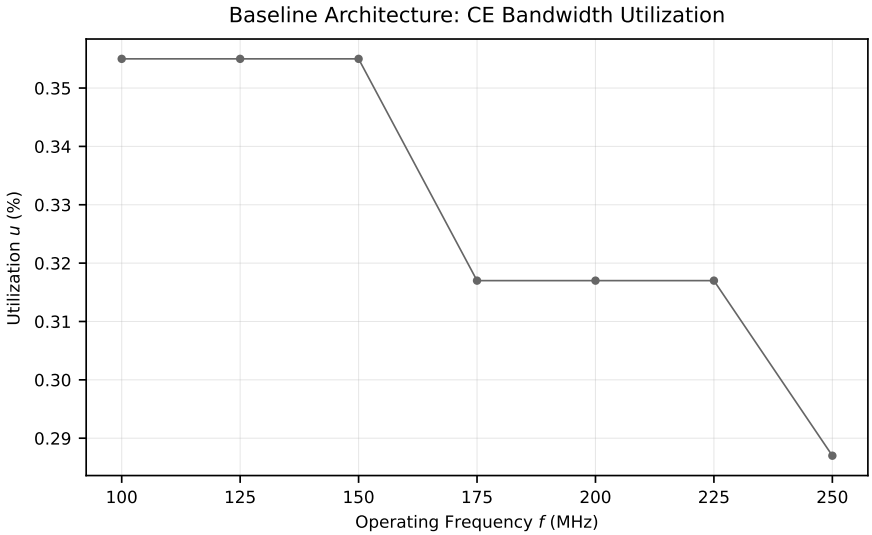


Figure 7.25: Baseline Architecture - CE Bandwidth Utilization across Operating Frequency f.

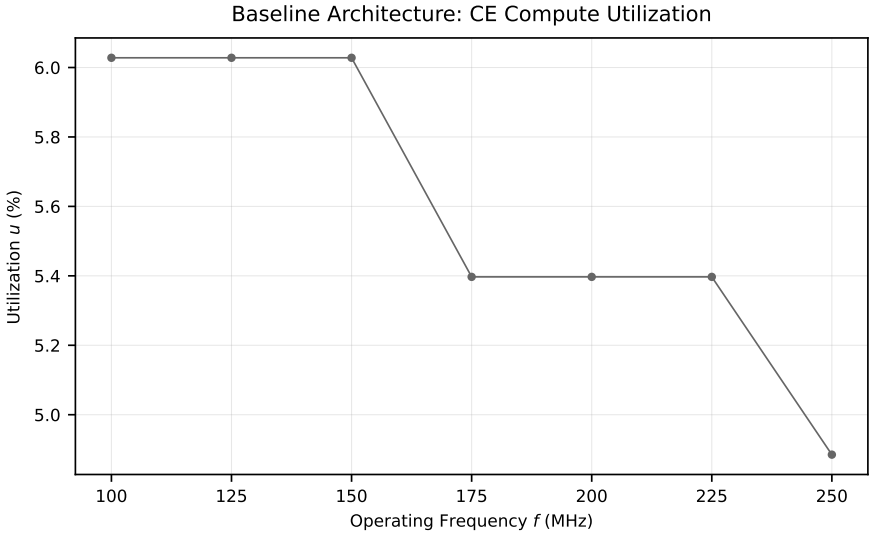


Figure 7.26: Baseline Architecture - CE Compute Utilization across Operating Frequency f.

In contrast, the proposed design achieves:

$$\mathrm{tp}_{\mathrm{ce,prp}}(f) = 0.133 \cdot f + 0.209 \tag{7.21}$$

raising compute utilization $u_{\mathrm{ce,cmp,prp}}$ to $\approx 28.6\%$. Across all utilization metrics (u_{bw} and $u_{\mathrm{ce,cmp}}$), plateau regions correspond to frequency bands with constant memory latency cycles, whereas discrete downward steps mark threshold boundaries (150 MHz $\rightarrow$ 175 MHz and 225 MHz $\rightarrow$ 250 MHz) where cycle quantization increases memory handshake overhead.

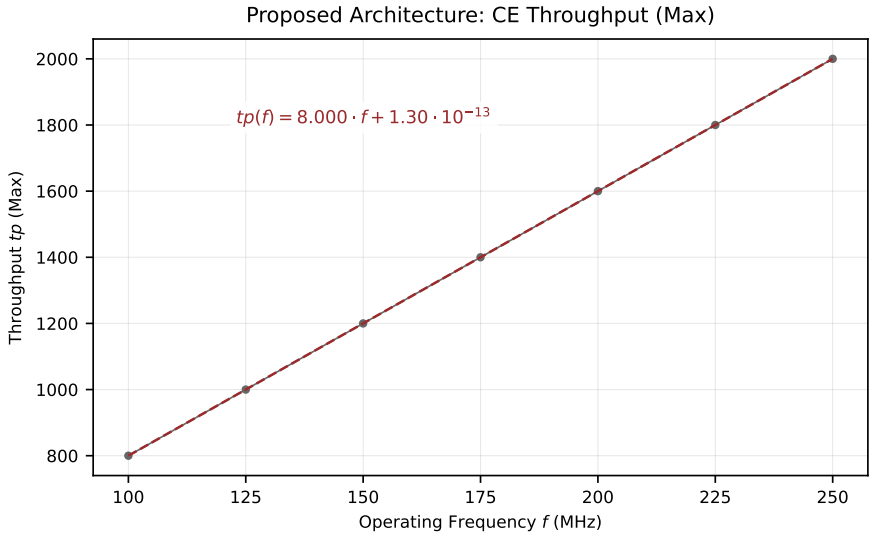


Figure 7.27: Proposed Architecture - CE Maximum Theoretical Throughput across Operating Frequency f.

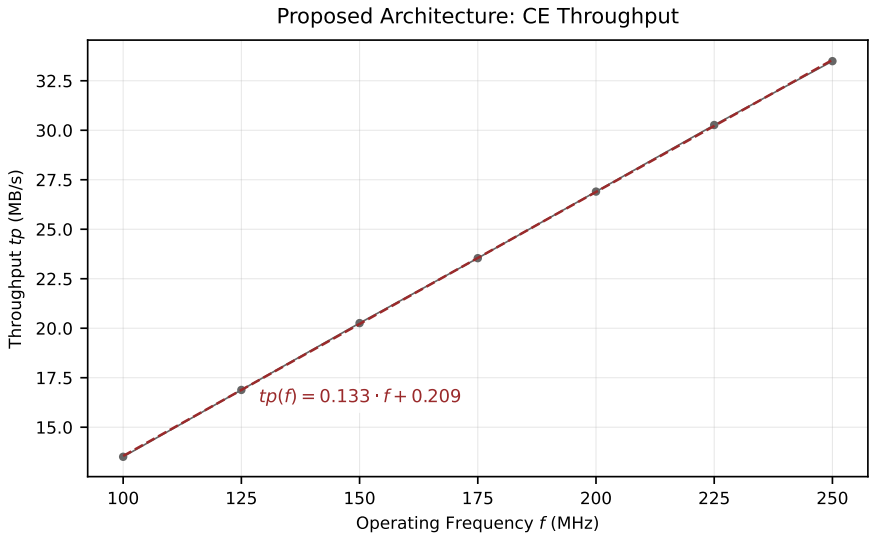


Figure 7.28: Proposed Architecture - CE Throughput across Operating Frequency f.

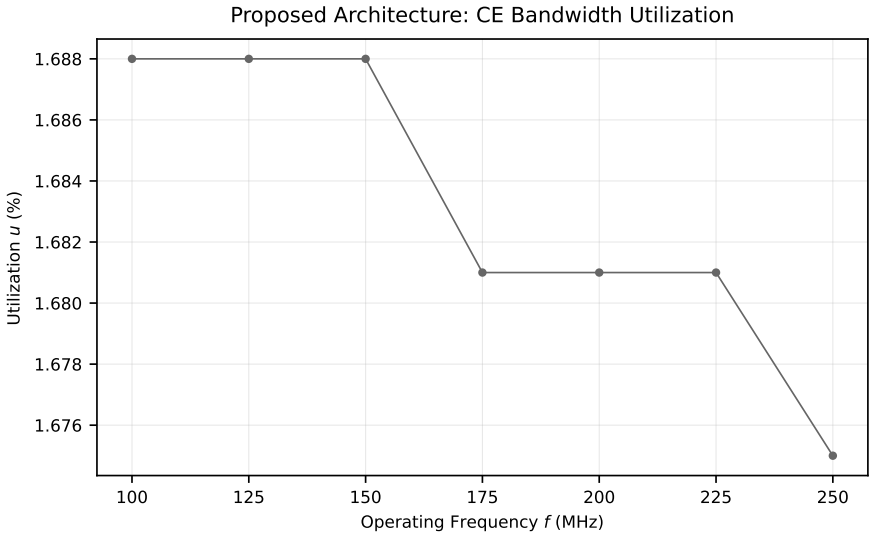


Figure 7.29: Proposed Architecture - CE Bandwidth Utilization across Operating Frequency f.

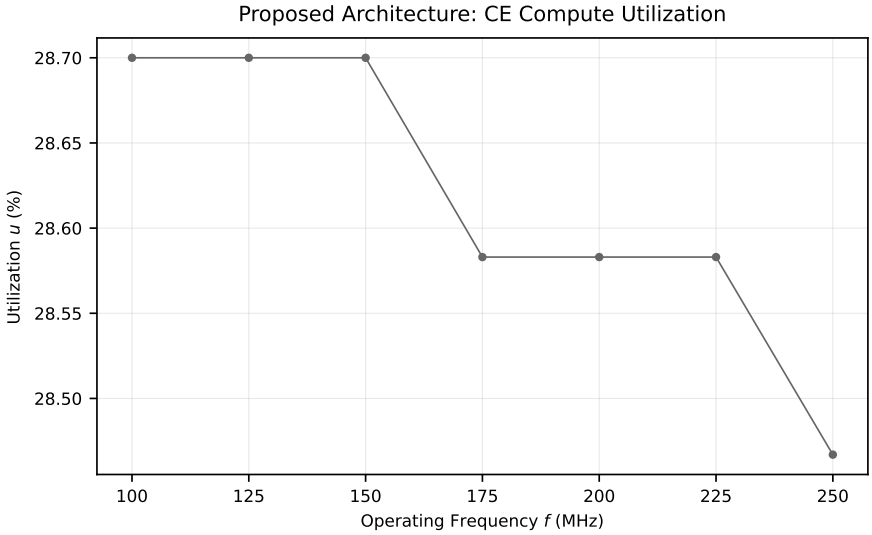


Figure 7.30: Proposed Architecture - CE Compute Utilization across Operating Frequency f.

7.1.3 Memory Latency Sweep

To evaluate memory interface efficiency and system robustness under varying external memory response delays, a memory latency sweep is performed across $l \in [15.0, 35.0]$ ns. As established in previous sections, the problem size is fixed at $N = 10000$. All simulations operate at a constant clock frequency $f = 100$ MHz ($T_{\text{clk}} = 10$ ns).

7.1.3.1 Simulation Time and Speedup Analysis

For the baseline architecture, execution time exhibits a step-like staircase profile as a function of external Memory Latency l. Because access delays are quantized into discrete integer clock cycles ($L = [\![l/T_{\text{clk}}]\!]$), the total simulation time $t_{\text{sim,bsl}}$ remains flat across sub-cycle latency intervals before stepping upward at cycle quantization boundaries:

$$t_{\text{sim,bsl}}(l) = 28200\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.22}$$

$$t_{\text{sim,bsl}}(l) = 31500\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.23}$$

$$t_{\text{sim,bsl}}(l) = 34800\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.24}$$

Due to unbuffered, direct memory accesses for every filter coefficient and sample, each additional cycle of latency (+10 ns) introduces significant cumulative stall overhead, increasing baseline execution time by 3300μs per latency cycle step.

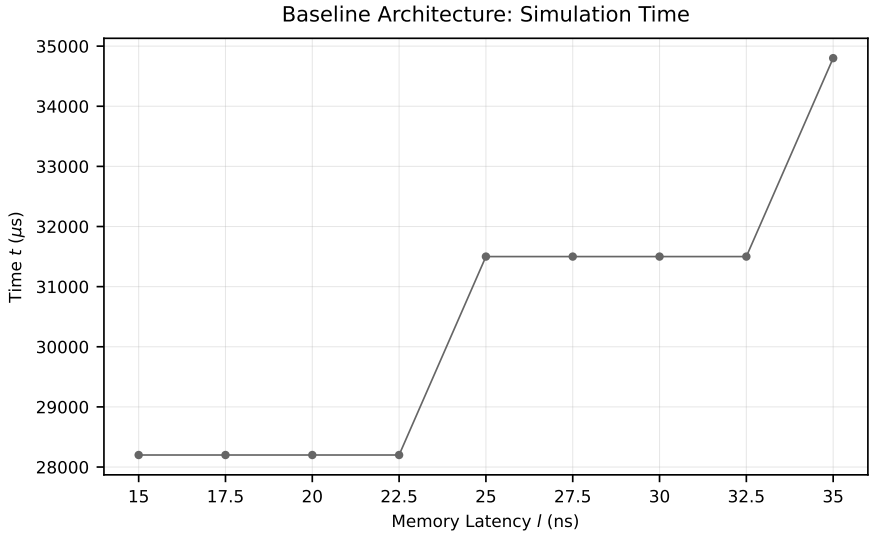


Figure 7.31: Baseline Architecture - Simulation Time across Memory Latency l.

For the proposed architecture, local stream buffering via the SE decouples compute execution from external memory access penalties. Consequently, the simulation time $t_{\text{sim,prp}}$ exhibits far higher resilience to memory latency increases:

$$t_{\text{sim,prp}}(l) = 5951.8\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.25}$$

$$t_{\text{sim,prp}}(l) = 5976.8\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.26}$$

$$t_{\text{sim,prp}}(l) = 6001.8\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.27}$$

Each cycle threshold transition increases total execution time by only 25.0μs, demonstrating that hardware pre-buffering successfully hides memory fetch latencies behind pipeline execution.

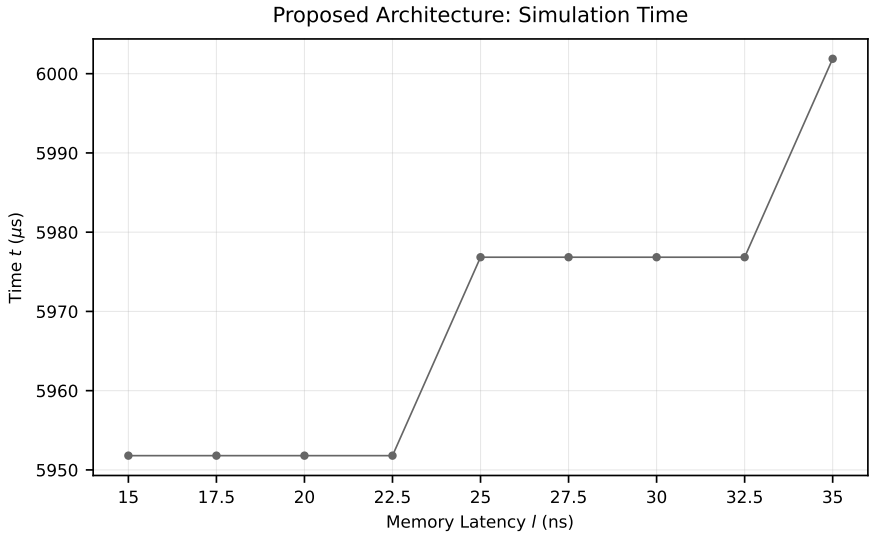


Figure 7.32: Proposed Architecture - Simulation Time across Memory Latency l.

The speedup increases as memory latency degrades:

$$s(l) \approx 4.738\times \quad \text{for } l \in [15.0, 22.5] \text{ ns} \tag{7.28}$$

$$s(l) \approx 5.270\times \quad \text{for } l \in [25.0, 32.5] \text{ ns} \tag{7.29}$$

$$s(l) \approx 5.798\times \quad \text{for } l = 35.0 \text{ ns} \tag{7.30}$$

Because the baseline architecture suffers severe cumulative penalties under higher memory latencies while the proposed design maintains near-constant processing speed, the speedup scales higher at larger values of l.

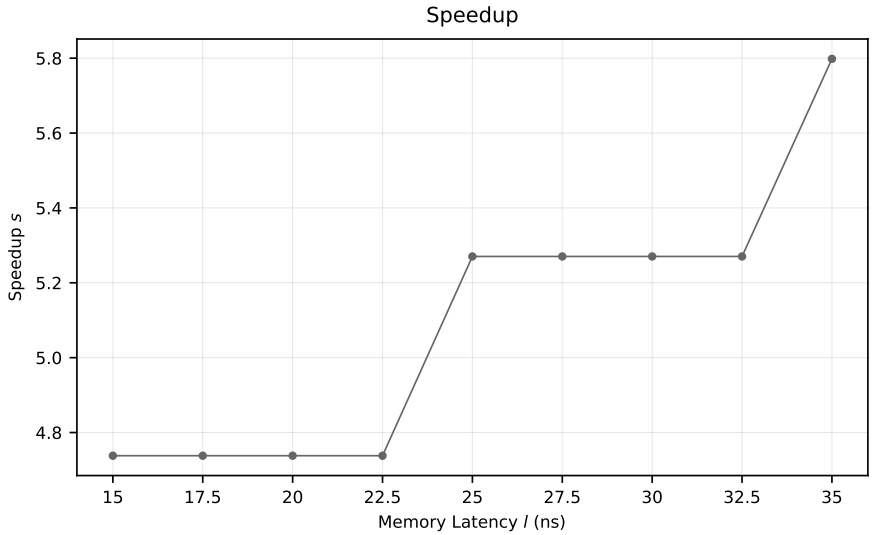


Figure 7.33: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Memory Latency l.

7.1.3.2 Memory Access Analysis

Because memory latency l affects transaction duration rather than the total volume of memory accesses, the MAF remains constant across all values of l for both architectures at a fixed problem size $N = 10000$.

For the baseline architecture, the memory access factor stays invariant at:

$$\text{MAF}_{\text{bsl}} = 33.000 \tag{7.31}$$

reflecting redundant scalar fetches for each tap operation. Consequently, memory communication time (t_{mem}) increases directly with memory latency cycle thresholds:

$$t_{\text{mem,bsl}}(l) = 13200.0\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.32}$$

$$t_{\text{mem,bsl}}(l) = 16500.0\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.33}$$

$$t_{\text{mem,bsl}}(l) = 19800.0\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.34}$$

Each latency cycle step introduces 3300.0μs of additional communication overhead, severely degrading overall system efficiency.

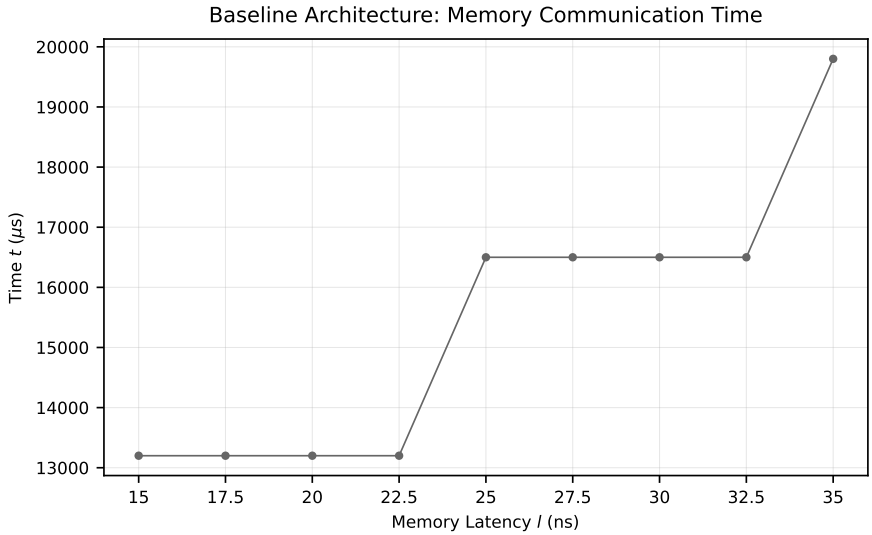


Figure 7.34: Baseline Architecture - Memory Communication Time across Memory Latency l.

For the proposed architecture, the memory access factor is similarly invariant:

$$\mathrm{MAF}_{\mathrm{prp}} = 2.003 \tag{7.35}$$

demonstrating high memory efficiency due to wide-bus vector pre-buffering via the SE. As a result, memory communication time (t_{mem}) is significantly reduced and shows high resilience to increased latency:

$$t_{\mathrm{mem,prp}}(l) = 450.7\mu\mathrm{s} \quad \text{for } l \in [15.0, 22.5]\ \mathrm{ns} \quad (L = 2) \tag{7.36}$$

$$t_{\mathrm{mem,prp}}(l) = 475.7\mu\mathrm{s} \quad \text{for } l \in [25.0, 32.5]\ \mathrm{ns} \quad (L = 3) \tag{7.37}$$

$$t_{\mathrm{mem,prp}}(l) = 500.7\mu\mathrm{s} \quad \text{for } l = 35.0\ \mathrm{ns} \quad (L = 4) \tag{7.38}$$

Each latency cycle step increases communication time by only 25.0μs, representing a $\approx 29\times$ to $40\times$ reduction in communication delay compared to the baseline design.

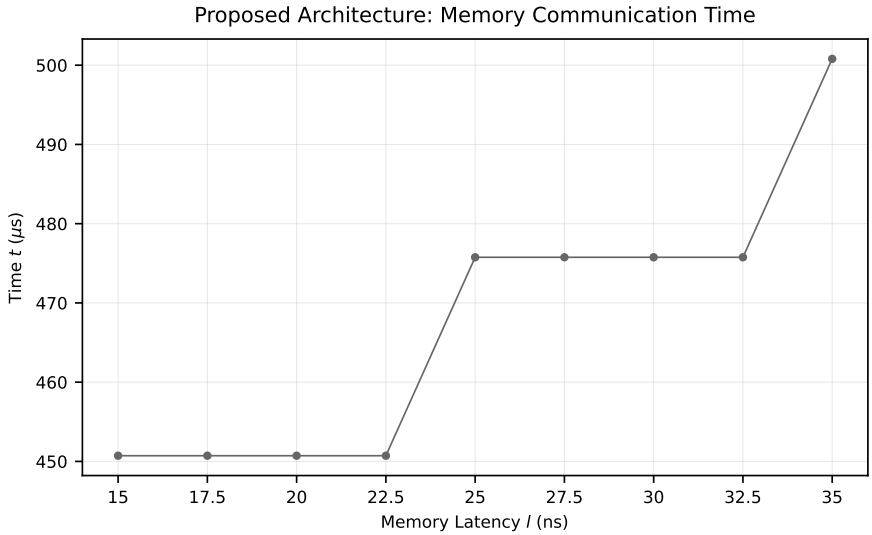


Figure 7.35: Proposed Architecture - Memory Communication Time across Memory Latency l.

7.1.3.3 Throughput and Bandwidth Utilization Analysis

To evaluate memory interface efficiency and compute pipeline performance under varying memory response delays, throughput (tp) and utilization (u) metrics are analyzed across $l \in [15.0, 35.0]$ ns at $f = 100$ MHz and $N = 10000$.

For the baseline architecture, the maximum theoretical throughput fo the MC-to-CE interface is $\text{tp}_{\text{max,mc-ce}} = 800.0$ MB/s. The effective MC-to-CE throughput ($\text{tp}_{\text{mc-ce}}$) follows a step-down pattern governed by discrete clock cycle quantization ($L = [\![l/T_{\text{clk}}]\!]$), as depicted in Figure 7.36:

$$\text{tp}_{\text{mc-ce}}(l) = 90.7 \text{ MB/s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.39}$$

$$\text{tp}_{\text{mc-ce}}(l) = 81.3 \text{ MB/s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.40}$$

$$\text{tp}_{\text{mc-ce}}(l) = 73.6 \text{ MB/s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.41}$$

As shown in Figure 7.37, this decline causes bus bandwidth utilization ($u_{\text{bw,mc-ce}}$) to drop from 11.35% to 10.16% and down to 9.20% across cycle thresholds.

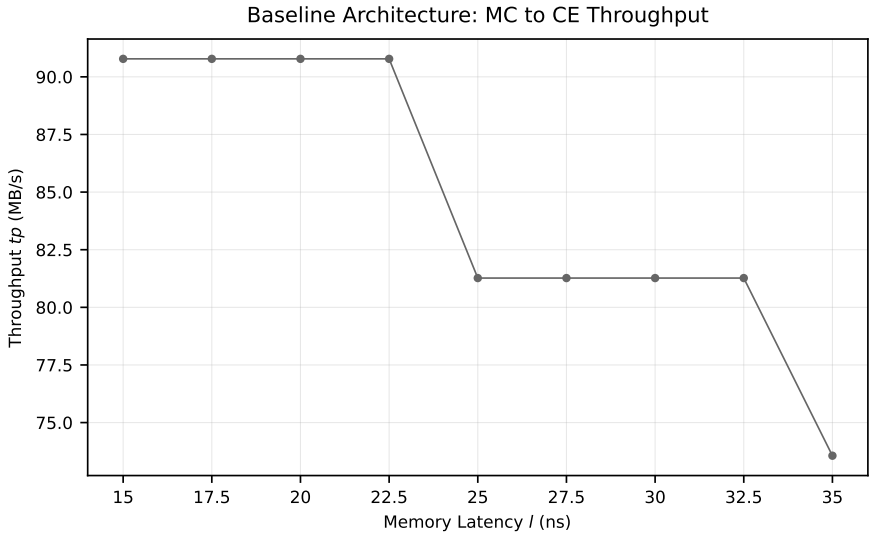


Figure 7.36: Baseline Architecture - MC to CE Throughput across Memory Latency l.

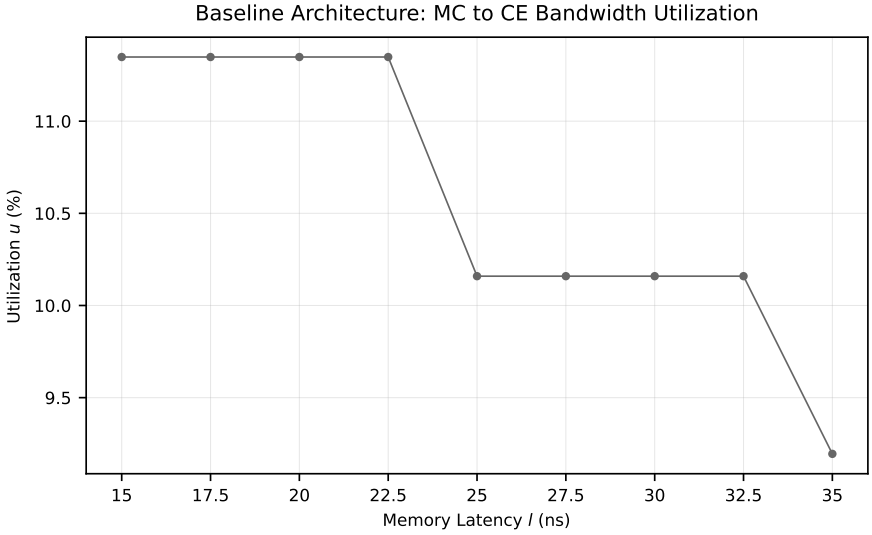


Figure 7.37: Baseline Architecture - MC to CE Bandwidth Utilization across Memory Latency l.

In contrast, the proposed architecture leverages pre-buffering via the SE to shield the core pipeline from memory latency penalties. With a maximum theoretical throughput of $\text{tp}_{\text{max,se-ce}} = 1600.0$ MB/s, the effective throughput ($\text{tp}_{\text{se-ce}}$) shown in Figure 7.38 remains highly stable across latency boundaries:

$$\text{tp}_{\text{se-ce}}(l) = 430.12 \text{ MB/s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.42}$$

$$\text{tp}_{\text{se-ce}}(l) = 428.32 \text{ MB/s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.43}$$

$$\text{tp}_{\text{se-ce}}(l) = 426.54 \text{ MB/s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.44}$$

As illustrated in Figure 7.39, this yields a consistent bandwidth utilization ($u_{\text{bw,se-ce}}$) between 26.88% and 26.66%, representing a $\approx 2.4\times$ to $2.9\times$ improvement over the baseline memory interface.

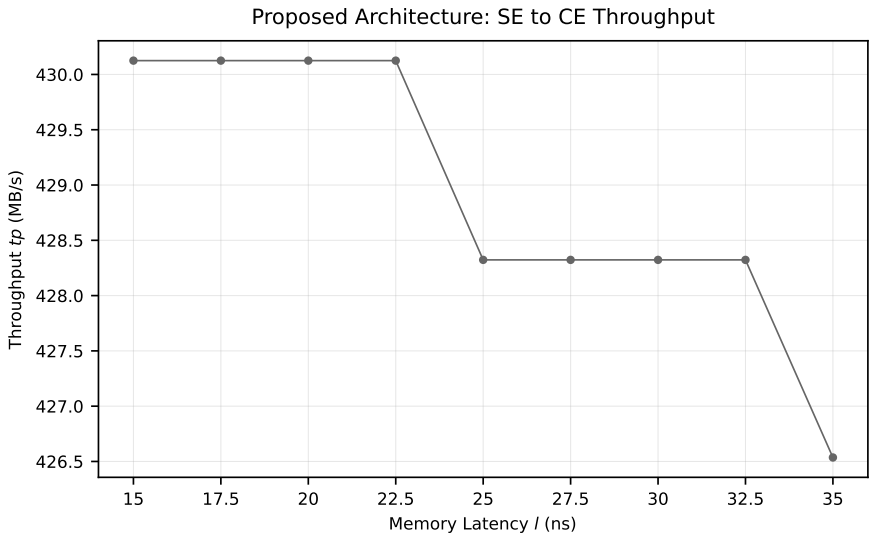


Figure 7.38: Proposed Architecture - SE to CE Throughput across Memory Latency l.

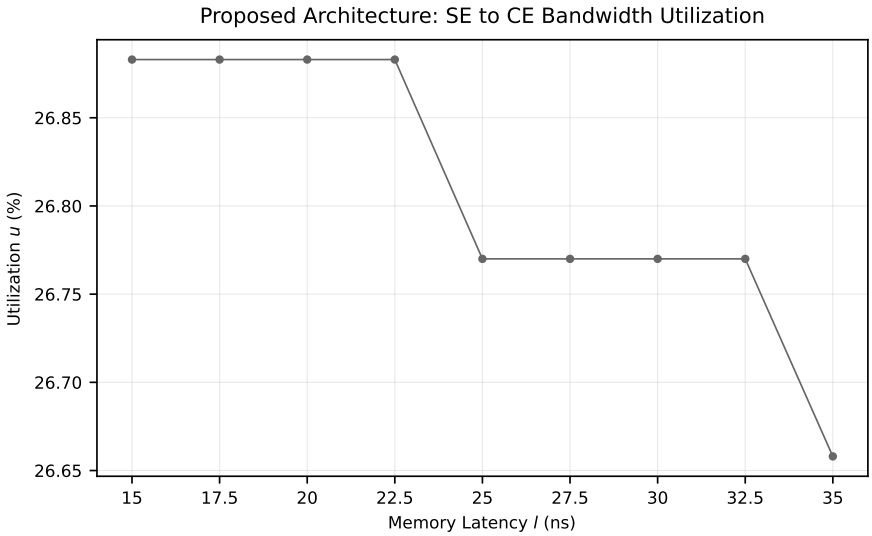


Figure 7.39: Proposed Architecture - SE to CE Bandwidth Utilization across Memory Latency l.

At the processing engine interface, where maximum theoretical compute throughput is bounded by $\text{tp}_{\text{max,ce}} = 800.0$ MB/s, severe memory stalls under unbuffered accesses degrade the baseline CE throughput ($\text{tp}_{\text{ce,bsl}}$) from 2.84 MB/s down to 2.30 MB/s (Figure 7.40). This causes baseline CE bandwidth utilization ($u_{\text{bw,ce,bsl}}$) to plummet to 0.287% (Figure 7.41) and compute utilization ($u_{\text{ce,cmp,bsl}}$) to drop from 6.03% to 4.89% (Figure 7.42).

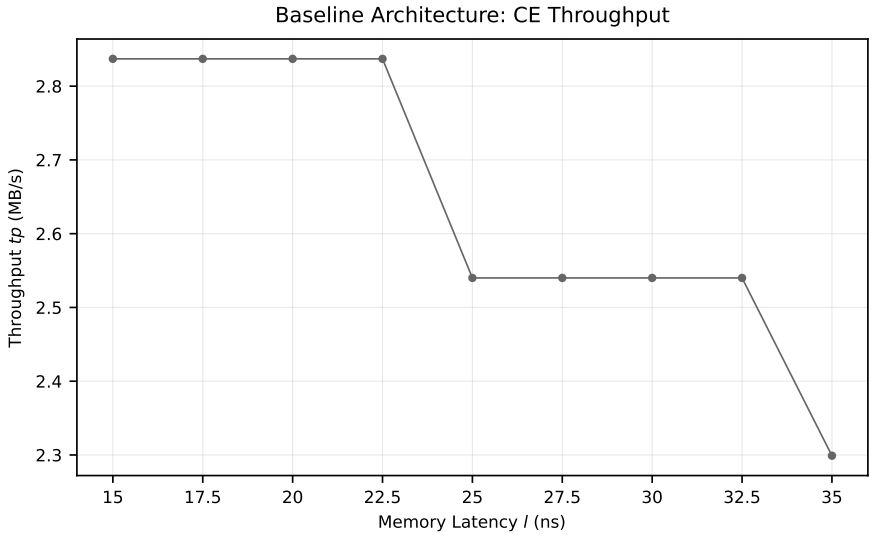


Figure 7.40: Baseline Architecture - CE Throughput across Memory Latency l.

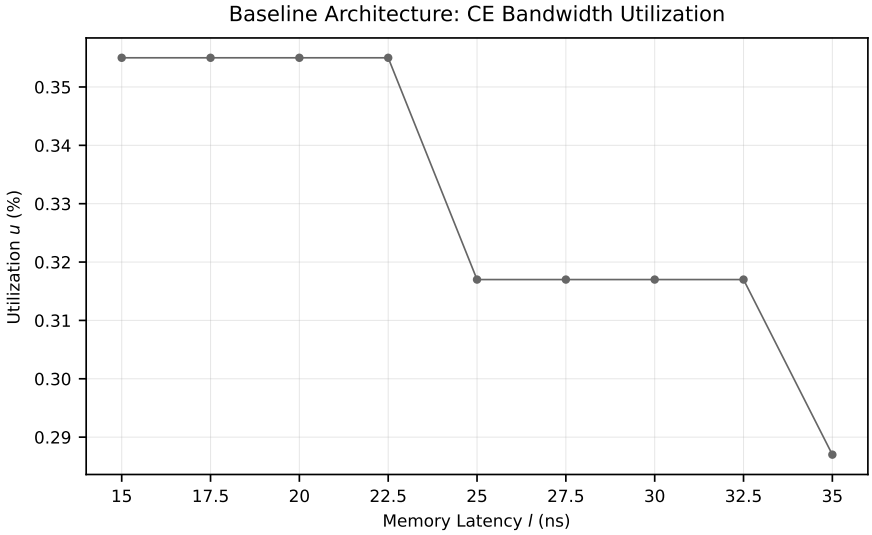


Figure 7.41: Baseline Architecture - CE Bandwidth Utilization across Memory Latency l.

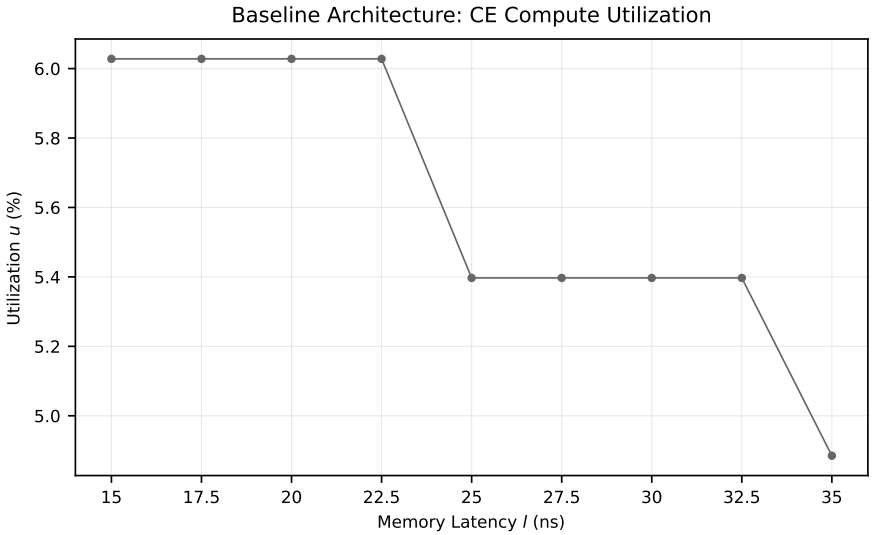


Figure 7.42: Baseline Architecture - CE Compute Utilization across Memory Latency l.

The proposed architecture reduces pipeline starvation, maintaining a high CE throughput ($\text{tp}_{\text{ce,prp}}$) between 13.51 MB/s and 13.39 MB/s, as seen in Figure 7.43. Consequently, proposed CE bandwidth utilization ($u_{\text{bw,ce,prp}}$) stays near 1.68% (Figure 7.44), while compute utilization ($u_{\text{ce,cmp}}$) is sustained at $\approx 28.5\% - 28.7\%$ (Figure 7.45), demonstrating reduced latency sensitivity.

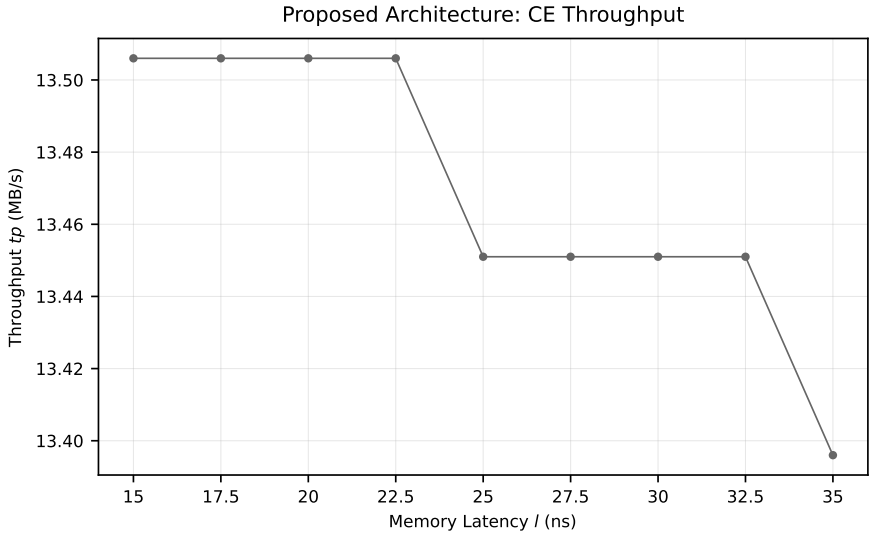


Figure 7.43: Proposed Architecture - CE Throughput across Memory Latency l.

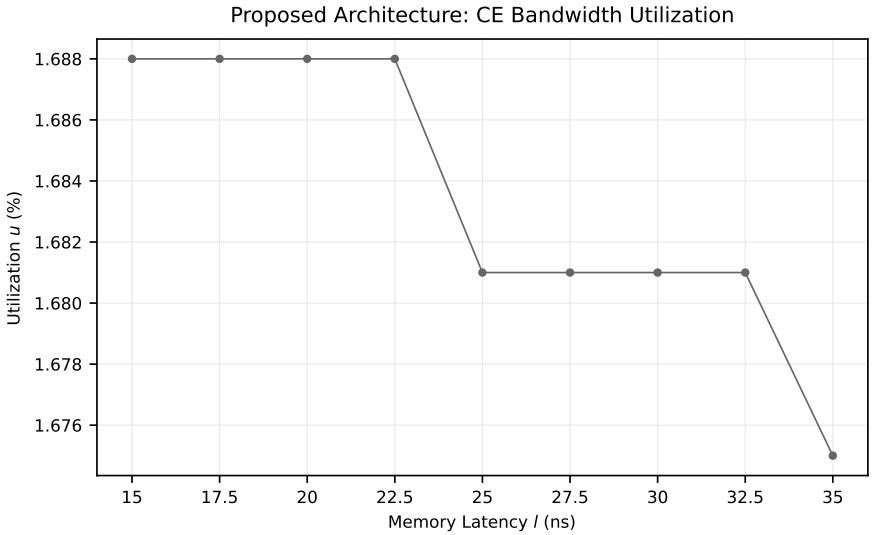


Figure 7.44: Proposed Architecture - CE Bandwidth Utilization across Memory Latency l.

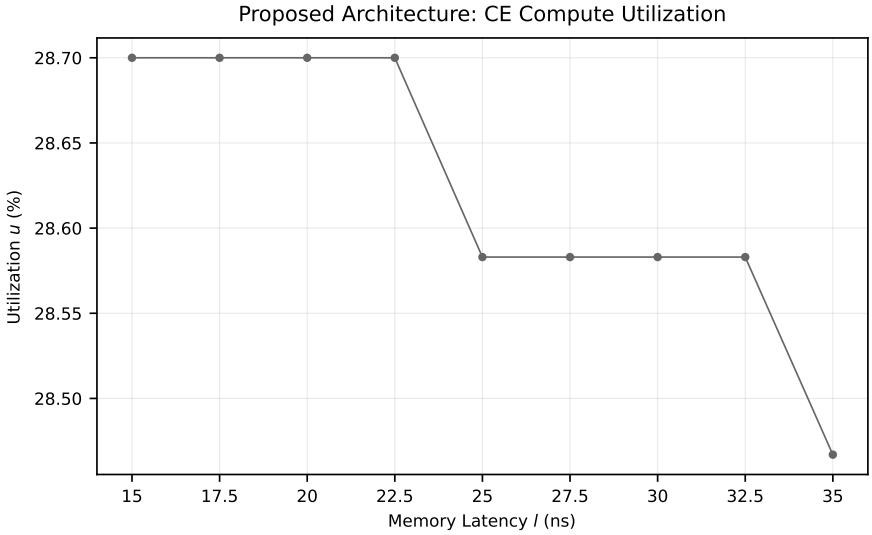


Figure 7.45: Proposed Architecture - CE Compute Utilization across Memory Latency l.

7.2 Matrix Multiplication

7.2.1 Problem Size Sweep

To analyze the scaling behavior of the Matrix Multiplication benchmark, the matrix dimension N is swept across $N \in [31, 126]$. In this setup, all SBs are configured in standard FIFO mode without hardware circular buffering, forcing the system to re-fetch elements across the 3-level nested loop iteration space. This section evaluates the impact of problem size scaling on execution latency, memory communication overheads, compute utilization, and memory access factors for both the Baseline and proposed architectures.

7.2.1.1 Simulation Time and Speedup Analysis

Because the problem formulation defines square input matrices ($A, B, C \in R^{N \times N}$), the total number of elements per matrix scales quadratically as N^2. Consequently, execution latency across both architectural variants exhibits an $O(N^2)$ relation over the evaluated sweep range.

For the baseline architecture, simulation execution time scales quadratically as shown in Figure 7.46, governed by the fitted regression model:

$$t_{\text{sim,bsl}}(N) = 40.135 \cdot N^2 - 2889.235 \cdot N + 6.23 \cdot 10^4 \quad [\mu\text{s}] \tag{7.45}$$

Without stream pre-buffering or local caching mechanisms, the baseline system executes single-element memory requests. While every transaction incurs memory latency stalls, it strictly fetches the target address without loading unused spatial metadata into local buffers.

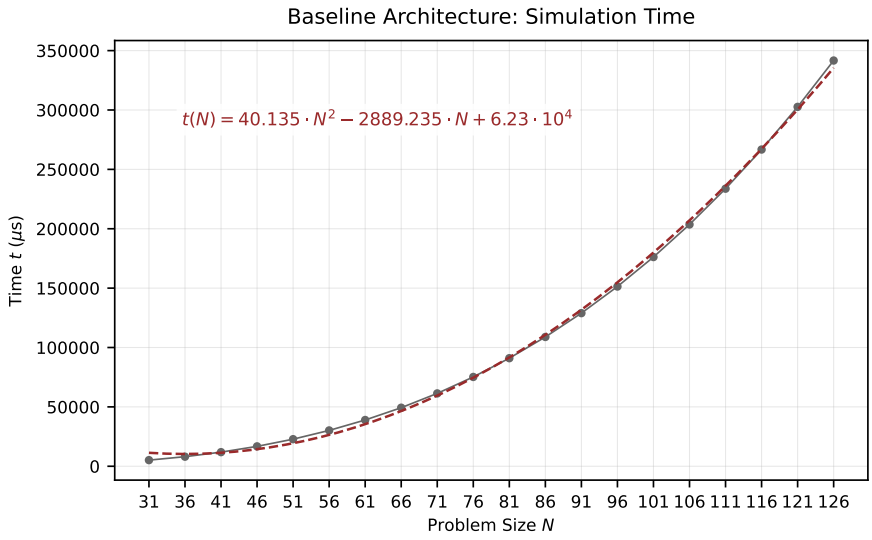


Figure 7.46: Baseline Architecture - Simulation Time across Problem Size N.

