CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# JIT-Based RVV Optimization for Implicit GEMM Convolution and FlashAttention

## With Cross-Architecture Autotuning, a Winograd $F(4, 3)$ Ablation, and AVX2 / AVX-512 / NEON Baselines

Master's thesis in High Performance Computer Systems

YUHAO WANG, TIANDU LI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

# JIT-Based RVV Optimization for Implicit GEMM Convolution and FlashAttention

With Cross-Architecture Autotuning, a Winograd $F(4, 3)$ Ablation, and AVX2 / AVX-512 / NEON Baselines

YUHAO WANG, TIANDU LI

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
*High Performance Computer Systems*
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

JIT-Based RVV Optimization for Implicit GEMM Convolution and FlashAttention With Cross-Architecture Autotuning, a Winograd $F(4, 3)$ Ablation, and AVX2 / AVX-512 / NEON Baselines
YUHAO WANG, TIANDU LI



Supervisor: Sonia Rani Gupta, Department of Computer Science and Engineering
Examiner: Miquel Pericàs, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
High Performance Computer Systems
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

# Abstract

Convolutional Neural Networks (CNNs) are fundamental to modern deep learning applications, including image classification, object detection, and autonomous systems. The computational efficiency of convolution operations is critical for real-time inference on resource-constrained devices. Traditional implementations rely on the im2col transformation followed by General Matrix Multiplication (GEMM), which incurs significant memory overhead due to explicit tensor expansion.

This thesis presents the **Implicit GEMM** convolution algorithm, which eliminates the im2col memory overhead by computing input coordinates dynamically during matrix multiplication. We implement and optimize this algorithm across four CPU vector backends—x86-64 with AVX2 and AVX-512, ARM with NEON, and RISC-V with the Vector Extension (RVV)—and complement it with a JIT-based FlashAttention kernel and an RVV Winograd $F(4, 3)$ ablation study.

The contributions of this thesis are:

1. A lightweight Just-In-Time (JIT) code generation framework for RVV that emits register-blocked, vector-length-agnostic Implicit GEMM micro-kernels at runtime, with loop unrolling and explicit register assignment.
2. A Vector Length Agnostic (VLA) RVV implementation evaluated across six VLEN configurations (128–8192 bits) on the gem5 simulator and on the BananaPi-F3 development board.
3. A hand-written AVX2 micro-kernel (6×16) and a portable ARM NEON micro-kernel (8×8) that share the same Implicit GEMM design, used to validate cross-architecture portability.
4. A lightweight RVV autotuner over {MR, LMUL, $k$-unroll} with register-budget pruning, contrasted with the AVX-512 autotuner to identify which tuning knobs transfer across architectures and which do not.
5. A JIT FlashAttention kernel as a non-convolution case study, and an RVV Winograd $F(4, 3)$ five-way ablation that separates algorithmic gains from implementation-level effects.

The implementations are integrated into the Intel oneDNN framework. On x86-64, the AVX2 Implicit GEMM achieves a peak of **164.68 GFLOPS** (9.5× over oneDNN's `gemm_convolution`), eliminating a 56.85 MB im2col buffer. Extended to AVX-512 with NUMA-aware tiling and a per-layer autotuner, the same design reaches

**599 GFLOPS** averaged over five VGG-16 layers and a single-layer peak of **1161 GFLOPS**, with cross-network speedups of **22×–228×** over a non-vectorized scalar baseline. On ARM NEON, the peak is **81.0 GFLOPS** (15.4× over a scalar reference). On RVV, the JIT kernel delivers up to **3.28×** over the scalar reference (gem5, VLEN=256) and is validated on the real BananaPi-F3 hardware (3.08×). Across the two autotuners, every layer except a few narrow-channel cases independently selects `MR=8`, indicating that the dominant tile parameter is governed by the architectural vector-register budget rather than by ISA-specific concerns.

This work demonstrates the effectiveness of Implicit GEMM as a memory-efficient alternative to traditional convolution methods, with particular relevance for emerging RISC-V platforms where optimized deep learning libraries remain limited, and provides a cross-architecture analysis of which optimization decisions transfer and which do not.

**Keywords:** Implicit GEMM, Convolution, Neural Networks, oneDNN, JIT Compilation, Cross-Architecture Optimization, Autotuning, AVX2, AVX-512, ARM NEON, RISC-V Vector Extension, FlashAttention, Winograd, High Performance Computing

# Acknowledgements

We would like to express our sincere gratitude to our examiner, Prof. Miquel Pericàs, for his invaluable guidance, research insights, and continuous support throughout this thesis. His expertise in high-performance computing and computer architecture has been instrumental in shaping this work.

We are equally thankful to our supervisor, Sonia Rani Gupta, for her consistent guidance and practical advice at every stage of the project. Her feedback on implementation details and evaluation methodology greatly improved the quality of this research.

We would also like to thank the Chalmers C3SE center for providing access to computational resources necessary for the x86-64 experiments.

Special thanks to the open-source communities behind oneDNN, RISC-V, and gem5, whose excellent software and documentation made this work possible.

Finally, we are deeply grateful to our families and friends for their patience, encouragement, and unwavering support during this journey.

Yuhao Wang and Tiandu Li, Gothenburg, June 2026

# List of Acronyms

Below is the list of acronyms used throughout this thesis, listed in alphabetical order:

| Acronym | Full Form |
|---|---|
| AVX | Advanced Vector Extensions |
| AVX2 | Advanced Vector Extensions 2 |
| AVX-512 | Advanced Vector Extensions 512-bit |
| BLAS | Basic Linear Algebra Subprograms |
| BLIS | BLAS-like Library Instantiation Software |
| CNN | Convolutional Neural Network |
| CPU | Central Processing Unit |
| DNN | Deep Neural Network |
| FMA | Fused Multiply-Add |
| FNV | Fowler–Noll–Vo (hash function) |
| GEBP | General Block-times-Panel (matrix-multiplication kernel) |
| GEPB | General Panel-times-Block |
| GEPP | General Panel-times-Panel |
| GEMM | General Matrix Multiplication |
| GFLOPS | Giga Floating-point Operations Per Second |
| GPU | Graphics Processing Unit |
| im2col | Image to Column (transformation) |
| ISA | Instruction Set Architecture |
| JIT | Just-In-Time (compilation) |
| LMUL | Length Multiplier (RVV) |
| NCHW | Batch, Channels, Height, Width (tensor layout) |
| NHWC | Batch, Height, Width, Channels (tensor layout) |
| OIHW | Output channels, Input channels, Height, Width |
| oneDNN | oneAPI Deep Neural Network Library |
| OpenMP | Open Multi-Processing |
| RISC-V | Reduced Instruction Set Computer - Five |
| RVV | RISC-V Vector Extension |
| SEW | Standard Element Width |
| SIMD | Single Instruction, Multiple Data |
| VLA | Vector Length Agnostic |
| VLEN | Vector Length (hardware vector register size) |
| VL | Vector Length (runtime vector length) |
| VGG | Visual Geometry Group |

# Contents

**Abstract** v

**Acknowledgements** vii

**List of Acronyms** viii

**List of Figures** xv

**List of Tables** xvii

**1 Introduction** 1
- 1.1 The Memory Bottleneck . . . 2
- 1.2 The Compute Challenge: Why Implicit GEMM needs JIT . . . 2
- 1.3 Goals and Contributions . . . 3
- 1.4 Claims and Evaluation Roadmap . . . 4
- 1.5 Thesis Organization . . . 5

**2 Background** 6
- 2.1 Convolutional Neural Networks . . . 6
  - 2.1.1 Convolution Operation . . . 6
  - 2.1.2 VGG16 Architecture . . . 7
  - 2.1.3 Benchmark Layer Inventory . . . 7
- 2.2 Convolution Implementation Strategies . . . 8
  - 2.2.1 Direct Convolution . . . 8
  - 2.2.2 im2col + GEMM . . . 8
  - 2.2.3 Implicit GEMM . . . 9
  - 2.2.4 Comparison of Approaches . . . 9
  - 2.2.5 Winograd Convolution . . . 9
    - 2.2.5.1 Winograd $F(m, r)$ Algorithm . . . 10
    - 2.2.5.2 $F(4, 3)$ Variant for $3 \times 3$ Convolutions . . . 10
    - 2.2.5.3 Computational Complexity . . . 11
- 2.3 FlashAttention . . . 11
- 2.4 Polynomial Approximation of $e^x$ . . . 12
- 2.5 GEMM Optimization Techniques . . . 14
  - 2.5.1 Cache Blocking (Tiling) . . . 14
  - 2.5.2 Micro-kernel Design . . . 14

2.5.3 Filter Packing . . . 14
2.6 Hardware Architectures . . . 15
2.6.1 x86-64 with AVX2 and AVX-512 . . . 15
2.6.2 RISC-V Vector Extension (RVV) . . . 15
2.7 oneDNN Framework . . . 15

**3 Methodology and Implementation 17**
3.1 JIT Code Generation Framework . . . 17
3.1.1 Direct Binary Emission . . . 17
3.1.2 Micro-kernel Structure Specialization . . . 18
3.1.3 Micro-kernel Generation Pipeline . . . 18
3.2 Implicit GEMM . . . 19
3.2.1 Algorithm Overview . . . 19
3.2.2 RVV Implementation . . . 20
3.2.2.1 Micro-kernel Adaptation: From Fixed-Width Tiles to 8×VL . . . 21
3.2.2.2 Data Layout and Filter Packing . . . 21
3.2.2.3 RVV Intrinsics Implementation . . . 22
3.2.2.4 Inline Assembly Optimization . . . 22
3.2.2.5 JIT Code Generation for RVV . . . 24
3.2.2.6 Implementation Comparison . . . 25
3.2.3 x86-64 AVX-512 Implementation . . . 25
3.2.3.1 Five Vectorization Strategies . . . 25
3.2.3.2 Filter Packing . . . 28
3.2.3.3 Two-Dimensional Register Blocking . . . 28
3.2.3.4 Padding Fast/Slow Path Dispatch . . . 29
3.2.3.5 Template Specialization . . . 30
3.2.4 Auto-tuner Framework . . . 32
3.2.4.1 Search Space and Pruning . . . 32
3.2.4.2 TILE_M: L2 Cache Tiling via OpenMP Chunk-size Control . . . 33
3.2.4.3 Per-shape Result Cache . . . 35
3.2.4.4 Comparison with the BananaPi RVV Sweep . . . 35
3.3 Case Study: JIT FlashAttention on RVV . . . 36
3.3.1 Algorithm Implementation . . . 37
3.3.2 Cephes exp() Approximation . . . 37
3.3.3 ABI Compliance and Reentrancy . . . 37
3.3.4 JIT Optimizations for FlashAttention . . . 38
3.4 Case Study: Winograd $F(4,3)$ on RVV . . . 39
3.4.1 Implementation Architecture . . . 40
3.4.2 RVV Vectorization Strategy . . . 40
3.4.3 Five-Way Ablation Study Design . . . 41
3.4.4 GEMM Micro-kernel Reuse . . . 42
3.4.5 Multi-Network Benchmark Suite . . . 42

**4 Results 44**
4.1 Evaluation Protocol . . . 44

4.2 Experimental Setup . . . 45
4.2.1 Benchmark Network Suite . . . 47
4.3 x86-64: Implicit GEMM vs. im2col . . . 47
4.3.1 Multi-Network x86-64 AVX2 Performance . . . 48
4.3.2 x86-64 AVX-512: vec_n Correctness Fix and Filter Packing (Cluster) . . . 50
4.4 RISC-V: Vector Length Agnostic Microbenchmark . . . 52
4.5 RISC-V: oneDNN Integration Performance . . . 53
4.5.1 Three-way Multi-Network Comparison at VLEN=512 . . . 53
4.5.2 JIT vs. Inline Assembly Analysis . . . 54
4.5.3 Speedup Discrepancy Explanation . . . 54
4.6 Multi-VLEN Performance Analysis: Implicit GEMM . . . 55
4.6.1 Overall Scalability and Saturation Points . . . 55
4.6.2 Key Findings: VLEN-Dependent Optimization Strategy . . . 56
4.6.3 Vectorization Axis: K-dim vs N-dim . . . 57
4.6.4 JIT Degradation at Large VLEN is a Working-Set Effect . . . 58
4.6.5 VLEN-Dependent Backend Selection . . . 59
4.7 RISC-V: oneDNN Winograd Convolution (Five-way Ablation) . . . 60
4.7.1 Cycle-level Results at VLEN=512 (gem5 `simTicks`) . . . 61
4.8 Real Hardware Validation: BananaPi-F3 . . . 62
4.8.1 Implicit GEMM: Three-way Comparison . . . 62
4.8.2 JIT Autotuning: Knob Sensitivity Analysis . . . 62
4.8.3 Winograd: Five-way Comparison on Real Hardware . . . 64
4.9 Memory Footprint . . . 65
4.10 Case Study: JIT FlashAttention . . . 65
4.10.1 Numerical Accuracy . . . 65
4.10.2 Polynomial-method comparison for $e^x$ . . . 65
4.10.3 Kernel Statistics . . . 66
4.10.4 Multi-VLEN Performance Evaluation . . . 66
4.11 Roofline Performance Analysis . . . 68
4.11.1 Peak Performance by VLEN . . . 68
4.11.2 Arithmetic Intensity Analysis . . . 68
4.11.3 Efficiency Analysis . . . 69
4.11.4 Roofline Findings Summary . . . 70
4.12 ARM NEON Performance (Apple Silicon) . . . 70
4.12.1 ARM NEON Micro-kernel Design . . . 71
4.12.2 Performance Results . . . 71
4.12.3 Cross-Architecture Comparison . . . 72
4.12.4 Analysis . . . 72
4.12.5 Multi-Network ARM NEON Performance . . . 73
4.12.6 Multi-Network Analysis . . . 73

**5 Discussion** **75**
5.1 Analysis of Performance Results . . . 75
5.1.1 x86-64 Performance Analysis . . . 75
5.1.2 Shape-Dependent Autotuning: TILE_M and Strategy Selection 75

5.1.3 Three Anomalies in the AVX-512 Optimization Waterfall . . . 76
5.1.4 Cross-Architecture Knob Taxonomy . . . 77
5.1.5 Lessons and Limitations of the AVX-512 Backend . . . 78
5.2 The Role of JIT in Algorithmic Viability . . . 78
5.2.1 From Memory Bottleneck to Integer Bottleneck . . . 78
5.2.2 Generation Cost vs Per-Call Saving . . . 79
5.2.3 Comparison with Other JITs . . . 79
5.3 RISC-V Vectorization: Challenges and Insights . . . 80
5.3.1 Interpreting the oneDNN Integration Results (VLEN=512) . . 80
5.3.2 The Cost of Agnosticism . . . 80
5.3.3 Reduction Latency . . . 81
5.3.4 VLA in Practice: One Binary, Two Orders of Magnitude of VLEN . . . 81
5.3.5 Real Hardware Validation: the 6% Gap . . . 82
5.3.6 Cross-architecture Validation Inside the RVV Backend . . . 82
5.3.7 Three Codegen Paths on the Same ISA . . . 82
5.3.8 gem5 as a Design Exploration Tool for Emerging Vector ISAs 83
5.4 Comparison with Related Work . . . 84
5.4.1 Indirect Convolution and Memory Overhead . . . 84
5.4.2 Comparison with Winograd . . . 84
5.4.3 Why Winograd RVV/JIT Gains are Small in Our gem5 Study 84
5.5 Case Study: FlashAttention Results . . . 85
5.5.1 Numerical Precision . . . 85
5.5.2 Choice of polynomial method for $e^x$ . . . 85
5.5.3 Framework Generality . . . 86
5.6 Limitations and Future Work . . . 86
5.6.1 Real Hardware Validation . . . 86
5.6.2 Support for Lower Precision . . . 86
5.6.3 Per-VLEN Panel Re-tiling . . . 86
5.6.4 Threats to Validity (Simulation and Benchmarking) . . . 87
5.7 Conclusion . . . 87

**6 Conclusion** **88**
6.1 Summary of Contributions . . . 88
6.1.1 Implicit GEMM Convolution Core (Primary) . . . 88
6.1.2 x86-64 Optimization (Baseline and Validation) . . . 88
6.1.3 x86-64 AVX-512 with Autotuning (Wider SIMD Validation) . 89
6.1.4 RISC-V Vector Implementation (Primary) . . . 89
6.1.5 Winograd Convolution on RISC-V Vector (Ablation Baseline) 90
6.1.6 JIT FlashAttention for LLM Inference (Primary) . . . 90
6.1.7 Framework Integration . . . 90
6.2 Key Findings . . . 90
6.3 Implications . . . 92
6.4 Future Directions . . . 92
6.5 Closing Remarks . . . 93