For the proposed architecture, execution time also follows a quadratic trend as shown in Figure 7.47, modeled by:

$$t_{\text{sim,prp}}(N) = 93.179 \cdot N^2 - 7585.117 \cdot N + 1.66 \cdot 10^5 \quad [\mu\text{s}] \tag{7.46}$$

The key factor behind the degraded performance of the proposed architecture lies in the interplay between column-wise strided memory layouts and the capacity constraints of the SCs. Each SC is parameterized with a depth $\text{SC}_{\text{DPT}} = 64$ and a width $\text{SC}_{\text{WDT}} = 8$ (DW = 64 bits), yielding a small local capacity C_{SC} of:

$$C_{\text{SC}} = \frac{\text{SC}_{\text{DPT}} \cdot \text{SC}_{\text{WDT}} \cdot \text{DW}}{8 \cdot 1000} = \frac{64 \cdot 8 \cdot 64}{8000} = 4.096 \quad [\text{KB}] \tag{7.47}$$

When traversing Matrix B along a column (where data is stored contiguously along rows in memory), every cache miss triggers a full burst transaction to fill an 8-element line ($\text{SC}_{\text{WDT}} = 8$). For a matrix size such as 50×50, fetching a single column element loads that word alongside 7 adjacent row elements. Because the kernel immediately requests the next vertical element in the column rather than the adjacent horizontal words, these pre-fetched elements offer zero spatial locality for the immediate operation.

Due to the limited 4.096 KB capacity and the lack of column-preserving circular buffering logic, two primary mechanisms cause severe cache thrashing:

1. **Conflict and Tag Replacement:** Disperse strided accesses from different matrix addresses map to overlapping cache sets, prematurely evicting previously buffered cache lines before their contents can be reused.

2. **Spatial Pollution:** Unused horizontal elements loaded during line fills pollute the small SC capacity, rapidly displacing valid data required for subsequent loop iterations.

Consequently, every column access results in a cache miss that forces a full burst transfer over the memory bus. Loading redundant line-fill data on every miss significantly increases the total memory communication time, causing the proposed architecture to exhibit higher simulation latency than the baseline design.

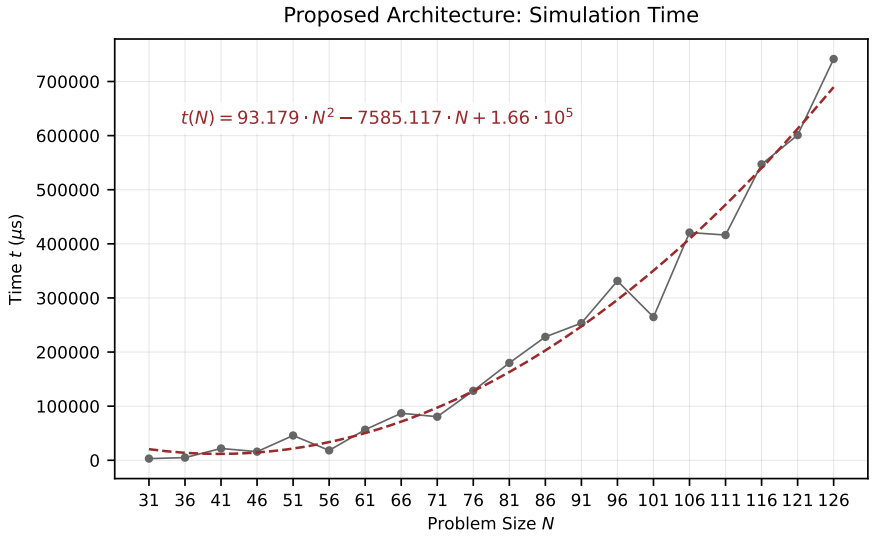


Figure 7.47: Proposed Architecture - Simulation Time across Problem Size N.

The resulting relative speedup $s(N)$ is shown in Figure 7.48. While local buffering and burst transfers provide brief performance advantages for small matrix dimensions ($N \leq 36$) and specific burst-aligned boundaries ($N = 56$), performance drops substantially as N increases. For larger matrices, spatial pollution and continuous SC line-fill thrashing dominate execution time, causing speedup to settle into a low regime ($s \approx 0.45 - 0.65$).

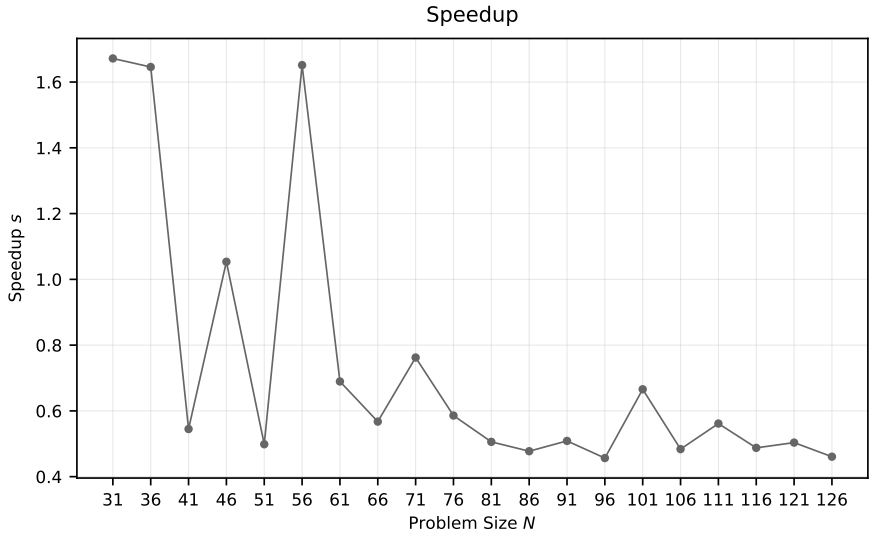


Figure 7.48: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Problem Size N.

7.2.1.2 Memory Access Analysis

To evaluate physical memory efficiency under dense matrix multiplication, memory traffic characteristics are examined through the MAF and the total memory communication time (t_{mem}).

For the baseline architecture, the MAF scales perfectly linearly with problem size N, as shown in Figure 7.49. The relationship is defined by:

$$\text{MAF}_{\text{bsl}}(N) = 2.000 \cdot N + 1.000 \tag{7.48}$$

This linear progression directly reflects the underlying 3-level nested loop structure without circular buffering in the SBs. To compute an $N \times N$ matrix product, matrix A's rows are fetched once per output row, whereas matrix B's columns are re-read N times across the outer loop. Because the baseline architecture executes non-buffered single-word requests, its memory transactions scale strictly in proportion to algorithmic re-fetches without introducing cache line fill overheads.

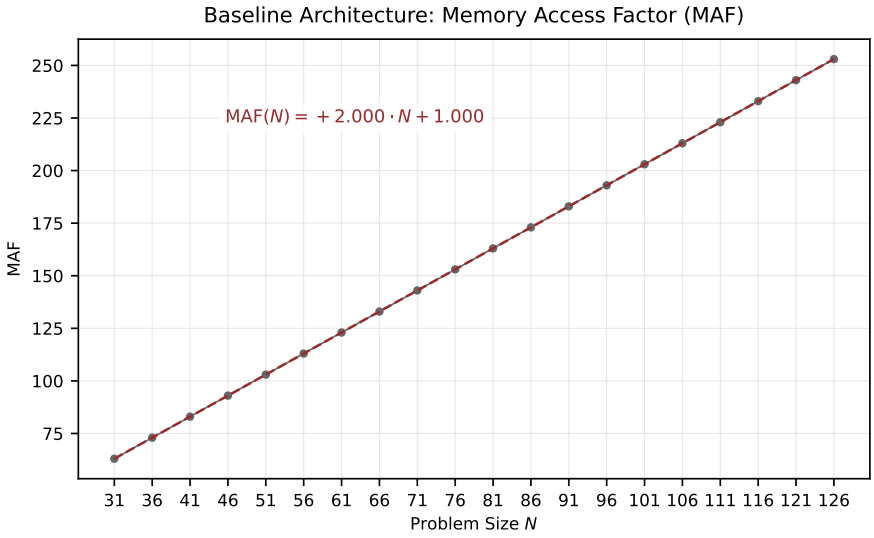


Figure 7.49: Baseline Architecture - Memory Access Factor (MAF) across Problem Size N.

The baseline memory communication time follows a smooth quadratic scaling curve, illustrated in Figure 7.50, modeled by the fitted second-order regression:

$$t_{\text{mem, bsl}}(N) = 18.880 \cdot N^2 - 1359.640 \cdot N + 2.93 \cdot 10^4 \quad [\mu\text{s}] \tag{7.49}$$

Because individual word requests incur predictable memory bus handshakes without burst stall variations, communication latency tracks total algorithmic element accesses directly.

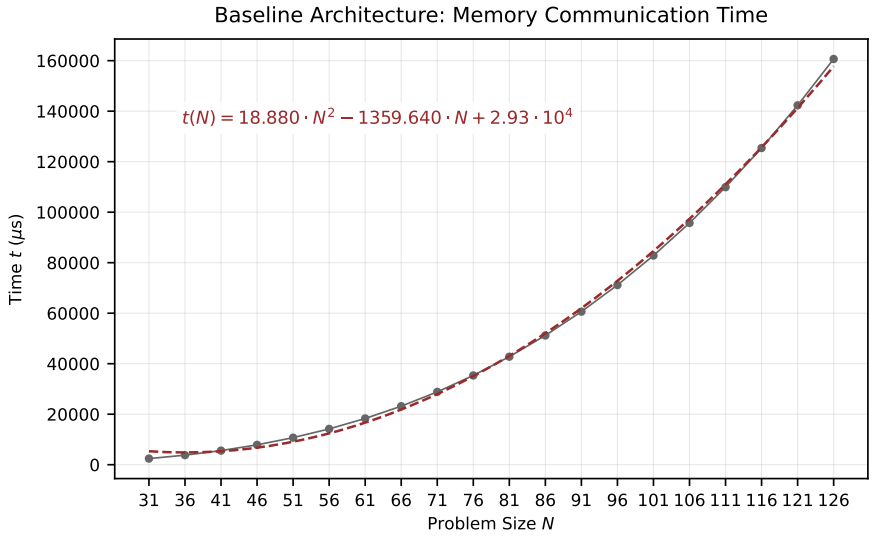


Figure 7.50: Baseline Architecture - Memory Communication Time across Problem Size N.

For the proposed architecture, the MAF exhibits drastically higher values and prominent non-monotonic oscillations, as shown in Figure 7.51. Rather than scaling moderately around $2N$, the MAF escalates up to ≈ 1000 at $N = 126$. This severe degradation is caused by severe thrashing within the SCs ($C_{\text{SC}} = 4.096$ KB). When traversing Matrix B along non-contiguous column strides, each SC miss forces a burst transfer of an entire cache line ($\text{SC}_{\text{WDT}} = 8$). Because horizontal adjacent elements loaded into the SC during line fills offer zero spatial locality for immediate vertical column operations, spatial pollution rapidly evicts useful lines. Continuous tag replacements force redundant multi-element line fills for nearly every scalar access, inflating the MAF far beyond the unbuffered baseline.

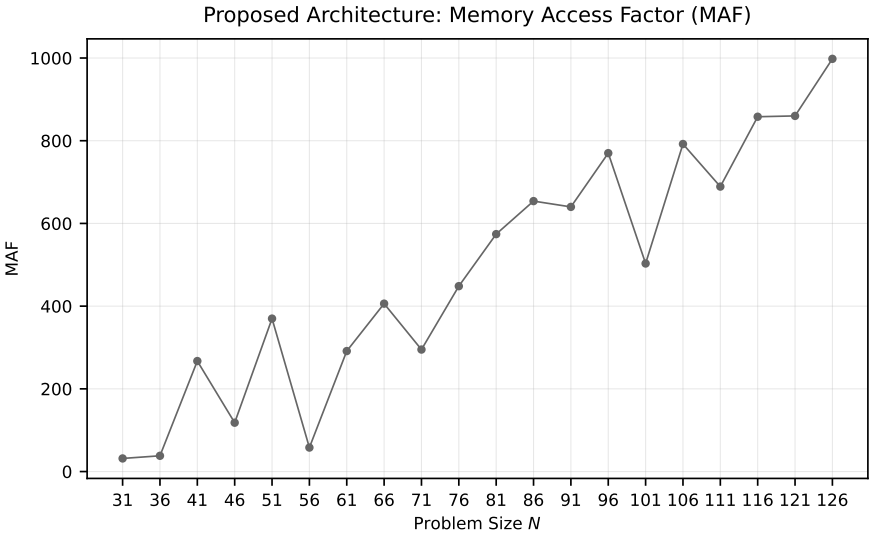


Figure 7.51: Proposed Architecture - Memory Access Factor (MAF) across Problem Size N.

The sharp periodic dips in the proposed MAF (e.g., at $N = 56$ and $N = 101$) mark matrix dimensions where row strides align favorably with the SC line mapping factors, temporarily mitigating conflict evictions.

The memory communication time for the proposed architecture, presented in Figure 7.52, reflects these SC line-fill penalties through quadratic growth described by:

$$t_{\text{mem, prp}}(N) = 45.382 \cdot N^2 - 3775.154 \cdot N + 8.29 \cdot 10^4 \quad [\mu\text{s}] \tag{7.50}$$

Due to continuous line-fill fetches over the memory bus during column traversals, memory communication time is more than double that of the baseline design for large N. These results demonstrate that under heavy cache pollution and premature line evictions — where the system can exploit neither local SC caching nor SB circular buffering — the proposed design incurs severe performance penalties. Because every miss triggers a burst transaction across the memory bus rather than a single request, the inflated latency of these memory transfers ultimately degrades total execution time below that of the unbuffered baseline.

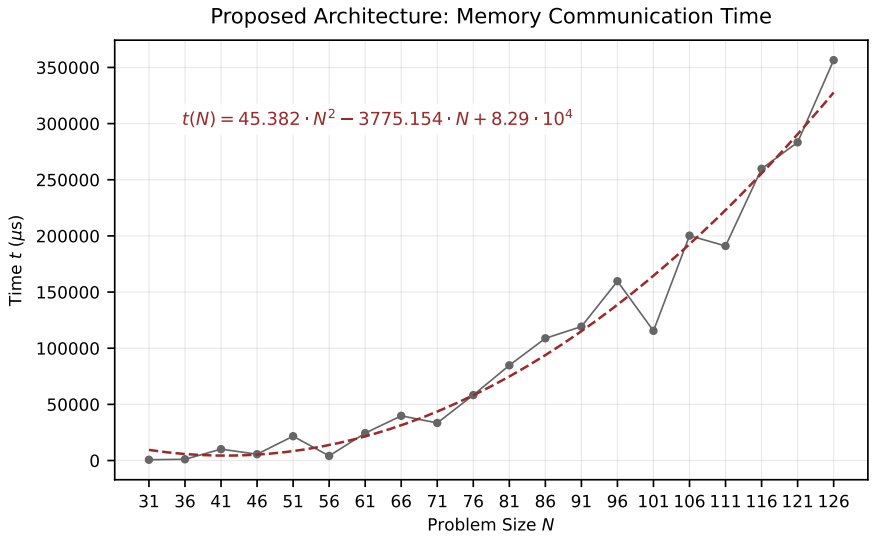


Figure 7.52: Proposed Architecture - Memory Communication Time across Problem Size N.

7.2.1.3 Throughput and Bandwidth Utilization Analysis

For the baseline architecture, memory transfers flow directly from the MC to the CE, bounded by a maximum theoretical throughput of 800 MB/s. As shown in Figure 7.53 and Figure 7.54, MC-to-CE throughput ($\text{tp}_{\text{mc-ce}}$) and bandwidth utilization ($u_{\text{bw,mc-ce}}$) scale smoothly according to second-order regression models:

$$\text{tp}_{\text{mc-ce}}(N) = -1.69 \cdot 10^{-4} \cdot N^2 + 0.038 \cdot N + 91.466 \quad [\text{MB/s}] \tag{7.51}$$

$$u_{\text{bw,mc-ce}}(N) = -2.10 \cdot 10^{-5} \cdot N^2 + 4.77 \cdot 10^{-3} \cdot N + 11.434 \quad [\%] \tag{7.52}$$

Due to unbuffered single-word requests incurring fixed memory access latencies, interconnect bandwidth utilization remains constrained within a narrow 11.55% – 11.71% range relative to the 800 MB/s limit.

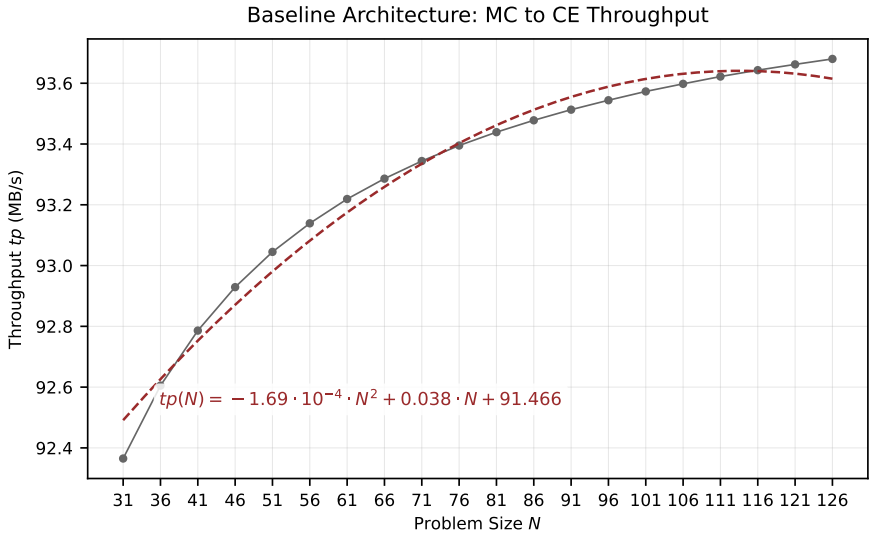


Figure 7.53: Baseline Architecture - MC to CE Throughput across Problem Size N.

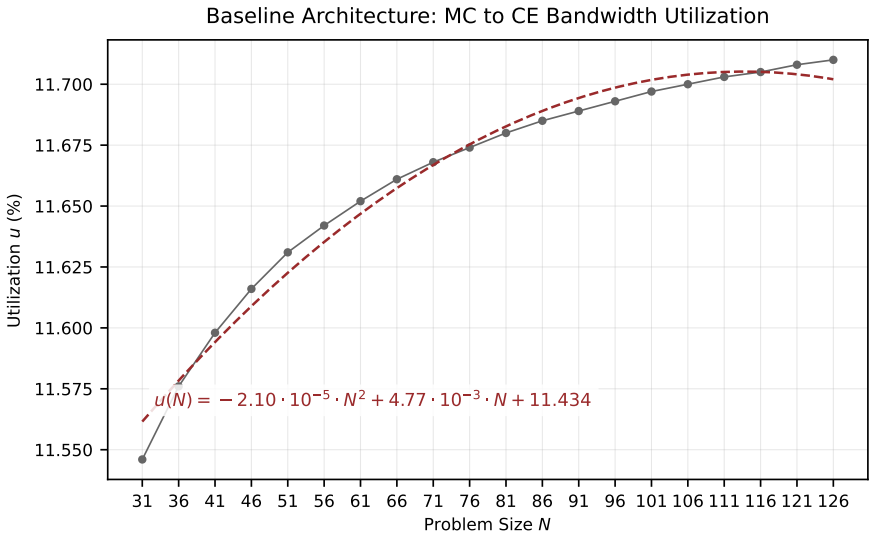


Figure 7.54: Baseline Architecture - MC to CE Bandwidth Utilization across Problem Size N.

In the proposed architecture, data is delivered from the SE to the CE over a wider interface with a maximum theoretical throughput of 1600 MB/s. However, as shown in Figure 7.55 and Figure 7.56, throughput ($\text{tp}_{\text{se-ce}}$) and utilization ($u_{\text{bw,se-ce}}$) exhibit severe non-monotonic fluctuations rather than smooth scaling.

At small problem sizes ($N \leq 36$) and specific burst-aligned dimensions ($N = 56$), the SE achieves peak throughputs up to ≈ 155 MB/s ($u_{\text{bw,se-ce}} \approx 9.6\%$). However, as N increases, spatial pollution and continuous SC cache line thrashing force long stall cycles while waiting for burst transactions over the memory bus. As a result, SE-to-CE throughput degrades sharply, dropping to ≈ 43 MB/s ($u_{\text{bw,se-ce}} < 3\%$) for larger matrices.

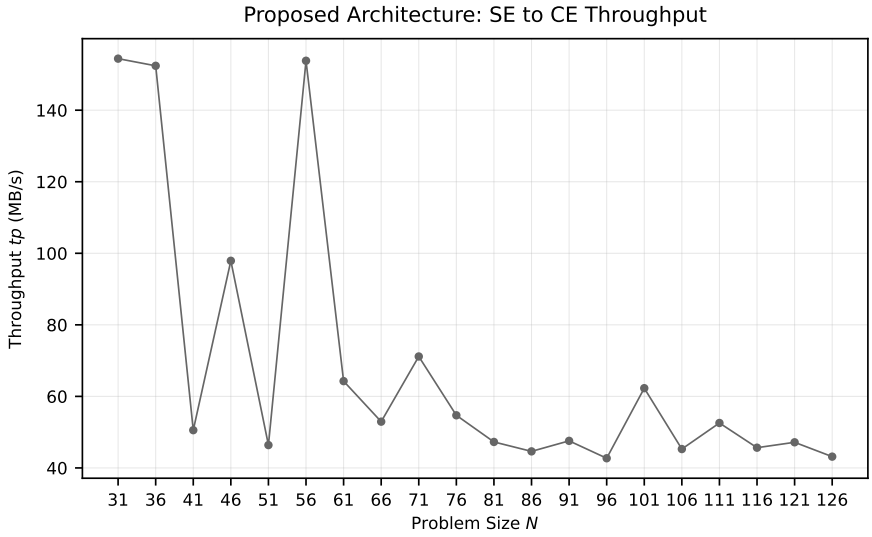


Figure 7.55: Proposed Architecture - SE to CE Throughput across Problem Size N.

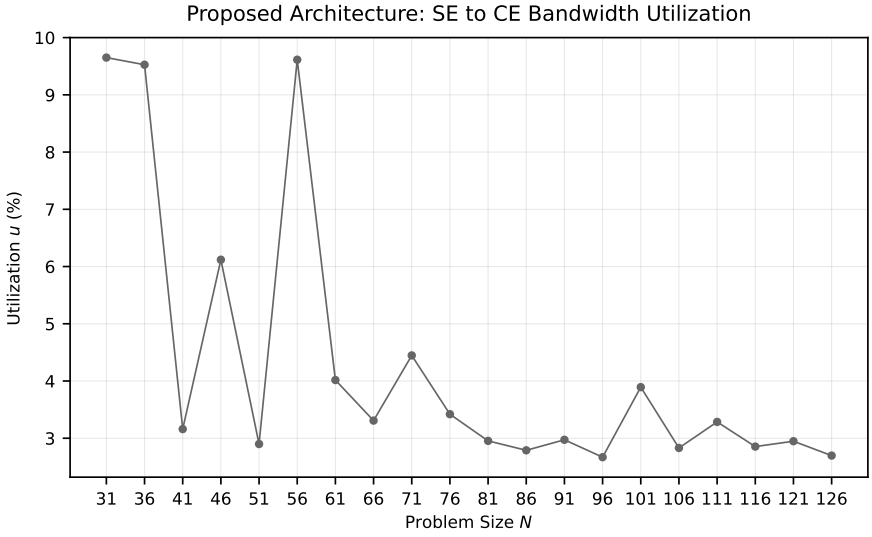


Figure 7.56: Proposed Architecture - SE to CE Bandwidth Utilization across Problem Size N.

Regarding the CE, bandwidth and compute utilization are evaluated relative to a maximum theoretical CE throughput limit of 800 MB/s for both architectural variants.

For the baseline architecture, CE throughput ($\mathrm{tp}_{\mathrm{ce,bsl}}$), bandwidth utilization ($u_{\mathrm{bw,ce,bsl}}$), and compute utilization ($u_{\mathrm{ce,cmp,bsl}}$) decay smoothly with increasing problem size, as illustrated in Figures 7.57, 7.58, and 7.59:

$$\mathrm{tp}_{\mathrm{ce,bsl}}(N) = 1.43 \cdot 10^{-4} \cdot N^2 - 0.033 \cdot N + 2.255 \quad [\mathrm{MB/s}] \tag{7.53}$$

$$u_{\mathrm{bw,ce,bsl}}(N) = 1.78 \cdot 10^{-5} \cdot N^2 - 4.05 \cdot 10^{-3} \cdot N + 0.281 \quad [\%] \tag{7.54}$$

$$u_{\mathrm{ce,cmp,bsl}}(N) = 7.36 \cdot 10^{-6} \cdot N^2 - 1.68 \cdot 10^{-3} \cdot N + 5.999 \quad [\%] \tag{7.55}$$

Because total simulation time is dominated by quadratic scaling ($O(N^2)$), constant and linear loop setup overheads become increasingly negligible as N expands. Consequently, the ratio of active compute operations to overall execution latency stabilizes, causing baseline CE compute utilization to undergo a minor asymptotic decrease from 5.96% at $N = 31$ down to 5.90% at $N = 126$.

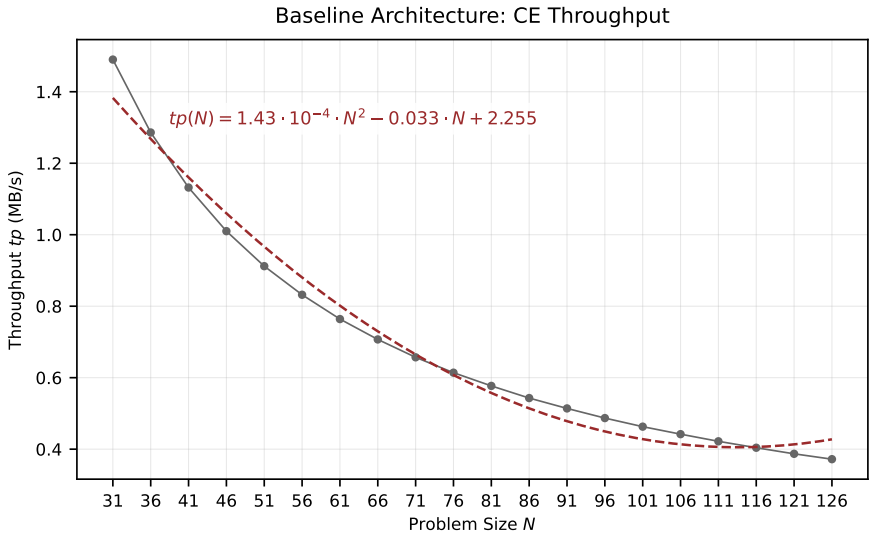


Figure 7.57: Baseline Architecture - CE Throughput across Problem Size N.

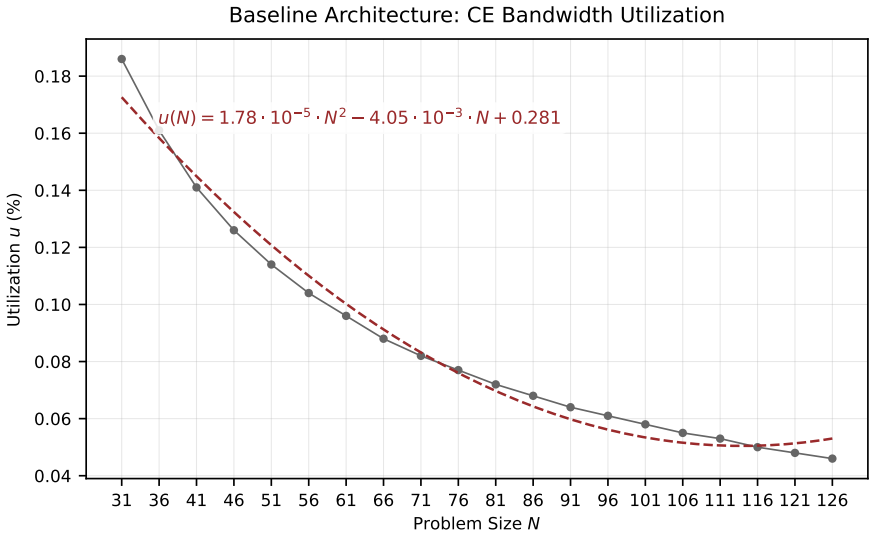


Figure 7.58: Baseline Architecture - CE Bandwidth Utilization across Problem Size N.

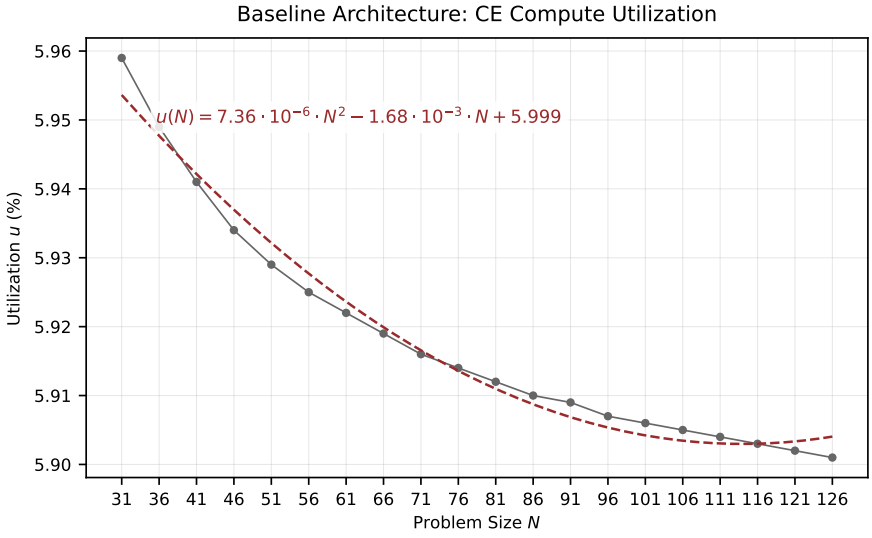


Figure 7.59: Baseline Architecture - CE Compute Utilization across Problem Size N.

For the proposed architecture, the CE metrics track the oscillating trend observed at the SE-to-CE interface, as depicted in Figures 7.60, 7.61, and 7.62. While small matrix sizes achieve higher initial CE throughput ($\text{tp}_{\text{ce,prp}} \approx 2.5$ MB/s, $u_{\text{bw,ce,prp}} \approx 0.31\%$) and compute utilization ($u_{\text{ce,cmp,prp}} \approx 10\%$), performance drops substantially as N expands. For larger matrices, continuous SC pollution keeps CE compute utilization starved below 3%.

This sharp contrast highlights the double-edged nature of line-fill caching in strided workloads: when local SC line fills repeatedly fail to deliver spatial locality and SB circular buffering is inactive, the lengthened bus transaction latency starves the CE execution units, yielding lower compute utilization than the simple unbuffered baseline design.

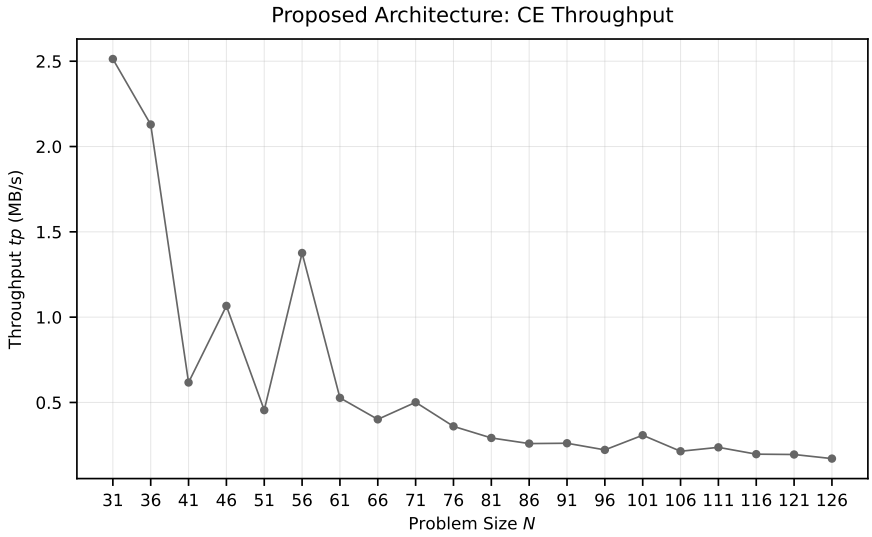


Figure 7.60: Proposed Architecture - CE Throughput across Problem Size N.

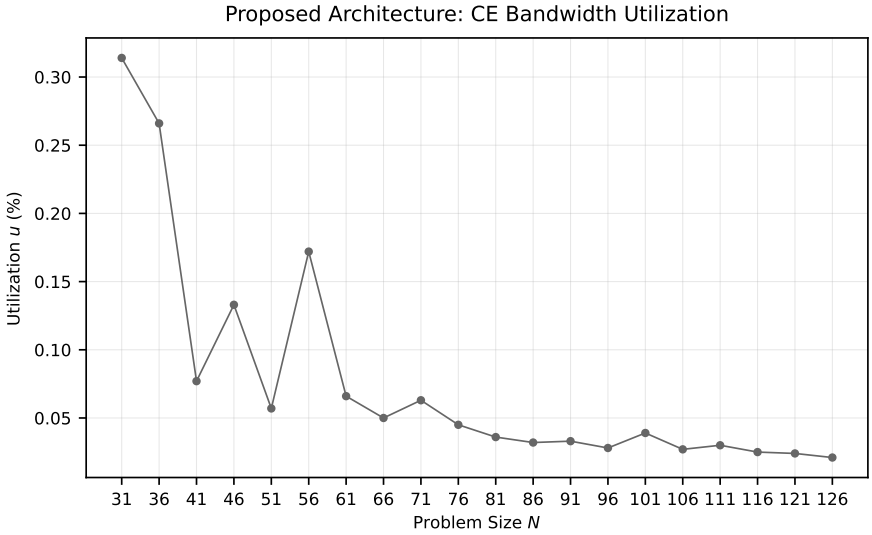


Figure 7.61: Proposed Architecture - CE Bandwidth Utilization across Problem Size N.

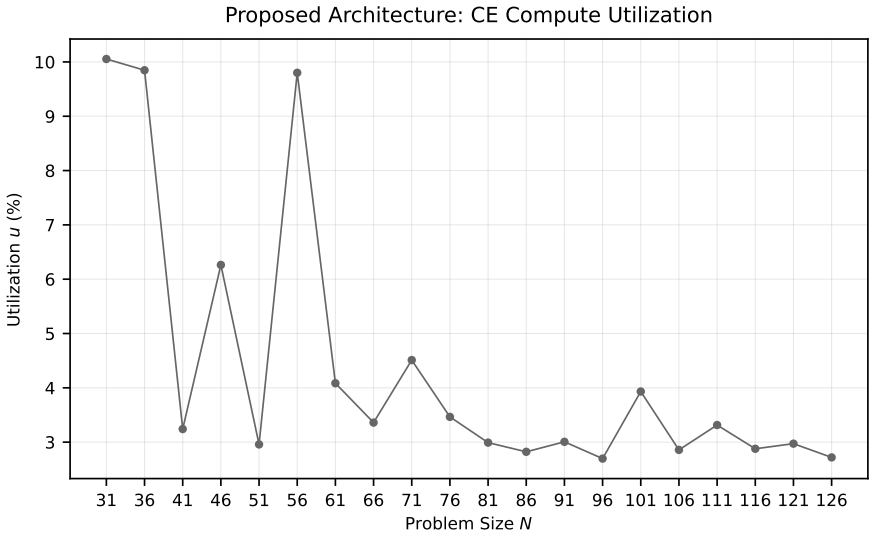


Figure 7.62: Proposed Architecture - CE Compute Utilization across Problem Size N.

7.2.2 Operating Frequency Sweep

To evaluate the scalability and performance sensitivity of both architectures to clock frequency variations during two-dimensional matrix multiplication, an operating frequency sweep is conducted across $f \in [100, 250]$ MHz. For all frequency sweep evaluations in this benchmark, the problem size is fixed at $N = 96$ because, as established in the problem size sweep analysis, $N = 96$ represents an operating regime where throughput performance has stabilized, thereby isolating the system's frequency response from dataset-scaling transients.

7.2.2.1 Simulation Time and Speedup Analysis

The relationship between operating frequency and execution time highlights fundamental differences in pipeline clocking efficiency and memory bottleneck management between the two system designs under multidimensional data access profiles.

In the baseline architecture, execution time decreases non-linearly with increasing clock frequency. As illustrated in Figure 7.63, the simulation time $t_{\text{sim,bsl}}$ follows an inverse relationship with operating frequency f:

$$t_{\text{sim,bsl}}(f) = \frac{1.28 \cdot 10^7}{f} + 2.03 \cdot 10^4 \quad [\mu\text{s}] \tag{7.56}$$

As frequency increases from 100 MHz to 250 MHz, execution time drops from 151240.000μs down to 74860.000μs. However, due to recurring single memory accesses that remain unscaled by clock frequency, the baseline architecture suffers from a large static offset of $2.03 \cdot 10^4 \mu$s, leading to diminishing returns at higher frequencies.

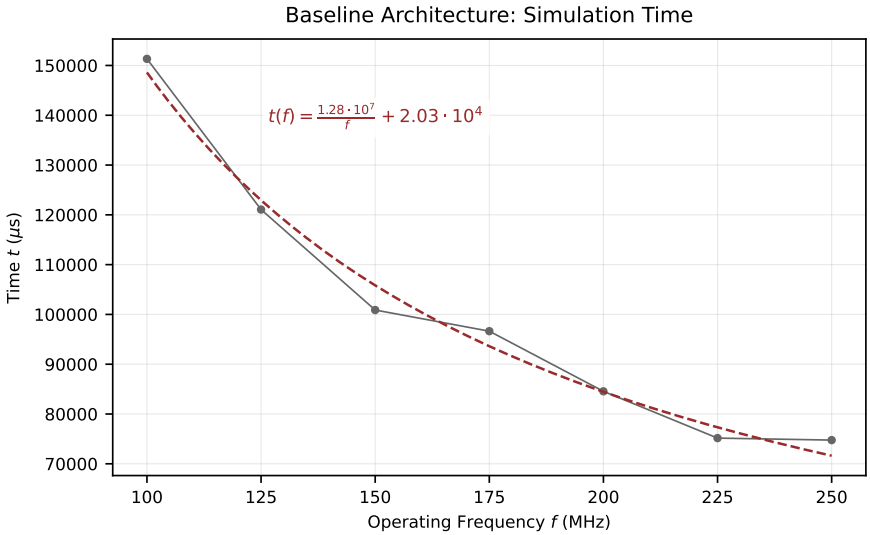


Figure 7.63: Baseline Architecture - Simulation Time across Operating Frequency f.

The proposed architecture also exhibits an inverse relationship with frequency. As shown in Figure 7.64, the simulation time $t_{\text{sim,prp}}$ scales according to:

$$t_{\text{sim,prp}}(f) = \frac{3.20 \cdot 10^7}{f} + 1.01 \cdot 10^4 \quad [\mu\text{s}] \tag{7.57}$$

As f increases from 100 MHz to 250 MHz, execution time decreases from 330882.000μs to 139870.000μs. While the frequency-dependent term remains high, the constant execution latency floor is reduced to $1.01 \cdot 10^4 \mu$s, representing a $\sim 50.2\%$ reduction in unscalable overhead compared to the baseline design.

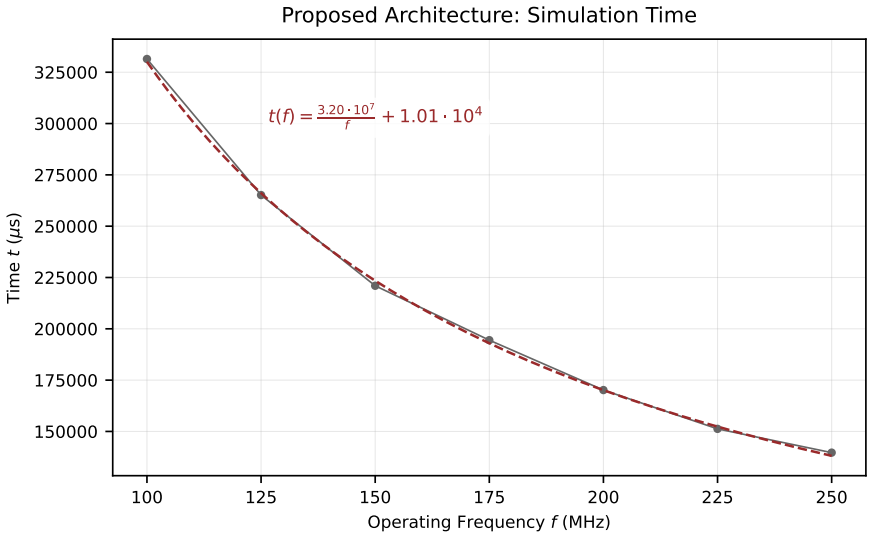


Figure 7.64: Proposed Architecture - Simulation Time across Operating Frequency f.

The resulting speedup $s(f)$ achieved by the proposed architecture relative to the baseline architecture across operating frequencies is depicted in Figure 7.65. Overall, the relative speedup improves from 0.46× at 100 MHz up to 0.53× at 250 MHz. The step-like jumps in the speedup curve occur because external memory latency is fixed in physical time, forcing it to scale in discrete clock cycles. As operating frequency increases, discrete latency cycle thresholds are crossed at 175 MHz and 250 MHz. Because these cycle-rounding penalties degrade the baseline architecture's efficiency more severely, each threshold crossing produces a step upward in relative speedup.

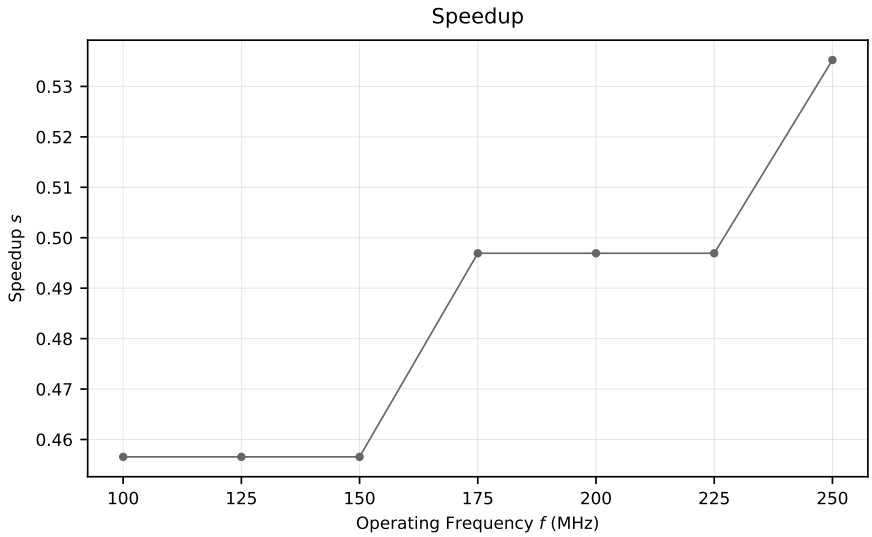


Figure 7.65: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Operating Frequency f.

7.2.2.2 Memory Access Analysis

To isolate the memory interface efficiency from computation across operating frequencies during matrix multiplication, we evaluate both the MAF and total memory communication time (t_{mem}) at a fixed problem size $N = 96$.

In the baseline architecture, the memory access factor remains strictly invariant across all operating frequencies at $\text{MAF}_{\text{bsl}} = 193.000$. Because the baseline system performs unbuffered single memory transfers for every element request, frequency scaling does not alter the structural number of memory accesses required per matrix computation loop iteration.

However, as shown in Figure 7.66, the baseline memory communication time ($t_{\text{mem,bsl}}$) exhibits non-monotonic behavior across frequency increments. While higher frequencies generally reduce execution time, discrete upward jumps occur between 150 MHz → 175 MHz and 225 MHz → 250 MHz. This behavior stems directly from the fixed physical response delay of the memory system. Because memory latencies round up to discrete integer clock cycles, operating at higher frequencies reduces the clock period T_{clk}, crossing threshold points where the MC

must wait an additional clock cycle per transaction. Due to the high number of individual transactions ($\text{MAF}_{\text{bsl}} = 193.000$), these accumulated single-cycle penalty steps produce visible spikes in total communication time.

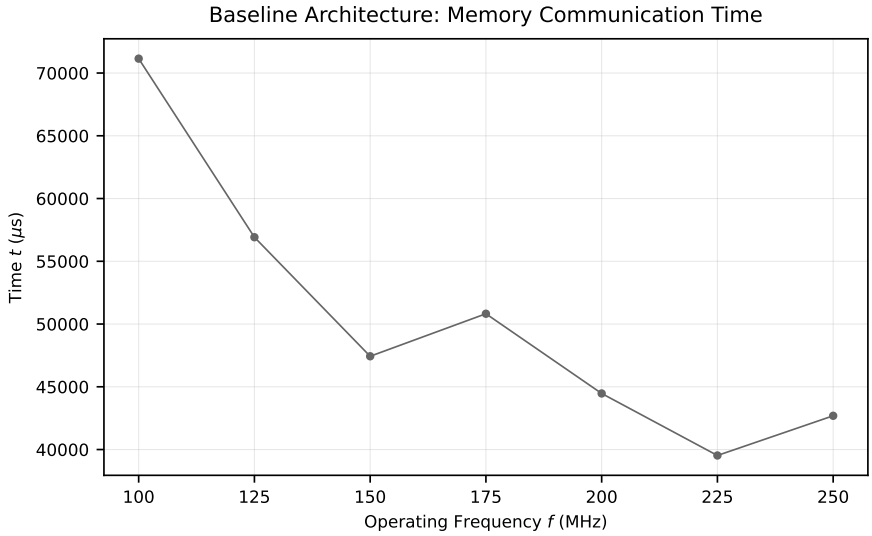


Figure 7.66: Baseline Architecture - Memory Communication Time across Operating Frequency f.

For the proposed architecture, the memory access factor also remains invariant across frequency variations at $\text{MAF}_{\text{prp}} = 770.000$. As discussed in Section 7.1.1.2, this high MAF value reflects the severe SC line-fill thrashing and spatial pollution caused by strided column-wise accesses over Matrix B.