**Bibliography** **94**

**A Implementation Code Excerpts** **I**
A.1 x86-64 AVX2 Micro-kernel . . . . . I
A.2 RVV Dot Product . . . . . II
A.3 Implicit Coordinate Computation . . . . . II
A.4 oneDNN Primitive Registration . . . . . III

**B Experimental Commands** **IV**
B.1 Building oneDNN for RISC-V . . . . . IV
B.2 Running gem5 Simulation . . . . . IV
B.3 x86-64 Benchmark . . . . . V

# List of Figures

1.1 System overview. Workloads (top-left) are dispatched by oneDNN to one of six implementations (red boxes indicate JIT-generated paths), each evaluated on the rightmost hardware or simulator. . . . . 5

4.1 Multi-network throughput across the four CPU vector backends evaluated in this thesis. Each panel is averaged across all benchmarked layers of a network. . . . . 49

4.2 Cumulative speedup of the AVX-512 Implicit GEMM pipeline across seven optimization stages, averaged over five representative cross-network layers (VGG-16 `conv2_1` at 112×112, 64→128 and `conv4_1` at 28×28, 256→512; ResNet-50 `res2_3x3` at 56×56, 64→64 and `res2_1x1e` at 56×56, 64→256; MobileNet-V2 `expand_1x1` at 112×112, 16→96). Stages: L1 naive scalar without auto-vectorization (`-fno-tree-vectorize`), L2 the same scalar code with auto-vectorization enabled (`-O3` default), L3 manual AVX-512 `vec_n` on the unpacked KRSC filter, L4 `vec_n` with filter packing, L5 2D register blocking (`regblock<MR,NR>`), L6 plus NUMA binding, L7 plus the TILE_M-aware autotuner. The pipeline reaches an end-to-end cumulative speedup of **122.6**× from L1 to L7 (4.4 GFLOPS to 536.4 GFLOPS averaged over the five layers). Three transitions are non-monotonic and are analyzed in Section 5.1.3: L1→L2 auto-vectorization drops the cumulative speedup from 1.0× to 0.8×, L3→L4 filter packing drops it from 8.9× to 7.9×, and L5→L6 NUMA binding drops it from 23.3× to 19.5×. . . . . 51

4.3 VLEN sweep on gem5 for JIT FlashAttention (N=128, D=64), six configurations from 128 to 8192 bits (lower is better). JIT is the fastest implementation up to VLEN=512; RVV intrinsics overtake at VLEN $\geq$ 1024 and saturate near VLEN=8192 at $\sim$ 4.7× over scalar. Increasing vector width past this crossover does not yield monotonic gains once the per-iteration vector already exceeds the kernel's working set. The Implicit GEMM counterpart of this sweep is analyzed in Section 4.6.3 (Figure 4.4). . . . . 56

4.4 Implicit GEMM throughput across VLEN for the two vectorization axes. Both K-axis kernels (JIT-K, asm-K) increase monotonically with VLEN; the N-axis kernel (JIT-N) peaks near VLEN=512–1024 and then drops as its packed panel exceeds L2. Holding the code generator fixed (JIT-K vs JIT-N) shows the difference follows the axis, not the JIT; holding the axis fixed (JIT-K vs asm-K) shows the K-axis trend reproduces across code generators. . . . . 58
4.5 JIT Implicit GEMM throughput across VLEN with an un-capped packed panel versus an L2-aware capped panel (`cap32`). The un-capped panel grows with VLEN and exceeds L2 beyond VLEN=1024; capping the panel keeps it L2-resident and maintains approximately constant throughput across the sweep. The result indicates that the wide-VLEN drop in this implementation is a per-VLEN tiling effect rather than a limitation of runtime code generation. . . . . 59
4.6 Winograd $F(4, 3)$ five-way ablation, simulation vs. real hardware. Panel (a) gem5 simTicks at VLEN=512 (lower is better, total over 31 layers across VGG-16, ResNet-18, ResNet-50): most of the gain comes from the Winograd algorithm itself, while the four Winograd variants are nearly indistinguishable under the in-order MinorCPU model. Panel (b) BananaPi-F3 GFLOPS at VLEN=256 on real out-of-order hardware: RVV-vectorized transforms now show a clear +75% benefit on VGG-16, while the JIT GEMM backend designed for Implicit GEMM does not match OpenBLAS on Winograd's tile-shaped GEMMs. 61
4.7 Autotuner picks across two architectures. Panel (a) RVV (BananaPi-F3, VLEN=256, 13 layers, knobs {MR, LMUL}); Panel (b) AVX-512 (Xeon Gold 6548N, 20 representative layers, knobs {MR, NR}). Each row is a layer; cell color encodes the speedup of the selected configuration. Red boxes mark the MR=8 columns: nearly every layer on both sides selects MR=8, the only autotuner knob that is portable across the two architectures. . . . . 63
4.8 Roofline model under gem5 parameters. Bandwidth ceilings (L1/L2/DRAM, solid sloped lines) intersect compute ceilings $P_{\text{peak}}(\text{VLEN})$ for the six evaluated VLEN configurations (dashed horizontal lines). Measured kernel points (Implicit GEMM and FlashAttention at multiple VLENs) are plotted at their arithmetic intensity. All measured points lie well to the right of every ridge point, confirming that the kernels are compute-bound under these bandwidth assumptions. . . . . 70
4.9 Cross-architecture comparison split into two views. Panel (a): absolute peak GFLOPS, comparable across rows. Panel (b): relative speedup over each platform's local baseline; bar color encodes whether the baseline is a vendor library or a scalar reference, so the speedup ratios are *not* directly comparable across rows. . . . . 72

# List of Tables


1.1 Claim-to-evidence map. . . . . 4

2.1 Benchmark $3 \times 3$ convolution layers used throughout this work. All layers use `stride=1`, `pad=1`, and a $3 \times 3$ filter ($R = S = 3$). $H \times W$ is the input spatial size, $C_{\text{in}}$ the input channel count, and $C_{\text{out}}$ the output channel count. . . . . 7
2.2 Comparison of Convolution Implementation Strategies . . . . . 10
2.3 Winograd $F(4, 3)$ Computation Stages . . . . . 11
2.4 Two polynomial methods for $e^x$, evaluated on the softmax input range $x \in [-10, 0]$ (post max-trick). Each row reports the maximum relative error observed on a dense grid of the interval. The Taylor rows use the textbook origin-centered form without range reduction; the Cephes row combines range reduction $x = n \ln 2 + g$ with a degree-5 minimax polynomial on $|g| \leq \ln 2/2$. . . . . 14

3.1 Micro-kernel Tile Sizes Across Different VLENs . . . . . 21
3.2 Register Allocation in RVV Inline Assembly Micro-kernel . . . . . 23
3.3 LMUL register budget for an 8-row tile ($(\text{MR} + 1) \cdot \text{LMUL}$ vectors) . 25
3.4 Comparison of Three RVV Implicit GEMM Implementations . . . . . 25
3.5 Five AVX-512 vectorization strategies for Implicit GEMM. "Loop order" is the three-deep nest after the M-loop; "Acc. location" indicates whether the K-loop reduction lives in a ZMM register or in an L1-resident buffer; "FMAs/inner step" counts the number of independent 16-wide FMAs issued per innermost iteration; "Inputs" lists which operands are reloaded inside the inner loop. . . . . 26
3.6 Register-budget analysis for the $(\text{MR}, \text{NR})$ candidates considered by the autotuner. "Accs." is the number of ZMM registers used by the accumulator panel; "Total ZMM" assumes $c_{\text{tmp}} = 1$. "Picks" counts how often the autotuner selects this $(\text{MR}, \text{NR})$ across the 28 evaluated layer shapes (the remaining two layers select `vec_k` rather than `regblock`, see Section 3.2.3.1). ZMMs above 32 spill to memory and are pruned at search-space generation. . . . . 29
3.7 Eleven precompiled function instantiations exposed by the AVX-512 backend. "Pick rule" indicates how the runtime dispatch in `autotuner.hpp` maps an autotuner-chosen `ConvConfig` to one of these functions. . . . . 31