As illustrated in Figure 7.67, the memory communication time of the proposed architecture ($t_{\text{mem,prp}}$) scales inversely with operating frequency according to:

$$t_{\text{mem,prp}}(f) = \frac{1.48 \cdot 10^7}{f} + 1.01 \cdot 10^4 \quad [\mu\text{s}] \tag{7.58}$$

Unlike the baseline design, memory transactions in the proposed architecture are executed via burst transfers. Consequently, discrete integer clock cycle quantization effects are effectively absorbed across the long burst transactions, allowing memory communication time to decrease smoothly from 159870.000μs down to 70860.000μs as f increases from 100 MHz to 250 MHz.

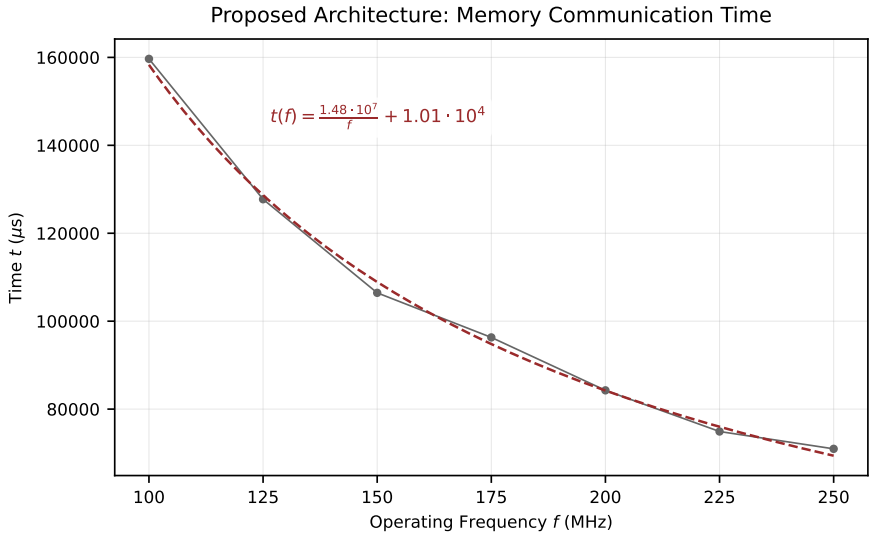


Figure 7.67: Proposed Architecture - Memory Communication Time across Operating Frequency f.

7.2.2.3 Throughput and Bandwidth Utilization Analysis

To evaluate memory bus efficiency and processing engine performance across operating frequencies, we analyze the throughput (tp) and utilization (u) metrics for both architectures at a fixed problem size N.

For the baseline architecture, the maximum theoretical MC-to-CE throughput ($\text{tp}_{\text{max,mc-ce}}$) scales linearly as a function of the operating frequency, as illustrated in Figure 7.68:

$$\text{tp}_{\text{max,mc-ce}}(f) \approx 8.000 \cdot f \tag{7.59}$$

reaching 2000 MB/s at 250 MHz. However, actual MC-to-CE throughput (shown in Figure 7.69) follows:

$$\text{tp}_{\text{mc-ce}}(f) = 0.635 \cdot f + 33.435 \tag{7.60}$$

affected by discrete clock-cycle quantization stalls between 150 MHz $\rightarrow$ 175 MHz and 225 MHz $\rightarrow$ 250 MHz. In regions where latency cycle requirements remain constant, both tp_{max} and measured throughput increase proportionally with frequency. Conversely, crossing latency threshold boundaries introduces additional stall cycles, causing measured throughput to deviate downward relative to capacity. Consequently, as depicted in Figure 7.70, the MC-to-CE bandwidth utilization ($u_{\text{bw,mc-ce}}$) exhibits step-down drops from 11.7% to 9.5% across the frequency range.

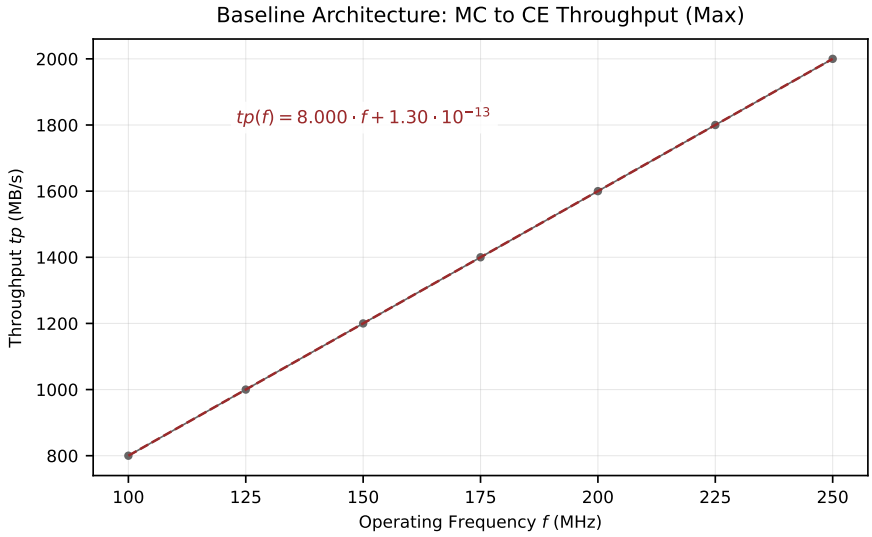


Figure 7.68: Baseline Architecture - MC to CE Maximum Theoretical Throughput across Operating Frequency f.

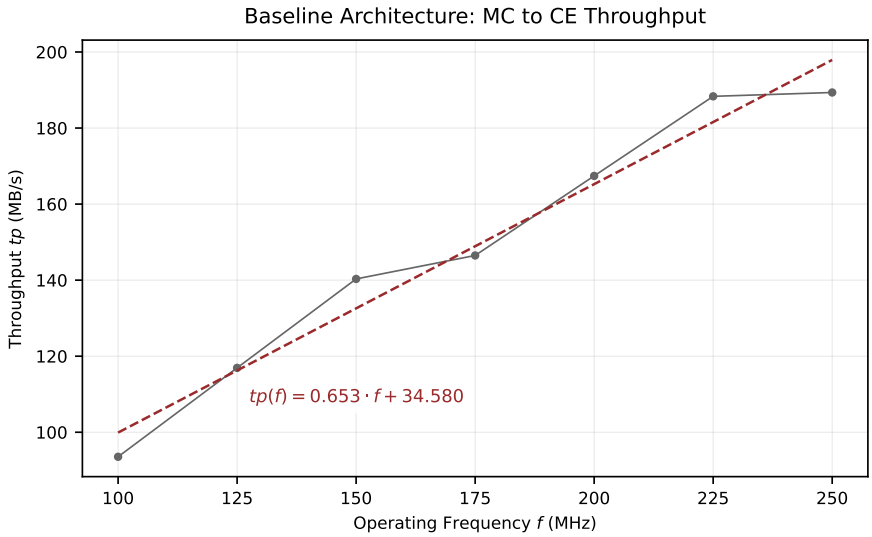


Figure 7.69: Baseline Architecture - MC to CE Throughput across Operating Frequency f.

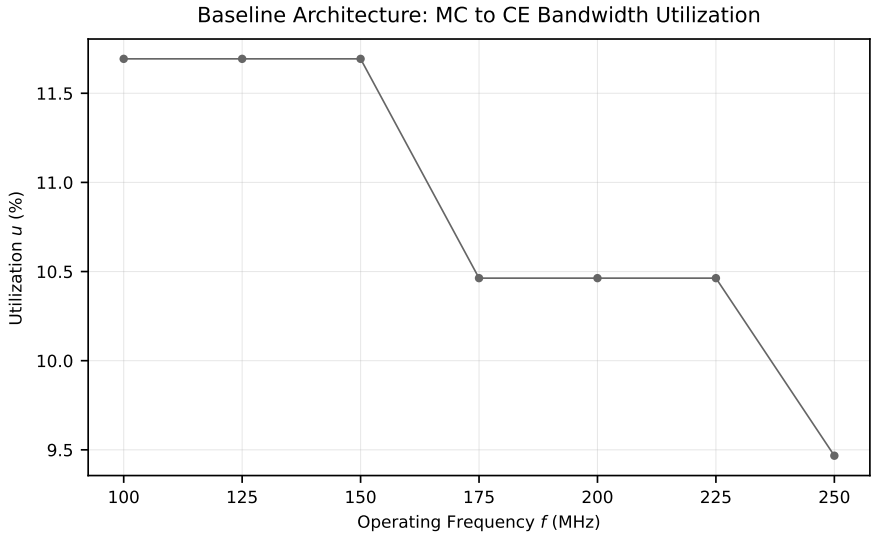


Figure 7.70: Baseline Architecture - MC to CE Bandwidth Utilization across Operating Frequency f.

For the proposed architecture, the SE-to-CE interface features a wider bus, yielding a theoretical maximum throughput (Figure 7.71) of:

$$\mathrm{tp}_{\mathrm{max,se\text{-}ce}}(f) \approx 16.000 \cdot f \tag{7.61}$$

and an effective throughput (Figure 7.72) of:

$$\mathrm{tp}_{\mathrm{se\text{-}ce}}(f) = 0.394 \cdot f + 4.148 \tag{7.62}$$

As plotted in Figure 7.73, bandwidth utilization $u_{\mathrm{bw,se\text{-}ce}}$ ranges between 2.67% and 2.53%, experiencing step-down drops across frequency transitions due to cycle quantization latency overhead.

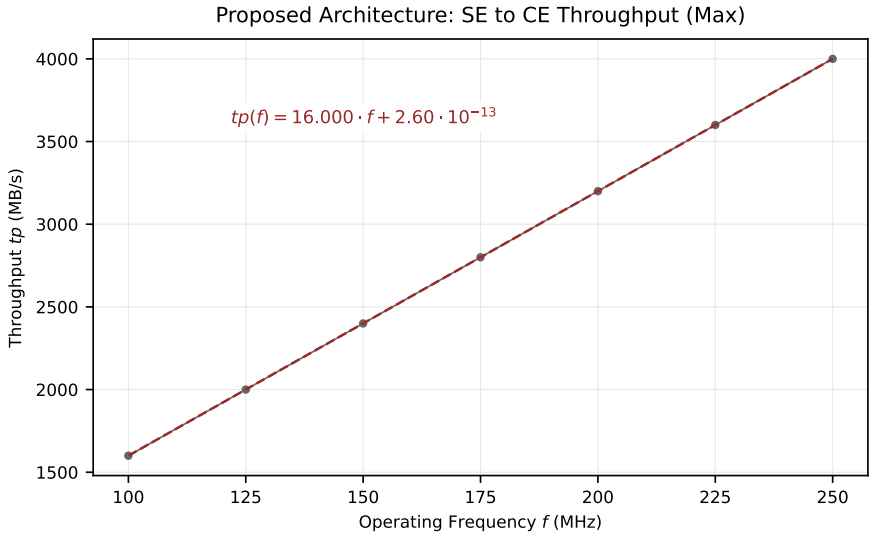


Figure 7.71: Proposed Architecture - SE to CE Maximum Theoretical Throughput across Operating Frequency f.

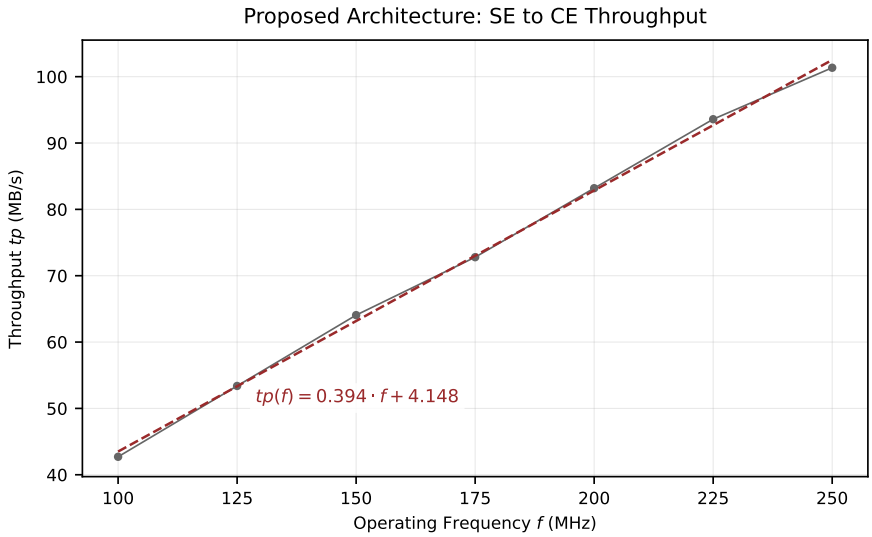


Figure 7.72: Proposed Architecture - SE to CE Throughput across Operating Frequency f.

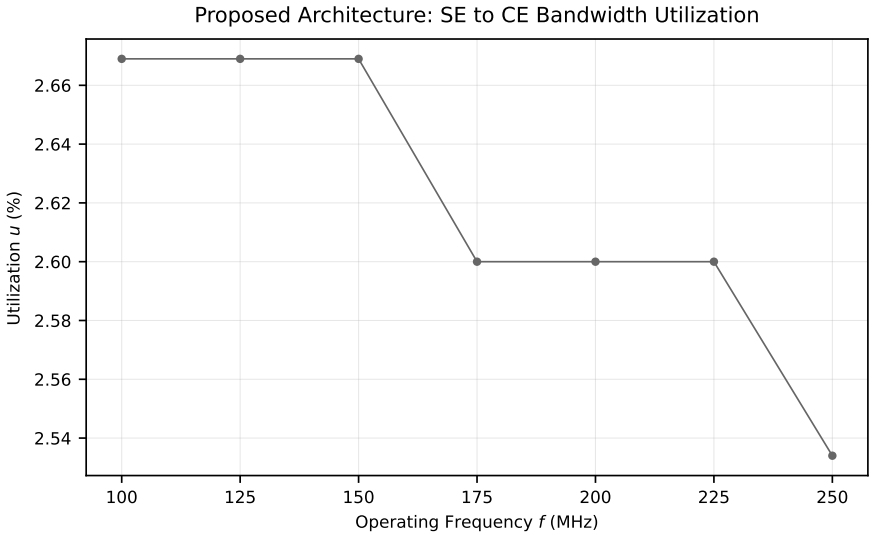


Figure 7.73: Proposed Architecture - SE to CE Bandwidth Utilization across Operating Frequency f.

The maximum CE theoretical throughput ($\text{tp}_{\text{max,ce}}$) scales identically in both architectures, as shown in Figure 7.74 and Figure 7.78:

$$\text{tp}_{\text{max,ce}}(f) \approx 8.000 \cdot f \tag{7.63}$$

However, the baseline CE throughput ($\text{tp}_{\text{ce,bsl}}$) shown in Figure 7.75 achieves only:

$$\text{tp}_{\text{ce,bsl}}(f) = 0.020 \cdot f + 1.045 \tag{7.64}$$

due to continuous memory stalls. This directly impacts baseline performance, resulting in low bandwidth utilization $u_{\text{bw,ce,bsl}}$ (Figure 7.76) and compute utilization $u_{\text{ce,cmp,bls}}$ between 4.8% and 5.9% (Figure 7.77).

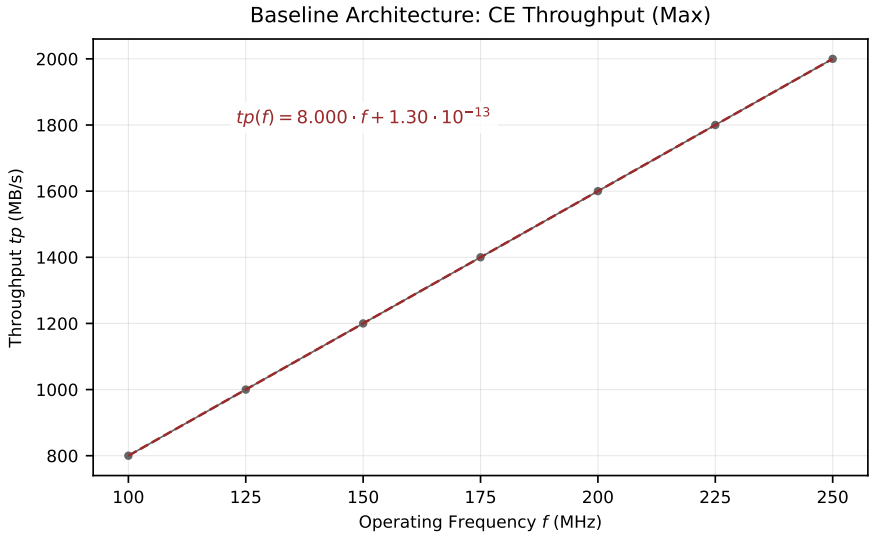


Figure 7.74: Baseline Architecture - CE Maximum Theoretical Throughput across Operating Frequency f.

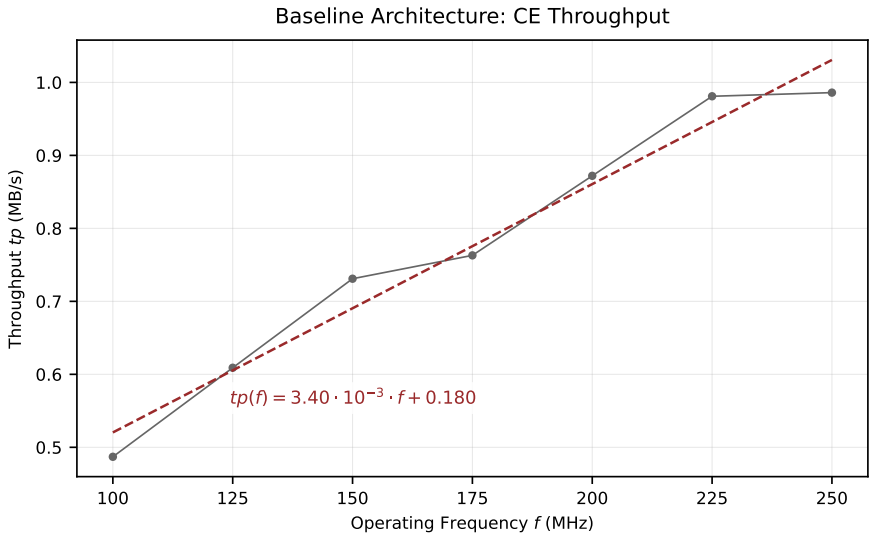


Figure 7.75: Baseline Architecture - CE Throughput across Operating Frequency f.

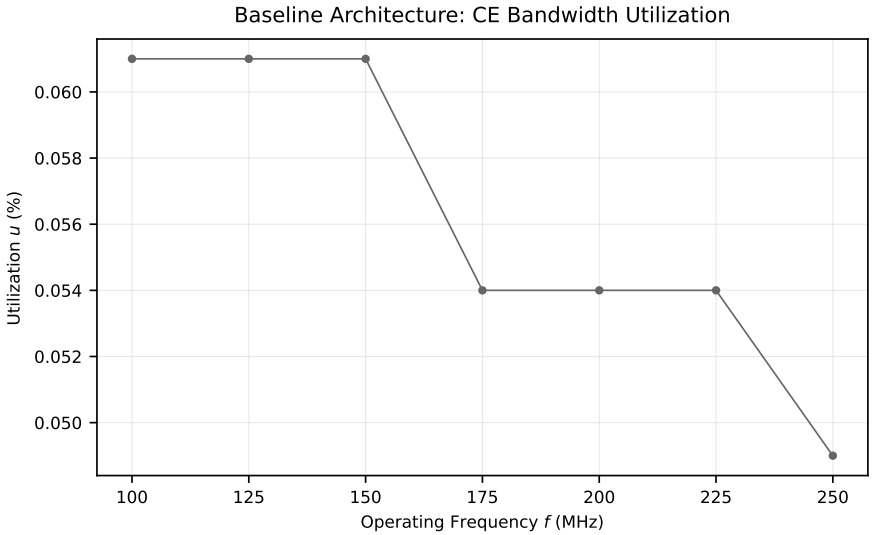


Figure 7.76: Baseline Architecture - CE Bandwidth Utilization across Operating Frequency f.

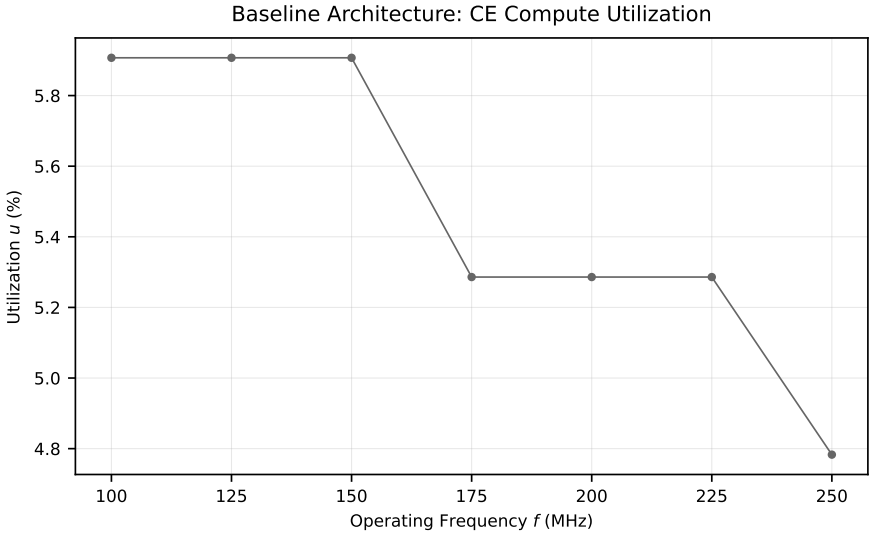


Figure 7.77: Baseline Architecture - CE Compute Utilization across Operating Frequency f.

In contrast, the proposed design (Figure 7.79) achieves:

$$\mathrm{tp}_{\mathrm{ce,prp}}(f) = 0.00205 \cdot f + 0.021 \tag{7.65}$$

with a CE bandwidth utilization $u_{\mathrm{bw,ce,prp}}$ scaling around 0.027% (Figure 7.80) and compute utilization $u_{\mathrm{ce,cmp,prp}}$ between 2.70% and 2.56% (Figure 7.81). Across all utilization metrics (u_{bw} and $u_{\mathrm{ce,cmp}}$), plateau regions correspond to frequency bands with constant memory latency cycles, whereas discrete downward steps mark threshold boundaries (150 MHz $\rightarrow$ 175 MHz and 225 MHz $\rightarrow$ 250 MHz) where cycle quantization increases memory handshake overhead.

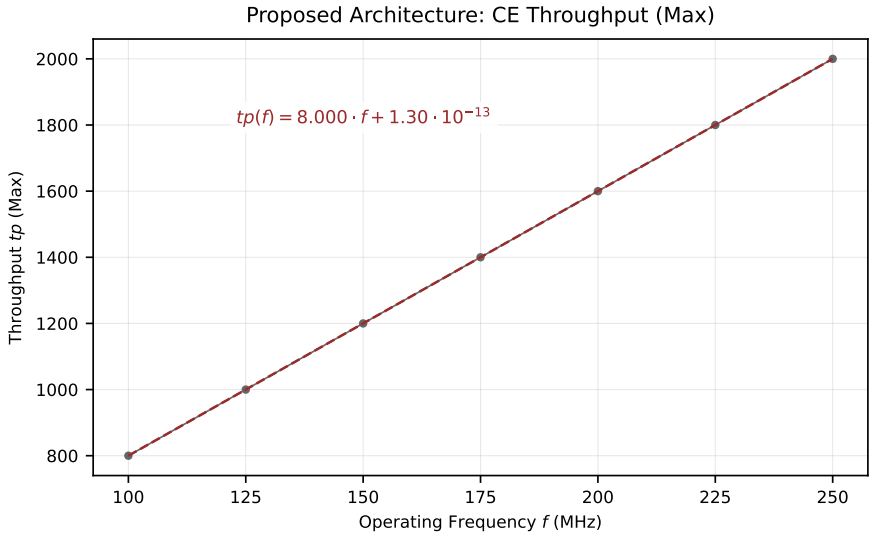


Figure 7.78: Proposed Architecture - CE Maximum Theoretical Throughput across Operating Frequency f.

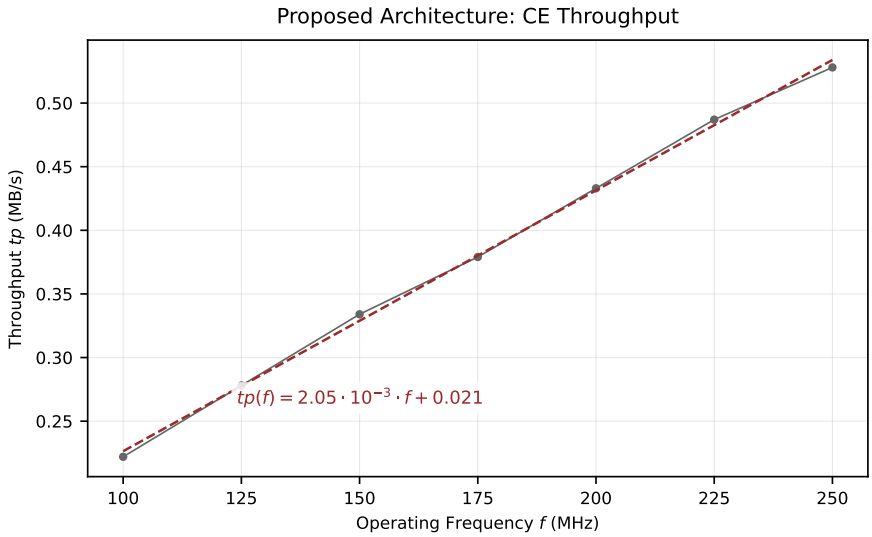


Figure 7.79: Proposed Architecture - CE Throughput across Operating Frequency f.

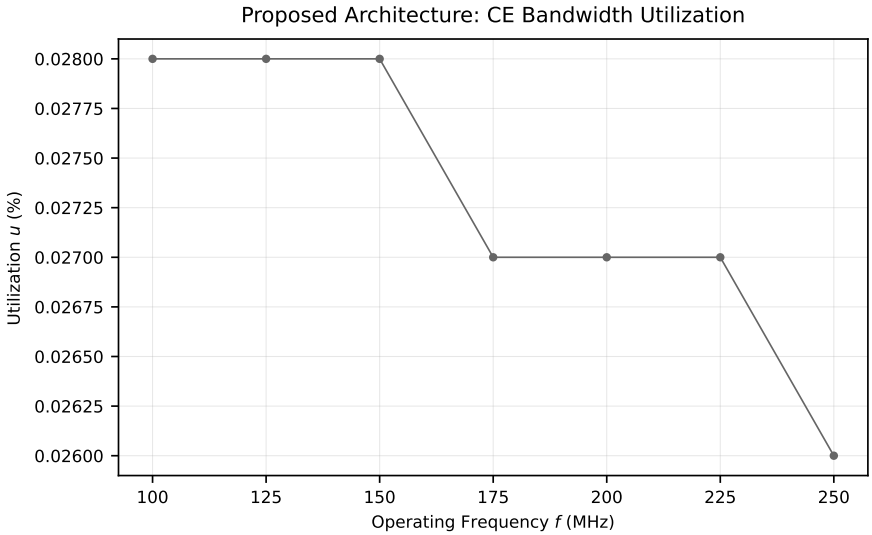


Figure 7.80: Proposed Architecture - CE Bandwidth Utilization across Operating Frequency f.

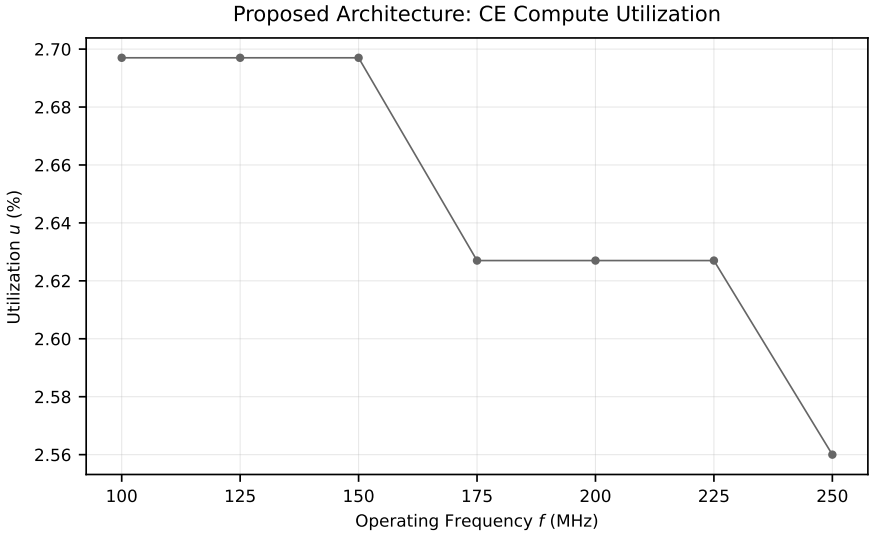


Figure 7.81: Proposed Architecture - CE Compute Utilization across Operating Frequency f.

7.2.3 Memory Latency Sweep

To evaluate memory interface efficiency and system robustness under varying external memory response delays, a memory latency sweep is performed across $l \in [15.0, 35.0]$ ns. As established in previous sections, the problem size is fixed at $N = 96$.

7.2.3.1 Simulation Time and Speedup Analysis

For the baseline architecture, as depicted in Figure 7.82, execution time exhibits a staircase profile as a function of external memory latency l. Because access delays are quantized into discrete integer clock cycles ($L = [\![l/T_{\text{clk}}]\!]$), the total simulation time $t_{\text{sim,bsl}}$ remains flat across sub-cycle latency intervals before stepping upward at cycle quantization boundaries:

$$t_{\text{sim,bsl}}(l) = 151300\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.66}$$

$$t_{\text{sim,bsl}}(l) = 169100\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.67}$$

$$t_{\text{sim,bsl}}(l) = 186900\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.68}$$

Due to unbuffered, direct memory accesses during matrix multiplication operations, each additional cycle of latency (+10 ns) introduces significant cumulative stall overhead, increasing baseline execution time by 17800μs per latency cycle step.

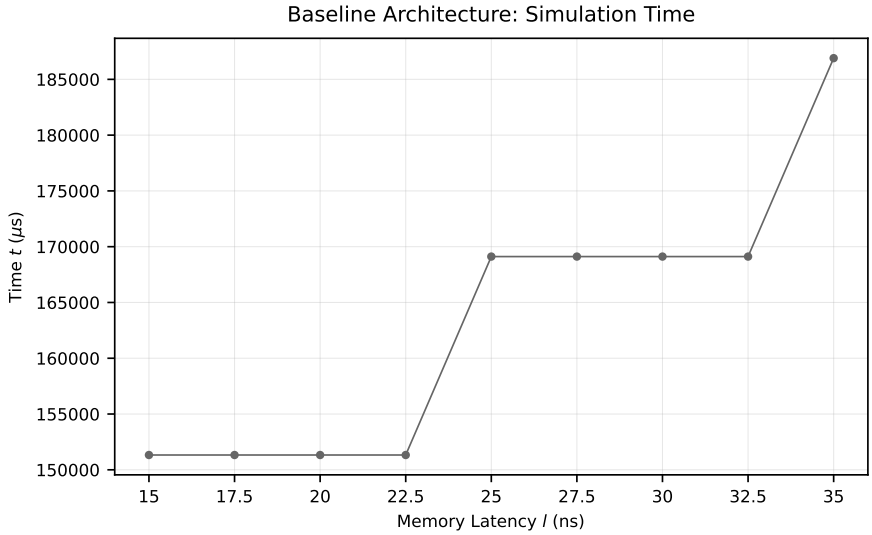


Figure 7.82: Baseline Architecture - Simulation Time across Memory Latency l.

For the proposed architecture, shown in Figure 7.83, the simulation time $t_{\text{sim,prp}}$ follows a similar step-wise scaling behavior across cycle quantization boundaries:

$$t_{\text{sim,prp}}(l) = 331460\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.69}$$

$$t_{\text{sim,prp}}(l) = 340330\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.70}$$

$$t_{\text{sim,prp}}(l) = 349200\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.71}$$

Each cycle threshold transition increases total execution time by 8870μs.

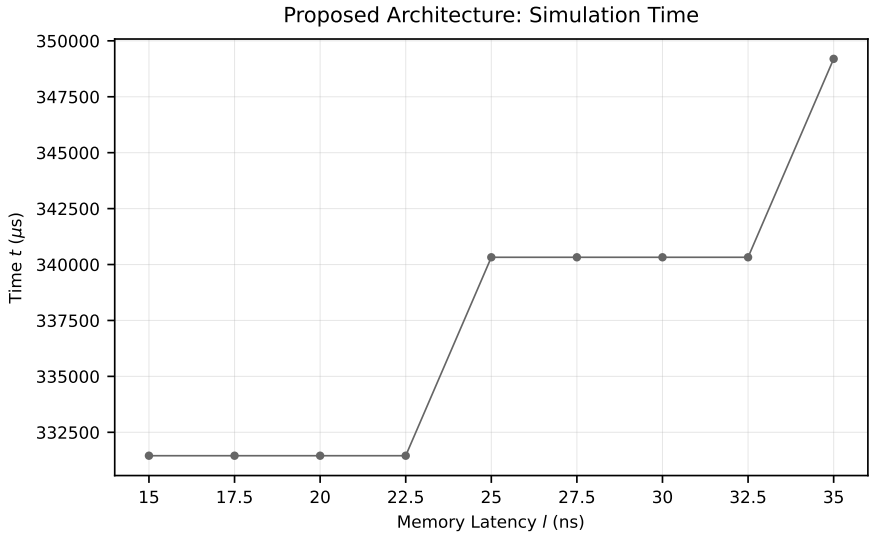


Figure 7.83: Proposed Architecture - Simulation Time across Memory Latency l.

As shown in Figure 7.84, the relative speedup achieved by the proposed architecture exhibits a step-wise profile across memory latency values:

$$s(l) \approx 0.455\times \quad \text{for } l \in [15.0, 22.5] \text{ ns} \tag{7.72}$$

$$s(l) \approx 0.496\times \quad \text{for } l \in [25.0, 32.5] \text{ ns} \tag{7.73}$$

$$s(l) \approx 0.536\times \quad \text{for } l = 35.0 \text{ ns} \tag{7.74}$$

Because the baseline architecture suffers higher percentage latency penalties under increased external memory delays relative to its base execution time, the relative speedup $s(l)$ scales higher as memory latency l increases.

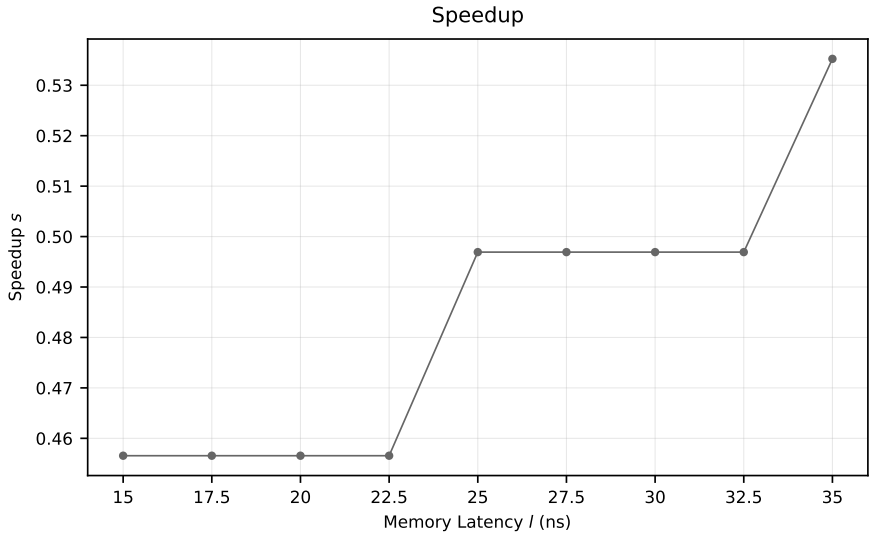


Figure 7.84: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Memory Latency l.

7.2.3.2 Memory Access Analysis

Because memory latency l affects transaction duration rather than the total volume of memory accesses, the MAF remains constant across all values of l for both architectures at a fixed problem size $N = 96$.

For the baseline architecture, the memory access factor stays invariant at:

$$\text{MAF}_{\text{bsl}} = 193.000 \tag{7.75}$$

reflecting unbuffered scalar memory access penalties during matrix multiplication operations. Consequently, memory communication time (t_{mem}) increases directly with memory latency cycle thresholds:

$$t_{\text{mem,bsl}}(l) = 71190.0\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.76}$$

$$t_{\text{mem,bsl}}(l) = 88980.0\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.77}$$

$$t_{\text{mem,bsl}}(l) = 106770.0\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.78}$$

Each latency cycle step introduces 17790.0μs of additional communication overhead, severely impacting baseline memory execution efficiency.

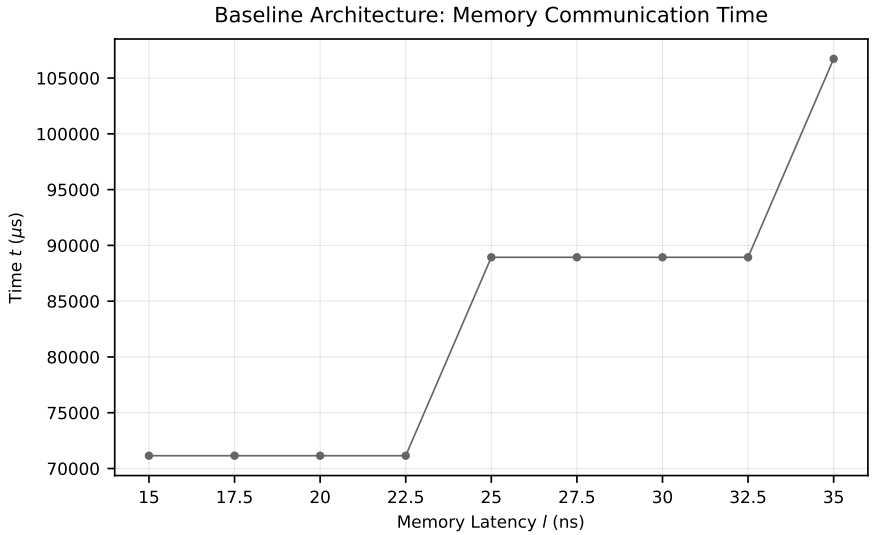


Figure 7.85: Baseline Architecture - Memory Communication Time across Memory Latency l.

For the proposed architecture, the memory access factor is similarly invariant:

$$\mathrm{MAF}_{\mathrm{prp}} = 770.000 \tag{7.79}$$

The memory communication time (t_{mem}) exhibits step-wise latency penalties across cycle quantization boundaries:

$$t_{\mathrm{mem,prp}}(l) = 159760.0\mu\mathrm{s} \quad \text{for } l \in [15.0, 22.5]\ \mathrm{ns} \quad (L = 2) \tag{7.80}$$

$$t_{\mathrm{mem,prp}}(l) = 168630.0\mu\mathrm{s} \quad \text{for } l \in [25.0, 32.5]\ \mathrm{ns} \quad (L = 3) \tag{7.81}$$

$$t_{\mathrm{mem,prp}}(l) = 177500.0\mu\mathrm{s} \quad \text{for } l = 35.0\ \mathrm{ns} \quad (L = 4) \tag{7.82}$$

Each latency cycle step increases memory communication time by 8870.0μs.

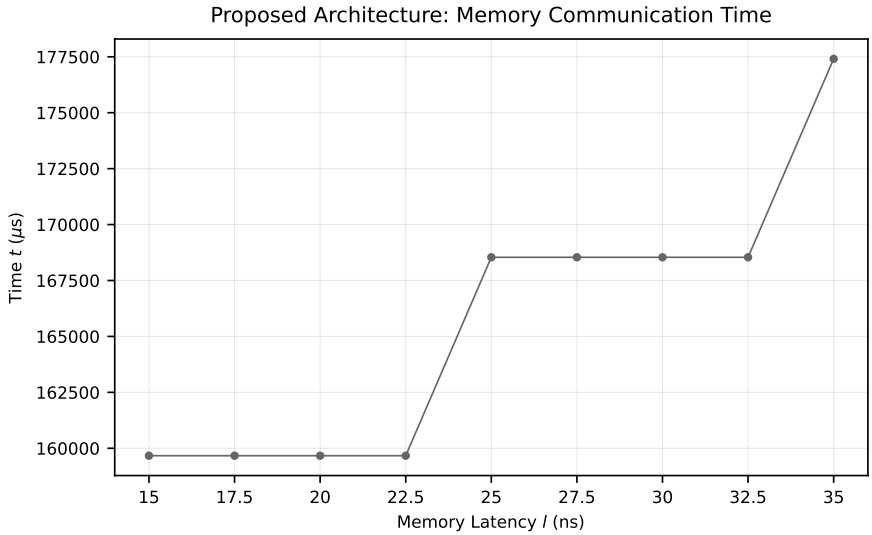


Figure 7.86: Proposed Architecture - Memory Communication Time across Memory Latency l.

7.2.3.3 Throughput and Bandwidth Utilization Analysis

For the baseline architecture, the maximum theoretical throughput of the MC-to-CE interface is $\text{tp}_{\text{max,mc-ce}} = 800.0$ MB/s. The effective MC-to-CE throughput ($\text{tp}_{\text{mc-ce}}$) follows a step-down pattern governed by discrete clock cycle quantization ($L = [\![l/T_{\text{clk}}]\!]$), as depicted in Figure 7.87:

$$\text{tp}_{\text{mc-ce}}(l) = 93.5 \text{ MB/s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.83}$$

$$\text{tp}_{\text{mc-ce}}(l) = 83.7 \text{ MB/s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.84}$$

$$\text{tp}_{\text{mc-ce}}(l) = 75.7 \text{ MB/s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.85}$$

As shown in Figure 7.88, this decline causes bus bandwidth utilization ($u_{\text{bw,mc-ce}}$) to drop from 11.69% to 10.46% and down to 9.46% across cycle thresholds.

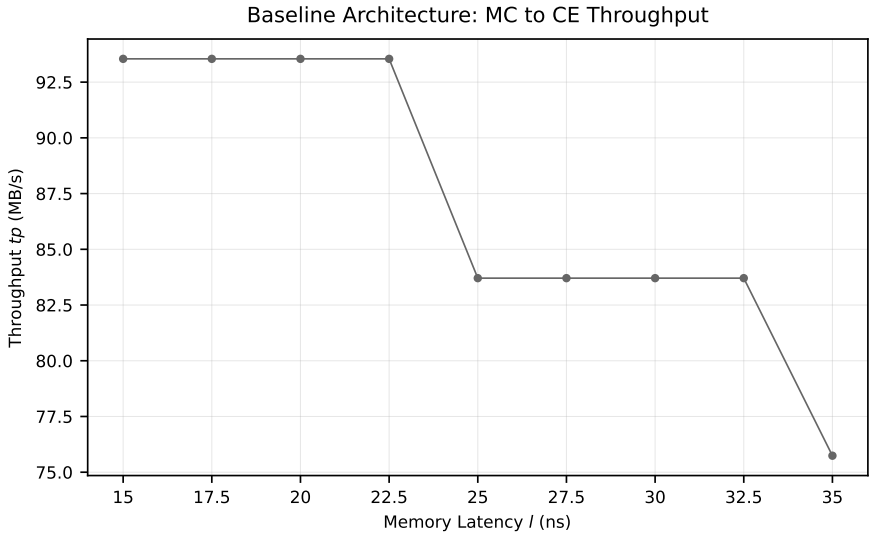


Figure 7.87: Baseline Architecture - MC to CE Throughput across Memory Latency l.

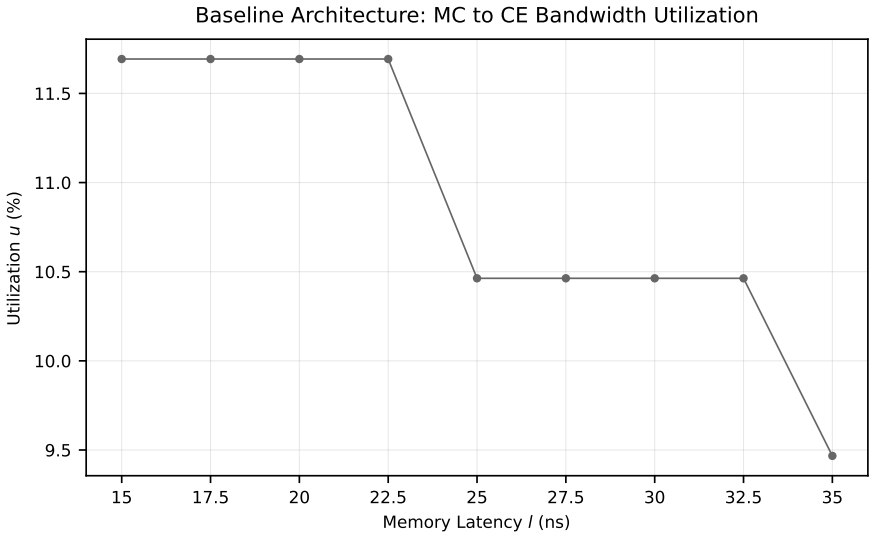


Figure 7.88: Baseline Architecture - MC to CE Bandwidth Utilization across Memory Latency l.

In contrast, the proposed architecture leverages pre-buffering via the SE to shield the core pipeline from memory latency penalties. With a maximum theoretical throughput of $\text{tp}_{\text{max,se-ce}} = 1600.0$ MB/s, the effective throughput ($\text{tp}_{\text{se-ce}}$) shown in Figure 7.89 remains highly stable across latency boundaries:

$$\text{tp}_{\text{se-ce}}(l) = 42.7 \text{ MB/s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.86}$$

$$\text{tp}_{\text{se-ce}}(l) = 41.6 \text{ MB/s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.87}$$

$$\text{tp}_{\text{se-ce}}(l) = 40.5 \text{ MB/s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.88}$$

As illustrated in Figure 7.90, this yields a consistent bandwidth utilization ($u_{\text{bw,se-ce}}$) between 2.67% and 2.53%.

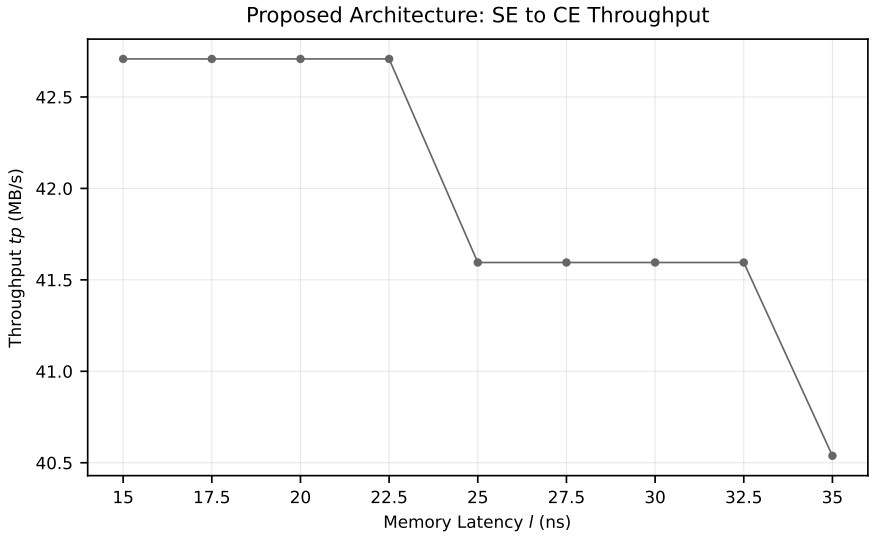


Figure 7.89: Proposed Architecture - SE to CE Throughput across Memory Latency l.

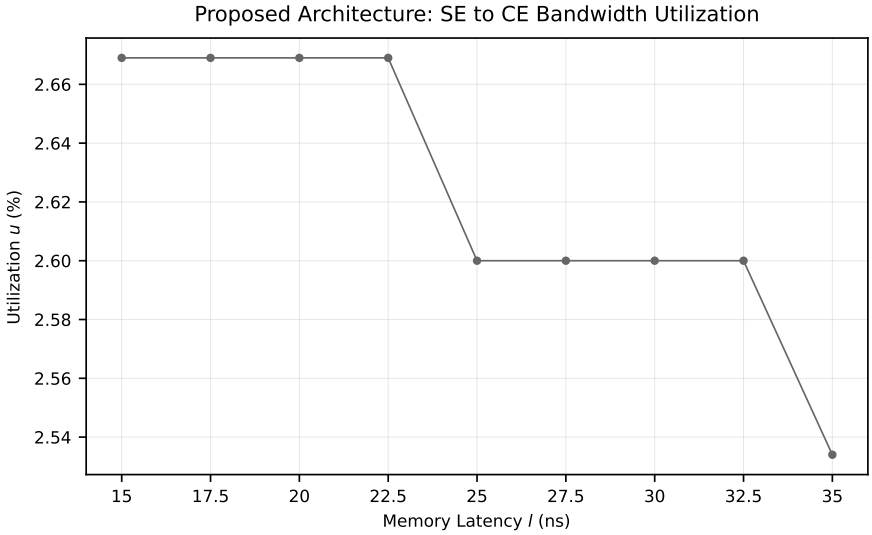


Figure 7.90: Proposed Architecture - SE to CE Bandwidth Utilization across Memory Latency l.

At the processing engine interface, where maximum theoretical compute throughput is bounded by $\text{tp}_{\text{max,ce}} = 800.0$ MB/s, severe single memory accesses degrade the baseline CE throughput ($\text{tp}_{\text{ce,bsl}}$) from 0.487 MB/s down to 0.394 MB/s (Figure 7.91). This causes baseline CE bandwidth utilization ($u_{\text{bw,ce,bsl}}$) to plummet to 0.049% (Figure 7.92) and compute utilization ($u_{\text{ce,cmp,bsl}}$) to drop from 5.91% to 4.78% (Figure 7.93).

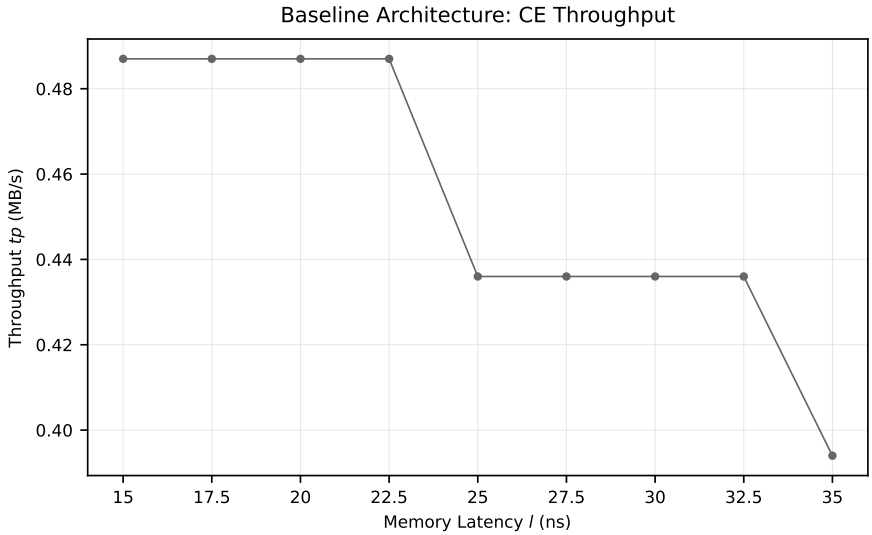


Figure 7.91: Baseline Architecture - CE Throughput across Memory Latency l.

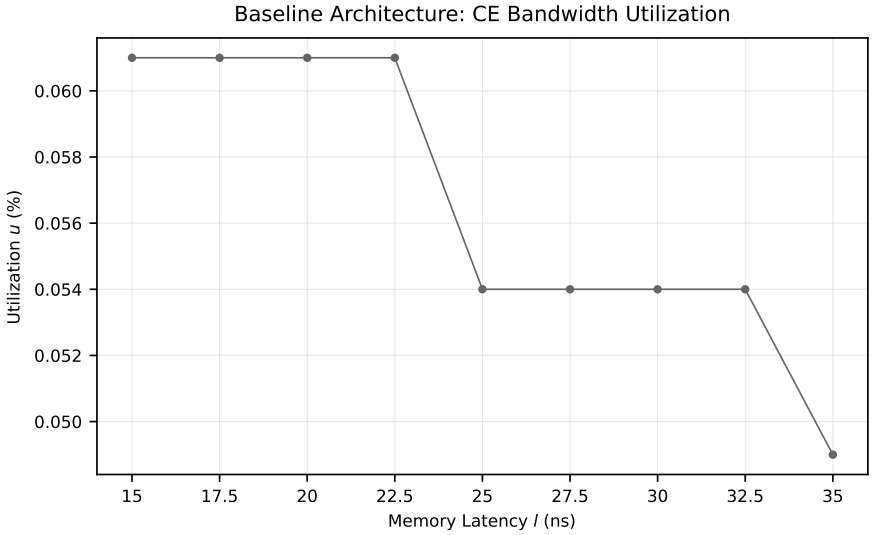


Figure 7.92: Baseline Architecture - CE Bandwidth Utilization across Memory Latency l.

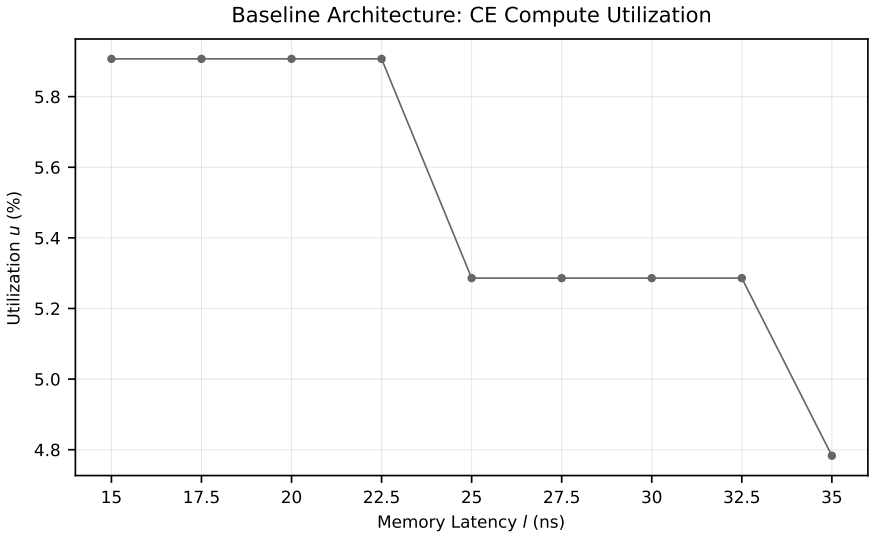


Figure 7.93: Baseline Architecture - CE Compute Utilization across Memory Latency l.

The proposed architecture detains a CE throughput ($tp_{ce,prp}$) between 0.222 MB/s and 0.211 MB/s, as seen in Figure 7.94. Consequently, proposed CE bandwidth utilization ($u_{bw,ce,prp}$) stays near 0.0260% to 0.0280% (Figure 7.95), while compute utilization ($u_{ce,cmp}$) is sustained at $\approx 2.56\% - 2.70\%$ (Figure 7.96), demonstrating reduced latency sensitivity.

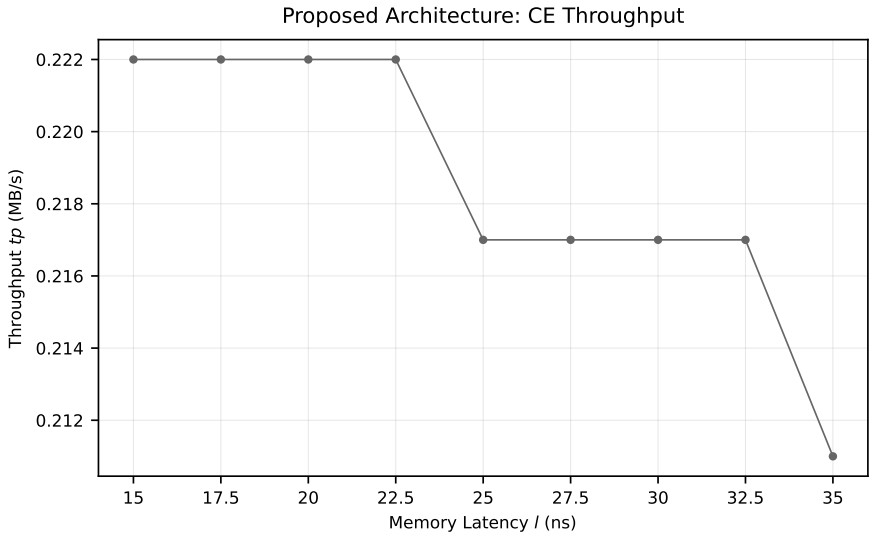


Figure 7.94: Proposed Architecture - CE Throughput across Memory Latency l.

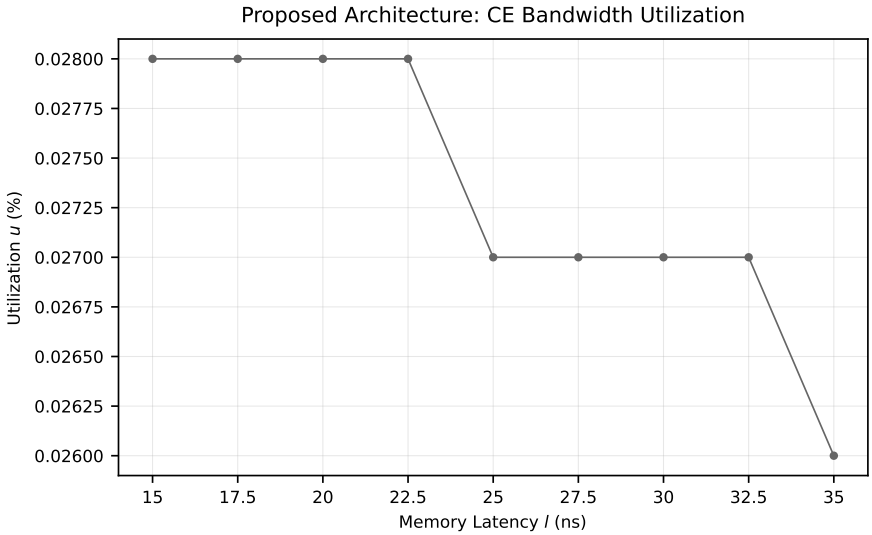


Figure 7.95: Proposed Architecture - CE Bandwidth Utilization across Memory Latency l.

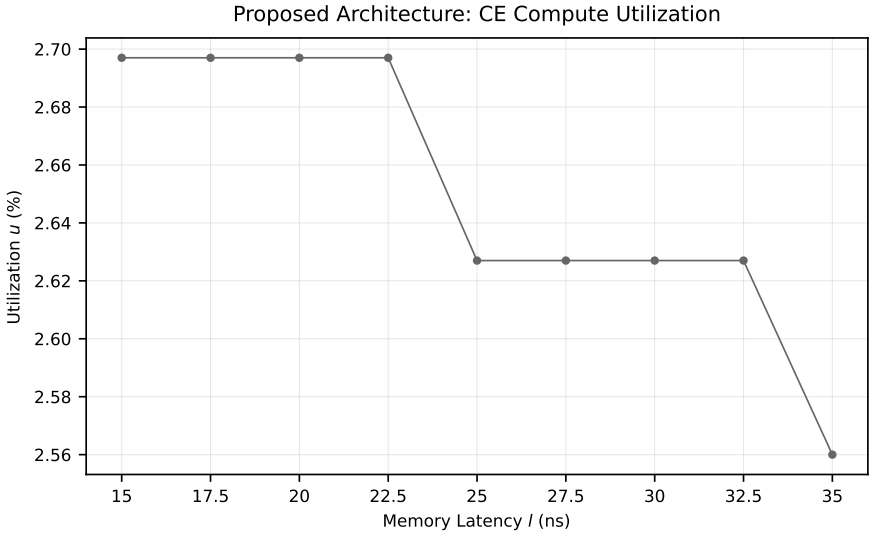


Figure 7.96: Proposed Architecture - CE Compute Utilization across Memory Latency l.

7.3 2D Convolution

7.3.1 Problem Size Sweep

To evaluate the execution performance of the 2D Convolution benchmark, simulations were conducted across a range of input map dimensions. As described in Section 6.4.3, the swept parameter N represents the spatial height and width of the square input map ($I_R \times I_C = N \times N$), which was varied from $N = 31$ to $N = 126$ in steps of 5.

7.3.1.1 Simulation Time and Speedup Analysis

In the baseline architecture, the MC-CE directly manages external memory requests without intermediate caching or buffering mechanisms. For every 2D sliding-window step, the MC-CE issues individual single-read operations to fetch each pixel and kernel element from external memory before writing back the output pixel. Because memory accesses are directly bound to the execution pipeline without localized buffering, the baseline architecture suffers severe memory stalling, leading to high overall simulation times ($t_{\text{sim,bsl}}$) as shown in Figure 7.97.

Mathematically, since the data grid scales quadratically with N, the baseline simulation time follows a quadratic relationship with respect to N, modeled by the regression equation:

$$t_{\text{sim,bsl}}(N) = 1.630 \cdot N^2 - 6.520 \cdot N + 6.550 \quad [\mu\text{s}] \tag{7.89}$$

The dominant quadratic coefficient ($1.630\,\mu$s) reflects the execution scaling factor under the $O(N^2)$ algorithmic complexity for processing an $N \times N$ map with a fixed 3×3 kernel.

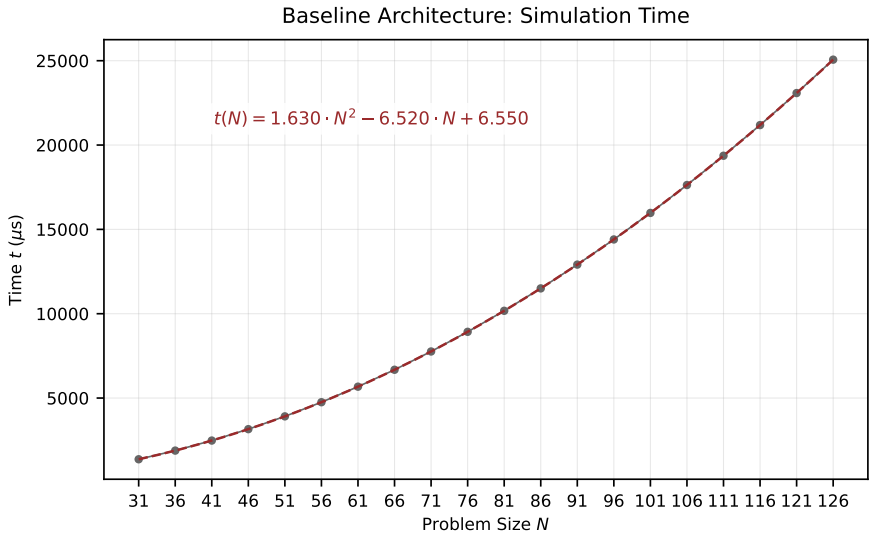


Figure 7.97: Baseline Architecture - Simulation Time across Problem Size N.

Conversely, the proposed architecture employs a SE between main memory and the MD-CE, offloading address generation and managing 4-level nested strided loops to control horizontal and

vertical spatial traversals. Additionally, the 3×3 kernel weights are pre-loaded once into the IS1 SB operating in CB mode, allowing continuous cyclic reuse across all pixels without issuing redundant external memory requests. By shielding the MD-CE from memory latency and keeping the pipeline continuously supplied through local stream buffers, simulation time ($t_{\text{sim,prp}}$) is dramatically reduced while maintaining a quadratic response, as depicted in Figure 7.98:

$$t_{\text{sim,prp}}(N) = 0.372 \cdot N^2 - 1.333 \cdot N + 2.580 \quad [\mu\text{s}] \tag{7.90}$$

Comparing the leading coefficients, the proposed architecture lowers the quadratic factor from $1.630\,\mu$s down to $0.372\,\mu$s. The small linear and constant terms account for the initial buffer loading phase before active streaming begins.

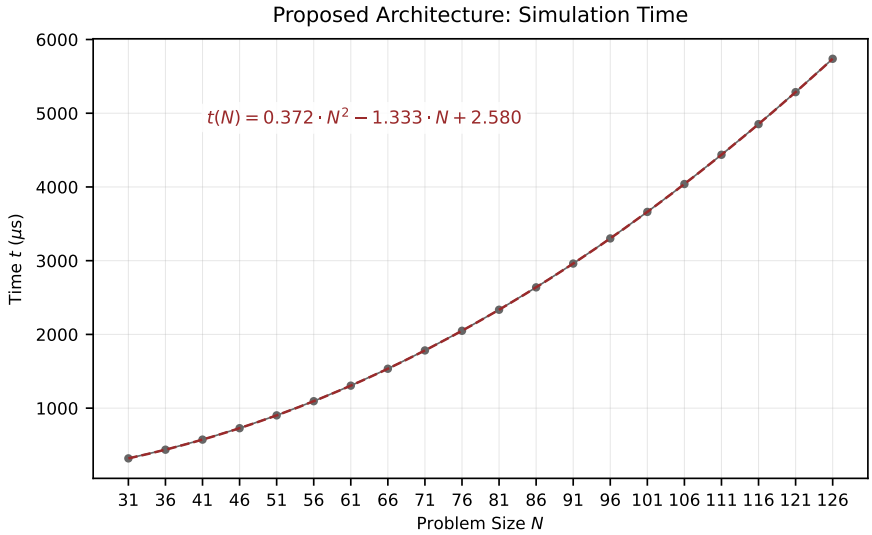


Figure 7.98: Proposed Architecture - Simulation Time across Problem Size N.

The relative speedup $s(N)$ achieved by the proposed architecture over the baseline architecture is shown in Figure 7.99. The empirical speedup curve follows the inverse regression model:

$$s(N) = 4.390 - \frac{2.625}{N} \tag{7.91}$$

Starting from $s(31) \approx 4.301$, the speedup monotonically increases with problem size N as the fixed initial buffer setup overheads are amortized over larger spatial grids. As $N \to \infty$, the transient $\frac{2.625}{N}$ term decays to zero, causing the speedup to asymptotically saturate toward an upper ceiling of $s \approx 4.390\times$, closely matching the theoretical ratio of the leading quadratic execution coefficients ($\frac{1.630}{0.372} \approx 4.381$).

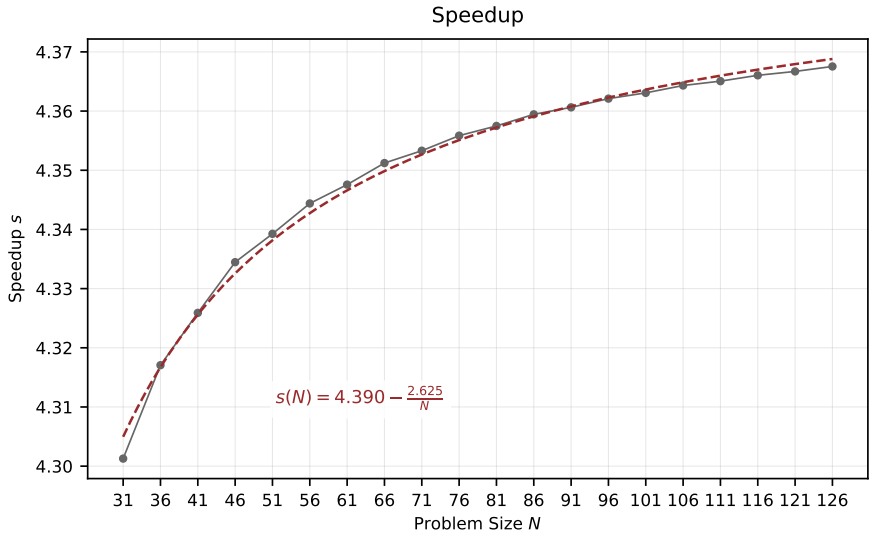


Figure 7.99: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Problem Size N.

7.3.1.2 Memory Access Analysis

In the baseline architecture, the MC-CE lacks intermediate stream buffering or local line caches, requiring direct memory transactions for every input element and kernel weight during the 2D sliding-window traversal. For a 3×3 kernel operating on an $(N-2) \times (N-2)$ output grid, calculating each output pixel requires 9 input image reads, 9 kernel weight fetches, and 1 output write transaction. Consequently, 19 discrete, unbuffered single-access operations are executed per output element, maintaining a constant MAF ($\text{MAF}_{\text{bsl}} = 19.000$) across all grid dimensions N.

Because each single-read and single-write transaction suffers full bus handshake and memory latency, the baseline memory communication time ($t_{\text{mem,bsl}}$) grows quadratically with N in accordance with the spatial grid size, as depicted in Figure 7.100:

$$t_{\text{mem,bsl}}(N) = 0.760 \cdot N^2 - 3.040 \cdot N + 3.040 \quad [\mu\text{s}] \tag{7.92}$$

The dominant quadratic coefficient ($0.760\,\mu$s) underscores the significant latency bottleneck imposed by unbuffered, single-request memory interactions during 2D spatial processing.

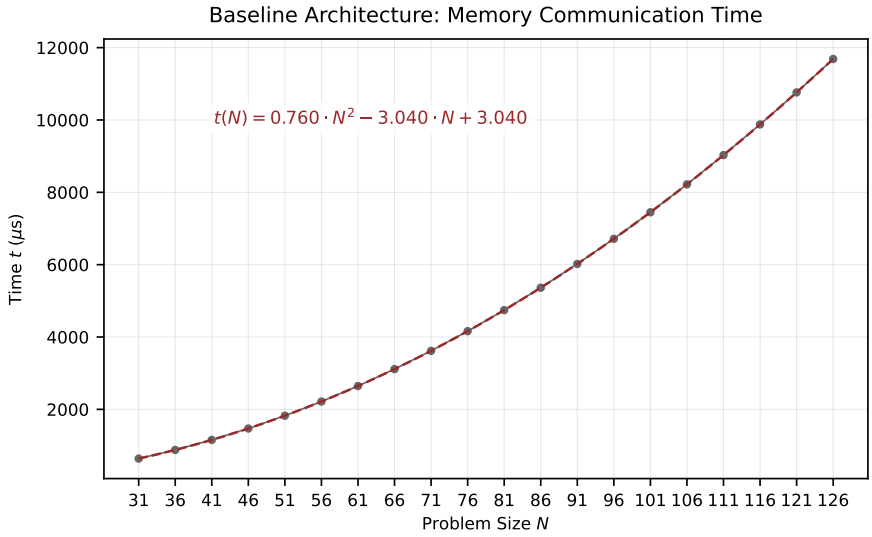


Figure 7.100: Baseline Architecture - Memory Communication Time across Problem Size N.

In contrast, the proposed architecture leverages the SE to decouple compute pipeline execution from memory access. By configuring the IS1 SB in CB mode, kernel weights are loaded once from external memory during initialization and cyclically reused across the entire spatial traversal, completely eliminating redundant coefficient fetches. Furthermore, data requests are transformed into high-throughput burst transfers.

As illustrated in Figure 7.101, the MAF for the proposed architecture (MAF_{prp}) follows an inverse decay relationship across grid sizes:

$$\text{MAF}_{\text{prp}}(N) = \frac{5.377}{N} + 1.989 \tag{7.93}$$

For smaller grid dimensions, the initial one-time pre-loading of the 9 kernel weights slightly inflates the access ratio. As N expands, this fixed initialization cost is rapidly amortized ($\frac{5.377}{N} \to 0$), driving MAF_{prp} asymptotically toward a steady-state lower bound of ~ 1.989 (representing effectively 1 image stream read and 1 output stream write per target coordinate).

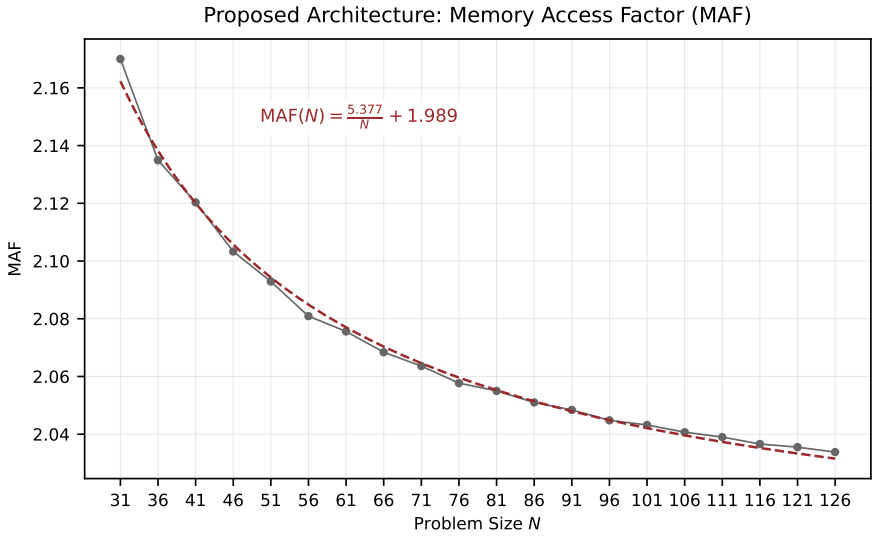


Figure 7.101: Proposed Architecture - Memory Access Factor (MAF) across Problem Size N.

By replacing unbuffered single transactions with optimized streaming bursts and local buffer reuse, memory bus latency is drastically reduced. As shown in Figure 7.102, the proposed memory communication time ($t_{\text{mem,prp}}$) remains quadratic with respect to N, but with a significantly suppressed execution coefficient:

$$t_{\text{mem,prp}}(N) = 0.045 \cdot N^2 - 0.090 \cdot N + 0.588 \quad [\mu\text{s}] \tag{7.94}$$

Comparing the leading quadratic terms reveals that the proposed architecture reduces external memory communication latency scaling from $0.760\,\mu$s down to $0.045\,\mu$s per N^2 element—achieving a $\sim 16.89\times$ reduction in overall memory bus communication time overhead.

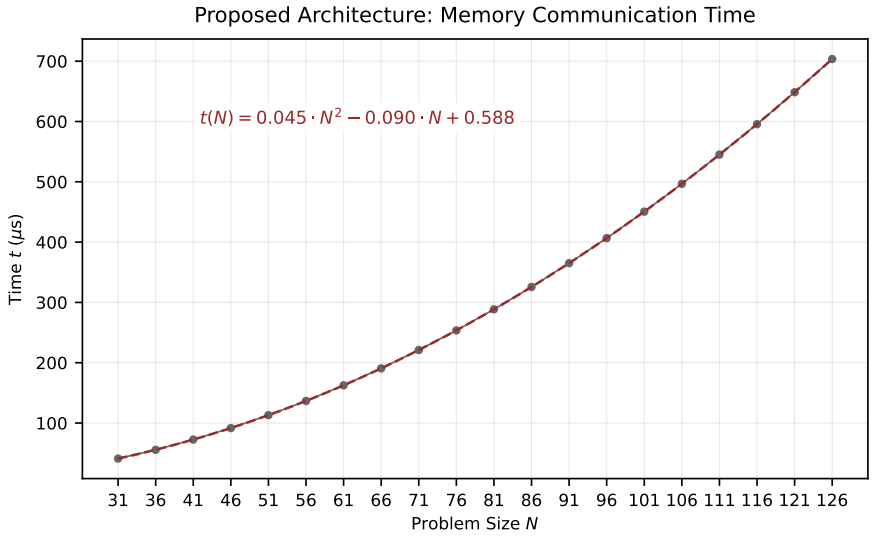


Figure 7.102: Proposed Architecture - Memory Communication Time across Problem Size N.

7.3.1.3 Throughput and Bandwidth Utilization Analysis

To further evaluate system performance, we analyze the CE throughput, bus bandwidth utilization, and internal execution efficiency across both architectures relative to their theoretical hardware limits.

In the baseline architecture, the direct coupling of memory accesses to the compute pipeline without caching or buffering mechanisms severely bottlenecks execution. As analyzed in Section 7.3.1.2, single-transaction memory accesses force the system to spend a significant portion of its runtime stalling for data. On the MC-to-CE interface — which has a maximum theoretical throughput $\text{tp}_{\text{max,mc-ce}} = 800.000\,\text{MB/s}$ — the memory-side transfer throughput ($\text{tp}_{\text{mc-ce}}$) remains flat across all grid dimensions N at 88.344 MB/s, yielding a low bandwidth utilization ($u_{\text{bw,mc-ce}}$) of 11.043%. Because the CE constantly stalls during these unbuffered handshakes, its effective operational data rate ($\text{tp}_{\text{ce,bsl}}$) reaches a constant 4.908 MB/s across all problem sizes N.

This severe memory stalling directly impacts overall hardware utilization. Relative to the maximum theoretical throughput ($\text{tp}_{\text{max,ce,bsl}}$) of 800.000 MB/s, the baseline architecture maintains a constant CE bandwidth utilization ($u_{\text{bw,ce,bsl}}$) of just 0.613%. Furthermore, because the internal arithmetic units sit idle during memory transfers, the baseline CE compute utilization ($u_{\text{ce,cmp,bsl}}$) remains constant at 6.135% across all values of N.

Conversely, the proposed architecture leverages the SE to decouple memory transfers from execution. Internally, the SE streams pre-buffered operand data directly to the CE over a dedicated high-bandwidth local interface rated for a maximum theoretical throughput $\text{tp}_{\text{max,se-ce}} = 1600.000\,\text{MB/s}$. As presented in Figure 7.103, the internal SE-to-CE throughput ($\text{tp}_{\text{se-ce}}$) follows a bounded inverse growth model:

$$\text{tp}_{\text{se-ce}}(N) = 387.785 - \frac{230.484}{N} \quad [\text{MB/s}] \tag{7.95}$$

Rising from 380.030 MB/s at $N = 31$ to an asymptotic steady state of 387.785 MB/s as $N \to \infty$.

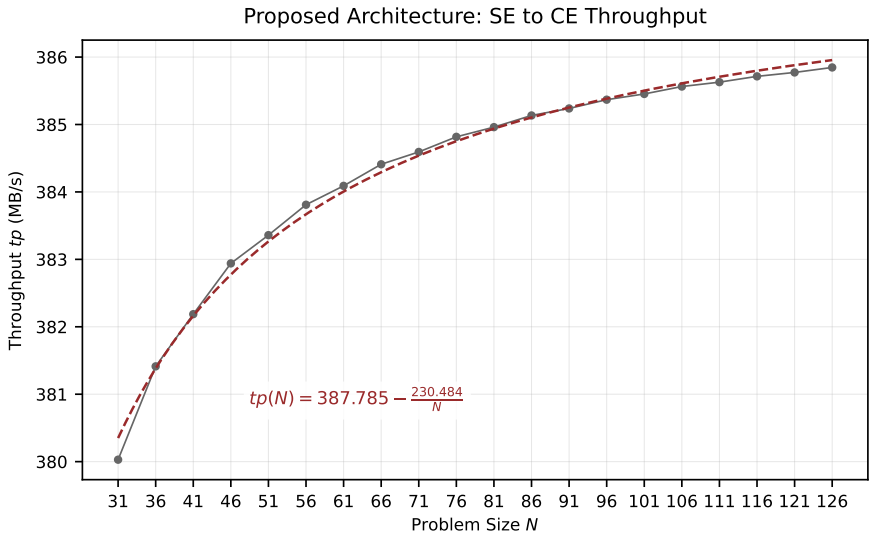


Figure 7.103: Proposed Architecture - SE to CE Throughput across Problem Size N.

Correspondingly, as illustrated in Figure 7.104, the local SE-to-CE bandwidth utilization ($u_{\text{bw,se-ce}}$) scales according to:

$$u_{\text{bw,se-ce}}(N) = 24.236 - \frac{14.399}{N} \quad [\%] \tag{7.96}$$

Scaling from 23.750% at $N = 31$ and stabilizing near 24.236% as $N \to \infty$.

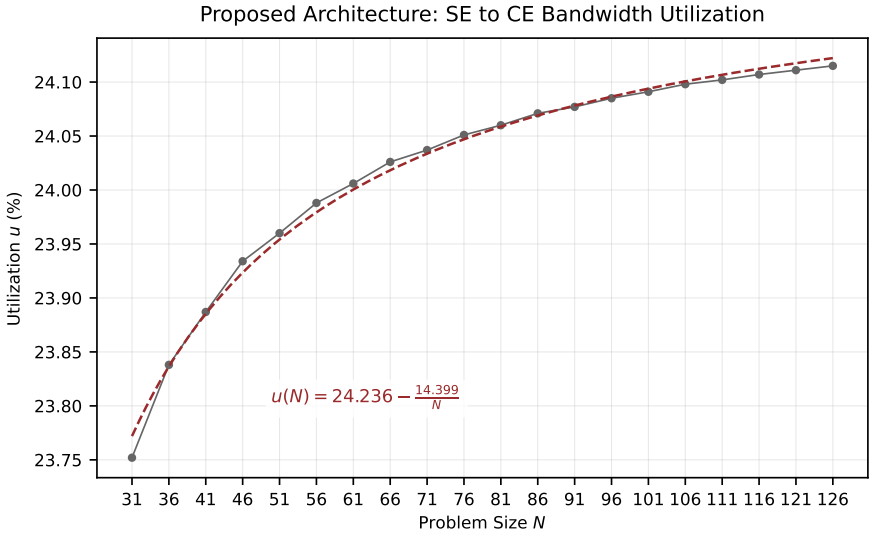


Figure 7.104: Proposed Architecture - SE to CE Bandwidth Utilization across Problem Size N.

Owing to local buffer reuse and stream-based transfers, external memory stalls are drastically

reduced. As demonstrated in Figure 7.105, the effective CE throughput ($\mathrm{tp}_{\mathrm{ce,prp}}$) is modeled by:

$$\mathrm{tp}_{\mathrm{ce,prp}}(N) = \frac{1310.490}{N^2 + 200.218} + 21.449 \quad [\mathrm{MB/s}] \tag{7.97}$$

For smaller grids, boundary pixel overhead and buffer fills briefly elevate the rate before stabilizing to a steady-state throughput of 21.449 MB/s as $N \to \infty$. Compared to the baseline's 4.908 MB/s, this represents an effective throughput increase of 4.37×.

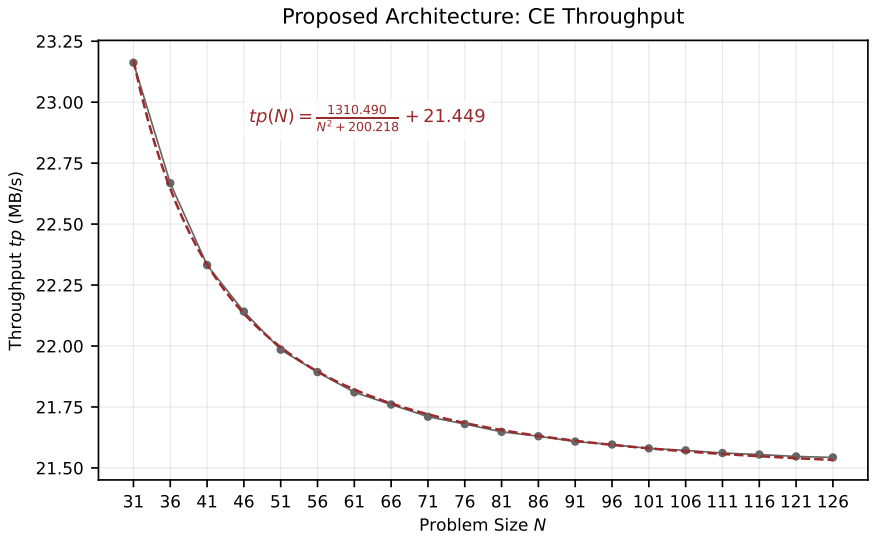


Figure 7.105: Proposed Architecture - CE Throughput across Problem Size N.

Figure 7.106 highlights the overall CE bandwidth utilization ($u_{\mathrm{bw,ce,prp}}$) against the maximum theoretical channel capacity $\mathrm{tp}_{\mathrm{max,ce,prp}} = 800.000$ MB/s:

$$u_{\mathrm{bw,ce,prp}}(N) = \frac{164.353}{N^2 + 197.349} + 2.681 \quad [\%] \tag{7.98}$$

As problem size N expands, $u_{\mathrm{bw,ce,prp}}$ settles to a steady value of 2.681% (compared to 0.613% in the baseline).

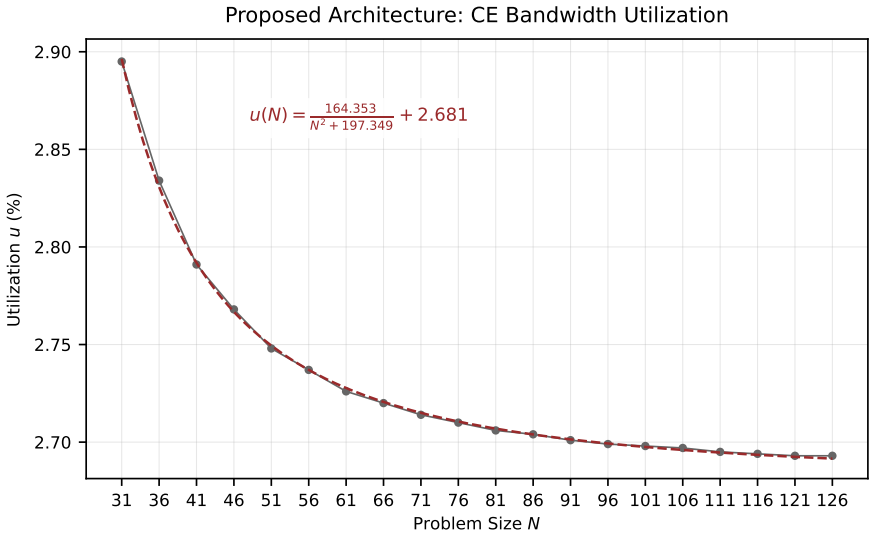


Figure 7.106: Proposed Architecture - CE Bandwidth Utilization across Problem Size N.

Finally, this continuous local data delivery enables the processing engine to execute with decreased stalling. As depicted in Figure 7.107, the CE compute utilization ($u_{\text{ce,cmp,prp}}$) scales as:

$$u_{\text{ce,cmp,prp}}(N) = \frac{1636.603}{N^2 + 201.102} + 26.811 \quad [\%] \tag{7.99}$$

As $N \to \infty$, $u_{\text{ce,cmp,prp}}$ settles to an asymptotic steady-state efficiency of 26.811%, yielding a 4.37× increase in compute utilization over the baseline's constant 6.135%.

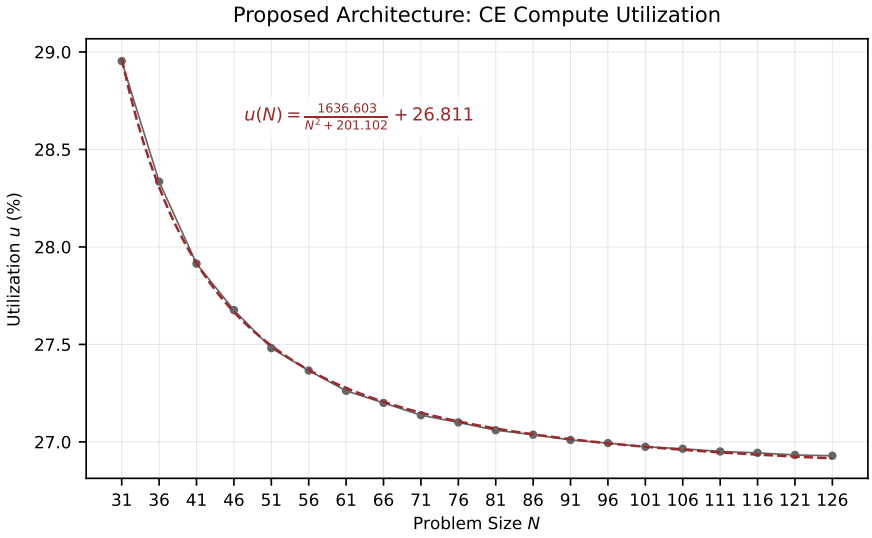


Figure 7.107: Proposed Architecture - CE Compute Utilization across Problem Size N.

7.3.2 Operating Frequency Sweep

To evaluate the scalability and performance sensitivity of both architectures to clock frequency variations, an operating frequency sweep is conducted across $f \in [100, 250]$ MHz. For all frequency sweep evaluations, the problem size is fixed at $N = 96$. This dataset size was specifically selected because, as established in Section 7.3.1.3, $N = 96$ represents an operating regime where throughput and execution efficiency have stabilized, thereby eliminating dataset-scaling transients from the frequency response analysis.

7.3.2.1 Simulation Time and Speedup Analysis

The relationship between operating frequency and execution time highlights fundamental differences in pipeline clocking efficiency between the two system designs.

In the baseline architecture, execution time decreases non-linearly with increasing clock frequency. As illustrated in Figure 7.108, the simulation time $t_{\text{sim,bsl}}$ follows an inverse relationship with operating frequency f:

$$t_{\text{sim,bsl}}(f) = \frac{1.22 \cdot 10^6}{f} + 1917.036 \quad [\mu\text{s}] \tag{7.100}$$

As frequency increases from 100 MHz to 250 MHz, execution time drops from 14389.036 μs down to 7112.036 μs. However, due to fixed memory access handshakes and unbuffered bus latency penalties that remain unscaled by clock frequency, the baseline architecture exhibits diminishing returns at higher operating frequencies.

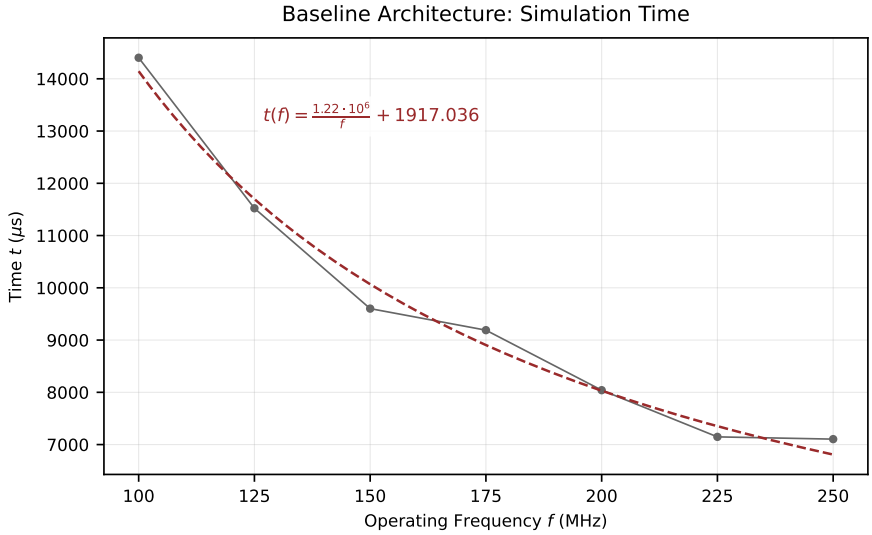


Figure 7.108: Baseline Architecture - Simulation Time across Operating Frequency f.

Conversely, the proposed architecture also exhibits an inverse relationship with frequency, but with drastically reduced overheads. As shown in Figure 7.109, the simulation time $t_{\text{sim,prp}}$ scales

as:

$$t_{\text{sim,prp}}(f) = \frac{3.27 \cdot 10^5}{f} + 25.829 \quad [\mu\text{s}] \tag{7.101}$$

As f increases from 100 MHz to 250 MHz, execution time decreases from 3295.829 μs to 1333.829 μs. The offset constant is reduced to just 25.829 μs, demonstrating that memory stalling overheads are substantially eliminated.