3.8 RVV Winograd Transform Memory Access Patterns . . . . . 41
3.9 Five-Way Winograd Ablation Study . . . . . 41

4.1 x86-64 Performance Comparison (VGG16, GFLOPS). All layers use a $3 \times 3$ filter with stride = 1, padding = 1. . . . . 47
4.2 x86-64 AVX2 Multi-Network Performance (Representative Layers) . . 48
4.3 RISC-V gem5 Performance on VGG16 layers (GFLOPS, VLEN=8192). All layers use a $3 \times 3$ filter with stride = 1, padding = 1; spatial sizes follow the 1/4-resolution layer set used for gem5 throughout this work. 52
4.4 oneDNN Integration: Three-way Performance Comparison on VGG16 layers (VLEN=8192 bits, gem5). All layers use a $3 \times 3$ filter with stride = 1, padding = 1. . . . . 54
4.5 Highest-throughput implementation by VLEN range (gem5 simulation results) . . . . . 59
4.6 BananaPi-F3 Implicit GEMM (VLEN=256, single-threaded). Average GFLOPS per network. . . . . 62
4.7 RVV autotuner picks on BananaPi-F3 (VLEN=256), annotated with the layer shape and the GEMM-equivalent $(M, K)$ that the micro-kernel sees. "Regime" classifies the layer into the shape regimes introduced in Section 4.3.1: B = balanced, N = narrow-mobile, D = deep-narrow. . . . . 63
4.8 FlashAttention Precision Validation (N=4, D=4) . . . . . 65
4.9 Two polynomial methods for $e^x$ on the softmax range $[-10, 0]$. Taylor entries use the textbook origin-centered form (no range reduction); the Cephes entry uses range reduction $x = n \ln 2 + g$ and a 5th-order minimax polynomial on $|g| \leq \ln 2/2$. The $x = 1$ column is informational—it is not the range that matters for softmax. . . . . 66
4.10 Theoretical Peak Performance by VLEN . . . . . 68
4.11 Arithmetic Intensity of Key Kernels . . . . . 68
4.12 Roofline Ridge Points $AI_{\text{ridge}} = P_{\text{peak}}/BW$ (FLOPs/Byte) Across VLEN and Memory Levels . . . . . 69
4.13 Implicit GEMM Efficiency by VLEN . . . . . 71
4.14 Micro-kernel Design Comparison Across Architectures . . . . . 71
4.15 ARM NEON Performance on Apple Silicon (VGG-style Layers) . . . 71
4.16 ARM NEON Multi-Network Performance on Apple Silicon (Representative Layers) . . . . . 73

5.1 Implicit GEMM vs Winograd Convolution . . . . . 84
5.2 FlashAttention Numerical Accuracy (gem5 simulation) . . . . . 85

# 1

# Introduction

Convolutional Neural Networks (CNNs) and Transformer-based models dominate modern inference workloads. On CPUs, performance is often limited not only by raw arithmetic throughput but also by data movement and the ability to generate tight inner loops for kernel-shaped computations such as convolution and attention.

While GPUs dominate training, CPU inference remains critical in many production settings due to cost, deployment simplicity, and tight integration with general-purpose systems software. This thesis presents a lightweight JIT code generation framework that combines hardware-aware kernel optimization with an automatic autotuner, and that targets multiple modern CPU vector ISAs from a single design: x86-64 (AVX2 and AVX-512) [1], ARM NEON, and the RISC-V Vector Extension (RVV) [2]. The framework is built on top of Intel oneDNN [3] and exercises both its JIT and its ahead-of-time (AOT) code paths. We provide and study our Implicit GEMM algorithmic implementations in depth on x86-64 and RVV, the two ISAs this thesis focuses on; the ARM NEON backend is additionally implemented as a portability showcase, demonstrating that the same single design carries over to a third mature SIMD ISA.

The thesis is organized around three main aspects:

- **JIT-based optimization of *Implicit GEMM convolution* with cross-architecture portability** across x86-64, ARM NEON, and RVV. A single Implicit GEMM design is specialized at runtime into three production-grade backends, integrated into oneDNN, and evaluated on both standard convolutions and the depthwise variant used by mobile networks.
- **A lightweight cross-architecture *autotuner*** that operates on a small, portable set of tile and unroll knobs (e.g., $M_R$, LMUL, $k$-unroll, $N_R$) with register-budget and channel-count pruning. Running the same autotuner skeleton on RVV and AVX-512 lets us identify which tuning decisions transfer across architectures and which are hardware-specific.
- ***FlashAttention* as a non-CNN case study** that evaluates the framework on a numerically sensitive, non-convolutional workload. The same JIT infrastructure emits a vectorized online softmax with a high-accuracy exp approximation, showing that the code-generation pipeline extends from CNN convolution to Transformer-style attention without redesign.

We additionally include an RVV *Winograd* $F(4,3)$ five-way ablation to separate algorithmic gains from implementation-level gains. Together, these components form a cross-architecture, multi-workload evaluation of JIT-based CPU kernel generation rather than an isolated RVV exercise.

## 1.1 The Memory Bottleneck

The conventional high-performance CPU approach in many libraries is to lower convolution into matrix multiplication via `im2col` followed by a BLAS GEMM call (e.g., popularized in early CPU/GPU frameworks such as Caffe [4]). While GEMM kernels are highly optimized, `im2col` materializes a large temporary matrix that can dominate memory traffic, inflate the working set, and degrade cache behavior—especially in early CNN layers with large spatial dimensions. In the common stride-1 case, the lowered matrix can be up to $K_H \times K_W$ times larger than the original feature map due to overlap-driven duplication, and explicit lowering can introduce both memory and runtime overhead [5].

Implicit GEMM [6] avoids explicit `im2col` expansion by computing input patch coordinates *on-the-fly* inside the GEMM loop nest. This reduces memory footprint and can substantially improve performance for memory-bound layers, but it introduces new compute overhead in the form of dynamic index/coordinate calculations. This trade-off is reflected in production libraries: oneDNN documents Implicit GEMM as a convolution fallback that reinterprets convolution as GEMM via scratchpad rearrangement [7]. Related work, such as MEC targets the same memory-overhead bottleneck by compact lowering and parallel small GEMMs [8], while optimized direct convolution demonstrates that eliminating auxiliary memory can improve both performance and scalability on CPUs [9].

## 1.2 The Compute Challenge: Why Implicit GEMM needs JIT

While Implicit GEMM solves the memory bottleneck, a naive formulation introduces a new challenge: coordinate computation overhead. If the mapping from a linear index $k$ to tensor coordinates $(c, r, s)$ is performed inside the compute loop, every step requires integer division and modulo operations, which are expensive on modern CPUs (often 10–20× slower than floating-point math). We address this with the standard packing structure: the coordinate mapping is performed once when gathering each input panel and is then amortized over the matrix multiplication, so the steady-state micro-kernel contains no integer index arithmetic.

This leaves a second question: how to obtain an efficient micro-kernel for each layer shape and vector length. A static compiler (like GCC or Clang) cannot fully specialize the kernel, because the tensor dimensions $(N, H, W, C)$ and, on RVV, the vector length are only known at runtime.

Just-In-Time (JIT) compilation addresses this through runtime specialization. By

generating the micro-kernel at runtime, we can:

1. Specialize per shape and vector length: emit a kernel tailored to the current layer and VLEN without recompilation, which is the property that lets a single RVV binary run across the VLEN range.
2. Loop unrolling: unroll the reduction loop based on the exact kernel size, reducing loop-control overhead.
3. Register allocation: fix register assignments for a specific micro-kernel tile (e.g., $6 \times 16$ on AVX-512), maximizing register reuse without spill/fill code.

Thus, this thesis presents both an algorithmic choice (Implicit GEMM) and a system-level implementation strategy: a micro-kernel JIT designed to make Implicit GEMM practical across x86-64, ARM NEON, and RVV from a single design, with extensions to a non-CNN case study (FlashAttention) and a Winograd ablation.

## 1.3 Goals and Contributions

**Goal.** Develop and evaluate a lightweight JIT-based code generation framework that combines hardware-aware kernel optimization with an automatic autotuner, and assess whether the same framework delivers practical CPU speedups across three modern vector ISAs (x86-64, ARM NEON, RISC-V Vector) and across two structurally different workloads (Implicit GEMM convolution and FlashAttention attention).