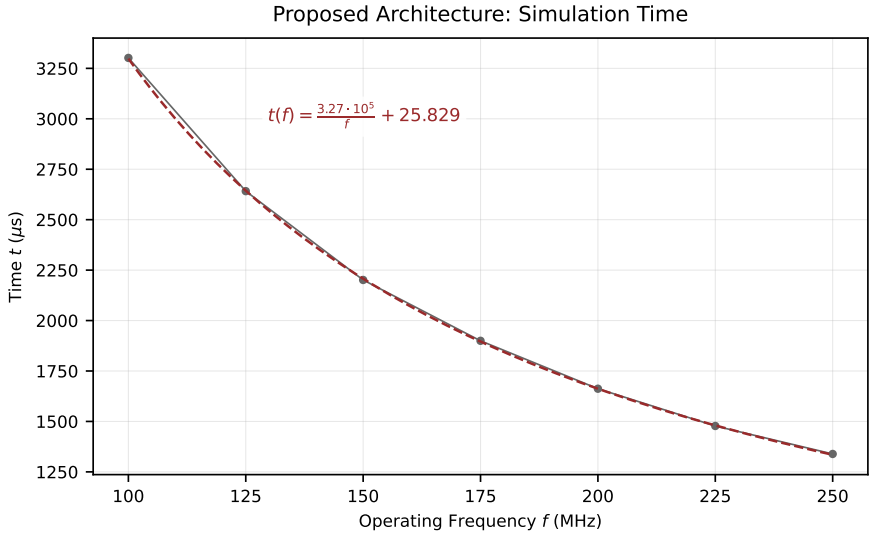


Figure 7.109: Proposed Architecture - Simulation Time across Operating Frequency f.

The resulting speedup $s(f)$ achieved by the proposed architecture over the baseline architecture across operating frequencies is depicted in Figure 7.110. Overall, the speedup grows from 4.37× at 100 MHz up to 5.33× at 250 MHz. The step-like jumps in the speedup curve occur because external memory latency is fixed in physical time, forcing the MC latency to scale in discrete clock cycles. As frequency increases, the cycle count remains constant until a frequency threshold is crossed, at which point the memory response time requires an additional clock cycle. Because these discrete latency penalties degrade the baseline architecture's efficiency more severely than the proposed design, each cycle threshold crossing produces a step jump in the speedup curve.

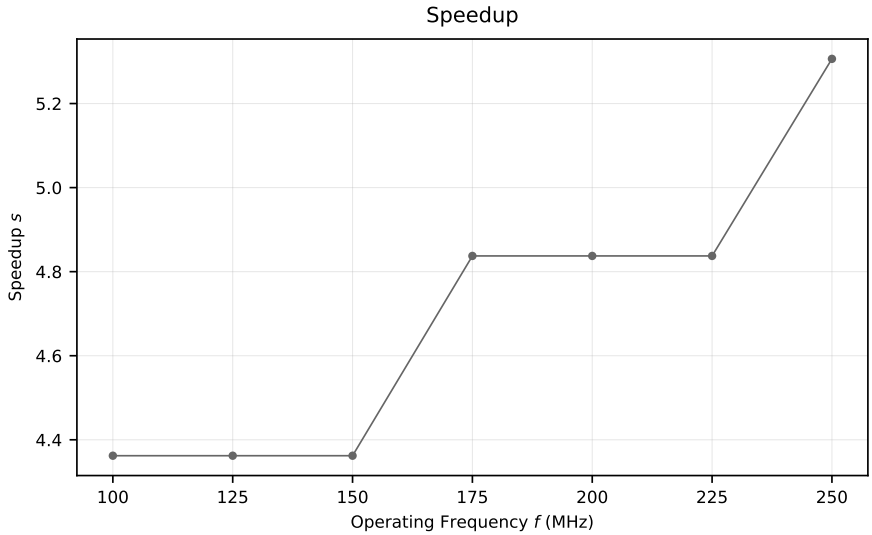


Figure 7.110: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Operating Frequency f.

7.3.2.2 Memory Access Analysis

To isolate the memory interface efficiency from computation across operating frequencies, we evaluate both the MAF and total memory communication time (t_{mem}) at a fixed problem size $N = 96$.

In the baseline architecture, the memory access factor remains strictly invariant across all operating frequencies at $\text{MAF}_{\text{bsl}} = 19.000$. Because the baseline system performs unbuffered single-data memory transfers for every kernel coefficient and map element fetch, frequency scaling does not alter the structural number of memory accesses required per computation.

However, as shown in Figure 7.111, the baseline memory communication time ($t_{\text{mem,bsl}}$) exhibits non-monotonic behavior across frequency increments. While higher frequencies generally reduce execution time, discrete upward jumps occur between $150\,\text{MHz} \rightarrow 175\,\text{MHz}$ and $225\,\text{MHz} \rightarrow 250\,\text{MHz}$. This behavior stems directly from the fixed physical response delay of the memory system. Because the latency must round up to discrete integer clock cycles, operating at higher frequencies reduces the clock period T_{clk}, crossing threshold points where the memory response takes an additional clock cycle per access. Because MAF_{bsl} is high (19.000), these accumulated single-cycle penalty steps cause notable spikes in the total communication delay.

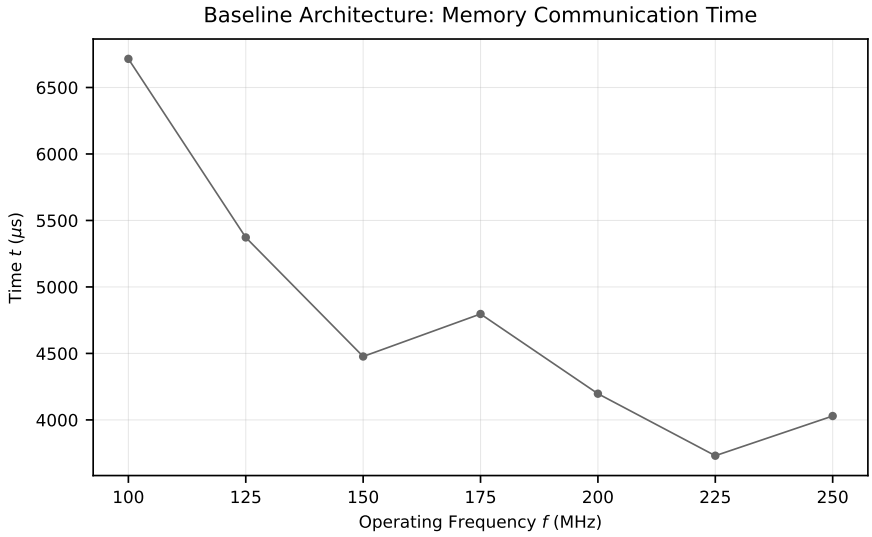


Figure 7.111: Baseline Architecture - Memory Communication Time across Operating Frequency f.

Conversely, the proposed architecture maintains a constant $\text{MAF}_{\text{prp}} = 2.045$ across all frequencies, demonstrating solid data reuse due to the SE's stream buffering mechanisms.

As illustrated in Figure 7.112, the memory communication time of the proposed architecture ($t_{\text{mem,prp}}$) follows an inverse relationship with operating frequency:

$$t_{\text{mem,prp}}(f) = \frac{3.77 \cdot 10^4}{f} + 25.829 \quad [\mu\text{s}] \tag{7.102}$$

Because burst-mode transfers and specialized line buffers drastically minimize total memory transactions ($\text{MAF} \approx 2.045$), integer clock cycle quantization effects on external latency are completely absorbed. This allows memory communication time to scale smoothly from $406.829\,\mu$s down to $176.629\,\mu$s as f increases from 100 MHz to 250 MHz.

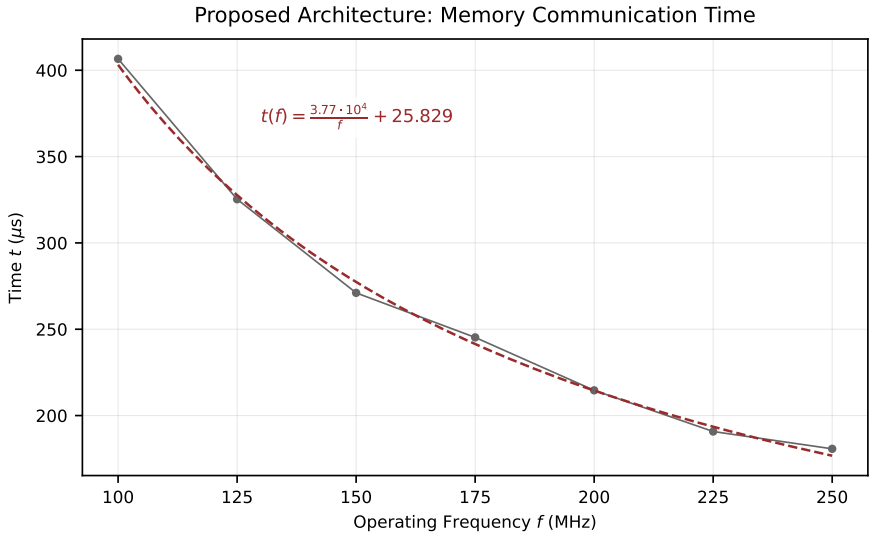


Figure 7.112: Proposed Architecture - Memory Communication Time across Operating Frequency f.

7.3.2.3 Throughput and Bandwidth Utilization Analysis

To evaluate memory bus efficiency and processing engine performance across operating frequencies, we analyze the throughput (tp) and utilization (u) metrics for both architectures at a fixed problem size $N = 96$.

For the baseline architecture, the maximum theoretical MC-to-CE throughput ($\text{tp}_{\text{max,mc-ce}}$) scales linearly as a function of the operating frequency:

$$\text{tp}_{\text{max,mc-ce}}(f) \approx 8.000 \cdot f \tag{7.103}$$

reaching 2000 MB/s at 250 MHz. However, actual MC-to-CE throughput follows:

$$\text{tp}_{\text{mc-ce}}(f) = 0.619 \cdot f + 32.433 \tag{7.104}$$

affected by discrete clock-cycle quantization stalls between 150 MHz $\rightarrow$ 175 MHz and 225 MHz $\rightarrow$ 250 MHz. In regions where latency cycle requirements remain constant, both tp_{max} and measured throughput increase proportionally with frequency. Conversely, crossing latency threshold boundaries introduces additional stall cycles, causing measured throughput to deviate downward relative to capacity. Consequently, the MC-to-CE bandwidth utilization ($u_{\text{bw,mc-ce}}$) remains constrained between 9.0% and 11.0% across the frequency range.

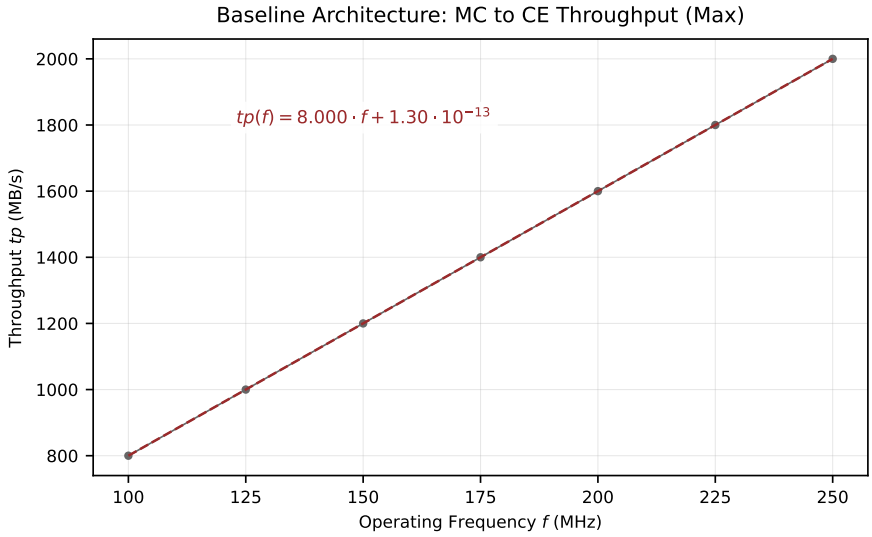


Figure 7.113: Baseline Architecture - MC to CE Maximum Theoretical Throughput across Operating Frequency f.

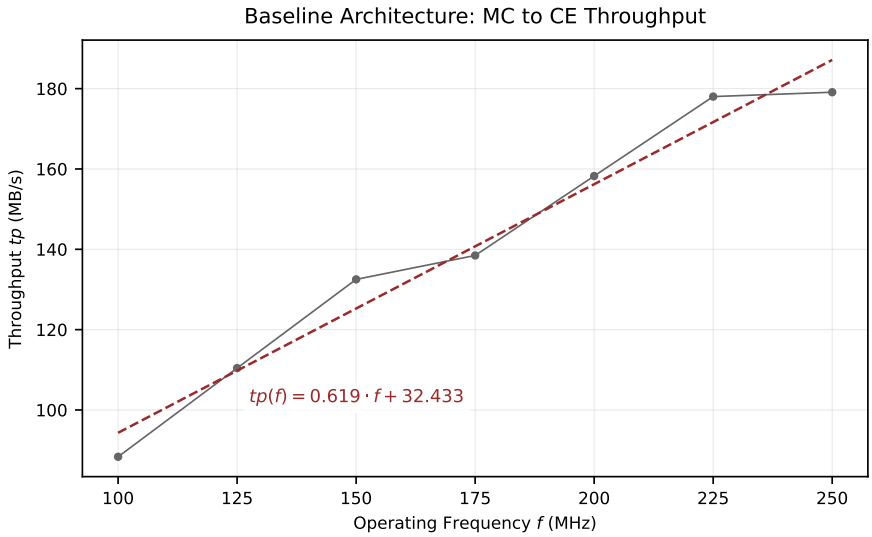


Figure 7.114: Baseline Architecture - MC to CE Throughput across Operating Frequency f.

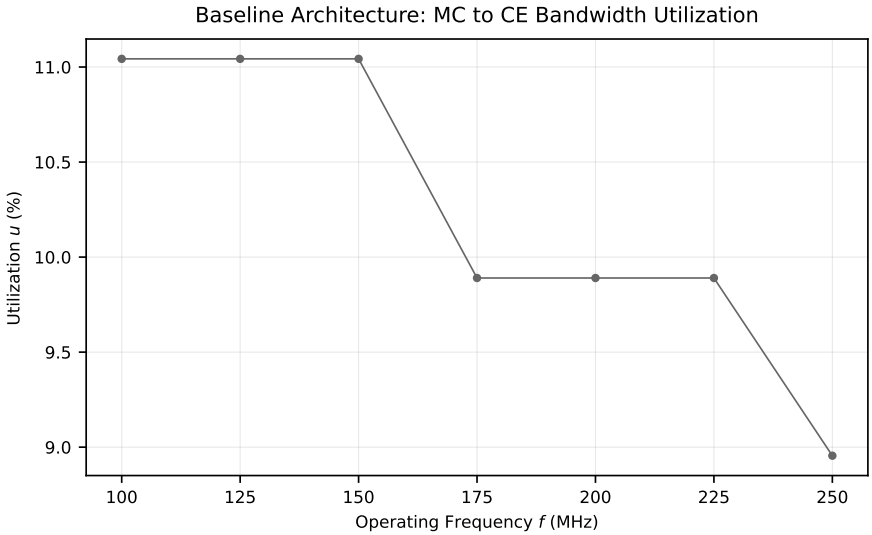


Figure 7.115: Baseline Architecture - MC to CE Bandwidth Utilization across Operating Frequency f.

For the proposed architecture, the SE-to-CE interface features a wider bus, yielding a theoretical maximum throughput of:

$$\mathrm{tp}_{\mathrm{max,se\text{-}ce}}(f) \approx 16.000 \cdot f \tag{7.105}$$

and an effective throughput fitted by:

$$\mathrm{tp}_{\mathrm{se\text{-}ce}}(f) = 3.774 \cdot f + 9.921 \tag{7.106}$$

Bandwidth utilization $u_{\mathrm{bw,se\text{-}ce}}$ scales near $23.76\% - 24.08\%$, representing a $\approx 2.3\times$ efficiency improvement over the MC-to-CE interface.

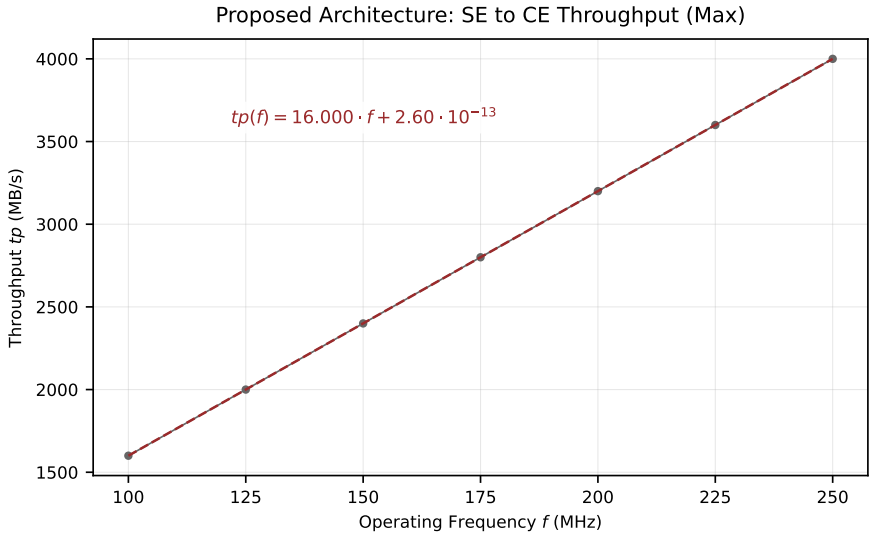


Figure 7.116: Proposed Architecture - SE to CE Maximum Theoretical Throughput across Operating Frequency f.

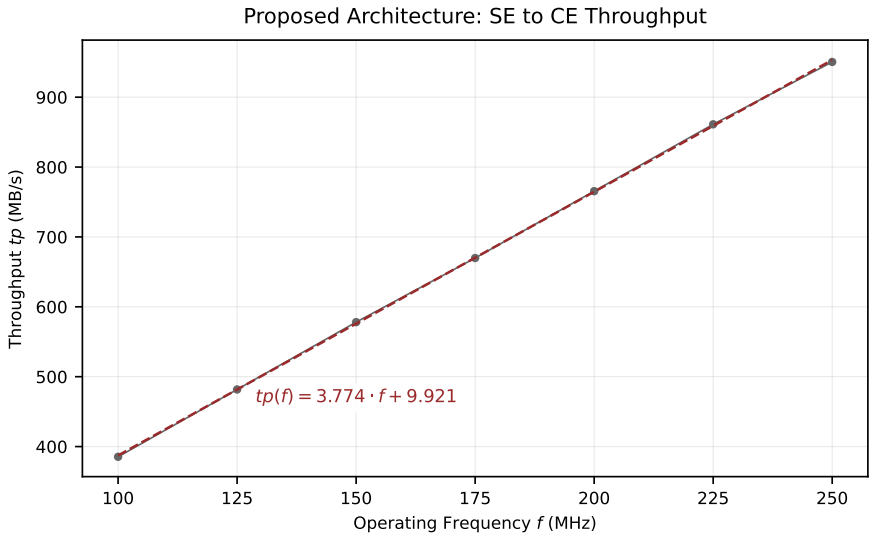


Figure 7.117: Proposed Architecture - SE to CE Throughput across Operating Frequency f.

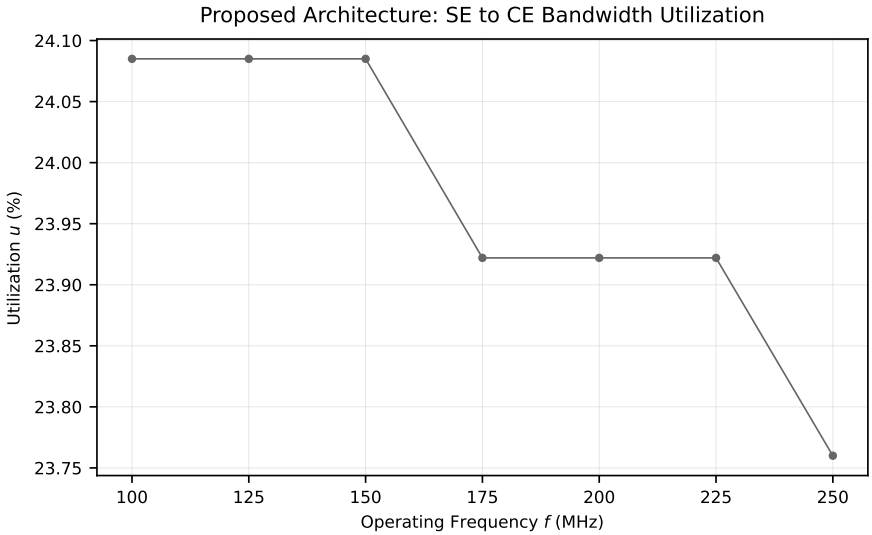


Figure 7.118: Proposed Architecture - SE to CE Bandwidth Utilization across Operating Frequency f.

The maximum CE theoretical throughput ($\text{tp}_{\text{max,ce}}$) scales identically in both architectures:

$$\text{tp}_{\text{max,ce}}(f) \approx 8.000 \cdot f \tag{7.107}$$

However, the baseline CE throughput ($\text{tp}_{\text{ce,bsl}}$) achieves only:

$$\text{tp}_{\text{ce,bsl}}(f) = 0.034 \cdot f + 1.802 \tag{7.108}$$

due to continuous unbuffered memory stalls, leading to a low compute utilization ($u_{\text{ce,cmp,bsl}}$) between 5.0% and 6.15%.

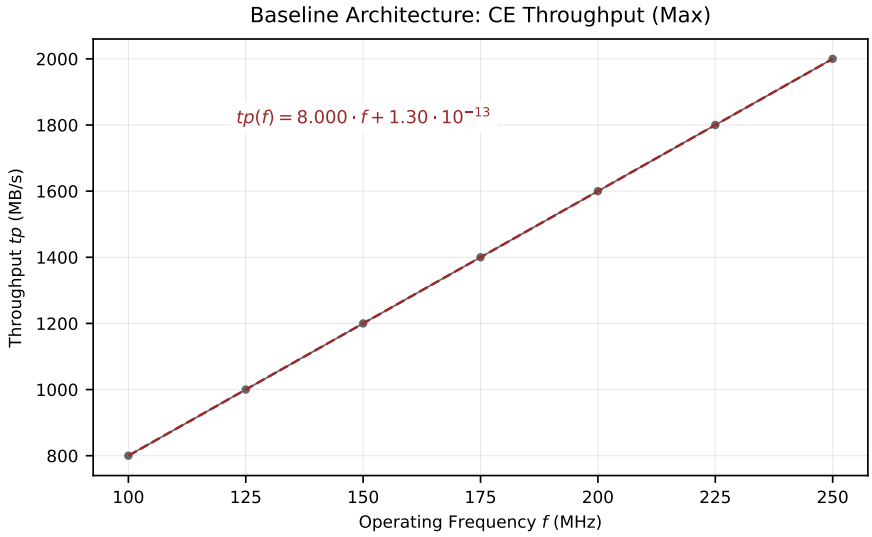


Figure 7.119: Baseline Architecture - CE Maximum Theoretical Throughput across Operating Frequency f.

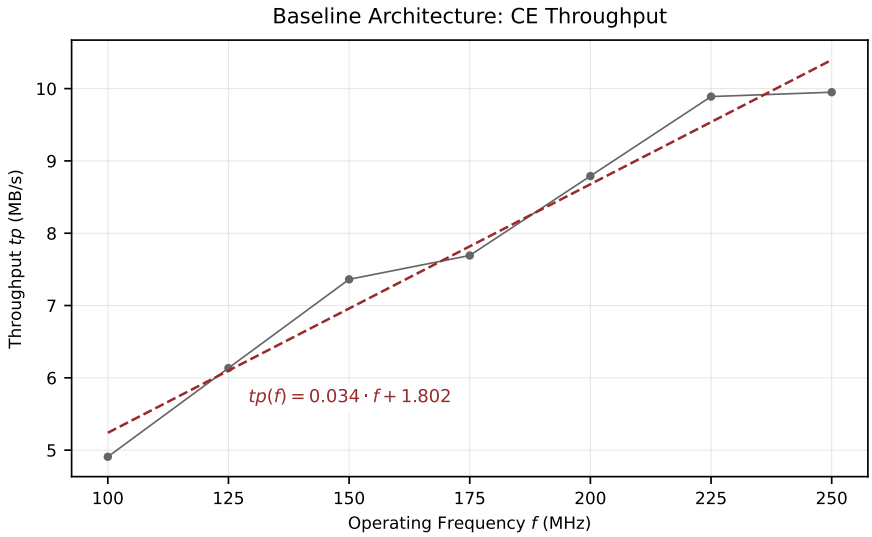


Figure 7.120: Baseline Architecture - CE Throughput across Operating Frequency f.

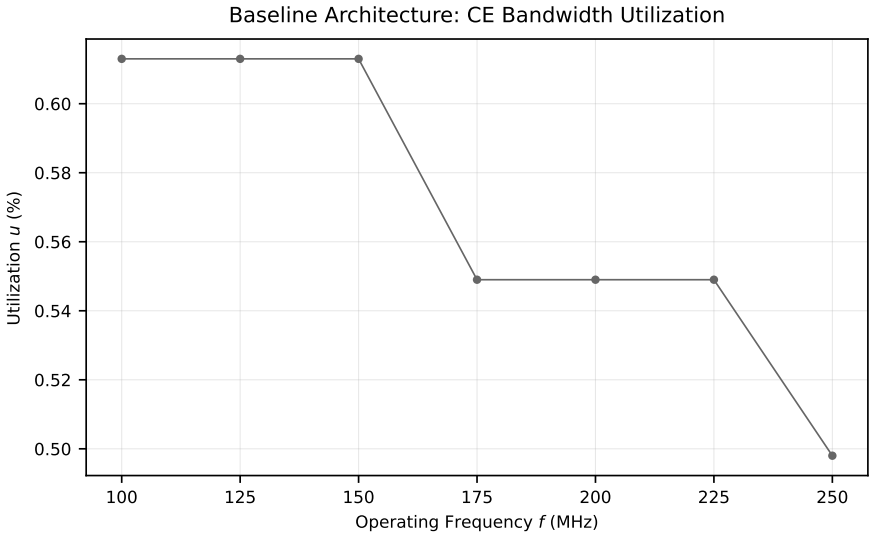


Figure 7.121: Baseline Architecture - CE Bandwidth Utilization across Operating Frequency f.

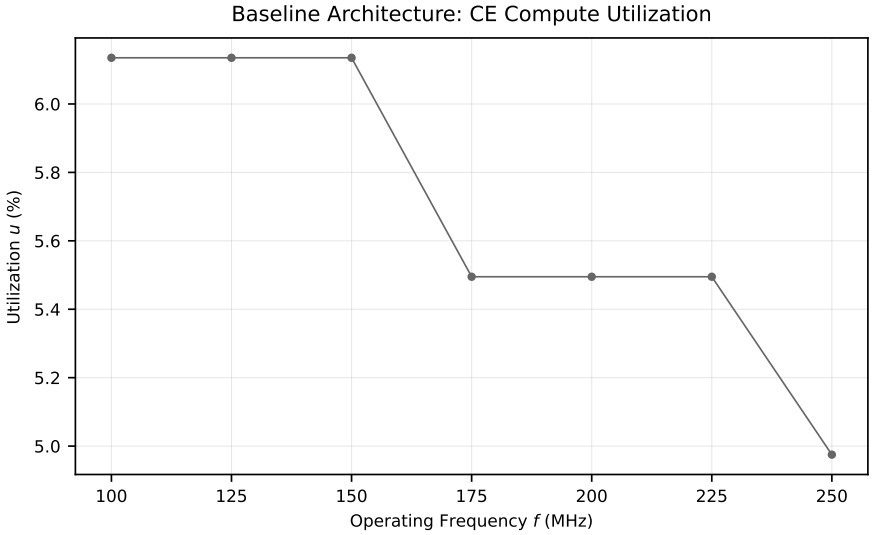


Figure 7.122: Baseline Architecture - CE Compute Utilization across Operating Frequency f.

In contrast, the proposed design achieves:

$$\mathrm{tp}_{\mathrm{ce,prp}}(f) = 0.212 \cdot f + 0.541 \tag{7.109}$$

raising compute utilization $u_{\mathrm{ce,cmp,prp}}$ to $\approx 26.65\% - 27.0\%$. Across all utilization metrics, plateau regions correspond to frequency bands with constant memory latency cycles, whereas discrete downward steps mark threshold boundaries ($150\,\mathrm{MHz} \rightarrow 175\,\mathrm{MHz}$ and $225\,\mathrm{MHz} \rightarrow 250\,\mathrm{MHz}$) where cycle quantization increases memory transaction's overhead.

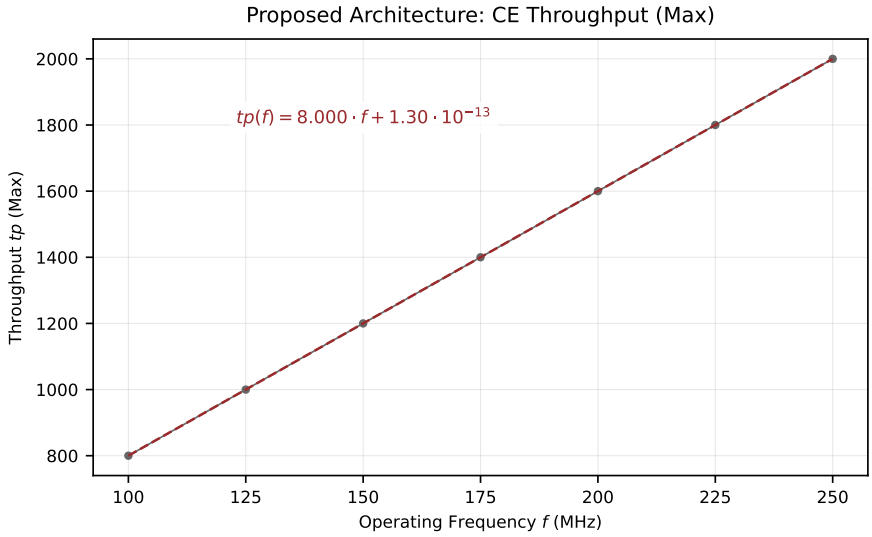


Figure 7.123: Proposed Architecture - CE Maximum Theoretical Throughput across Operating Frequency f.

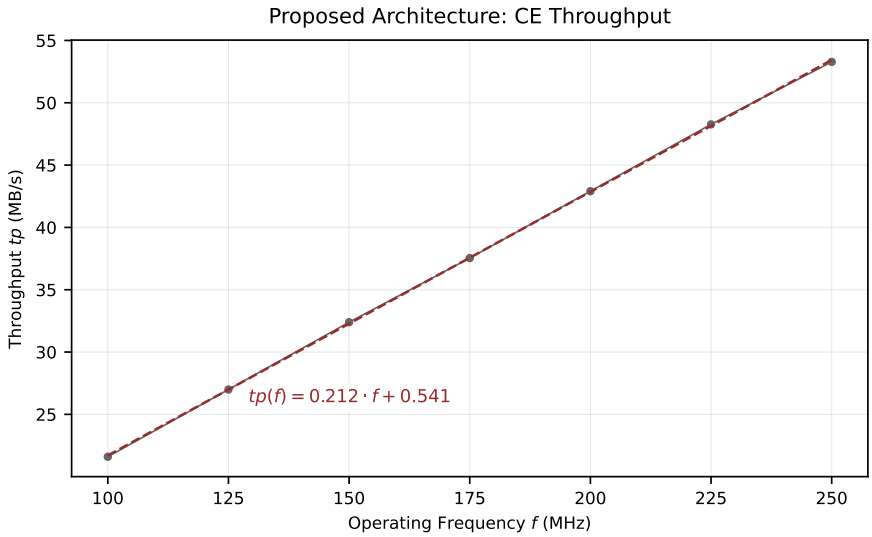


Figure 7.124: Proposed Architecture - CE Throughput across Operating Frequency f.

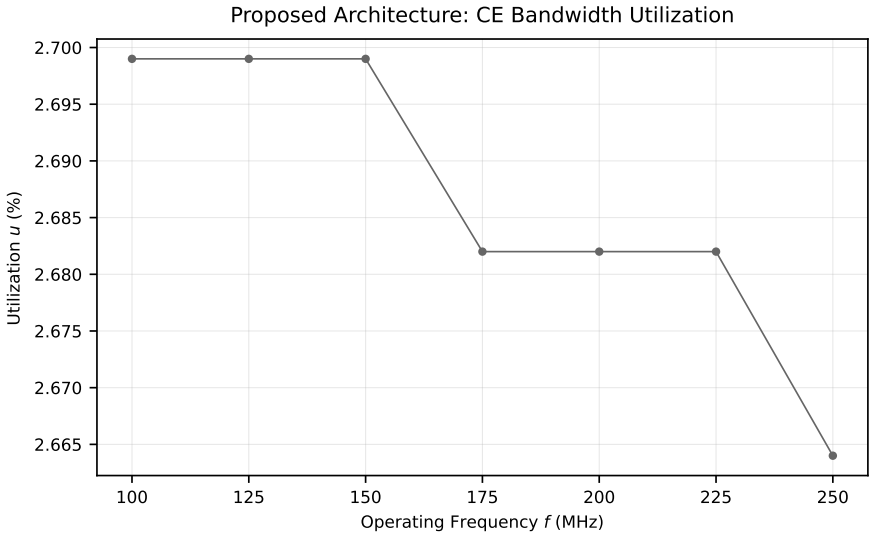


Figure 7.125: Proposed Architecture - CE Bandwidth Utilization across Operating Frequency f.

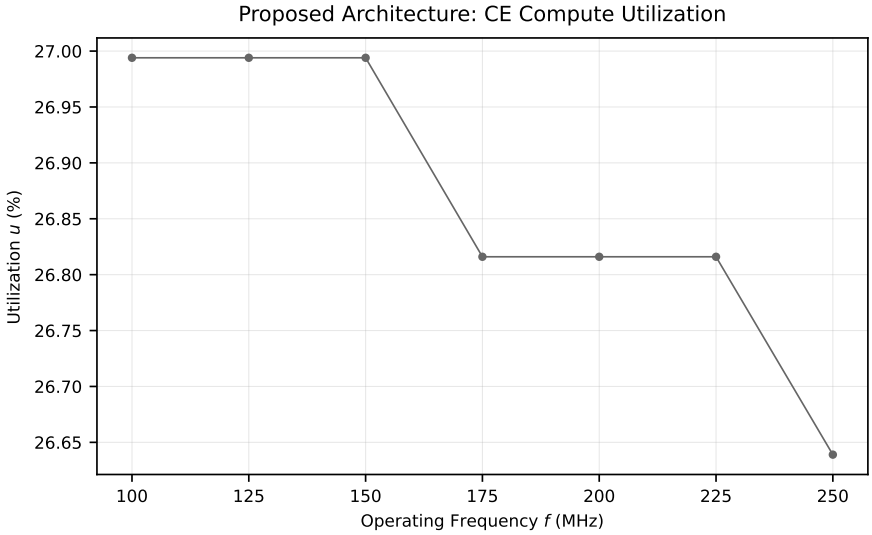


Figure 7.126: Proposed Architecture - CE Compute Utilization across Operating Frequency f.

7.3.3 Memory Latency Sweep

To evaluate memory interface efficiency and system robustness under varying external memory response delays, a memory latency sweep is performed across $l \in [15.0, 35.0]$ ns. As established in previous sections, the problem size is fixed at $N = 96$. All simulations operate at a constant clock frequency $f = 100$ MHz ($T_{\text{clk}} = 10$ ns).

7.3.3.1 Simulation Time and Speedup Analysis

For the baseline architecture, execution time exhibits a step-like staircase profile as a function of external memory latency l. Because access delays are quantized into discrete integer clock cycles ($L = [\![l/T_{\text{clk}}]\!]$), the total simulation time $t_{\text{sim,bsl}}$ remains flat across sub-cycle latency intervals before stepping upward at cycle quantization boundaries:

$$t_{\text{sim,bsl}}(l) = 14400\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.110}$$

$$t_{\text{sim,bsl}}(l) = 16080\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.111}$$

$$t_{\text{sim,bsl}}(l) = 17760\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.112}$$

Due to unbuffered, direct memory accesses for every kernel/map element during processing, each additional cycle of latency (+10 ns) introduces significant cumulative stall overhead, increasing baseline execution time by 1680μs per latency cycle step.

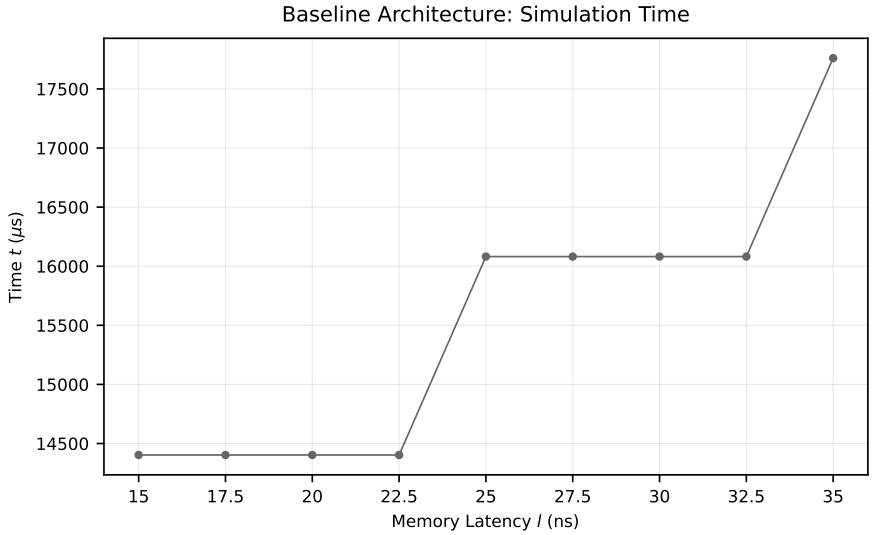


Figure 7.127: Baseline Architecture - Simulation Time across Memory Latency l.

For the proposed architecture, local stream buffering via the SE decouples compute execution from external memory access penalties. Consequently, the simulation time $t_{\text{sim,prp}}$ exhibits far higher resilience to memory latency increases:

$$t_{\text{sim,prp}}(l) = 3301.8\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.113}$$

$$t_{\text{sim,prp}}(l) = 3324.4\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.114}$$

$$t_{\text{sim,prp}}(l) = 3347.0\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.115}$$

Each cycle threshold transition increases total execution time by only 22.6μs, demonstrating that hardware pre-buffering successfully hides memory fetch latencies behind pipeline execution.

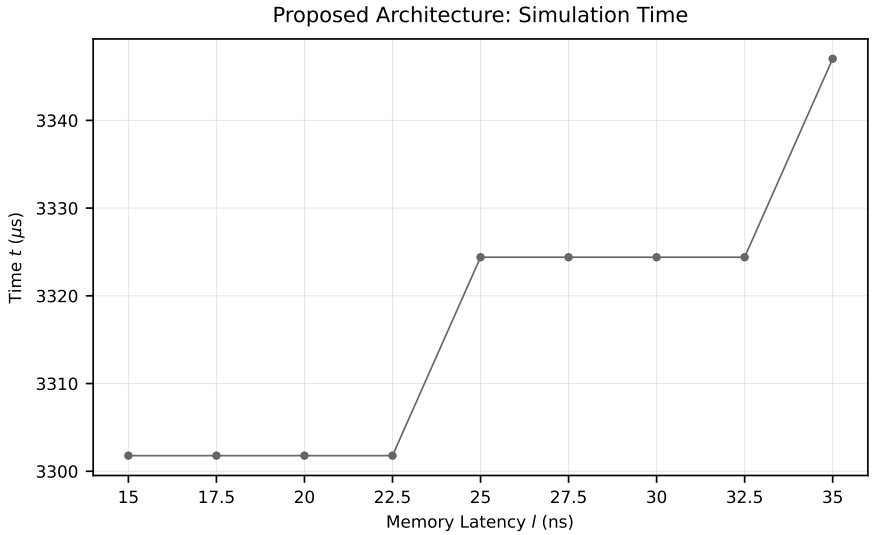


Figure 7.128: Proposed Architecture - Simulation Time across Memory Latency l.

The speedup increases as memory latency degrades:

$$s(l) \approx 4.361 \times \quad \text{for } l \in [15.0, 22.5] \text{ ns} \tag{7.116}$$

$$s(l) \approx 4.837 \times \quad \text{for } l \in [25.0, 32.5] \text{ ns} \tag{7.117}$$

$$s(l) \approx 5.306 \times \quad \text{for } l = 35.0 \text{ ns} \tag{7.118}$$

Because the baseline architecture suffers severe cumulative penalties under higher memory latencies while the proposed design maintains near-constant processing speed, the speedup scales higher at larger values of l.

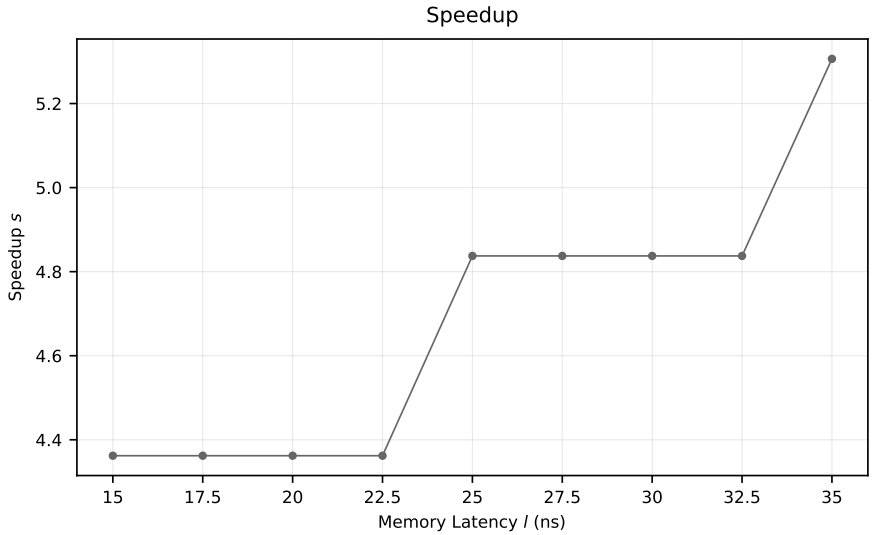


Figure 7.129: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Memory Latency l.

7.3.3.2 Memory Access Analysis

Because memory latency l affects transaction duration rather than the total volume of memory accesses, the MAF remains constant across all values of l for both architectures at a fixed problem size $N = 96$.

For the baseline architecture, the memory access factor stays invariant at:

$$\mathrm{MAF}_{\mathrm{bsl}} = 19.000 \tag{7.119}$$

reflecting redundant scalar fetches for each operation. Consequently, memory communication time (t_{mem}) increases directly with memory latency cycle thresholds:

$$t_{\mathrm{mem,bsl}}(l) = 6720.0\mu\mathrm{s} \quad \text{for } l \in [15.0, 22.5]\ \mathrm{ns} \quad (L = 2) \tag{7.120}$$

$$t_{\mathrm{mem,bsl}}(l) = 8400.0\mu\mathrm{s} \quad \text{for } l \in [25.0, 32.5]\ \mathrm{ns} \quad (L = 3) \tag{7.121}$$

$$t_{\mathrm{mem,bsl}}(l) = 10080.0\mu\mathrm{s} \quad \text{for } l = 35.0\ \mathrm{ns} \quad (L = 4) \tag{7.122}$$

Each latency cycle step introduces 1680.0μs of additional communication overhead, severely degrading overall system efficiency.

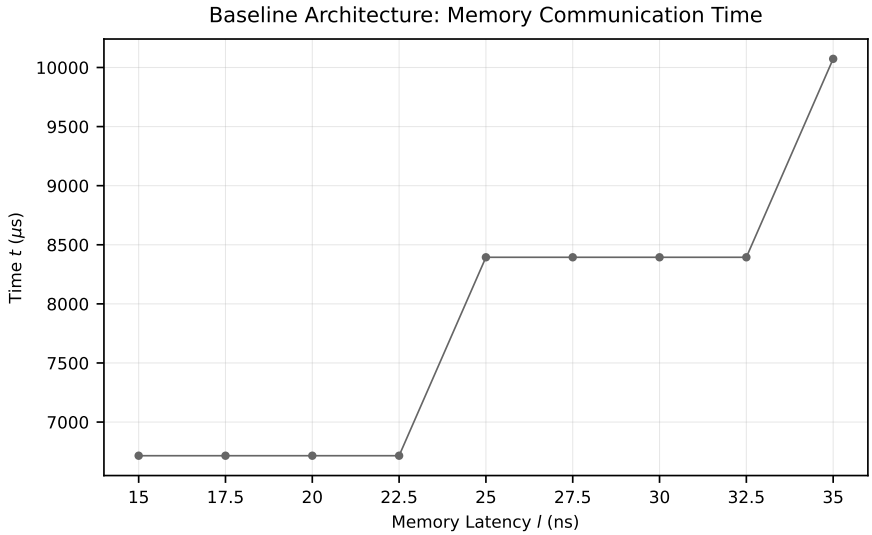


Figure 7.130: Baseline Architecture - Memory Communication Time across Memory Latency l.

For the proposed architecture, the memory access factor is similarly invariant:

$$\mathrm{MAF}_{\mathrm{prp}} = 2.045 \tag{7.123}$$

demonstrating high memory efficiency due to pre-buffering and data reuse via the SE. As a result, memory communication time (t_{mem}) is significantly reduced and shows high resilience to increased latency:

$$t_{\mathrm{mem,prp}}(l) = 406.6\mu\mathrm{s} \quad \text{for } l \in [15.0, 22.5]\ \mathrm{ns} \quad (L = 2) \tag{7.124}$$

$$t_{\mathrm{mem,prp}}(l) = 429.2\mu\mathrm{s} \quad \text{for } l \in [25.0, 32.5]\ \mathrm{ns} \quad (L = 3) \tag{7.125}$$

$$t_{\mathrm{mem,prp}}(l) = 451.8\mu\mathrm{s} \quad \text{for } l = 35.0\ \mathrm{ns} \quad (L = 4) \tag{7.126}$$

Each latency cycle step increases communication time by only 22.6μs, representing a $\approx 16.5\times$ to $22.3\times$ reduction in communication delay compared to the baseline design.

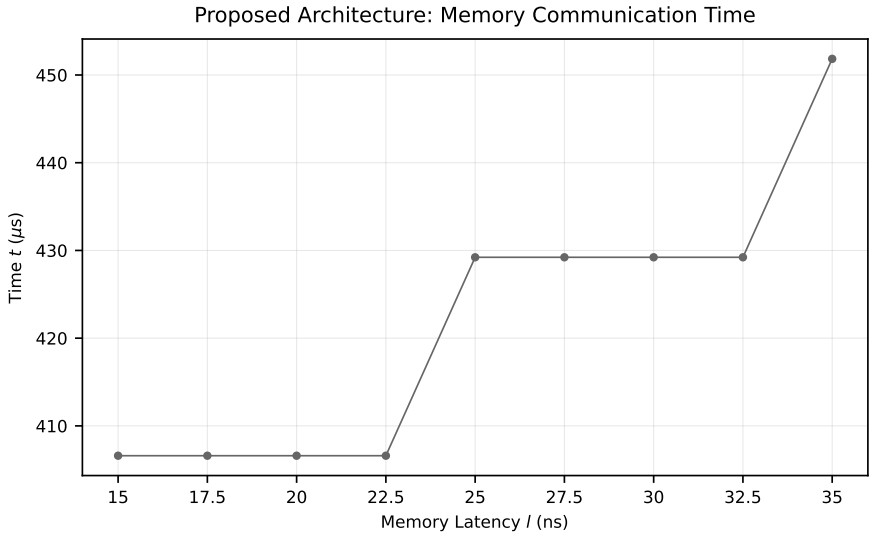


Figure 7.131: Proposed Architecture - Memory Communication Time across Memory Latency l.

7.3.3.3 Throughput and Bandwidth Utilization Analysis

For the baseline architecture, the maximum theoretical throughput of the MC-to-CE interface is $\text{tp}_{\text{max,mc-ce}} = 800.0$ MB/s. The effective MC-to-CE throughput ($\text{tp}_{\text{mc-ce}}$) follows a step-down pattern governed by discrete clock cycle quantization ($L = [\![l/T_{\text{clk}}]\!]$), as depicted in Figure 7.132:

$$\text{tp}_{\text{mc-ce}}(l) = 88.3 \text{ MB/s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.127}$$

$$\text{tp}_{\text{mc-ce}}(l) = 79.1 \text{ MB/s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.128}$$

$$\text{tp}_{\text{mc-ce}}(l) = 71.7 \text{ MB/s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.129}$$

As shown in Figure 7.133, this decline causes bus bandwidth utilization ($u_{\text{bw,mc-ce}}$) to drop from 11.04% to 9.89% and down to 8.96% across cycle thresholds.

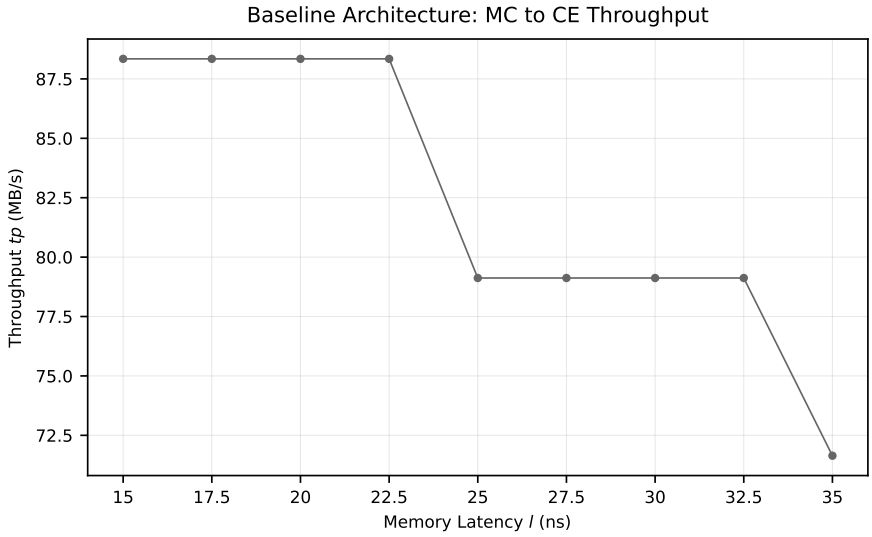


Figure 7.132: Baseline Architecture - MC to CE Throughput across Memory Latency l.

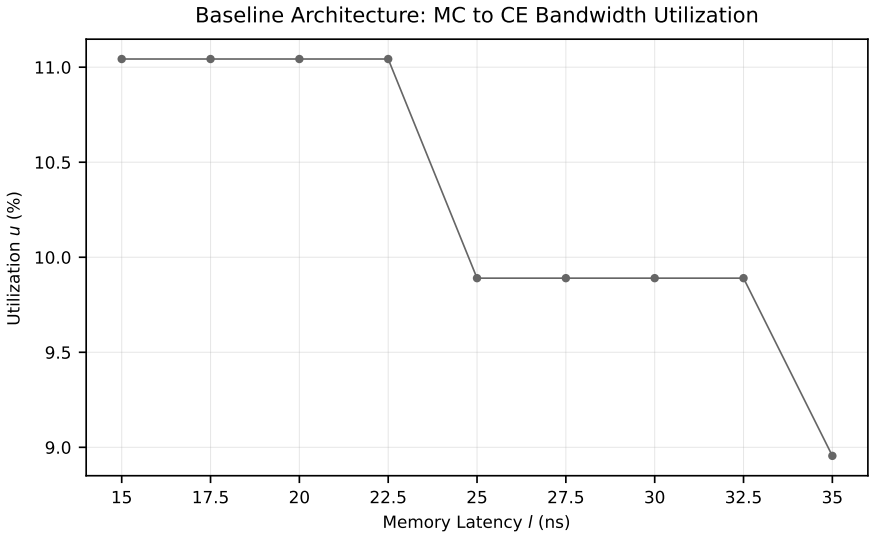


Figure 7.133: Baseline Architecture - MC to CE Bandwidth Utilization across Memory Latency l.

In contrast, the proposed architecture leverages pre-buffering via the SE to shield the core pipeline from memory latency penalties. With a maximum theoretical throughput of $\text{tp}_{\text{max,se-ce}} = 1600.0$ MB/s, the effective throughput ($\text{tp}_{\text{se-ce}}$) shown in Figure 7.134 remains highly stable across latency boundaries:

$$\text{tp}_{\text{se-ce}}(l) = 385.34 \text{ MB/s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.130}$$

$$\text{tp}_{\text{se-ce}}(l) = 382.74 \text{ MB/s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.131}$$

$$\text{tp}_{\text{se-ce}}(l) = 380.17 \text{ MB/s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.132}$$

As illustrated in Figure 7.135, this yields a consistent bandwidth utilization ($u_{\text{bw,se-ce}}$) between 24.08% and 23.76%, representing a $\approx 2.2\times$ to $2.7\times$ improvement over the baseline memory interface.

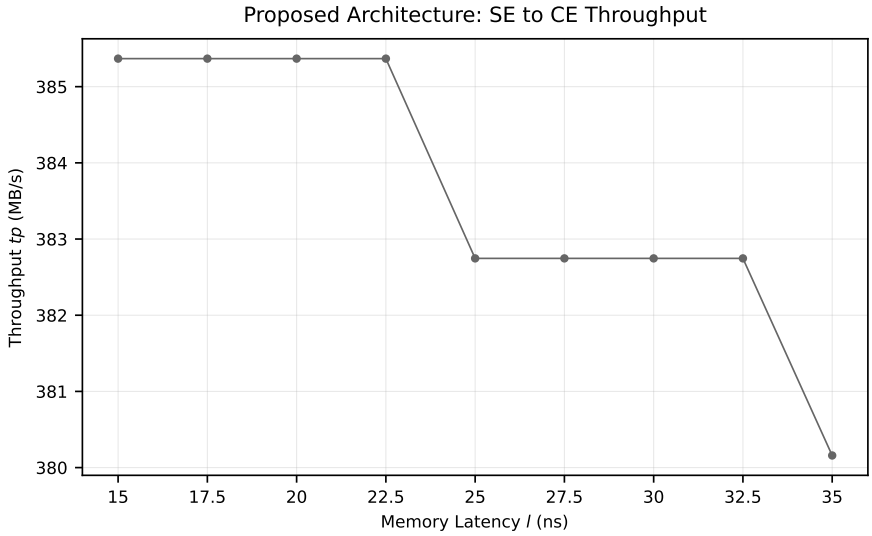


Figure 7.134: Proposed Architecture - SE to CE Throughput across Memory Latency l.

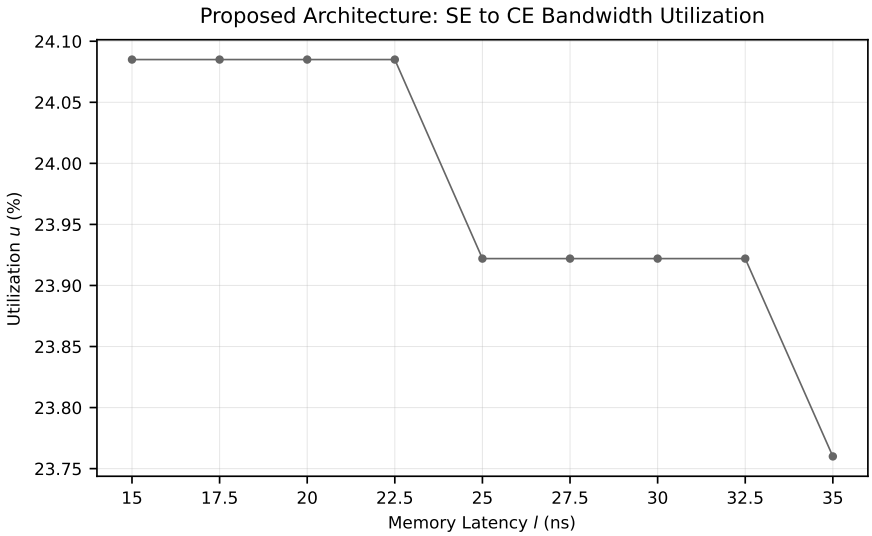


Figure 7.135: Proposed Architecture - SE to CE Bandwidth Utilization across Memory Latency l.

At the processing engine interface, where maximum theoretical compute throughput is bounded by $\text{tp}_{\text{max,ce}} = 800.0$ MB/s, severe memory stalls under unbuffered accesses degrade the baseline CE throughput ($\text{tp}_{\text{ce,bsl}}$) from 4.91 MB/s down to 3.98 MB/s (Figure 7.136). This causes baseline CE bandwidth utilization ($u_{\text{bw,ce,bsl}}$) to plummet to 0.50% (Figure 7.137) and compute utilization ($u_{\text{ce,cmp,bsl}}$) to drop from 6.14% to 4.98% (Figure 7.138).

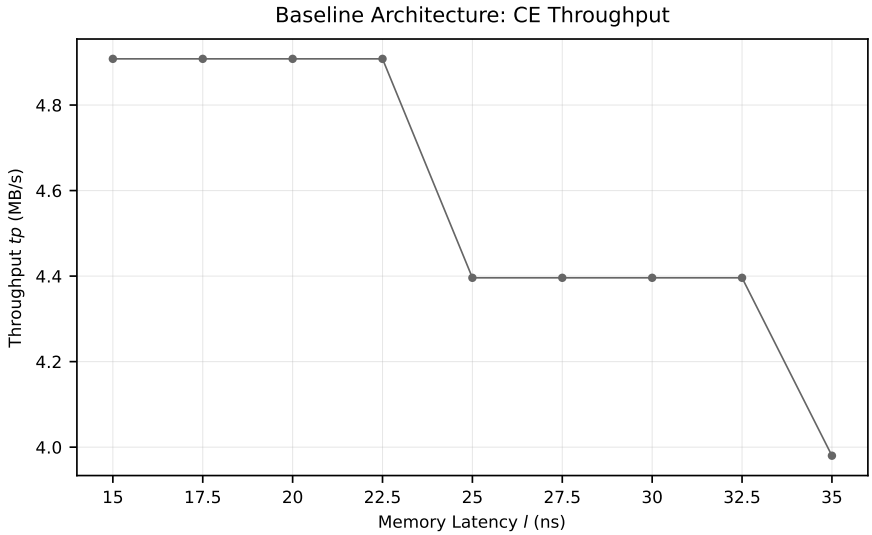


Figure 7.136: Baseline Architecture - CE Throughput across Memory Latency l.

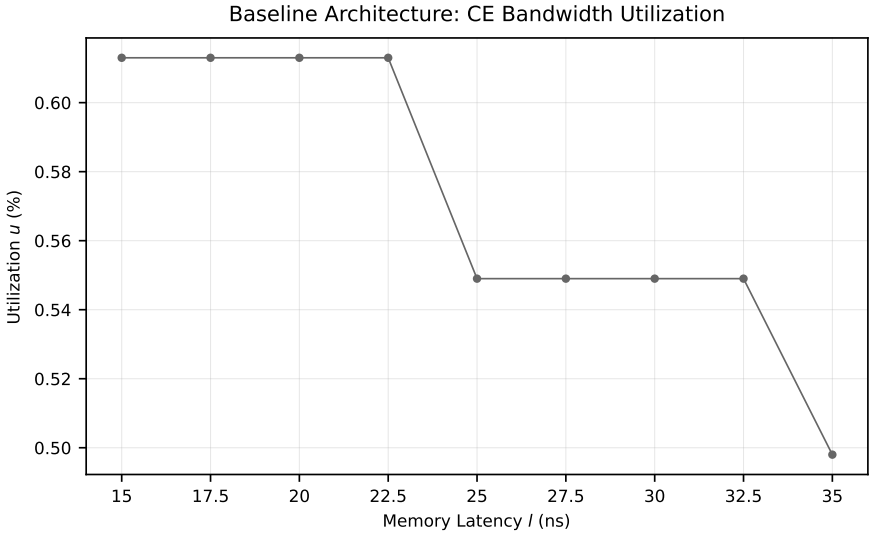


Figure 7.137: Baseline Architecture - CE Bandwidth Utilization across Memory Latency l.

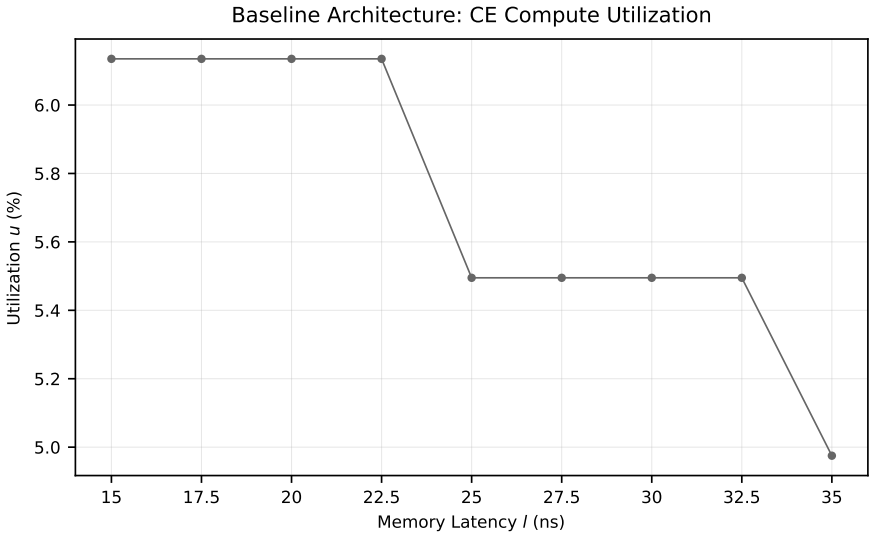


Figure 7.138: Baseline Architecture - CE Compute Utilization across Memory Latency l.

The proposed architecture reduces pipeline starvation, maintaining a high CE throughput ($\text{tp}_{\text{ce,prp}}$) between 21.59 MB/s and 21.31 MB/s, as seen in Figure 7.139. Consequently, proposed CE bandwidth utilization ($u_{\text{bw,ce,prp}}$) stays near 2.68% (Figure 7.140), while compute utilization ($u_{\text{ce,cmp}}$) is sustained at $\approx 26.6\% - 27.0\%$ (Figure 7.141), demonstrating reduced latency sensitivity.

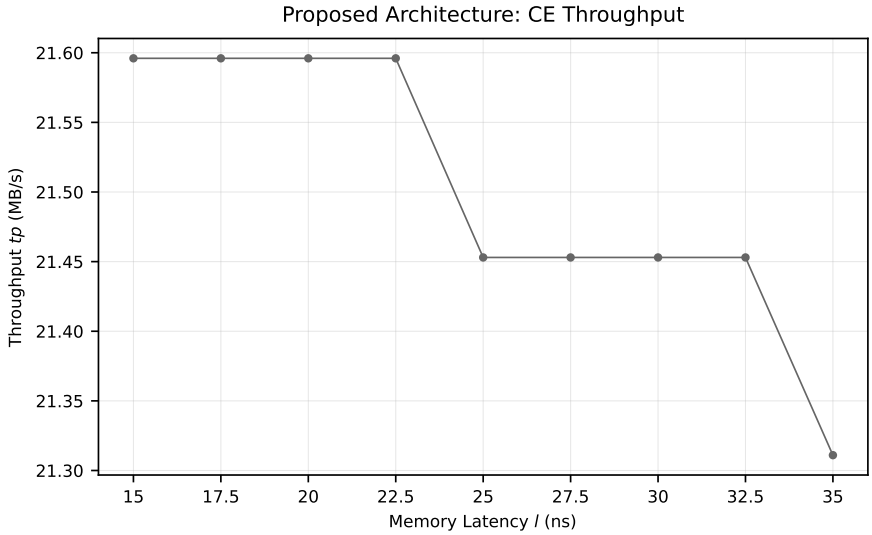


Figure 7.139: Proposed Architecture - CE Throughput across Memory Latency l.

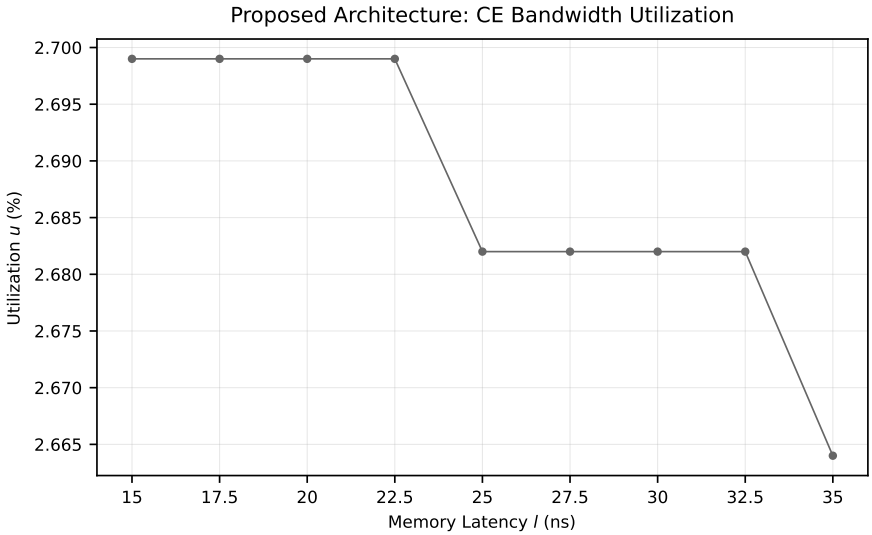


Figure 7.140: Proposed Architecture - CE Bandwidth Utilization across Memory Latency l.

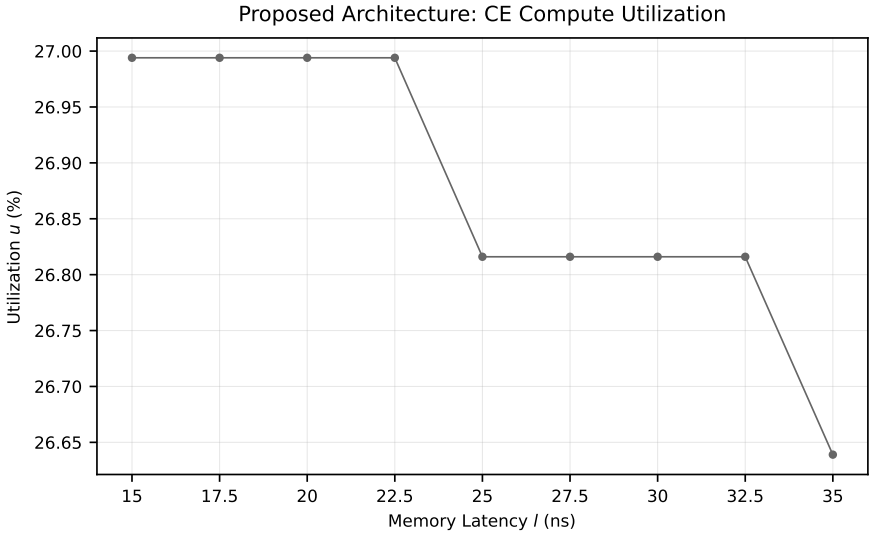


Figure 7.141: Proposed Architecture - CE Compute Utilization across Memory Latency l.

7.4 3D Convolution

7.4.1 Problem Size Sweep

To evaluate the execution performance of the 3D Convolution benchmark, simulations were conducted across a range of isotropic volumetric problem sizes. As described in Section 6.4.4, the swept parameter N represents the edge length of the 3D spatial tensor ($I_R \times I_C \times I_D = N \times N \times N$), which was varied from $N = 10$ to $N = 25$ voxel elements per dimension.

7.4.1.1 Simulation Time and Speedup Analysis

In the baseline architecture, memory transactions are issued directly by the MC-CE without caching or streaming buffers. Computing the $3 \times 3 \times 3$ filtering operation requires repeated single-read memory operations for every voxel across the 3D volume. Because volumetric memory access pattern striding across horizontal, vertical, and depth dimensions scales cubically with N^3, memory bus transfer latency heavily dominates overall execution time. As shown in Figure 7.142, the baseline simulation time ($t_{\text{sim,bsl}}$) increases smoothly along a cubic trajectory with problem size N.

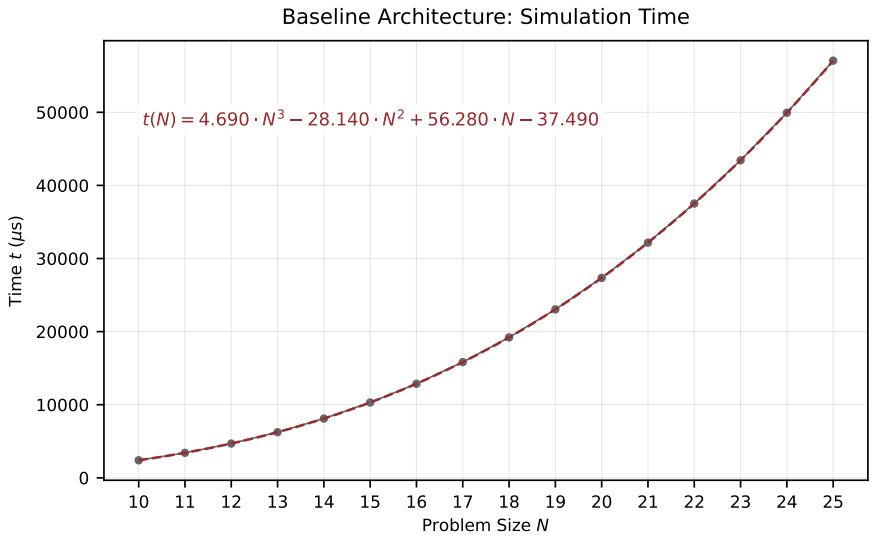


Figure 7.142: Baseline Architecture - Simulation Time across Problem Size N.

Conversely, the proposed architecture offloads memory management to the SE, decoupling address generation from computation. The IS1 SB operates in CB mode, caching the 27 elements of the $3 \times 3 \times 3$ filter kernel once to enable local coefficient reuse. The IS0 SB utilizes a 6-level nested strided loop hierarchy to fetch the volumetric image data.

While the proposed architecture maintains substantially lower overall execution latency compared to the baseline, the simulation time curve ($t_{\text{sim,prp}}$) exhibits prominent latency spikes at specific problem sizes ($N = 16$ and $N = 22 - 23$), as depicted in Figure 7.143.

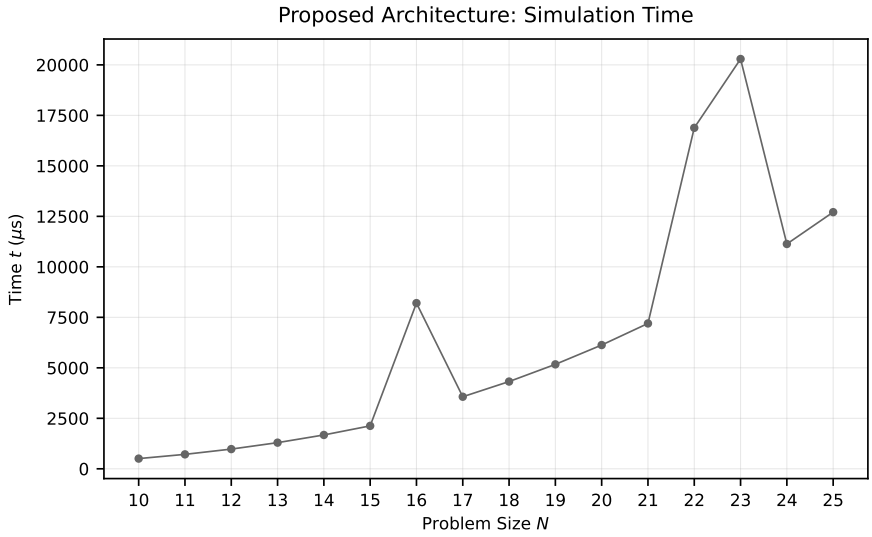


Figure 7.143: Proposed Architecture - Simulation Time across Problem Size N.

These localized performance penalties stem from conflict-driven cache thrashing within the SC during 3D memory traversal. Each line fill loads contiguous burst elements ($SC_{WDT} = 8$) into the local SC. However, 3D tensor traversal requires strided indexing across row (N) and plane (N^2) boundaries. For specific dimensions of N, the memory address strides cause distinct spatial data points with different address tags to repeatedly map into overlapping cache sets within the local capacity ($C_{SC} = 4.096$ KB).

This set aliasing leads to early cache line evictions before the pre-fetched horizontal data can be reused across subsequent vertical or depth loop iterations. Upon returning to access those evicted elements, the SE experiences repetitive cache misses, forcing additional memory bus line-fill transactions and creating temporary spikes in simulation time.

As a direct result of these cache thrashing events, the speedup profile $s(N)$ shown in Figure 7.144 experiences noticeable sharp drops at $N = 16$ (falling to $\approx 1.57\times$) and $N = 22 - 23$ (dropping to $\approx 2.22\times - 2.14\times$). Outside these conflict-heavy dimensions, the pre-buffering mechanism effectively insulates the execution pipeline, sustaining an acceleration factor near $s \approx 4.5\times - 4.85\times$.

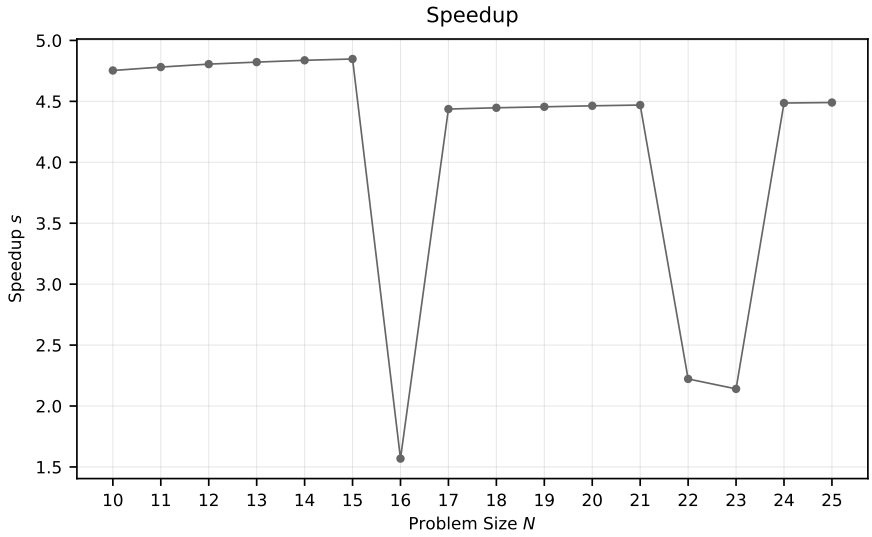


Figure 7.144: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Problem Size N.

7.4.1.2 Memory Access Analysis

To further analyze how memory traffic influences overall performance, we evaluate the MAF and total memory communication time across isotropic volume sizes N.

For the baseline architecture, every voxel processing operation directly issues scalar memory requests to external memory without caching mechanisms. Consequently, the baseline MAF remains strictly constant at $\mathrm{MAF}_{\mathrm{bsl}}(N) = 55.000$ across all problem sizes $N \in [10, 25]$. Driven by this unmitigated scalar transfer overhead across $3 \times 3 \times 3$ filtering windows, the baseline memory communication time ($t_{\mathrm{mem,bsl}}$) scales cubically with volume growth:

$$t_{\mathrm{mem,bsl}}(N) = 2.200 \cdot N^3 - 13.200 \cdot N^2 + 26.400 \cdot N - 17.600 \quad [\mu\mathrm{s}] \tag{7.133}$$

As shown in Figure 7.145, $t_{\mathrm{mem,bsl}}$ climbs from $1135.20\,\mu$s at $N = 10$ to $26756.40\,\mu$s at $N = 25$.

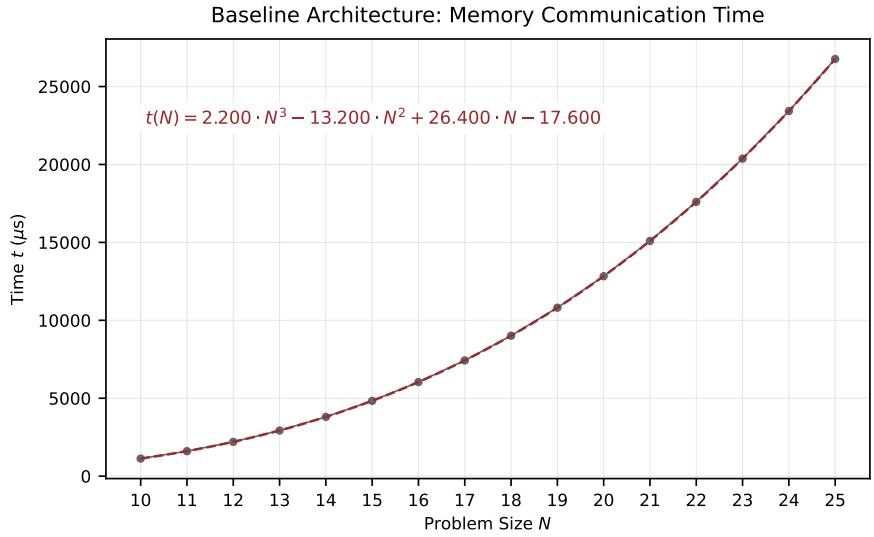


Figure 7.145: Baseline Architecture - Memory Communication Time across Problem Size N.

In contrast, the proposed architecture substantially reduces memory overhead by pre-loading kernel data via the SE and caching $3 \times 3 \times 3$ kernel weights in CB mode. However, as illustrated in Figure 7.146, the proposed MAF (MAF_{prp}) exhibits sharp increases at specific problem dimensions — reaching $\text{MAF}_{\text{prp}} \approx 54.8$ at $N = 16$, and peaking between 32.1 and 34.2 for $N \in [22, 23]$. Outside these conflict points, MAF_{prp} remains exceptionally low (ranging between 2.6 and 4.8).

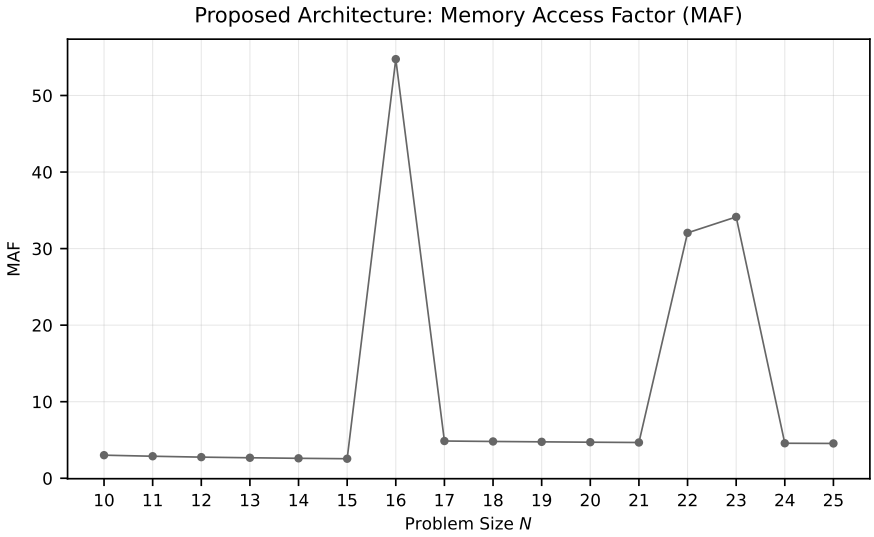


Figure 7.146: Proposed Architecture - Memory Access Factor (MAF) across Problem Size N.

These MAF surges directly cause corresponding spikes in memory communication time ($t_{\text{mem,prp}}$), as depicted in Figure 7.147. At $N = 16$, $t_{\text{mem,prp}}$ increases to $3386\,\mu$s, and reaches a maximum of

7108 μs at $N = 23$. This behavior confirms that the execution time slowdowns identified in Section 7.4.1.1 are directly driven by conflict-induced SC cache thrashing and tag replacement under specific volumetric stride configurations.

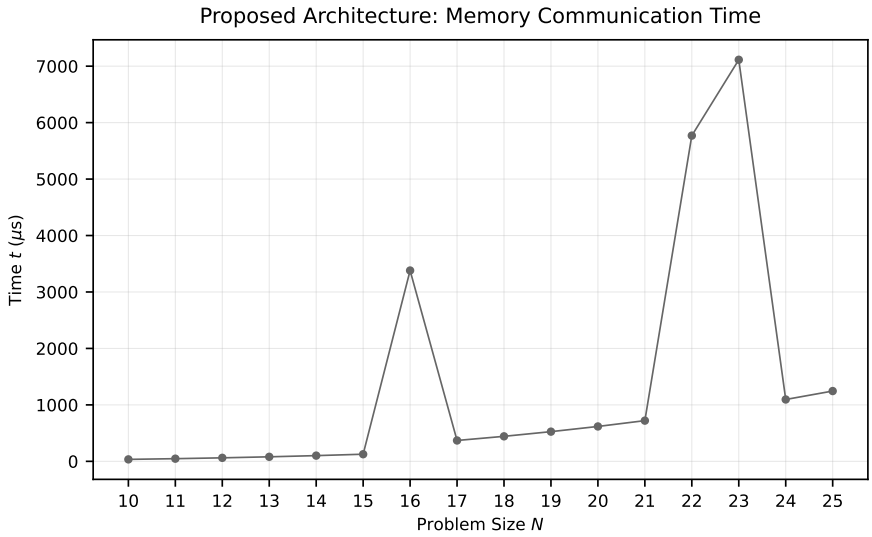


Figure 7.147: Proposed Architecture - Memory Communication Time across Problem Size N.

7.4.1.3 Throughput and Bandwidth Utilization Analysis

To evaluate complete system performance under 3D convolution workload scaling, we analyze the CE throughput, interface bandwidth utilization, and execution efficiency across problem dimensions $N \in [10, 25]$.

In the baseline architecture, the continuous scalar handshakes required by unbuffered memory transfers create massive memory stalling. On the MC-to-CE bus—which features a maximum theoretical throughput $\text{tp}_{\text{max,mc-ce}} = 800.000\,\text{MB/s}$—the baseline memory transfer rate remains flat at $\text{tp}_{\text{mc-ce}} = 92.111\,\text{MB/s}$ across all problem sizes N, resulting in a constant bandwidth utilization of $u_{\text{bw,mc-ce}} = 11.514\%$. Consequently, the effective data operational rate ($\text{tp}_{\text{ce,bsl}}$) of the CE remains severely constrained at 1.706 MB/s. Measured against the maximum theoretical channel bandwidth ($\text{tp}_{\text{max,ce,bsl}} = 800.000\,\text{MB/s}$), the baseline CE bandwidth utilization ($u_{\text{bw,ce,bsl}}$) is limited to 0.213%. Furthermore, because arithmetic units sit idle during scalar memory fetches, the baseline compute utilization ($u_{\text{ce,cmp,bsl}}$) stays constant at 5.970%.

In contrast, the proposed architecture decouples memory fetching from computation by streaming volumetric operands from the SE to the CE over a high-performance interconnect with a maximum theoretical capacity $\text{tp}_{\text{max,se-ce}} = 1600.000\,\text{MB/s}$. Outside of memory conflict bounds, the SE-to-CE throughput ($\text{tp}_{\text{se-ce}}$) reaches peak values between 437.21 MB/s and 446.54 MB/s for $N \in [10, 15]$, maintaining steady-state transfers near 410 MB/s to 413 MB/s for non-conflicting

sizes. As shown in Figure 7.148, these transfers map to the bandwidth utilization ($u_{\text{bw,se-ce}}$) depicted in Figure 7.149. However, cache thrashing at specific volumetric strides causes performance degradation: at $N = 16$, $\text{tp}_{\text{se-ce}}$ drops sharply to 144.62 MB/s ($u_{\text{bw,se-ce}} = 9.04\%$), while at $N \in [22, 23]$, throughput falls to 197.10 – 204.81 MB/s ($u_{\text{bw,se-ce}} \approx 12.32\% - 12.80\%$).

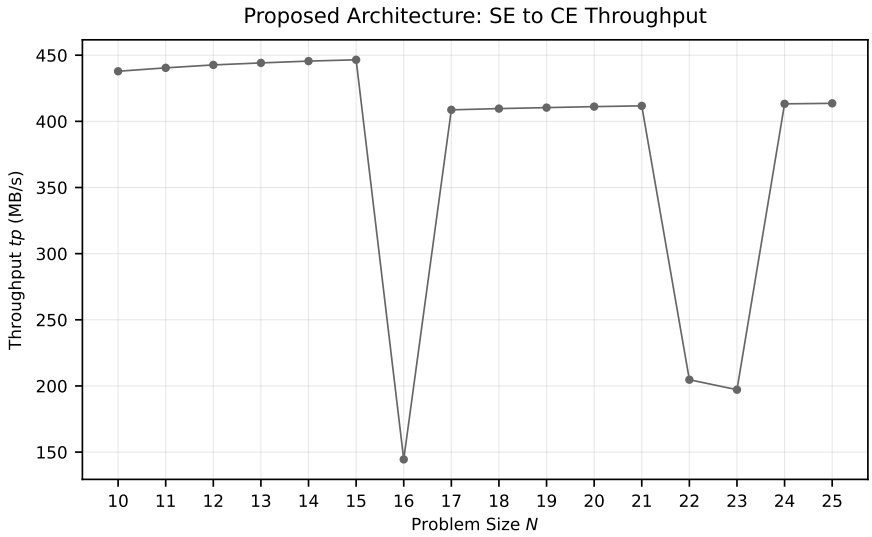


Figure 7.148: Proposed Architecture - SE to CE Throughput across Problem Size N.

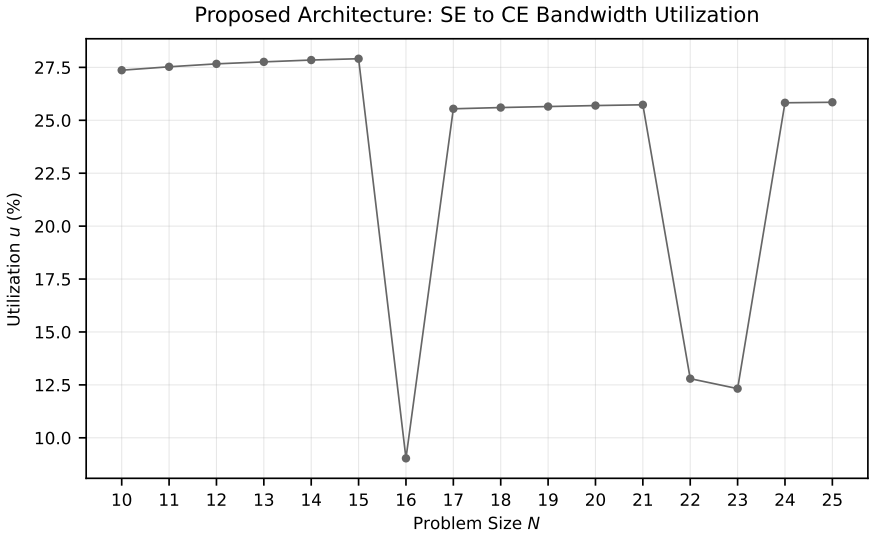


Figure 7.149: Proposed Architecture - SE to CE Bandwidth Utilization across Problem Size N.

This streaming throughput directly dictates overall processing engine performance. Under ideal volumetric dimensions ($N \in [10, 15]$), the proposed CE effective throughput ($\text{tp}_{\text{ce,prp}}$) reaches up to 8.53 MB/s — a 5.00× speedup over the baseline's 1.706 MB/s — and maintains a steady operational throughput of ≈ 7.65 MB/s across non-conflicting grid sizes ($N \in [17, 21]$ and $N \in$

[24,25]). As shown in Figure 7.150, these throughput variations directly correlate with the CE channel bandwidth utilization ($u_{\text{bw,ce,prp}}$) in Figure 7.151, which peaks at 1.066% before settling near 0.957% outside conflict regions. At the conflict points $N = 16$ and $N \in [22,23]$, increased memory stall cycles lower $\text{tp}_{\text{ce,prp}}$ to 2.68 MB/s and 3.65 – 3.80 MB/s, respectively.

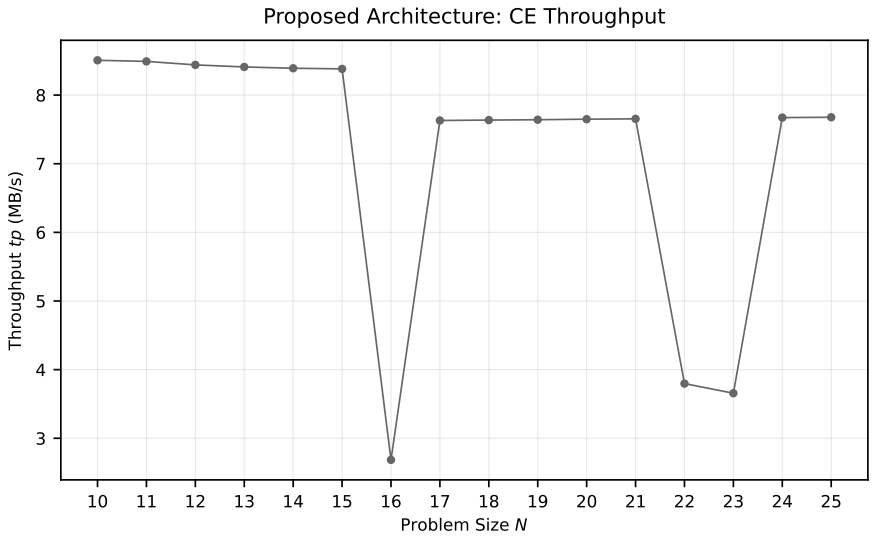


Figure 7.150: Proposed Architecture - CE Throughput across Problem Size N.

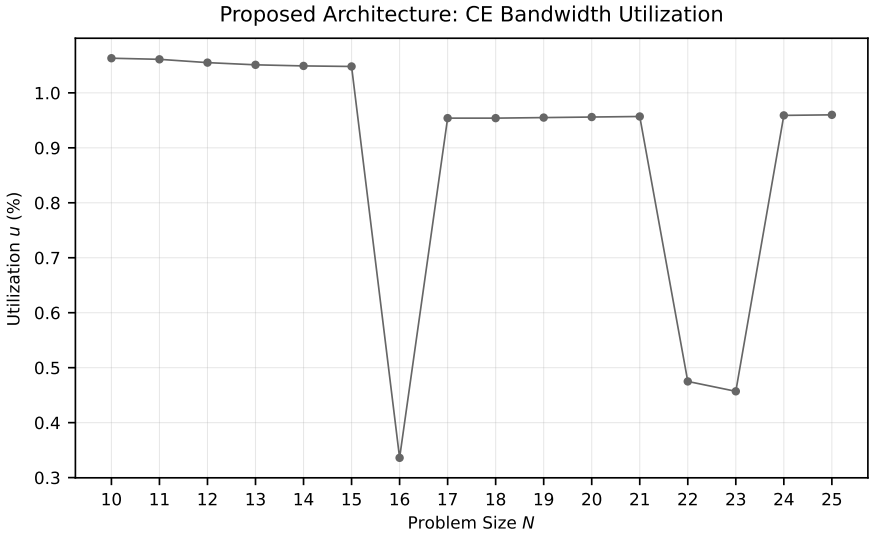


Figure 7.151: Proposed Architecture - CE Bandwidth Utilization across Problem Size N.