**Contributions.**

1. **Cross-ISA Implicit GEMM engine integrated into oneDNN.** A single Implicit GEMM design realized as three production-grade backends—x86-64 (AVX2 and AVX-512), ARM NEON, and RISC-V Vector (RVV)—and integrated into oneDNN through both Just-in-Time (JIT) and ahead-of-time (AOT) code paths. The engine covers both standard $3 \times 3$ convolutions and the depthwise variant, with OpenMP thread-level parallelization.
2. **x86-64 backends (AVX2 / AVX-512).** High-throughput AVX2 and AVX-512 implementations integrated into oneDNN, including five vectorization strategies, two-dimensional register blocking against the 32-ZMM file, and a packed filter layout that removes a layout-induced correctness pitfall. The same effort covers the depthwise variant used in MobileNet-V2 and RegNetY, reaching 20–56× over scalar across nine evaluated layers.[1]
3. **Vector Length Agnostic (VLA) RVV backend.** A custom RVV micro-kernel JIT that emits VLA machine code, scaling without rebuild from VLEN=128 to VLEN=8192 bits. The same kernel is rigorously simulated across six VLEN configurations on gem5 and validated on real hardware (BananaPi-F3 development board, VLEN=256).

[1] Detailed discussion of the strategies, register blocking, packing, and padding dispatch is presented in Chapter 3 (Methodology), with results in Chapter 4.

Table 1.1: Claim-to-evidence map.

| Claim | Evidence (where to look) |
|---|---|
| C1: Implicit GEMM reduces memory bottlenecks and improves memory-bound layers. | Table/figure: x86-64 speedup vs oneDNN im2col; memory-footprint analysis. |
| C2: JIT specialization removes index-math overhead in Implicit GEMM. | Microbenchmark or kernel-instruction analysis; JIT vs non-JIT comparisons. |
| C3: RVV VLA kernels provide portable speedups; best backend depends on VLEN. | Multi-VLEN sweep; oneDNN integration results under gem5. |
| C4: The JIT framework generalizes beyond convolution (FlashAttention). | FlashAttention correctness and performance vs VLEN; numerical error tables. |
| C5: Winograd provides algorithmic gains; backend differences may be small under simulation constraints. | Five-way Winograd ablation (Direct-Conv vs Winograd variants) and limitation analysis. |

4. **Lightweight cross-architecture autotuner.** An autotuner over a small, portable knob set ($M_R$, $N_R$, $k$-unroll, LMUL, TILE_M) with register-budget and channel-count pruning, and per-shape JSON-cached results to amortize tuning cost across inferences. Its most effective single knob, TILE_M, alone contributes a 1.8–2.4× end-to-end uplift on five representative networks. Running the same skeleton on RVV and AVX-512 yields a knob taxonomy that separates architecture-portable parameters from hardware-specific ones.

5. **FlashAttention as non-CNN case study (RVV).** A complete JIT FlashAttention kernel for RVV that evaluates the framework beyond convolutional workloads. The kernel implements an online softmax with the max-trick for numerical stability and uses a Cephes-style minimax polynomial exp approximation in place of `libm`; this evaluates the JIT on numerically sensitive control flow and shows that the same code-generation pipeline extends from CNN convolution to Transformer-style attention.

6. **RVV Winograd $F(4, 3)$ five-way ablation.** A Winograd implementation, adapted from the open-source `convWinograd` library of Dolz et al. [10] and ported to RVV, evaluated with a five-way ablation that separates algorithmic gains (Winograd over direct convolution) from implementation-level gains (vectorized transforms, custom GEMM, JIT-generated GEMM), reusing the same $8 \times$ VL JIT micro-kernel built for Implicit GEMM.

## 1.4 Claims and Evaluation Roadmap

Following common empirical-evaluation practice in systems and compiler optimization work, we structure the evaluation around a small set of explicit claims and map each claim to concrete evidence (tables/figures). Each optimization is motivated by a hypothesis, tested by an experiment, and interpreted with limitations.

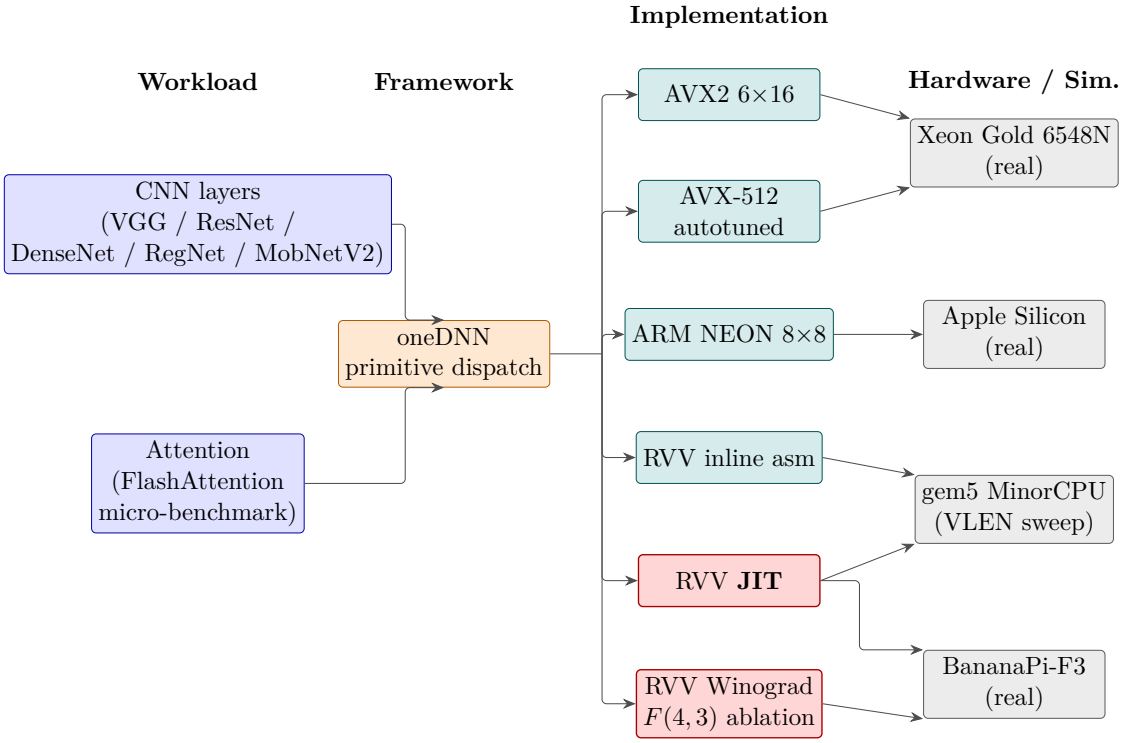


Figure 1.1: System overview. Workloads (top-left) are dispatched by oneDNN to one of six implementations (red boxes indicate JIT-generated paths), each evaluated on the rightmost hardware or simulator.

## 1.5 Thesis Organization

Chapter 2 reviews background on convolution, Implicit GEMM, and vector architectures. Chapter 3 describes the JIT framework and the x86-64/RVV kernel implementations and their integration into oneDNN. Chapter 4 presents experimental results with a consistent evaluation protocol and discusses limitations. Chapter 5 concludes with key findings and future directions.

# 2

# Background

This chapter provides the theoretical foundations necessary for understanding Implicit GEMM convolution, including CNN fundamentals, convolution algorithms, and the target hardware architectures.

## 2.1 Convolutional Neural Networks

Convolutional Neural Networks are a class of deep neural networks particularly effective for processing structured grid data such as images. A typical CNN consists of multiple layers including convolutional layers, pooling layers, and fully connected layers.

### 2.1.1 Convolution Operation

The discrete 2D convolution operation for a single output channel can be expressed as:

$$O[n,k,p,q] = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} I[n,c,h,w] \cdot F[k,c,r,s] \tag{2.1}$$

where:

- $I \in \mathbb{R}^{N \times C \times H \times W}$ is the input tensor
- $F \in \mathbb{R}^{K \times C \times R \times S}$ is the filter tensor
- $O \in \mathbb{R}^{N \times K \times P \times Q}$ is the output tensor
- $h = p \cdot \text{stride}_h - \text{pad}_h + r$
- $w = q \cdot \text{stride}_w - \text{pad}_w + s$

The computational complexity is $O(N \cdot K \cdot P \cdot Q \cdot C \cdot R \cdot S)$, which can be expressed as $2 \cdot N \cdot K \cdot P \cdot Q \cdot C \cdot R \cdot S$ floating-point operations (multiply-accumulate counted as 2 operations).

Throughout this thesis, tensors are stored in the NCHW layout (batch, channel, height, width), consistent with the indexing used above; all kernel implementations and memory-layout discussion in the following chapters assume this layout.