Finally, Figure 7.152 demonstrates the impact of stream-buffered data delivery on internal arithmetic efficiency. Outside cache thrashing regions, the proposed CE compute utilization ($u_{\text{ce,cmp,prp}}$) reaches 29.76% for small problem dimensions and maintains a steady state of 26.78% across larger non-conflicting sizes. Compared to the baseline's 5.970% compute utilization, the

proposed architecture achieves a 4.49× boost in overall execution efficiency. Even under severe conflict conditions at $N = 16$ ($u_{\text{ce,cmp,prp}} = 9.39\%$) and $N = 23$ ($u_{\text{ce,cmp,prp}} = 12.78\%$), the proposed design maintains higher compute unit utilization than the baseline.

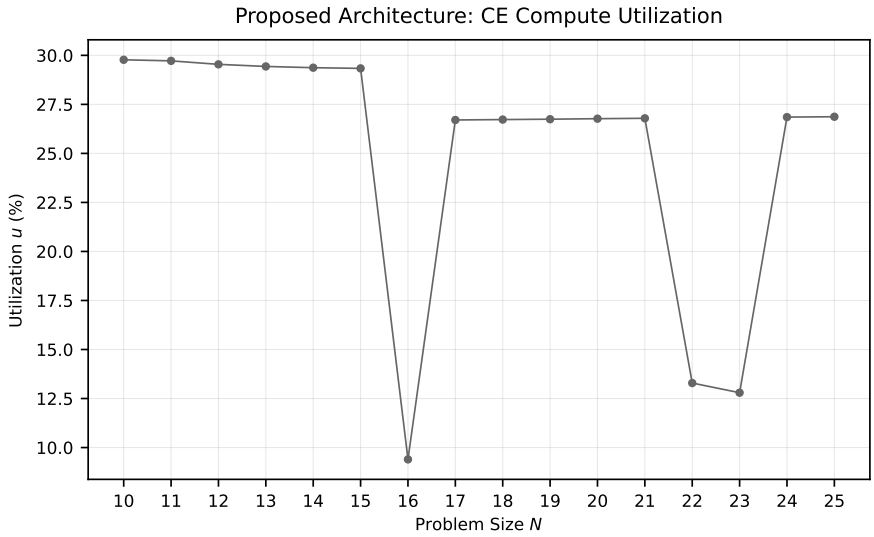


Figure 7.152: Proposed Architecture - CE Compute Utilization across Problem Size N.

7.4.2 Operating Frequency Sweep

To evaluate the scalability and performance sensitivity of both architectures to clock frequency variations, an operating frequency sweep is conducted across $f \in [100, 250]$ MHz. For all frequency sweep evaluations, the problem size is fixed at $N = 19$. This dataset size was specifically selected because, as established in Section 7.4.1.3, $N = 19$ represents a representative non-conflicting operating regime where CE throughput and execution efficiency have stabilized, thereby eliminating dataset-scaling transients from the frequency response analysis.

7.4.2.1 Simulation Time and Speedup Analysis

The relationship between operating frequency and execution time highlights fundamental differences in pipeline clocking efficiency between the two system designs.

In the baseline architecture, execution time decreases non-linearly with increasing clock frequency. As illustrated in Figure 7.153, the simulation time $t_{\text{sim,bsl}}$ follows an inverse relationship with operating frequency f:

$$t_{\text{sim,bsl}}(f) = \frac{1.95 \cdot 10^6}{f} + 3085.534 \quad [\mu\text{s}] \tag{7.134}$$

As frequency increases from 100 MHz to 250 MHz, execution time drops from 23048.800 μs down to 11370.000 μs. However, due to fixed memory access handshakes and unbuffered bus latency

penalties that remain unscaled by clock frequency, the baseline architecture exhibits diminishing returns at higher operating frequencies.

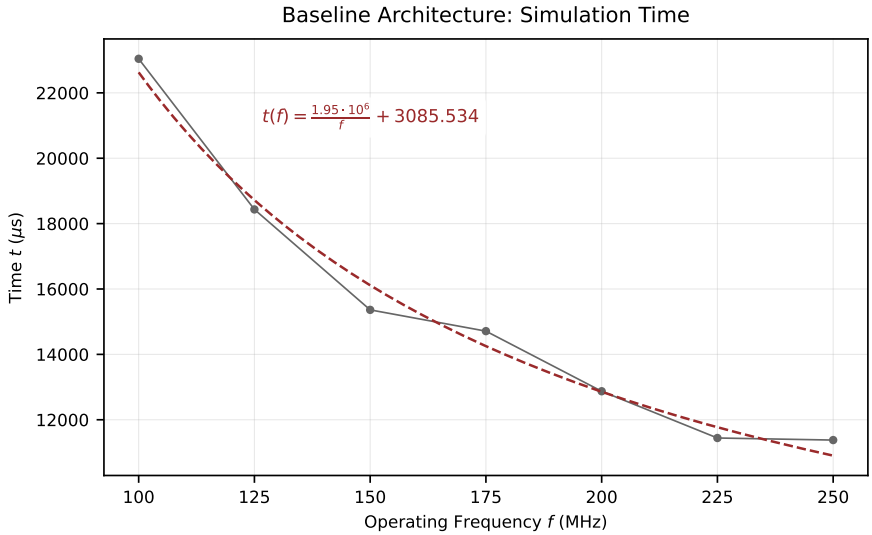


Figure 7.153: Baseline Architecture - Simulation Time across Operating Frequency f.

Conversely, the proposed architecture also exhibits an inverse relationship with frequency, but with drastically reduced overheads. As shown in Figure 7.154, the simulation time $t_{\text{sim,prp}}$ scales as:

$$t_{\text{sim,prp}}(f) = \frac{5.13 \cdot 10^5}{f} + 33.377 \quad [\mu\text{s}] \tag{7.135}$$

As f increases from 100 MHz to 250 MHz, execution time decreases from 5058.400 μs to 2092.300 μs. The offset constant is reduced to just 33.377 μs, demonstrating that memory stalling overheads are substantially eliminated.

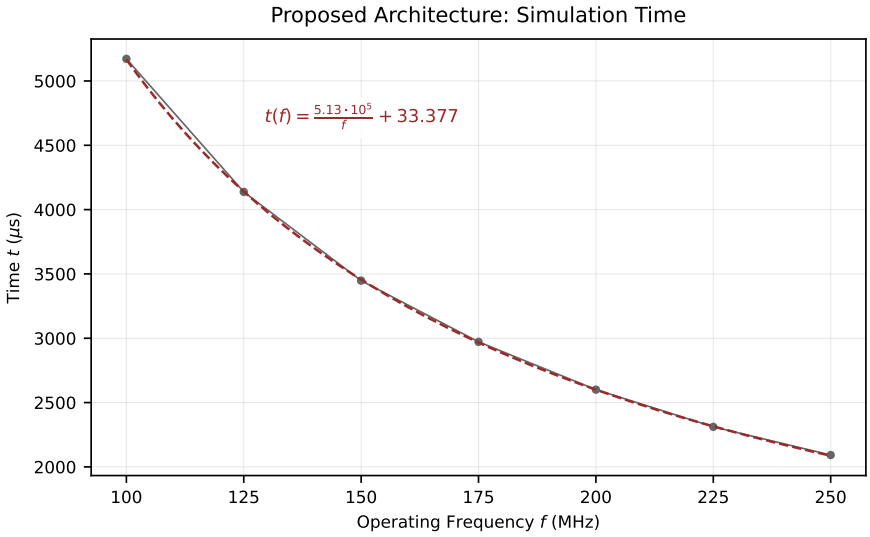


Figure 7.154: Proposed Architecture - Simulation Time across Operating Frequency f.

The resulting speedup $s(f)$ achieved by the proposed architecture over the baseline architecture across operating frequencies is depicted in Figure 7.155. Overall, the speedup grows from 4.56× at 100 MHz up to 5.43× at 250 MHz. The step-like jumps in the speedup curve occur because external memory latency is fixed in physical time, forcing the memory latency to scale in discrete clock cycles. As frequency increases, the cycle count remains constant until a frequency threshold is crossed, at which point the memory response time requires an additional clock cycle. Because these discrete latency penalties degrade the baseline architecture's efficiency more severely than the proposed design, each cycle threshold crossing produces a step jump in the speedup curve.

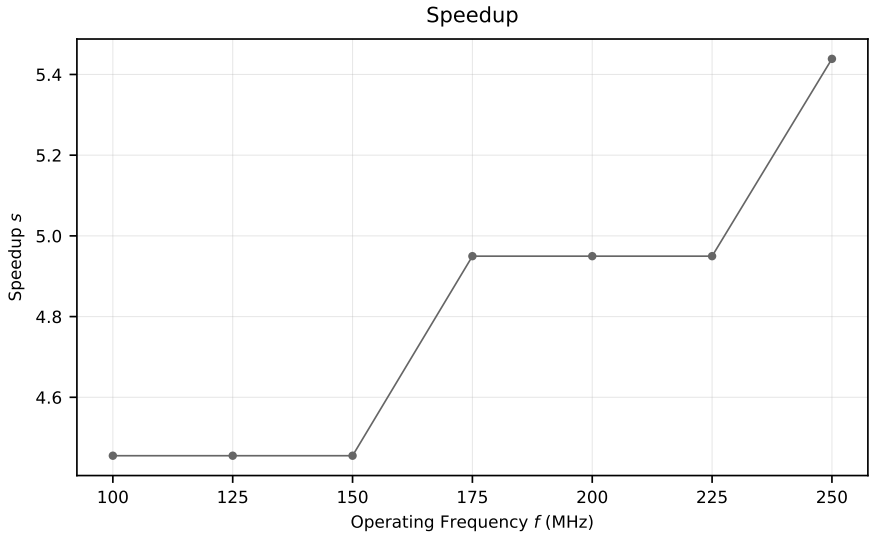


Figure 7.155: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Operating Frequency f.

7.4.2.2 Memory Access Analysis

To examine external memory efficiency during the frequency sweep at $N = 19$, we analyze the MAF and memory communication time across $f \in [100, 250]$ MHz.

In the baseline architecture, the system lacks data caching mechanisms, requiring unbuffered scalar fetches for every arithmetic operand. As a result, the baseline MAF (MAF_{bsl}) remains strictly constant at 55.000 across all operating frequencies. Due to these unbuffered transactions, the memory communication time ($t_{\text{mem,com,bsl}}$) scales non-monotonically with frequency, as illustrated in Figure 7.156. Starting at 10800.000 μs at 100 MHz, $t_{\text{mem,com,bsl}}$ drops to 6480.000 μs at 250 MHz, exhibiting intermittent upward latency spikes (such as 7714.286 μs at 175 MHz) caused by discrete cycle threshold crossings in external memory response times.

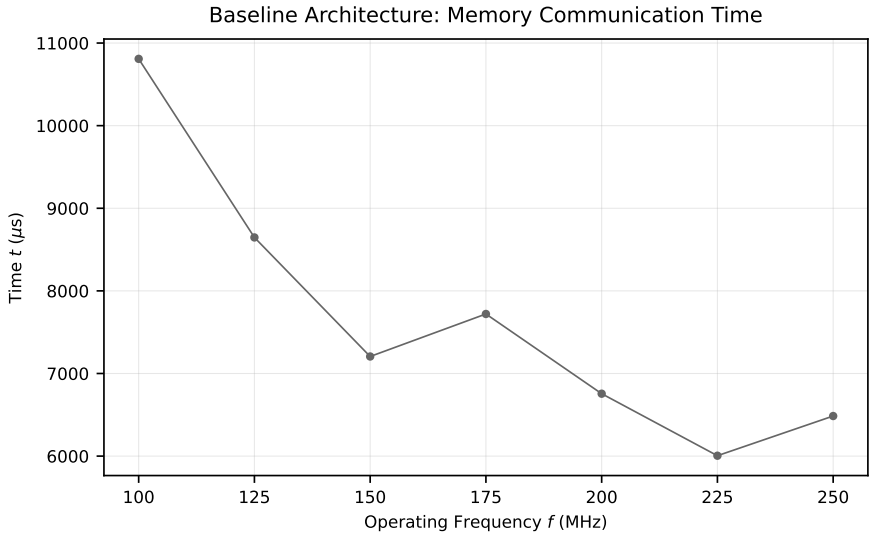


Figure 7.156: Baseline Architecture - Memory Communication Time across Operating Frequency f.

In contrast, the proposed architecture exploits local stream-buffering within the SE to optimize memory accesses. Because the algorithmic access pattern is invariant with respect to clock rate, the proposed MAF (MAF_{prp}) remains flat across all frequencies at a drastically reduced value of 4.758 — representing an 11.56× reduction in external memory accesses compared to the baseline's 55.000.

As depicted in Figure 7.157, the reduced memory traffic enables the proposed memory communication time ($t_{\text{mem,com,prp}}$) to scale smoothly according to an inverse frequency model:

$$t_{\text{mem,com,prp}}(f) = \frac{4.88 \cdot 10^4}{f} + 33.377 \quad [\mu\text{s}] \tag{7.136}$$

As f increases from 100 MHz to 250 MHz, $t_{\text{mem,com,prp}}$ decreases from 521.377 μs to 228.577 μs. Compared to the baseline architecture, this represents a 20.71× to 28.35× reduction in external memory communication time across the evaluated frequency spectrum, demonstrating the efficacy of stream-buffered data reuse in mitigating memory latency bottlenecks.

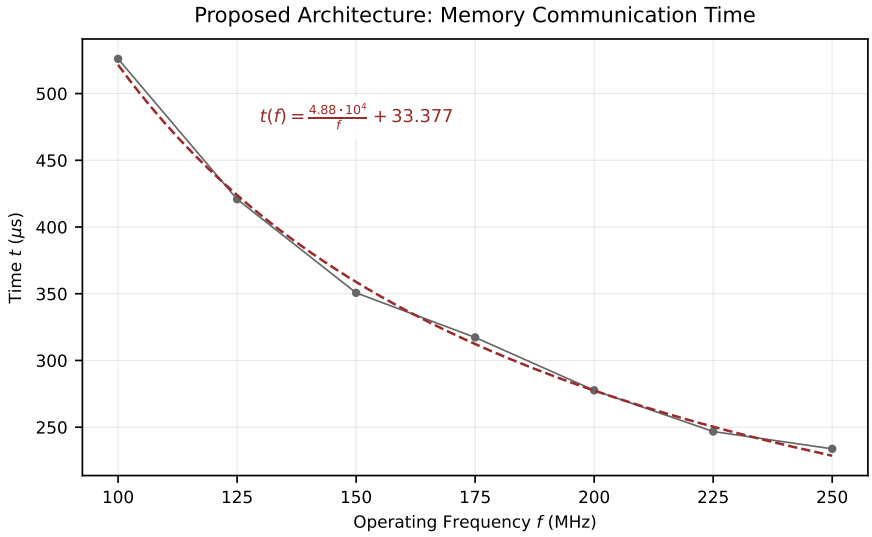


Figure 7.157: Proposed Architecture - Memory Communication Time across Operating Frequency f.

7.4.2.3 Throughput and Bandwidth Utilization Analysis

In the baseline architecture, the maximum theoretical throughput for the MC-to-CE bus ($\mathrm{tp}_{\mathrm{max,mc\text{-}ce,bsl}}$) scales linearly with frequency as illustrated in Figure 7.158:

$$\mathrm{tp}_{\mathrm{max,mc\text{-}ce,bsl}}(f) \approx 8.000 \cdot f \quad [\mathrm{MB/s}] \tag{7.137}$$

increasing from 800.000 MB/s at 100 MHz to 2000.000 MB/s at 250 MHz.

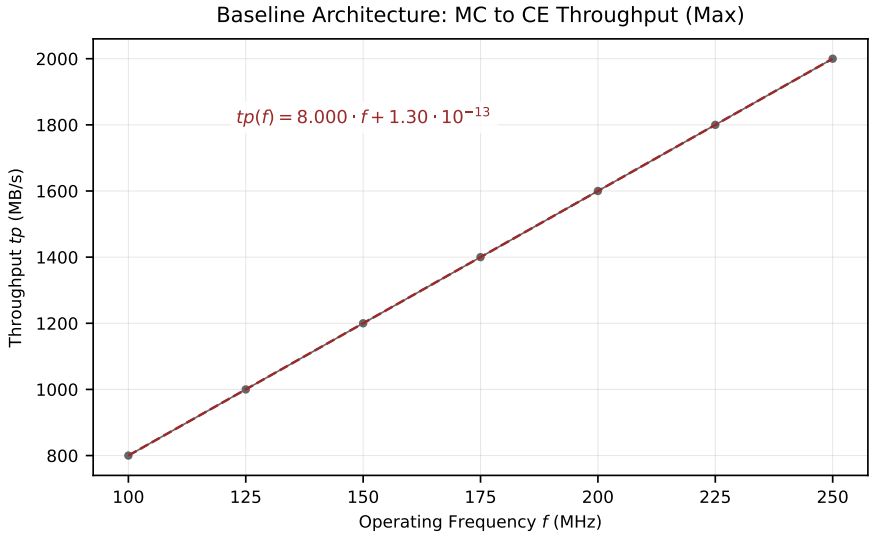


Figure 7.158: Baseline Architecture - MC to CE Maximum Theoretical Throughput across Operating Frequency f.

However, severe memory stall cycles prevent the baseline system from utilizing this available bandwidth. As depicted in Figure 7.159, the achieved MC-to-CE throughput ($\text{tp}_{\text{mc-ce,bsl}}$) scales at a modest rate of:

$$\text{tp}_{\text{mc-ce,bsl}}(f) = 0.644 \cdot f + 33.985 \quad [\text{MB/s}] \tag{7.138}$$

growing from 92.111 MB/s at 100 MHz to 186.826 MB/s at 250 MHz. As shown in Figure 7.160, this results in a low bus bandwidth utilization ($u_{\text{bw,mc-ce,bsl}}$) bounded between 9.34% and 11.51%.

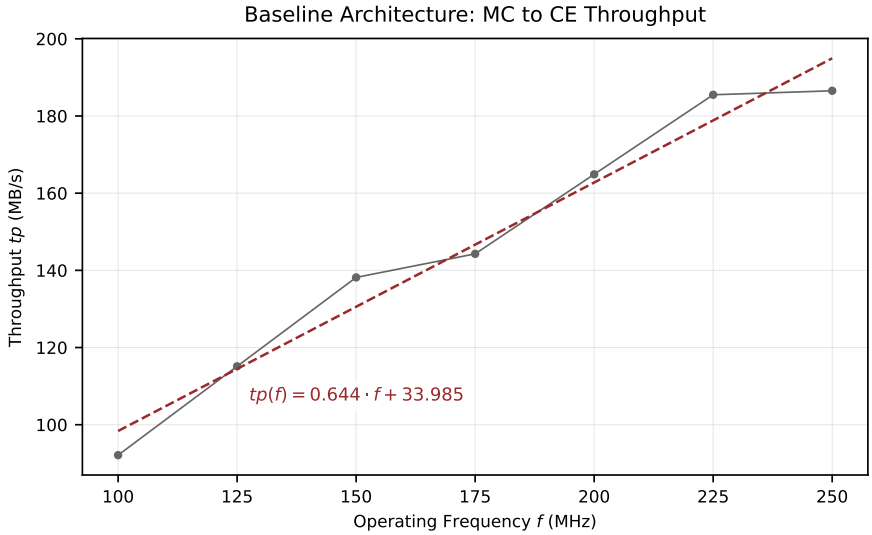


Figure 7.159: Baseline Architecture - MC to CE Throughput across Operating Frequency f.

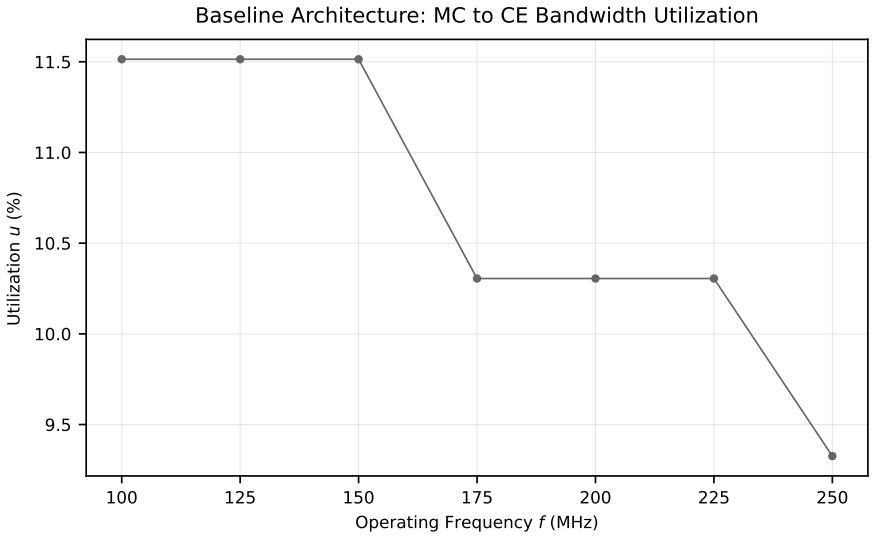


Figure 7.160: Baseline Architecture - MC to CE Bandwidth Utilization across Operating Frequency f.

The baseline CE maximum theoretical throughput ($\mathrm{tp}_{\mathrm{max,ce,bsl}}$) is depicted in Figure 7.161:

$$\mathrm{tp}_{\mathrm{max,ce,bsl}}(f) \approx 8.000 \cdot f \quad [\mathrm{MB/s}] \tag{7.139}$$

However, operational throughput ($\mathrm{tp}_{\mathrm{ce,bsl}}$) is severely constrained due to scalar memory fetch overheads. As shown in Figure 7.162, it scales as:

$$\mathrm{tp}_{\mathrm{ce,bsl}}(f) = 0.012 \cdot f + 0.630 \quad [\mathrm{MB/s}] \tag{7.140}$$

increasing from 1.706 MB/s at 100 MHz to 3.458 MB/s at 250 MHz.

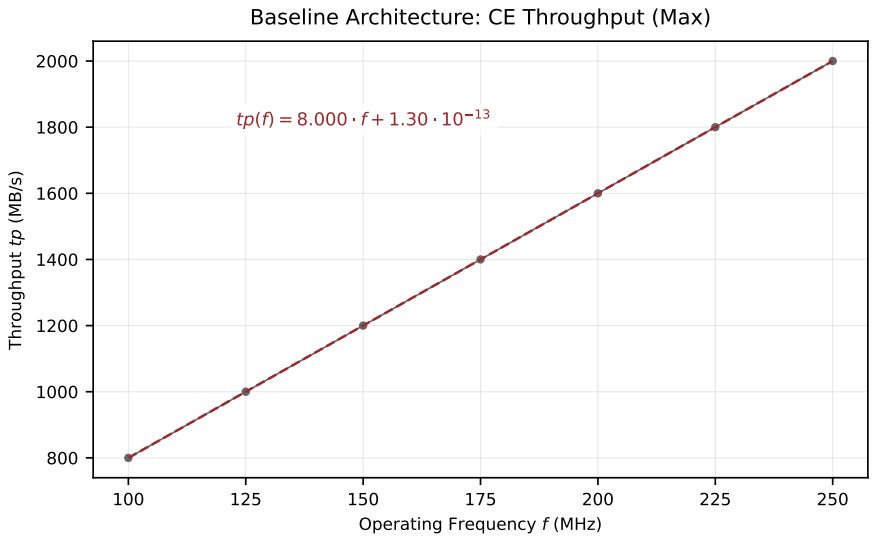


Figure 7.161: Baseline Architecture - CE Maximum Theoretical Throughput across Operating Frequency f.

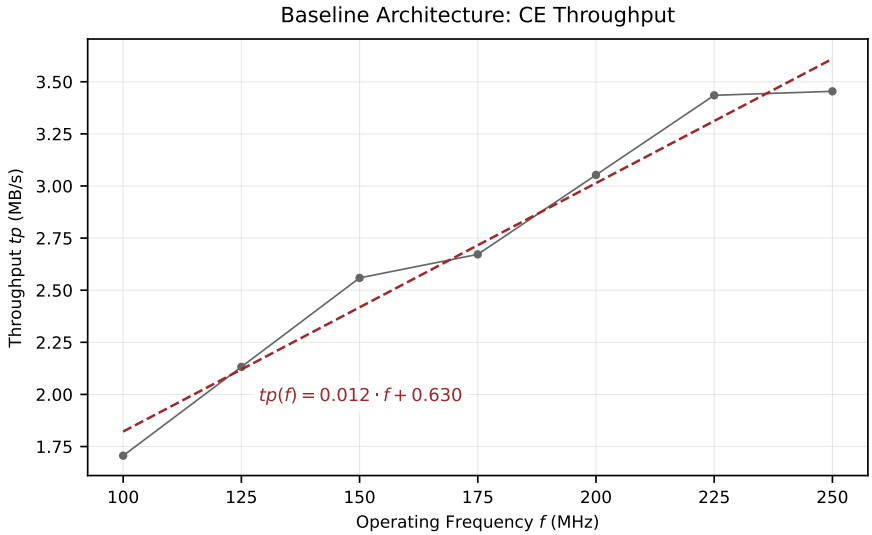


Figure 7.162: Baseline Architecture - CE Throughput across Operating Frequency f.

As illustrated in Figure 7.163, the baseline CE bandwidth utilization ($u_{\text{bw,ce,bsl}}$) drops from 0.213% down to 0.173% as clock frequency increases. Furthermore, because arithmetic units sit idle while awaiting scalar operands, Figure 7.164 demonstrates that the baseline compute utilization ($u_{\text{ce,cmp,bsl}}$) remains limited to 4.84% – 5.97% across all frequencies.

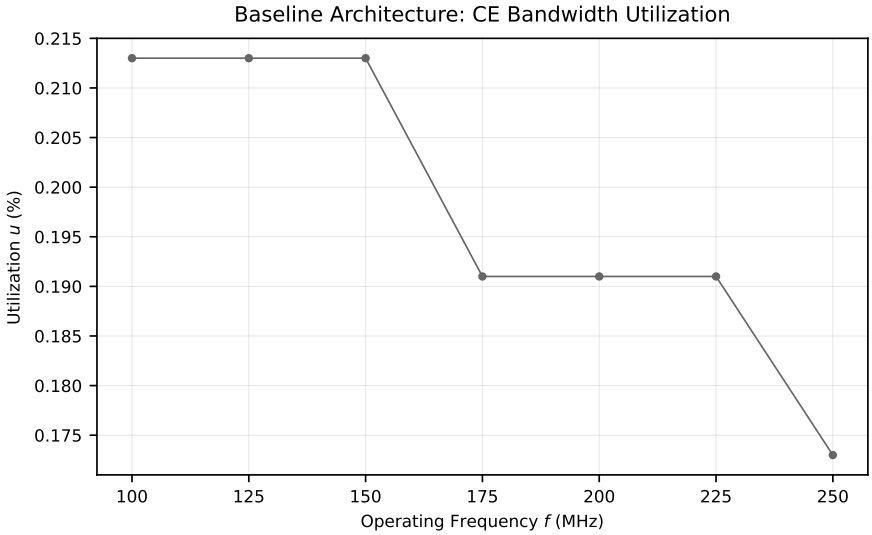


Figure 7.163: Baseline Architecture - CE Bandwidth Utilization across Operating Frequency f.

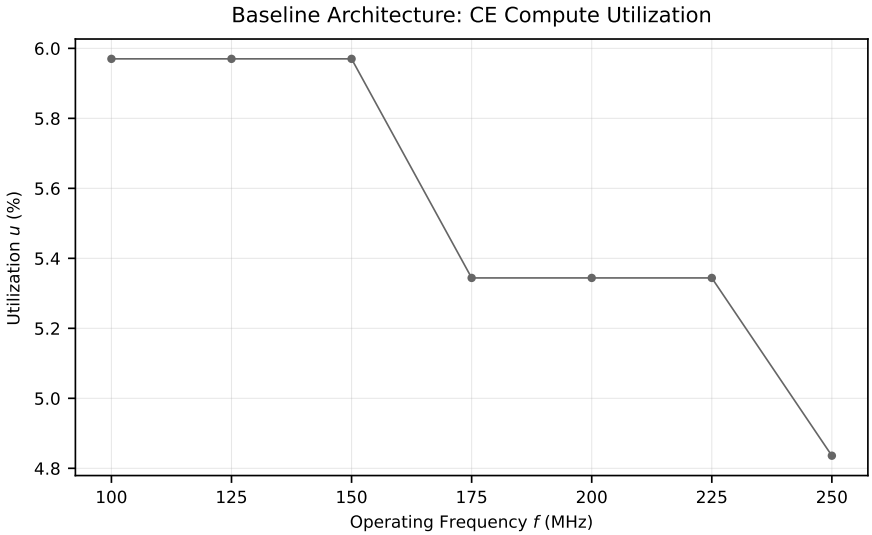


Figure 7.164: Baseline Architecture - CE Compute Utilization across Operating Frequency f.

In contrast, the proposed architecture decouples memory accesses from arithmetic execution via high-bandwidth stream buffering. As shown in Figure 7.165, the maximum theoretical throughput on the SE-to-CE interconnect ($\mathrm{tp}_{\mathrm{max,se\text{-}ce}}$) scales linearly as:

$$\mathrm{tp}_{\mathrm{max,se\text{-}ce}}(f) \approx 16.000 \cdot f \quad [\mathrm{MB/s}] \tag{7.141}$$

reaching 4000.000 MB/s at 250 MHz.

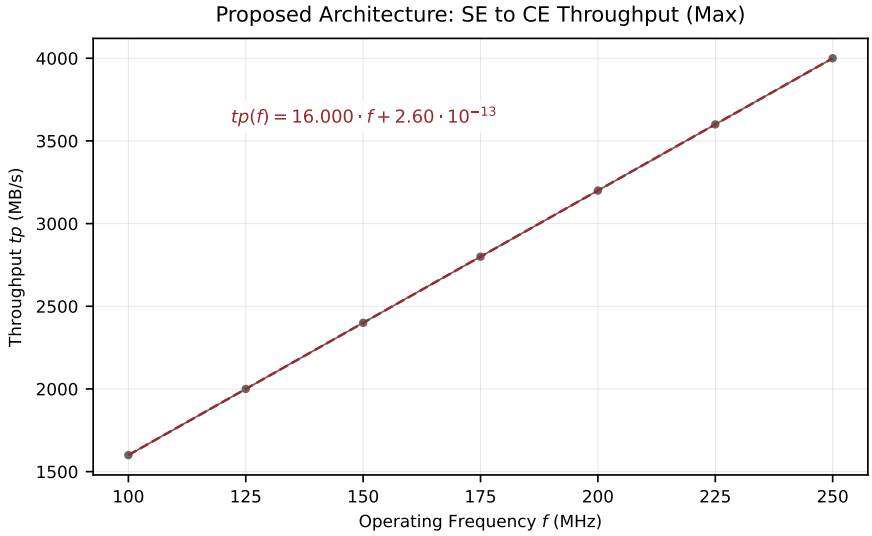


Figure 7.165: Proposed Architecture - SE to CE Maximum Theoretical Throughput across Operating Frequency f.

As shown in Figure 7.166, the proposed SE-to-CE transfer rate ($\text{tp}_{\text{se-ce}}$) scales efficiently according to:

$$\text{tp}_{\text{se-ce}}(f) = 4.033 \cdot f + 8.733 \quad [\text{MB/s}] \tag{7.142}$$

increasing from 412.033 MB/s at 100 MHz to 1017.033 MB/s at 250 MHz. Figure 7.167 shows that this maintains a robust local interconnect bandwidth utilization ($u_{\text{bw,se-ce}}$) near 25.43% – 25.75% across the entire frequency range.

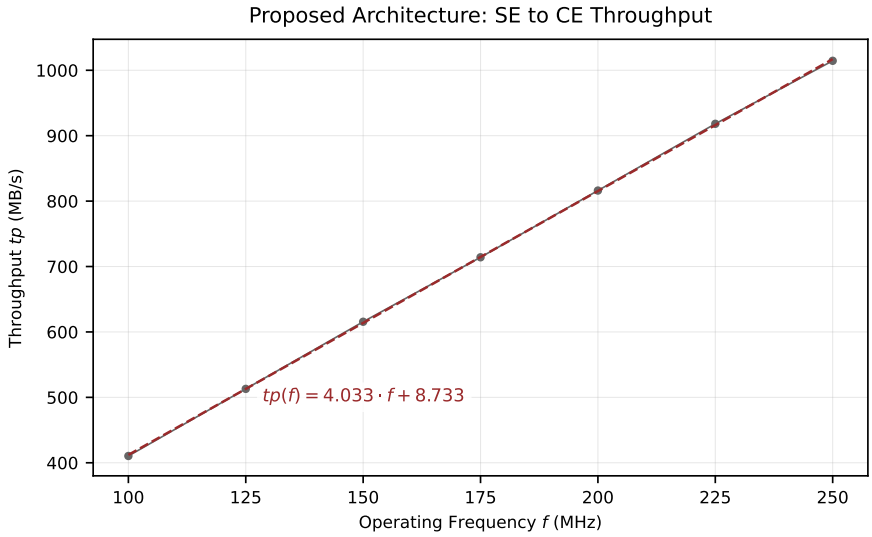


Figure 7.166: Proposed Architecture - SE to CE Throughput across Operating Frequency f.

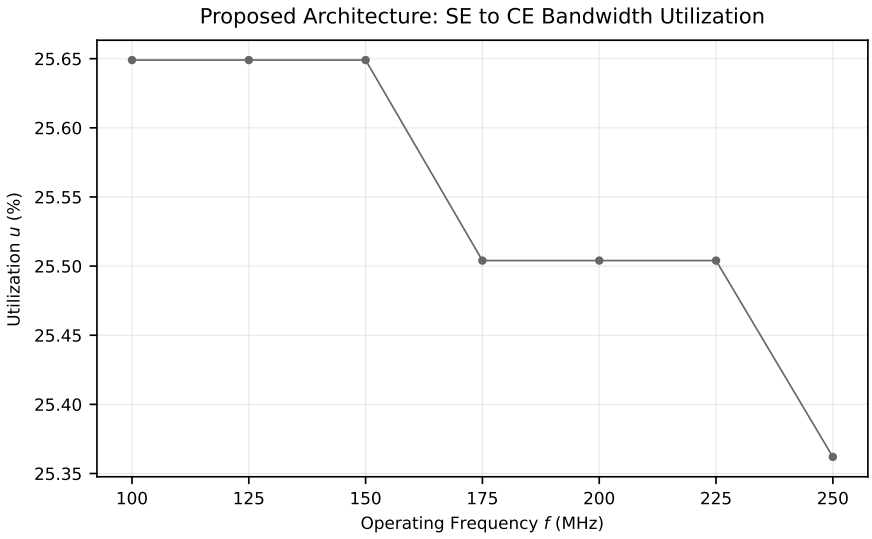


Figure 7.167: Proposed Architecture - SE to CE Bandwidth Utilization across Operating Frequency f.

The proposed CE maximum theoretical throughput ($\mathrm{tp}_{\mathrm{max,ce,prp}}$) is shown in Figure 7.168:

$$\mathrm{tp}_{\mathrm{max,ce,prp}}(f) \approx 8.000 \cdot f \quad [\mathrm{MB/s}] \tag{7.143}$$

This streaming performance translates directly to effective compute execution. As depicted in Figure 7.169, the proposed CE throughput ($\mathrm{tp}_{\mathrm{ce,prp}}$) scales as:

$$\mathrm{tp}_{\mathrm{ce,prp}}(f) = 0.075 \cdot f + 0.160 \quad [\mathrm{MB/s}] \tag{7.144}$$

growing from 7.770 MB/s at 100 MHz to 18.790 MB/s at 250 MHz—achieving a 4.56× to 5.43× speedup over the baseline architecture across operating frequencies.

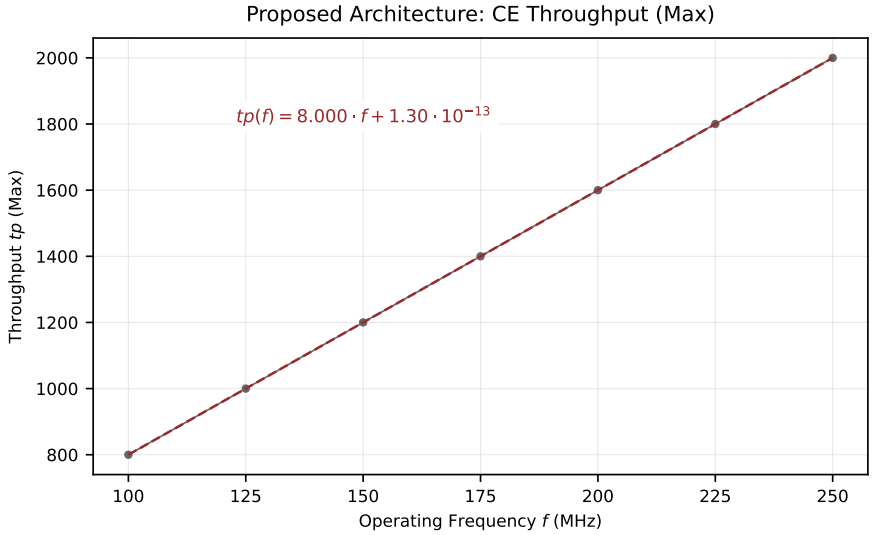


Figure 7.168: Proposed Architecture - CE Maximum Theoretical Throughput across Operating Frequency f.

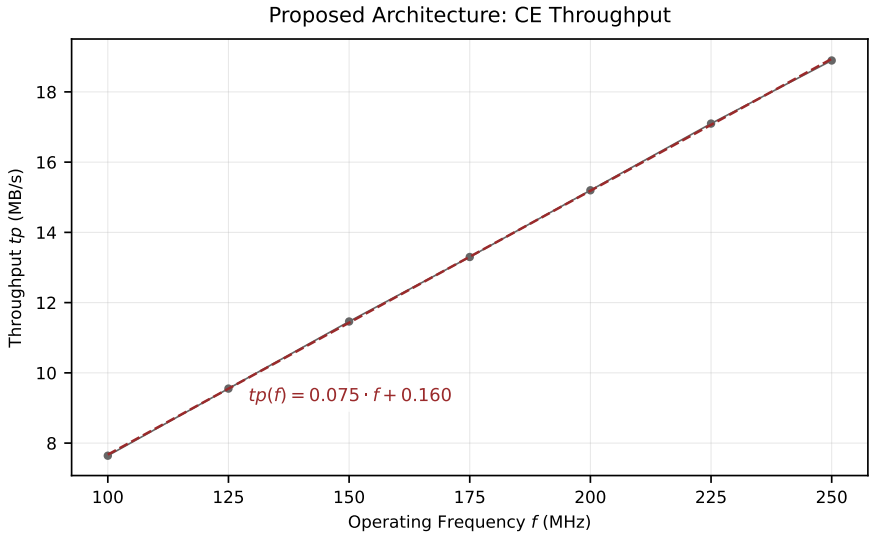


Figure 7.169: Proposed Architecture - CE Throughput across Operating Frequency f.

As shown in Figure 7.170, the CE bandwidth utilization ($u_{\text{bw,ce,prp}}$) remains consistently stable between 0.945% and 0.955%. Finally, as shown in Figure 7.171, the proposed compute utilization ($u_{\text{ce,cmp,prp}}$) stays high within 26.45% – 26.74% across all clock frequencies, representing a 4.49× to 5.56× higher compute utilization compared to the baseline system.

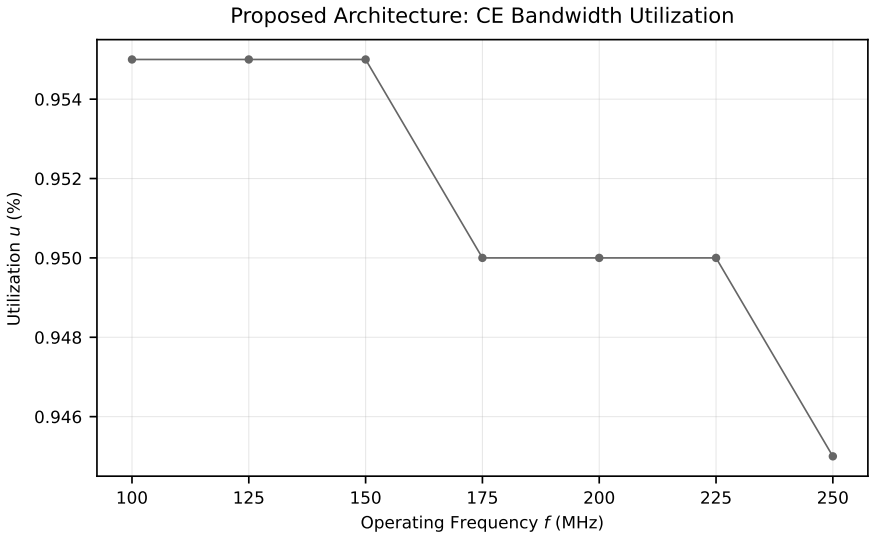


Figure 7.170: Proposed Architecture - CE Bandwidth Utilization across Operating Frequency f.

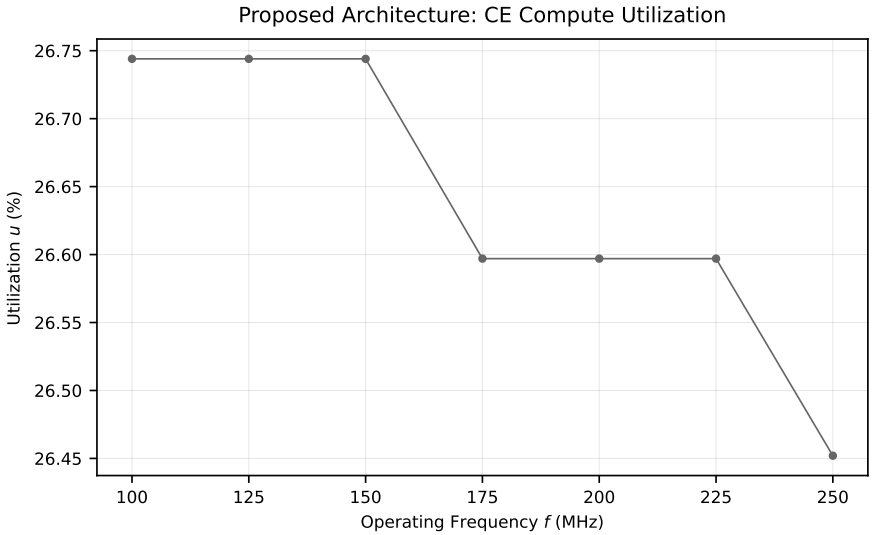


Figure 7.171: Proposed Architecture - CE Compute Utilization across Operating Frequency f.

7.4.3 Memory Latency Sweep

To evaluate memory interface efficiency and system robustness under varying external memory response delays, a memory latency sweep is performed across $l \in [15.0, 35.0]$ ns. As established in previous sections, the problem size is fixed at $N = 19$. All simulations operate at a constant clock frequency $f = 100$ MHz ($T_{\text{clk}} = 10$ ns).

7.4.3.1 Simulation Time and Speedup Analysis

For the baseline architecture, execution time exhibits a step-like staircase profile as a function of the memory latency l. Because access delays are quantized into discrete integer clock cycles ($L = [\![l/T_{\text{clk}}]\!]$), the total simulation time $t_{\text{sim,bsl}}$ remains flat across sub-cycle latency intervals before stepping upward at cycle quantization boundaries:

$$t_{\text{sim,bsl}}(l) = 23040.0\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.145}$$

$$t_{\text{sim,bsl}}(l) = 25740.0\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.146}$$

$$t_{\text{sim,bsl}}(l) = 28440.0\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.147}$$

Due to unbuffered, direct memory accesses for every volume element, each additional cycle of latency (+10 ns) introduces significant cumulative stall overhead, increasing baseline execution time by 2700.0μs per latency cycle step (a 23.44% overall increase across the sweep).

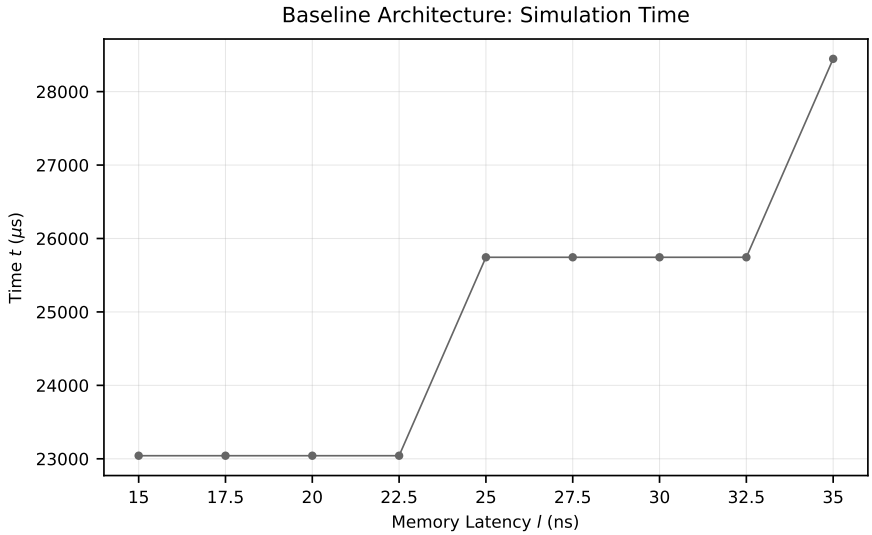


Figure 7.172: Baseline Architecture - Simulation Time across Memory Latency l.