### 2.1.2 VGG16 Architecture

VGG16 [11] is a deep CNN architecture consisting of 16 weight layers, primarily using $3 \times 3$ convolution filters. Its uniform architecture makes it an ideal benchmark for convolution optimization studies. The network processes $224 \times 224$ RGB images through five convolutional blocks with increasing channel depths (64, 128, 256, 512, 512).

### 2.1.3 Benchmark Layer Inventory

Throughout this thesis we evaluate the kernels on representative $3 \times 3$ convolution layers drawn from VGG16, ResNet-18, and ResNet-50. Table 2.1 lists the exact shape of each evaluated layer; it is used as a reference both for the multi-network results in Chapter 4 and for interpreting the autotuner picks (large vs. small $K$, large vs. small $H \times W$). All layers use stride = 1 and padding = 1, which are also the conditions under which Winograd $F(4, 3)$ is applicable. Spatial dimensions for VGG16 are reported at the 1/4 resolution actually used in our gem5 runs (input $56 \times 56$); for ResNet-18 and ResNet-50 we follow the canonical block shapes after the initial stride-2 stem.

Table 2.1: Benchmark $3 \times 3$ convolution layers used throughout this work. All layers use `stride=1`, `pad=1`, and a $3 \times 3$ filter ($R = S = 3$). $H \times W$ is the input spatial size, $C_{\text{in}}$ the input channel count, and $C_{\text{out}}$ the output channel count.

| Network | Layer | $H \times W$ | $C_{\text{in}}$ | $C_{\text{out}}$ | Notes |
|---|---|---|---|---|---|
| VGG16 | conv1_2 | $56 \times 56$ | 64 | 64 | shallow, narrow channels |
| VGG16 | conv2_1 | $28 \times 28$ | 64 | 128 | channel growth |
| VGG16 | conv2_2 | $28 \times 28$ | 128 | 128 | balanced |
| VGG16 | conv3_1 | $14 \times 14$ | 128 | 256 | channel growth |
| VGG16 | conv3_2 | $14 \times 14$ | 256 | 256 | balanced |
| VGG16 | conv3_3 | $14 \times 14$ | 256 | 256 | balanced |
| VGG16 | conv4_1 | $7 \times 7$ | 256 | 512 | deep, small spatial |
| VGG16 | conv4_2 | $7 \times 7$ | 512 | 512 | deep, small spatial |
| VGG16 | conv4_3 | $7 \times 7$ | 512 | 512 | deep, small spatial |
| VGG16 | conv5_1 | $7 \times 7$ | 512 | 512 | deepest stage |
| ResNet-18 | layer1.0 | $56 \times 56$ | 64 | 64 | basic block, shallow |
| ResNet-18 | layer2.0 | $28 \times 28$ | 128 | 128 | basic block |
| ResNet-18 | layer3.0 | $14 \times 14$ | 256 | 256 | basic block |
| ResNet-18 | layer4.0 | $7 \times 7$ | 512 | 512 | basic block, deep |
| ResNet-50 | layer1.0 (3×3) | $56 \times 56$ | 64 | 64 | bottleneck middle |
| ResNet-50 | layer2.0 (3×3) | $28 \times 28$ | 128 | 128 | bottleneck middle |
| ResNet-50 | layer3.0 (3×3) | $14 \times 14$ | 256 | 256 | bottleneck middle |
| ResNet-50 | layer4.0 (3×3) | $7 \times 7$ | 512 | 512 | bottleneck middle, deep |

The layers cluster into three regimes: (i) *shallow / narrow* ($C_{\text{in}}, C_{\text{out}} \leq 128$, large $H \times W$), where memory traffic per FMA is high; (ii) *balanced* ($C \approx 256$, $H \times W \approx 14^2$),

which is the regime in which the autotuner has the most freedom; and (iii) *deep / small-spatial* ($C \geq 512$, $H \times W \leq 7^2$), where the inner $K$ dimension dominates and tile-size choices are constrained by register pressure rather than by cache. We refer back to this classification when discussing autotuner picks in Section 4.8.2.

## 2.2 Convolution Implementation Strategies

A convolution can be realized through several distinct strategies that trade memory, arithmetic, and access-pattern regularity against one another. This thesis considers four: direct convolution, im2col + GEMM, Implicit GEMM, and the Winograd minimal-filtering algorithm; FFT-based convolution [12] is a further option that becomes competitive only for large filters and is not pursued here. Direct convolution and highly optimized direct kernels are already well studied [9], so the contribution of this thesis is not direct convolution itself but a portable Implicit GEMM formulation driven by a micro-kernel JIT, which retains a tuned GEMM micro-kernel while avoiding the im2col buffer.

### 2.2.1 Direct Convolution

Direct convolution computes each output element in place, accumulating the products of the input receptive field with the filter weights through a loop nest over the output positions and the reduction dimensions $(C, R, S)$, with no intermediate buffer or matrix reshaping:

$$O[n, k, p, q] = \sum_{c,r,s} I[n, c, h, w] \cdot W[k, c, r, s], \tag{2.2}$$

where $h = p \cdot \text{stride}_h - \text{pad}_h + r$ and $w = q \cdot \text{stride}_w - \text{pad}_w + s$. It requires zero auxiliary memory and, with careful loop ordering and vectorization, reaches high throughput on multi-core CPUs [9]. Its drawback is that the loop nest follows the convolution geometry rather than the regular dense-matrix structure that hardware GEMM micro-kernels are tuned for, so approaching peak performance requires tuning the schedule per layer shape. The strategies that follow keep this zero-buffer memory advantage while recovering the regularity of a GEMM micro-kernel.

### 2.2.2 im2col + GEMM

The im2col (image to column) transformation [13] rearranges input patches into columns of a matrix, converting convolution into matrix multiplication:

$$\text{Output} = \text{Filter}_{\text{reshaped}} \times \text{im2col}(\text{Input}) \tag{2.3}$$

The transformation creates a matrix of size $(C \cdot R \cdot S) \times (N \cdot P \cdot Q)$, representing all input patches required for the convolution. While this enables use of highly optimized BLAS GEMM routines, the memory overhead is significant:

$$\text{Memory}_{\text{im2col}} = N \cdot P \cdot Q \cdot C \cdot R \cdot S \cdot \text{sizeof(float)} \tag{2.4}$$

For VGG16 conv2_1 layer ($112 \times 112 \times 64 \rightarrow 128$, $3 \times 3$ filter), this expansion requires approximately 27.56 MB of additional memory.

### 2.2.3 Implicit GEMM

Implicit GEMM [6] eliminates the im2col memory overhead by computing input coordinates on-the-fly. The central idea is that the im2col matrix can be represented implicitly:

$$A[m,k] = \begin{cases} I[n,c,h,w] & \text{if } 0 \le h < H \text{ and } 0 \le w < W \\ 0 & \text{otherwise (padding)} \end{cases} \tag{2.5}$$

where the indices are computed from:

$$n = m/(P \cdot Q) \tag{2.6}$$
$$p = (m \mod (P \cdot Q))/Q \tag{2.7}$$
$$q = m \mod Q \tag{2.8}$$
$$c = k/(R \cdot S) \tag{2.9}$$
$$r = (k \mod (R \cdot S))/S \tag{2.10}$$
$$s = k \mod S \tag{2.11}$$
$$h = p \cdot \text{stride}_h - \text{pad}_h + r \tag{2.12}$$
$$w = q \cdot \text{stride}_w - \text{pad}_w + s \tag{2.13}$$

Unlike direct convolution, which iterates in the convolution's own geometry, Implicit GEMM presents the same computation to a dense GEMM micro-kernel and therefore inherits register blocking, filter packing, and BLAS-style scheduling, while the on-the-fly index arithmetic above replaces the materialized im2col buffer. This combination—GEMM micro-kernel efficiency without im2col memory—is what distinguishes it from both im2col + GEMM and direct convolution, and is the formulation this thesis makes portable through the micro-kernel JIT.

### 2.2.4 Comparison of Approaches

The two GEMM-based strategies differ mainly in how they obtain the lowered matrix: im2col + GEMM materializes it explicitly, whereas Implicit GEMM reconstructs each entry on demand. Table 2.2 summarizes the resulting trade-offs in memory footprint, bandwidth, and per-element compute.