For the proposed architecture, local stream buffering via the SE decouples compute execution from external memory access penalties. Consequently, the simulation time $t_{\text{sim,prp}}$ exhibits far higher resilience to memory latency increases:

$$t_{\text{sim,prp}}(l) = 5171.9\mu\text{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.148}$$

$$t_{\text{sim,prp}}(l) = 5201.2\mu\text{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.149}$$

$$t_{\text{sim,prp}}(l) = 5230.4\mu\text{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.150}$$

Each cycle threshold transition increases total execution time by only 29.24μs, demonstrating that hardware pre-buffering and local caching successfully hide memory fetch latencies behind pipeline execution (yielding an overall execution time increase of merely 1.13% across a 133% latency increase).

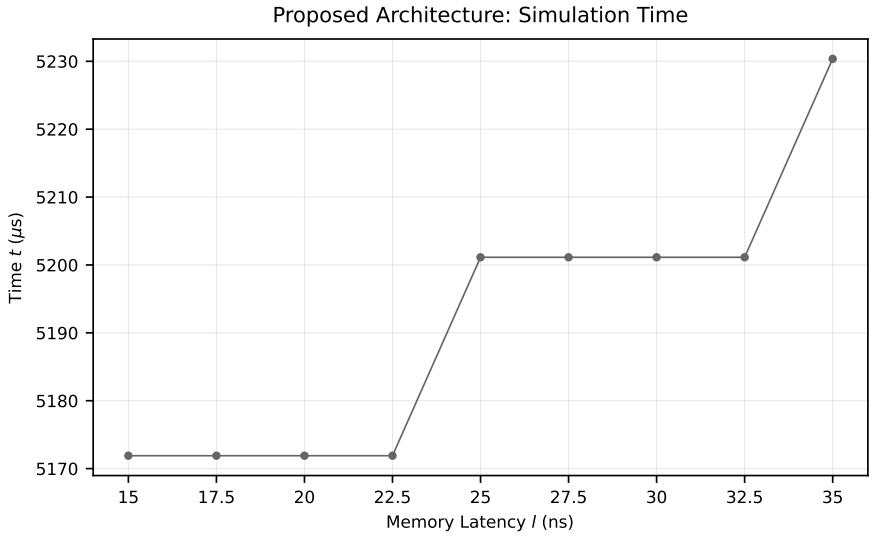


Figure 7.173: Proposed Architecture - Simulation Time across Memory Latency l.

The speedup increases as memory latency degrades:

$$s(l) \approx 4.455\times \quad \text{for } l \in [15.0, 22.5] \text{ ns} \tag{7.151}$$

$$s(l) \approx 4.949\times \quad \text{for } l \in [25.0, 32.5] \text{ ns} \tag{7.152}$$

$$s(l) \approx 5.437\times \quad \text{for } l = 35.0 \text{ ns} \tag{7.153}$$

Because the baseline architecture suffers severe cumulative penalties under higher memory latencies while the proposed design maintains near-constant processing speed, the speedup scales higher at larger values of l.

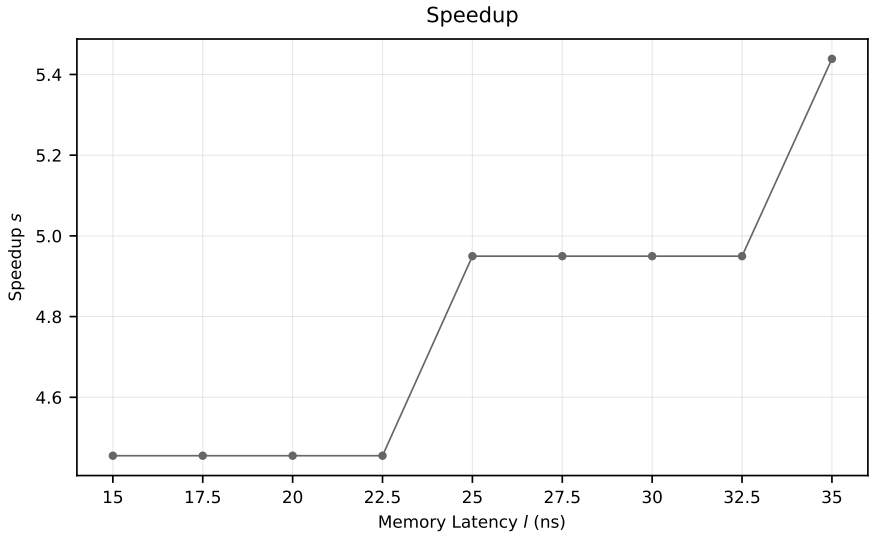


Figure 7.174: Speedup achieved by the Proposed Architecture relative to the Baseline Architecture across Memory Latency l.

7.4.3.2 Memory Access Analysis

The MAF measures the average number of external memory transfers required per output element produced. Because it depends solely on data reuse structures and algorithm access patterns, it remains completely invariant to hardware access latencies l.

For the baseline architecture, the absence of local caching or pre-buffering forces repeated external memory fetches for every voxel during the 3D convolution processing. Consequently, the baseline MAF remains constant across all memory latencies:

$$\mathrm{MAF}_{\mathrm{bsl}}(l) = 55.000 \quad \text{for all } l \in [15.0, 35.0] \text{ ns} \tag{7.154}$$

However, the actual total memory communication time $t_{\mathrm{mem,bsl}}$ scales directly with the cycle-quantized access latency $L = [\![l/T_{\mathrm{clk}}]\!]$:

$$t_{\mathrm{mem,bsl}}(l) = 10800.0\mu\mathrm{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.155}$$

$$t_{\mathrm{mem,bsl}}(l) = 13500.0\mu\mathrm{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.156}$$

$$t_{\mathrm{mem,bsl}}(l) = 16200.0\mu\mathrm{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.157}$$

Because each unbuffered memory request experiences the full penalty of increased memory latency, $t_{\mathrm{mem,bsl}}$ increases by 2700.0μs per latency cycle step, accounting for the vast majority of total execution time degradation in the baseline design.

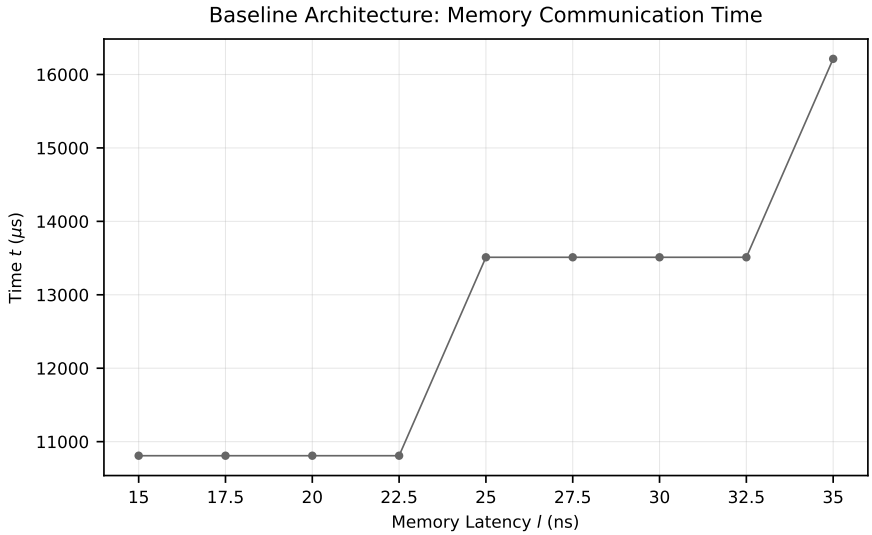


Figure 7.175: Baseline Architecture - Memory Communication Time across Memory Latency l.

For the proposed architecture, stream buffering allied with local caching within the SE reduce redundant spatial fetches across adjacent volumetric windows. This reduces the memory access factor by over an order of magnitude:

$$\mathrm{MAF}_{\mathrm{prp}}(l) = 4.758 \quad \text{for all } l \in [15.0, 35.0] \text{ ns} \tag{7.158}$$

Due to this drastically reduced external transfer volume, total memory communication time $t_{\mathrm{mem,prp}}$ stays minimal and exhibits extremely low sensitivity to latency step increments:

$$t_{\mathrm{mem,prp}}(l) = 526.0\mu\mathrm{s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.159}$$

$$t_{\mathrm{mem,prp}}(l) = 555.2\mu\mathrm{s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.160}$$

$$t_{\mathrm{mem,prp}}(l) = 584.4\mu\mathrm{s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.161}$$

Each latency step increases memory communication time by only 29.2μs, which demonstrates the low sensitivity to latency variations of the proposed design.

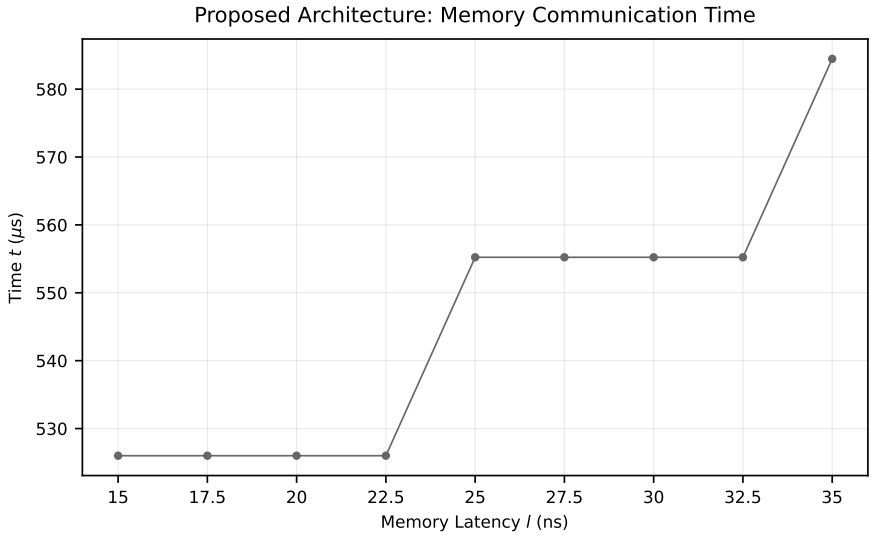


Figure 7.176: Proposed Architecture - Memory Communication Time across Memory Latency l.

7.4.3.3 Throughput and Bandwidth Utilization Analysis

For the baseline architecture, the maximum theoretical throughput of the MC-to-CE interface is $\mathrm{tp}_{\mathrm{max,mc\text{-}ce}} = 800.0$ MB/s. The effective MC-to-CE throughput ($\mathrm{tp}_{\mathrm{mc\text{-}ce}}$) follows a step-down pattern governed by discrete clock cycle quantization ($L = [\![l/T_{\mathrm{clk}}]\!]$), as depicted in Figure 7.177:

$$\mathrm{tp}_{\mathrm{mc\text{-}ce}}(l) = 92.16 \text{ MB/s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.162}$$

$$\mathrm{tp}_{\mathrm{mc\text{-}ce}}(l) = 82.47 \text{ MB/s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.163}$$

$$\mathrm{tp}_{\mathrm{mc\text{-}ce}}(l) = 74.62 \text{ MB/s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.164}$$

As shown in Figure 7.178, this decline causes bus bandwidth utilization ($u_{\mathrm{bw,mc\text{-}ce}}$) to drop from 11.52% to 10.31% and down to 9.33% across cycle thresholds.

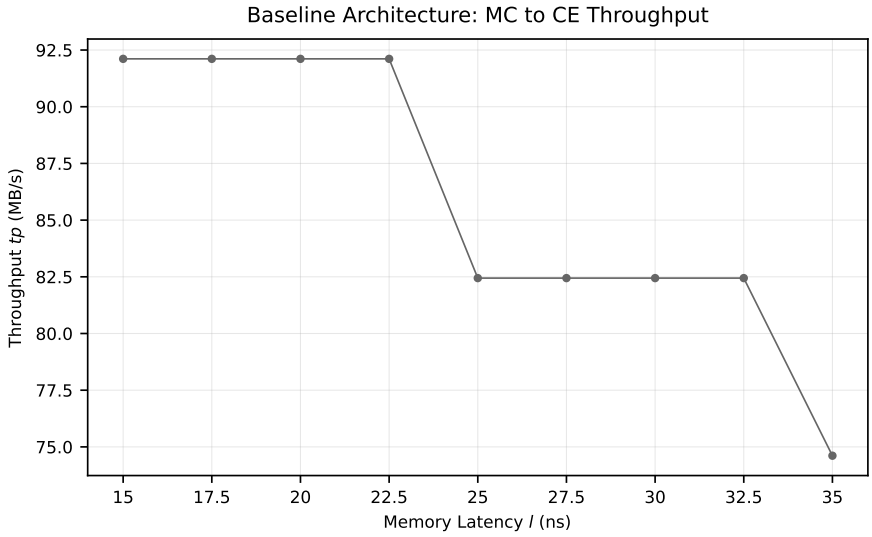


Figure 7.177: Baseline Architecture - MC to CE Throughput across Memory Latency l.

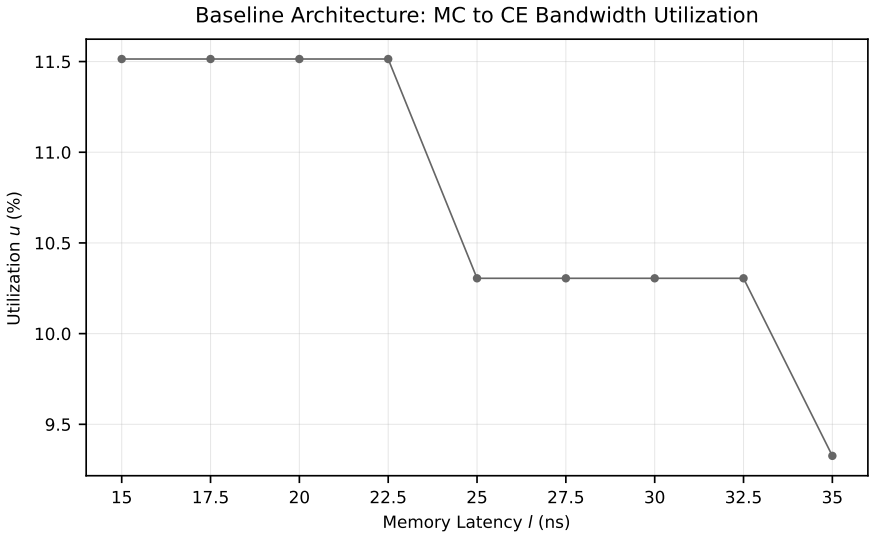


Figure 7.178: Baseline Architecture - MC to CE Bandwidth Utilization across Memory Latency l.

In contrast, the proposed architecture leverages pre-buffering and local caching via the SE to shield the core pipeline from memory latency penalties. With a maximum theoretical throughput of $\mathrm{tp}_{\mathrm{max,se\text{-}ce}} = 1600.0$ MB/s, the effective throughput ($\mathrm{tp}_{\mathrm{se\text{-}ce}}$) shown in Figure 7.179 follows a slight step-down pattern governed by discrete cycle quantization ($L = [\![l/T_{\mathrm{clk}}]\!]$):

$$\mathrm{tp}_{\mathrm{se\text{-}ce}}(l) = 410.37 \text{ MB/s} \quad \text{for } l \in [15.0, 22.5] \text{ ns} \quad (L = 2) \tag{7.165}$$

$$\mathrm{tp}_{\mathrm{se\text{-}ce}}(l) = 408.07 \text{ MB/s} \quad \text{for } l \in [25.0, 32.5] \text{ ns} \quad (L = 3) \tag{7.166}$$

$$\mathrm{tp}_{\mathrm{se\text{-}ce}}(l) = 405.79 \text{ MB/s} \quad \text{for } l = 35.0 \text{ ns} \quad (L = 4) \tag{7.167}$$

As illustrated in Figure 7.180, this yields a consistent bandwidth utilization ($u_{\text{bw,se-ce}}$) dropping slightly from 25.65% to 25.50% and down to 25.36%, representing a $\approx 2.2\times$ to $2.7\times$ improvement over the baseline memory interface.

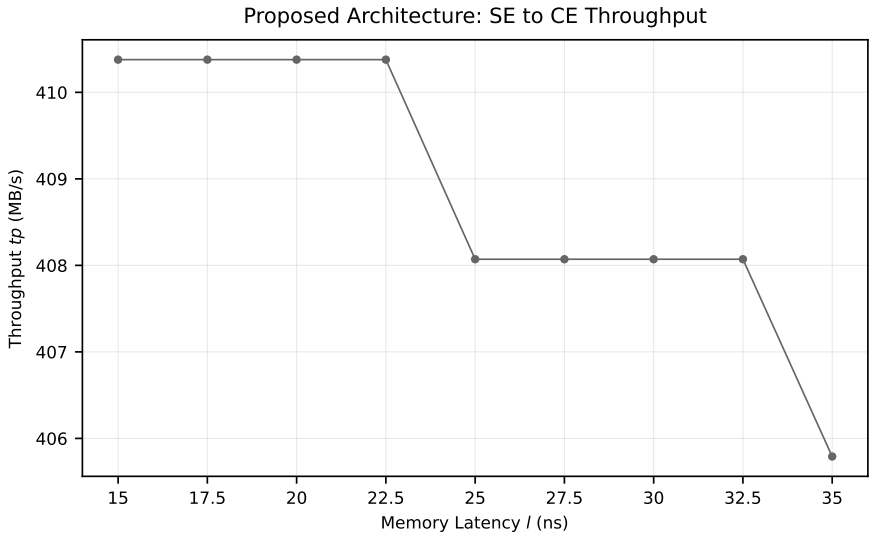


Figure 7.179: Proposed Architecture - SE to CE Throughput across Memory Latency l.

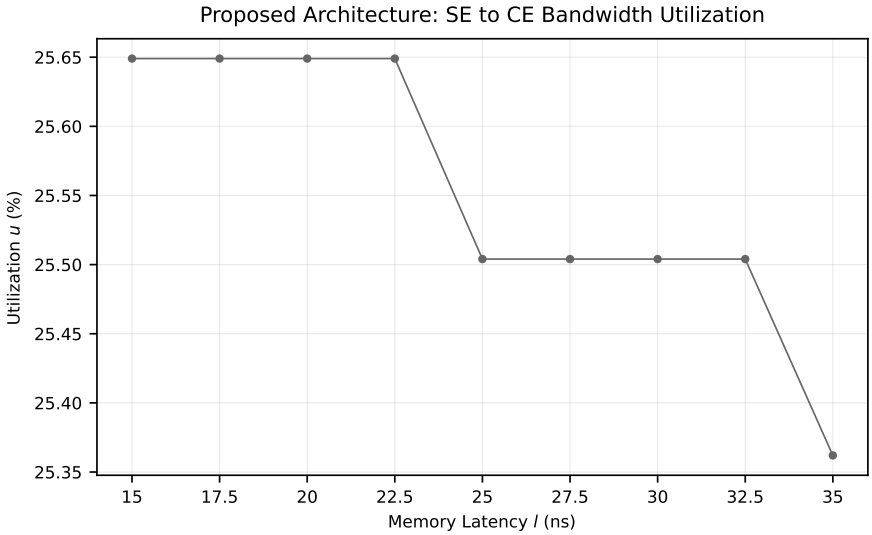


Figure 7.180: Proposed Architecture - SE to CE Bandwidth Utilization across Memory Latency l.

At the processing engine interface, where maximum theoretical compute throughput is bounded by $\text{tp}_{\text{max,ce}} = 800.0$ MB/s, severe memory stalls under unbuffered accesses degrade the baseline CE throughput ($\text{tp}_{\text{ce,bsl}}$) from 1.71 MB/s down to 1.36 MB/s (Figure 7.181). This causes baseline CE bandwidth utilization ($u_{\text{bw,ce,bsl}}$) to plummet to 0.17% (Figure 7.182) and compute utilization ($u_{\text{ce,cmp,bsl}}$) to drop from 5.96% to 4.83% (Figure 7.183).

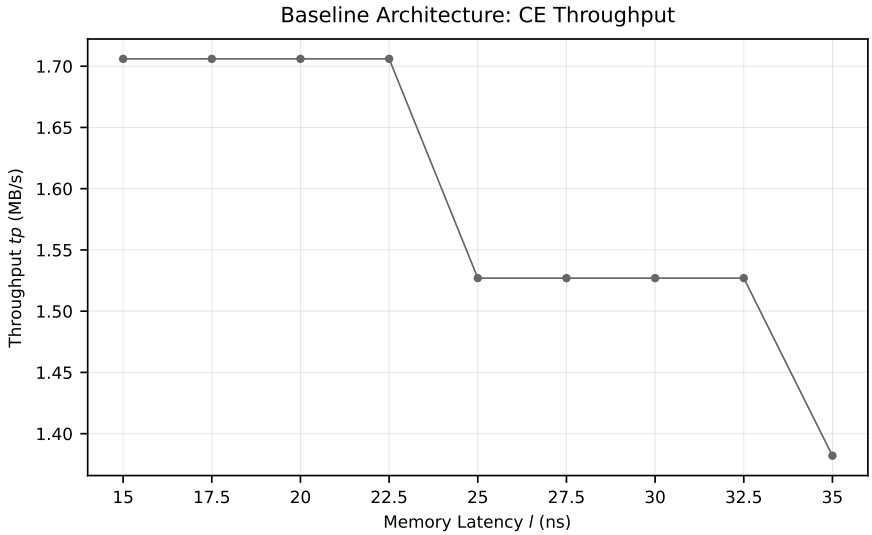


Figure 7.181: Baseline Architecture - CE Throughput across Memory Latency l.

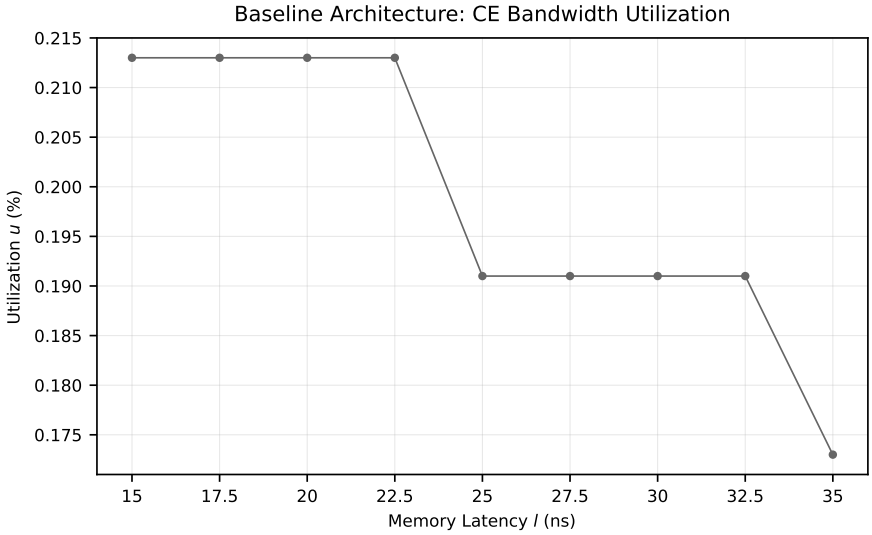


Figure 7.182: Baseline Architecture - CE Bandwidth Utilization across Memory Latency l.

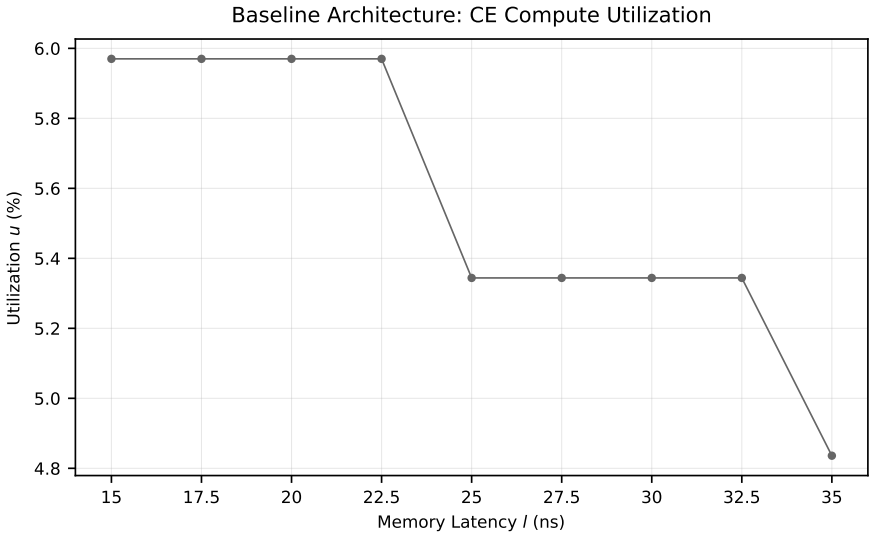


Figure 7.183: Baseline Architecture - CE Compute Utilization across Memory Latency l.

The proposed architecture reduces pipeline starvation, maintaining a high CE throughput ($\text{tp}_{\text{ce,prp}}$) between 7.64 MB/s and 7.56 MB/s, as seen in Figure 7.184. Consequently, proposed CE bandwidth utilization ($u_{\text{bw,ce,prp}}$) stays near 0.95% (Figure 7.185), while compute utilization ($u_{\text{ce,cmp,prp}}$) is sustained at $\approx 26.45\% - 26.75\%$ (Figure 7.186), demonstrating reduced latency sensitivity.

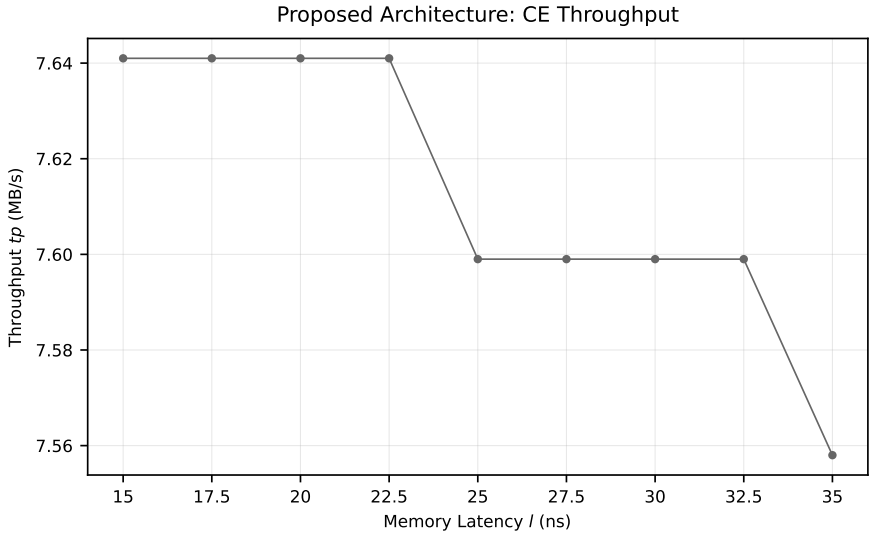


Figure 7.184: Proposed Architecture - CE Throughput across Memory Latency l.

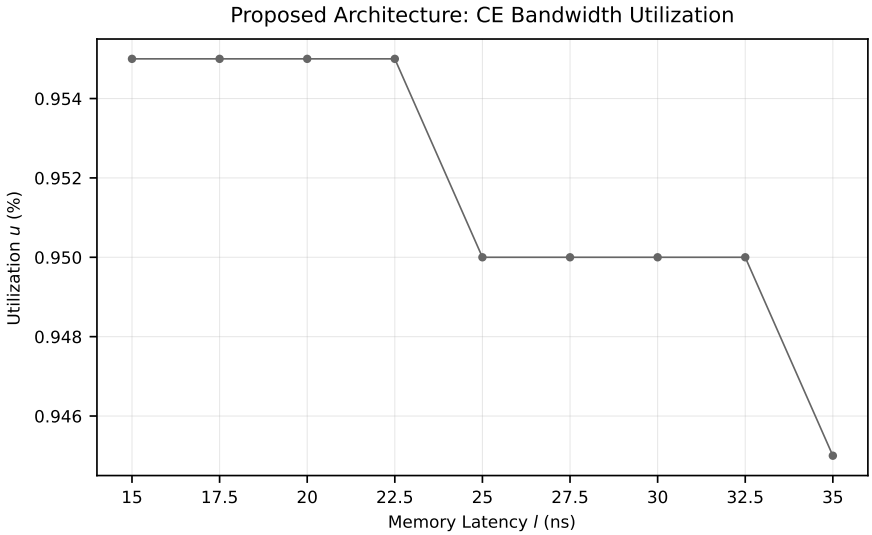


Figure 7.185: Proposed Architecture - CE Bandwidth Utilization across Memory Latency l.

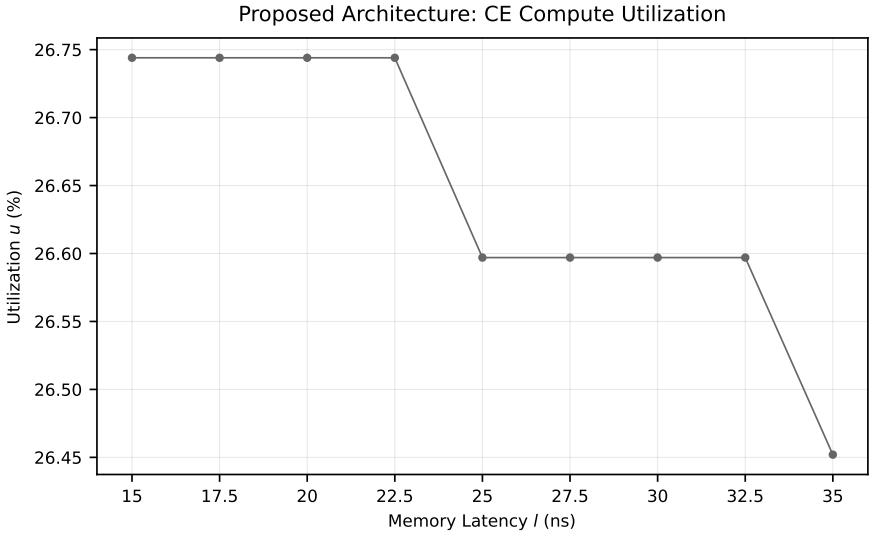


Figure 7.186: Proposed Architecture - CE Compute Utilization across Memory Latency l.

7.5 Discussion

In this section, we summarize and discuss the experimental results presented in the previous sections for each benchmark.

Across all benchmarks, the frequency and latency sweeps demonstrate consistent, physically coherent trends. As expected, throughput scales proportionally with the operating frequency and degrades as memory latency increases. The discrete step-like discontinuities observed across both sweeps are physical artifacts resulting from integer clock cycle quantization ($L = [\![l/T_{\text{clk}}]\!]$), marking operational boundaries where incremental changes in physical time force the handshake logic to allocate an additional clock cycle.

Furthermore, the response across frequency and latency sweeps directly mirrors the performance of each architecture when processing a specific workload. While the absolute metric values and relative architectural speedups shift based on the N value selected for each benchmark, the response curves to frequency and latency scaling remain structurally predictable around the reference values established by the selected problem size N. Consequently, the problem size sweep serves as the primary characterization of structural architectural efficiency, whereas the frequency and latency sweeps act as cross-cutting validations of operational stability. The following subsections discuss the results obtained for each benchmark regarding the problem size sweep.

7.5.1 FIR Filter

For the FIR filter benchmark, the proposed architecture demonstrated substantially superior execution performance and resource efficiency compared to the baseline architecture. This performance gap is directly rooted in how each architecture handles memory interaction and feeds data to the compute core.

In the baseline architecture, execution is severely memory-bound because the CE interacts directly with main memory without any local buffering or caching mechanisms. Fetching every input operand and filter tap through unbuffered, single-element transactions forces the processing pipeline to suffer constant memory stalls. As a result, the baseline operates with a high memory access footprint (MAF $= 33.0$) and a low compute unit utilization ($\approx 6.03\%$).

In contrast, the proposed architecture overcomes these memory bottlenecks by placing the SE as a dedicated CE feeding interface. By configuring the IS1 SB in CB mode, the 16 filter coefficients are pre-loaded into the SB and continuously reused across all sliding-window operations. Combined with memory burst transfers that exploit spatial locality, the proposed design eliminates redundant external reads, drastically reduces memory communication time, and thereby mitigates the processing pipeline starvation. This structural decoupling reduces the MAF to its theoretical minimum (MAF $\rightarrow 2.0$), yielding a sustained speedup of $\approx 4.74\times$ and raising compute unit utilization to $\approx 28.57\%$.

7.5.2 Matrix Multiplication

For the Matrix Multiplication benchmark, the proposed architecture exhibited substantially inferior execution performance and resource efficiency compared to the baseline architecture across the evaluated problem sizes. This performance degradation stems from the combined impact of continuous stream re-loading under FIFO mode and severe cache thrashing during strided column traversals.

Because both IS0 and IS1 SBs are configured in standard FIFO mode without hardware circular buffering, local data reuse is reduced. Consequently, the SCH is forced to continuously dispatch load requests to the IS0 and IS1 LDRs to repeatedly push matrix operands into the SBs across the 3-level nested loop. In contrast, operating a SB in CB mode requires only a single initial pre-load operation, after which the buffer continuously feeds the CE without the need to issue further memory requests regarding that stream.

While the baseline architecture also re-fetches elements across loop iterations, its direct unbuffered single-word transactions strictly request the exact target address ($\text{MAF}_{\text{bsl}} = 2.000 \cdot N + 1.000$), maintaining steady compute utilization ($\approx 5.90\% - 5.96\%$). Conversely, in the proposed architecture, traversing Matrix B column-wise forces every miss detected by the IS1 LDR to trigger an 8-element burst transaction to fill an IS1 SC line. Because row-major layout places vertical column elements non-contiguously in memory, the 7 adjacent horizontal words loaded alongside each target element offer zero spatial locality for immediate kernel steps.

This structural mismatch causes two severe thrashing mechanisms:

1. **Conflict and Tag Replacement:** Disperse strided accesses map to overlapping cache sets, prematurely evicting previously buffered lines before their data can be reused.

2. **Spatial Pollution:** Unused horizontal words pollute the small SC capacity, displacing valid data required for subsequent loop iterations.

Because burst transfers occupy the memory bus for significantly longer durations than single requests, recurring misses during column traversals amplify the bus occupancy. Rather than benefiting from burst efficiency, the system repeatedly executes long burst line fills for useless data, inflating total memory communication time beyond double that of the baseline, driving the MAF to ≈ 1000 at $N = 126$, starving the CE below 3% utilization, and dropping relative speedup into a low regime ($\approx 0.45 - 0.65$).

7.5.3 2D Convolution

For the 2D Convolution benchmark, the proposed architecture demonstrated superior execution efficiency and processing performance compared to the baseline architecture across all evaluated input grid dimensions. This performance gain is achieved by leveraging localized stream buffering, high local cache hit rates, and high-throughput burst transfers managed by the SE.

In the baseline architecture, execution is severely bottlenecked by unbuffered single-word memory transactions. Computing each output pixel in the 2D sliding-window traversal requires 19

discrete, single-element memory accesses — maintaining a constant MAF of 19.000 where every fetch incurs handshake overheads. Consequently, memory communication time scales heavily, causing continuous pipeline stalls that restrict CE compute utilization to a low constant rate of 6.135%.

Conversely, the proposed architecture mitigates external memory latency by capitalizing on three core advantages of the SE:

1. **Circular Buffering:** The 3×3 kernel weights are pre-loaded once into the IS1 SB operating in CB mode. Once initialized, the IS1 SB continuously feeds coefficients into the processing pipeline without requiring the SCH to issue further external memory requests, completely eliminating redundant coefficient fetches.

2. **Localized Caching:** For the image input stream (IS0), spatial locality during the sliding-window traversal enables high hit rates in the local IS0 SC. The IS0 LDR fetches pixel data directly from the local IS0 SC without triggering external memory line fills and inducing bus latency.

3. **Burst Transactions:** Instead of direct memory requests, the SE allows burst memory requests, which can mitigate the memory communication time.

By combining coefficient pre-loading, high local cache hit rates, and burst-based memory cache line fills, the proposed architecture drives the MAF asymptotically down toward ≈ 1.989. This suppresses external memory communication latency scaling from $0.760\,\mu$s down to $0.045\,\mu$s per N^2 element ($\approx 16.89\times$ latency reduction). As a result, the processing engine receives a continuous supply of operand data, boosting the CE compute utilization to 26.811% and delivering a sustained speedup improvement of $\approx 4.390\times$ over the baseline design.

7.5.4 3D Convolution

For the 3D Convolution benchmark, the proposed architecture demonstrated superior execution efficiency and processing performance compared to the baseline architecture across most of the volumetric tensor dimensions. This performance gain is achieved by leveraging complete coefficient reuse via circular buffering, spatial locality caching, and multi-element burst line fills managed by the SE.

In the baseline architecture, execution is severely memory-bound due to unbuffered single-word memory transactions. Computing each output voxel in the $3 \times 3 \times 3$ sliding-window traversal requires issuing 55 discrete scalar memory requests across external memory. Maintaining a constant MAF of 55.000, every single-element fetch incurs full bus handshake overheads. Driven by cubic scaling (N^3), memory communication time grows heavily, causing continuous pipeline stalls that restrict CE compute utilization to a low constant rate of 5.970%.

Conversely, the proposed architecture mitigates external memory latency by capitalizing on the primary microarchitectural mechanisms of the SE:

1. **Circular Buffering:** The 27 filter coefficients of the $3 \times 3 \times 3$ kernel are pre-loaded once into the IS1 SB operating in CB mode. Once initialized, the weights are completely reused across all volumetric sliding-window steps, eliminating redundant external coefficient fetches.

2. **Localized Caching:** Spatial locality during volumetric traversals enables high local hit rates in the IS0 SC. The IS0 LDR services pixel requests directly from the local cache without triggering external line fills and inducing bus latency.

3. **Burst Transactions:** Unbuffered single-element fetches are replaced with multi-element burst line fills, amortizing bus handshake overheads across contiguous spatial data.

Despite these architectural advantages, volumetric memory traversals introduce dimension-dependent performance degradation. For specific strides ($N = 16$ and $N \in [22, 23]$), 3D indexing across row (N) and plane (N^2) boundaries maps non-contiguous elements into overlapping SC sets. This set aliasing induces conflict-driven cache thrashing and premature line evictions, inflating the MAF up to ≈ 54.8 and temporarily dropping the compute utilization ($9.78\% - 12.78\%$) and the speedup ($\approx 1.57\times - 2.22\times$).

Outside these conflict-heavy dimensions, the pre-buffering and localized caching mechanisms operate at peak efficiency. The proposed architecture suppresses the baseline MAF down toward $\approx 2.6 - 4.8$, raising steady-state compute utilization to 26.78% (and up to 29.76% for smaller volumes) and delivering a sustained acceleration factor between $\approx 4.5\times - 4.85\times$ over the baseline design.

7.6 Summary

Overall, the experimental evaluation across the benchmark suite validates the core objectives of this thesis. By modeling and implementing a hardware-configurable SE, the results confirm that memory-decoupling and localized data reuse substantially transform the execution efficiency and resource utilization of computing substrates over traditional memory-coupled designs. However, the benchmarking also exposed clear structural limitations when the SE encounters specific multi-dimensional memory access patterns.

Across the evaluated workload space, system performance is fundamentally governed by data reusability. When the programmed access pattern maximizes data reuse, the SE effectively isolates the processing core from external memory delays by combining circular buffering and burst-oriented local caching, achieving high localized data reuse.

Conversely, the performance degradation is a direct function of memory access patterns that have low data reusability under row-major memory layout constraints. During non-contiguous vertical or strided traversals, the lack of dedicated column-traversal hardware logic forces burst transactions to load undesired adjacent data. This spatial pollution causes cache thrashing and set-aliasing conflict evictions, prematurely displacing useful data and repeatedly triggering high-latency external memory accesses that bottleneck the system throughput.

In conclusion, the SE achieves considerable hardware acceleration when memory access patterns align with its primary mechanisms — specialized streaming units (SBs) with circular buffering and burst-oriented caching for contiguous spatial reuse. Mitigating its limitations under highly strided access patterns will require future micro-architectural enhancements, such as extending the SB module to support programmable temporal circular buffering. Adding the capability to dynamically lock loaded buffers and circulate over local data segments on demand could dramatically increase data reuse and processing efficiency for dense linear algebra workloads such as the matrix multiplication.

Chapter 8

Conclusion

This chapter concludes the thesis by summarizing the primary achievements and inherent architectural limitations of the proposed work, reflecting on the overall research outcomes, and presenting prospective directions for future work.

8.1 Achievements and Limitations

The primary goal of this thesis was to explore how memory decoupling and localized data reuse impact reconfigurable compute architectures. All proposed specific objectives outlined in Section 1.3 were successfully fulfilled. To enable a quantitative comparative study, two distinct compute substrates — the MC-CE and the MD-CE — were developed alongside a functional external memory subsystem. Comprehensive benchmarking across representative multi-dimensional workloads demonstrated for most cases that offloading address generation and data prefetching to a specialized streaming unit significantly mitigates memory stalls, reduces application runtime, and maximizes compute unit utilization.

Despite these successful achievements, several architectural and modeling limitations were established to maintain a focused scope for this exploratory study:

- **Simplified Memory Access Latency Model:** The external memory subsystem relies on a representative model with fixed latency parameters configured within the memory controller to simulate access delays. Consequently, it does not fully capture the variable response latencies, bus contention, and dynamic transaction delays characteristic of real-world physical memory controllers (e.g., DRAM controllers).
- **Abstract Memory Addressing Granularity:** The implemented MB and the internal SCs use an element-addressable indexing scheme, where each memory address directly maps to a complete data element according to the configured system data width. In modern physical memory arrays, addresses typically point to individual bytes or words. Implementing standard byte-addressable memory retrieval would require multi-byte alignment, word-masking logic, and multi-cycle fetch sequences rather than single-location access.

- **Substrate Scale and Topology Limits:** The implemented CEs (MC-CE and MD-CE) serve as concise functional abstractions of a computing substrate rather than large-scale CGRA fabrics (such as 4×4 or 6×6 arrays). Because the primary objective was to evaluate the core impact of memory decoupling rather than the scalability of the SE across varying array topologies, the compute fabrics were intentionally kept compact and focused.

8.2 Summary

In summary, this research presents an exploratory architectural evaluation of memory-decoupled CGRA substrates driven by a hardware-configurable SE. By isolating data movement logic from core computation, the proposed design effectively decouples address calculation routines and memory transaction management from the execution pipeline.

The experimental findings confirm that incorporating localized stream buffering and programmable address generation units yields substantial performance gains over conventional memory-coupled designs. The offloading of memory orchestration simplifies the processing element fabric, offering a viable design paradigm for reducing silicon area overhead, lowering power consumption, and achieving higher compute efficiency in future reconfigurable processors.

8.3 Future Work

Building upon the insights and limitations identified in this thesis, several compelling avenues for future research are highlighted:

- **Integration of Realistic Memory Subsystems:** Replacing the simplified memory model with cycle-accurate DRAM simulation frameworks (such as Ramulator 2) or standard physical memory interfaces (e.g., AXI4-based memory controllers) will enable the SE to be evaluated against non-deterministic latency, bank conflicts, and realistic bus contention.
- **Adaptation to Varied and Scalable CGRA Topologies:** Investigating how the SE scales when driving larger, heterogeneous CGRA fabrics (e.g., 4×4 or 8×8 PE arrays). Adapting the SE to interface with diverse interconnect topologies may require structural evolution within the internal scheduling and stream cache management units.
- **FPGA Prototyping and In-Silicon Validation:** Deploying the complete system on a physical FPGA platform to gather real-world hardware metrics, validate timing closing at target operational frequencies, and measure exact power, performance, and area (Power, Performance, and Area (PPA)) trade-offs.

In summary, this study demonstrates the quantitative benefits of memory decoupling and provides practical design insights that can inform future work on stream-based reconfigurable computing architectures.

References

[1] O. Ragheb et al., "CGRA-ME 2.0: A Research Framework for Next-Generation CGRA Architectures and CAD," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024, pp. 642–649. DOI: 10.1109/IPDPSW63119.2024.00124

[2] A. Podobas, K. Sano, and S. Matsuoka, "A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.

[3] M. Wijtvliet, H. Corporaal, and A. Kumar, "CGRA Background and Related Work," in *Blocks, Towards Energy-efficient, Coarse-grained Reconfigurable Architectures*, Springer, 2021, pp. 15–60.

[4] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse Grained Reconfigurable Architectures in the Past 25 Years: Overview and Classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016, pp. 235–244. DOI: 10.1109/SAMOS.2016.7818353

[5] R. Hou et al., "Efficient Data Streaming with On-Chip Accelerators: Opportunities and Challenges," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 312–320. DOI: 10.1109/HPCA.2011.5749739

[6] D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra, "CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–26, 2019.

[7] L. Crespo, N. Neves, P. Tomás, and N. Roma, "A Survey on Stream-Based Architectures: From Accelerators to CPUs," *Proceedings of the IEEE*, vol. 113, no. 8, pp. 713–751, 2025. DOI: 10.1109/JPROC.2025.3642972

[8] B. Jacob, D. Wang, and S. Ng, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[9] R. Micheloni, A. Marelli, and K. Eshghi, *Inside Solid State Drives (SSDs)*. Springer, 2013.

[10] J.-H. Lee, J.-S. Lee, and S.-D. Kim, "A New Cache Architecture Based on Temporal and Spatial Locality," *Journal of Systems Architecture*, vol. 46, no. 15, pp. 1451–1467, 2000.

[11] U. J. Otozi, C.-O. Chioma, C. E. Peter, and M. N. Raymond, "Virtual and Cache Memory: Implications for Enhanced Performance of the Computer System," *International Journal of Computer Applications*, vol. 975, p. 8887, 2018.

[12] M. Abbaszadeh, "Efficient Data Streaming for a Tightly-Coupled CGRA," M.S. thesis, University of Toronto (Canada), 2023.

[13] G. P. Midoes, "UVE Streaming Engine: Hardware Implementation and Evaluation," 2024.