### 2.2.5 Winograd Convolution

Winograd's minimal filtering algorithm [14] reduces the arithmetic complexity of small convolutions by transforming the problem into the Winograd domain, performing

Table 2.2: Comparison of Convolution Implementation Strategies

| **Aspect** | **im2col + GEMM** | **Implicit GEMM** |
|---|---|---|
| Memory Overhead | $O(N \cdot P \cdot Q \cdot C \cdot R \cdot S)$ | $O(1)$ |
| Memory Bandwidth | $2\times$ (read input, read im2col) | $1\times$ |
| Computation Overhead | None | Coordinate computation |
| Flexibility | Fixed after transform | Dynamic indexing |
| Cache Efficiency | Good (contiguous access) | Requires careful design |

element-wise multiplication, and transforming back. Lavin and Gray [15] adapted this algorithm for deep neural networks, demonstrating significant speedups for $3 \times 3$ convolutions.

#### 2.2.5.1 Winograd $F(m, r)$ Algorithm

For a 1D convolution of an $m$-element output with an $r$-tap filter, the direct computation requires $m \cdot r$ multiplications. The Winograd algorithm reduces this to $m + r - 1$ multiplications at the cost of additional additions. For 2D convolution with $F(m \times m, r \times r)$, the computation becomes:

$$Y = A^T \left[ (G \cdot F \cdot G^T) \odot (B^T \cdot d \cdot B) \right] A \tag{2.14}$$

where:

- $F \in \mathbb{R}^{r \times r}$ is the filter
- $d \in \mathbb{R}^{(m+r-1) \times (m+r-1)}$ is the input tile
- $Y \in \mathbb{R}^{m \times m}$ is the output tile
- $G$, $B^T$, $A^T$ are constant transformation matrices
- $\odot$ denotes element-wise multiplication

#### 2.2.5.2 $F(4, 3)$ Variant for $3 \times 3$ Convolutions

We use $F(4, 3)$, which computes a $4 \times 4$ output tile from a $6 \times 6$ input tile with a $3 \times 3$ filter. The transformation matrices are:

$$G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix}, \quad B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \tag{2.15}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix} \tag{2.16}$$

#### 2.2.5.3 Computational Complexity

The four stages of Winograd convolution and their complexities are:

Table 2.3: Winograd $F(4,3)$ Computation Stages

| **Stage** | **Operation** | **Complexity** |
|---|---|---|
| Filter Transform | $U = G \cdot F \cdot G^T$ | $O(K \cdot C \cdot t^2)$ (one-time) |
| Input Transform | $V = B^T \cdot d \cdot B$ | $O(N_{\text{tiles}} \cdot C \cdot t^2)$ |
| Batched GEMM | $M[e,v] = U[e,v] \cdot V[e,v]$ | $O(t^2 \cdot K \cdot C \cdot N_{\text{tiles}})$ |
| Output Transform | $Y = A^T \cdot M \cdot A$ | $O(N_{\text{tiles}} \cdot K \cdot t \cdot m)$ |

where $t = m + r - 1 = 6$ is the tile size and $N_{\text{tiles}}$ is the number of spatial tiles. For $3 \times 3$ convolutions, Winograd $F(4,3)$ reduces the number of multiplications by a factor of $\frac{m^2 \cdot r^2}{(m+r-1)^2} = \frac{36}{36} = 1$ in the GEMM stage, but the GEMM operates on smaller matrices ($K \times N_{\text{tiles}}$ instead of $K \times (N \cdot P \cdot Q)$), and the total multiply count is reduced from $2 \cdot N \cdot K \cdot P \cdot Q \cdot C \cdot 9$ to approximately $2 \cdot 36 \cdot K \cdot C \cdot N_{\text{tiles}}$, yielding a theoretical speedup of $\frac{9 \cdot m^2}{(m+r-1)^2} = \frac{9 \cdot 16}{36} = 4\times$ for the GEMM-dominated regime.

## 2.3 FlashAttention

Self-attention is the central operation of Transformer models. Given query, key, and value matrices $Q, K, V \in \mathbb{R}^{N \times d}$ (where $N$ is the sequence length and $d$ is the head dimension), standard scaled dot-product attention computes

$$Y = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V. \tag{2.17}$$

**The IO bottleneck of standard attention.** A direct implementation of Eq. 2.17 materializes the full attention matrix $S = QK^\top \in \mathbb{R}^{N \times N}$ in memory. For long sequences this matrix dominates both memory footprint ($O(N^2)$) and memory traffic: it must be written after the first matmul, read by the softmax, written back, and finally read again by the second matmul. On CPUs this dominates the runtime well before arithmetic does.

**FlashAttention.** FlashAttention [16] avoids materializing $S$ by processing $K$ and $V$ in tiles and folding the softmax into the same loop. Concretely, the algorithm walks over the $N$ keys in blocks; for each block it computes a partial score, updates the output incrementally, and never stores the full $N \times N$ matrix. The result is mathematically identical to Eq. 2.17 but with $O(N)$ extra memory instead of $O(N^2)$ and a single pass over $K$ and $V$.

**Online softmax with the max-trick.** The challenge in this tiled formulation is the softmax: a naive softmax needs the global maximum and the global denominator, both of which require seeing the full row of scores. FlashAttention solves this with an online recurrence that maintains, after processing the first $j$ blocks, a running maximum $m^{(j)}$ and a running denominator $\ell^{(j)}$, and rescales the partial output $O^{(j)}$ whenever a new maximum is found:

$$m^{(j)} = \max\Big(m^{(j-1)},\ \max_i s_i^{(j)}\Big), \tag{2.18}$$

$$\ell^{(j)} = \ell^{(j-1)} \cdot e^{m^{(j-1)} - m^{(j)}} \ + \ \sum_i e^{s_i^{(j)} - m^{(j)}}, \tag{2.19}$$

$$O^{(j)} = O^{(j-1)} \cdot e^{m^{(j-1)} - m^{(j)}} \ + \ \sum_i e^{s_i^{(j)} - m^{(j)}} \cdot V_i^{(j)}. \tag{2.20}$$

After the last block this state equals exactly the row-wise softmax-times-value of Eq. 2.17. The subtraction by $m^{(j)}$ inside every exponent is the standard "max trick" that keeps $e^{(\cdot)}$ in a numerically safe range.

**Why `exp` accuracy matters here.** Every score in the inner loop produces an $e^{(\cdot)}$ call, and every block boundary multiplies the accumulator by another $e^{m^{(j-1)} - m^{(j)}}$ rescaling factor. Errors in these exponentials feed directly into both the denominator $\ell^{(j)}$ and the running output $O^{(j)}$ and accumulate linearly with sequence length. A cheap-but-inaccurate $e^x$ approximation therefore degrades end-to-end attention precision, which motivates the more careful approximation discussed in the next section.

## 2.4 Polynomial Approximation of $e^x$

Vector kernels evaluate $e^x$ many times per inner iteration (one per softmax score in FlashAttention). Calling `libm`'s `expf` is too expensive: it pays a function-call overhead and is not vectorized. We therefore inline a polynomial approximation. Two families of polynomial methods are in active use—truncated Taylor series and minimax (Remez) polynomials—and they make different trade-offs. The choice between them is what motivates our final design.

**Approach 1: Taylor series around 0.** The Taylor expansion of $e^x$ at the origin is

$$e^x \ \approx \ 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}. \tag{2.21}$$

This is the most direct polynomial method: it is simple to derive, trivial to evaluate, and remains the subject of active optimization work in the vector-math community (e.g., choice of expansion point, hybrid Taylor / Estrin schemes, table-augmented variants). For the FlashAttention setting, however, the unmodified version has two limitations:

- *Accuracy is concentrated near the expansion point.* A low-order truncation has its smallest error at the expansion point and worsens monotonically away from

it. For the 2nd-order truncation $1 + x + x^2/2$, the relative error is around 8% at $x = 1$, around 36% at $x = -1$, and exceeds 30% on either side of the origin by $|x| = 2$. The Lagrange remainder $|R_n(x)| \leq e^{|x|}\,|x|^{n+1}/(n+1)!$ predicts the trend; reaching $10^{-3}$ relative error over $[-1, 1]$ already requires degree $\sim 6$.

- *Used as-is, it does not cover the softmax range.* The max-trick keeps softmax inputs in $x \in [-T,\, 0]$ where $T$ scales with the per-row score spread (commonly $T \in [10, 87]$ for `float32`). Without an argument-reduction step, every fixed-degree Taylor truncation is dominated by its highest-order monomial as $|x|$ grows: at $x = -5$, the degree-2 polynomial evaluates to 8.5 while $e^{-5} \approx 6.7{\times}10^{-3}$. The polynomial is being asked to fit $e^x$ on the wrong interval; raising the degree alone narrows the error band near zero but does not close the gap on the full softmax range.

A Taylor approach can be made competitive by adding the same supporting transformations that minimax polynomials normally use—in particular, range reduction. In our setting we instead choose the minimax route directly, because the polynomial fit and the supporting transformations are already a matched pair in the Cephes reference implementation.

**Approach 2: minimax (Remez) polynomial with range reduction—the Cephes construction.** Production math libraries use a second polynomial method that decouples the two issues above. The argument is first reduced so that the polynomial only has to be accurate on a small, fixed interval, and the polynomial coefficients are then fitted to that interval to minimize the *worst-case* error rather than the error at one point. Concretely:

1. *Range reduction.* Decompose $x = n \cdot \ln(2) + g$ with integer $n$ chosen so that $|g| \leq \ln(2)/2$. Then $e^x = 2^n \cdot e^g$. The factor $2^n$ is built directly by writing $n$ into the exponent field of an IEEE-754 `float32` (essentially free).

2. *Minimax polynomial on the reduced argument.* On the small interval $[-\ln(2)/2,\, \ln(2)/2]$, $e^g$ is approximated by a fixed-degree polynomial $P(g) = c_0 + c_1 g + c_2 g^2 + \cdots + c_d g^d$. The coefficients are chosen by the Remez algorithm, which produces the polynomial that minimizes $\max_g |P(g) - e^g|$ over the interval (the *minimax* criterion). The Cephes library [17] ships such 5th- and 6th-order polynomials.

The combined evaluation is one shift, a handful of FMAs, and a bit-cast, with no branches. Table 2.4 contrasts the two polynomial methods on the input range that the softmax max-trick produces, in the form in which each is most commonly used (unmodified Taylor; range-reduced minimax).

The structural point the table makes is that the dominant source of error on the softmax range is not the polynomial fit itself but the absence of argument reduction: raising the Taylor degree from 2 to 6 narrows the error band near zero but still leaves the worst-case error orders of magnitude away from acceptable. Once range reduction maps every input into a small fixed interval, a moderate-degree minimax polynomial is sufficient to reach $\sim 10^{-5}$ over the full softmax range. We therefore use the Cephes-style polynomial method throughout this work. Taylor-based polynomial

Table 2.4: Two polynomial methods for $e^x$, evaluated on the softmax input range $x \in [-10, 0]$ (post max-trick). Each row reports the maximum relative error observed on a dense grid of the interval. The Taylor rows use the textbook origin-centered form without range reduction; the Cephes row combines range reduction $x = n \ln 2 + g$ with a degree-5 minimax polynomial on $|g| \leq \ln 2/2$.

| **Polynomial method** | **Degree** | **Range reduction** | **Max relative error on** $[-10, 0]$ |
|---|---|---|---|
| Taylor series $(1 + x + x^2/2)$ | 2 | no | $\gtrsim 10^6$ |
| Taylor series, degree 6 | 6 | no | $\gtrsim 10^2$ |
| Cephes: range reduction + degree-5 minimax polynomial | 5 | yes | $\sim 3 \times 10^{-5}$ |

schemes are not inherently unsuitable—with range reduction and a higher-degree fit they remain competitive, and recent vector-math work continues to refine them—but the range-reduced minimax form is the choice that best matches the accuracy budget of a FlashAttention kernel, where every softmax iteration evaluates $e^{(\cdot)}$ and the rescaling factor is reapplied at every block boundary.

## 2.5 GEMM Optimization Techniques

### 2.5.1 Cache Blocking (Tiling)

Modern CPUs have multi-level cache hierarchies (L1, L2, L3) with varying sizes and latencies. Cache blocking partitions matrices into tiles that fit in cache, maximizing data reuse. The blocking parameters $M_C$, $N_C$, and $K_C$ are chosen based on cache sizes:

- $M_C \times K_C$ block of matrix A fits in L2 cache
- $K_C \times N_R$ panel of matrix B fits in L1 cache
- $M_R \times N_R$ micro-tile of matrix C fits in registers

### 2.5.2 Micro-kernel Design

The innermost loop of GEMM is the micro-kernel, which computes a small tile of the output matrix. For x86-64 with AVX2, a typical micro-kernel size is $6 \times 16$:

- 6 rows fit in 12 YMM registers (2 registers per row for 16 columns)
- 16 columns = 2 AVX2 vectors (8 floats each)
- Achieves high register utilization and instruction-level parallelism

### 2.5.3 Filter Packing

To enable efficient SIMD access, filters are packed into a layout optimized for the micro-kernel:

$$\text{PackedFilter}[\text{tile}, k, \text{lane}] = \text{Filter}[\text{tile} \cdot N_R + \text{lane}, c, r, s] \tag{2.22}$$

This transformation is performed once and amortized across all convolutions with the same filter.

## 2.6 Hardware Architectures

### 2.6.1 x86-64 with AVX2 and AVX-512

AVX2 (Advanced Vector Extensions 2) extends x86-64 with 256-bit vector registers (YMM0-YMM15) and fused multiply-add (FMA) instructions [1]. Key characteristics:

- 16 YMM registers, each holding 8 single-precision floats
- `vfmadd231ps`: fused multiply-add, $a = a + b \times c$
- `vbroadcastss`: broadcast scalar to all vector lanes
- Throughput: up to 2 FMA operations per cycle

AVX-512 widens the same model to 512-bit registers (ZMM0-ZMM31), doubling both the SIMD width (16 single-precision floats per register) and the architectural register count. The thesis targets both extensions; the key characteristics above (FMA-based accumulation, scalar broadcast) carry over unchanged, and the larger ZMM file is what enables the wider $6 \times 16$ register-blocked micro-kernel discussed in Chapter 3.

### 2.6.2 RISC-V Vector Extension (RVV)

RVV [2] provides scalable vector processing with Vector Length Agnostic (VLA) programming. Key characteristics:

- VLEN: hardware vector length (implementation-defined, 128-8192 bits)
- VL: runtime vector length set by `vsetvli` instruction
- LMUL: vector length multiplier (1/8 to 8)
- `vfmacc.vf`: vector-scalar fused multiply-accumulate
- `vfredusum.vs`: vector reduction sum

The VLA model allows the same code to run efficiently on different hardware implementations:

$$\text{Elements per vector} = \frac{\text{VLEN} \times \text{LMUL}}{\text{SEW}} \tag{2.23}$$

## 2.7 oneDNN Framework

Intel oneDNN (oneAPI Deep Neural Network Library) [3] is a cross-platform performance library for deep learning applications. Key features:

- Primitive-based API: convolution, pooling, normalization, etc.

- JIT code generation for optimal performance
- Support for x86-64, AArch64, and emerging RISC-V platforms
- Memory format abstraction (NCHW, NHWC, blocked formats)

The primitive dispatch system allows multiple implementations to coexist, with runtime selection based on hardware capabilities and problem characteristics.

# 3

# Methodology and Implementation

This chapter describes how we implement and integrate a JIT-optimized kernel stack for CPU inference, with an emphasis on repeatable design decisions. Rather than presenting a collection of isolated tricks, we organize the methodology around the same questions that drive the evaluation in Chapter 4: (i) what bottleneck is being targeted, (ii) what code shape enables the hardware to exploit parallelism, and (iii) what trade-offs are introduced (portability, maintenance, and overhead).

The chapter has three parts. First, we present a lightweight micro-kernel JIT framework and explain why runtime specialization is essential for Implicit GEMM. Second, we describe our RVV (RISC-V Vector) implementation under a Vector Length Agnostic (VLA) model, including three backends (intrinsics, inline assembly, JIT) to expose the performance/portability spectrum. Third, we present two case studies that exercise different aspects of the same toolchain: FlashAttention (numerically sensitive control flow) and Winograd $F(4, 3)$ (transform-heavy convolution with a five-way ablation design).

## 3.1 JIT Code Generation Framework

Implicit GEMM avoids explicit `im2col` expansion, but a naive inline formulation shifts work into integer-heavy address calculations. We keep this address arithmetic out of the steady-state loop by computing it once in the packing step (Section 5.2.1), leaving a dense inner product over packed data. Our methodological choice is then to treat JIT compilation as an enabling mechanism for that inner product: we emit a register-blocked micro-kernel whose tile shape and register assignment are fixed in the instruction stream, so the steady-state execution resembles hand-tuned assembly while remaining portable across layer shapes and, on RVV, across vector lengths.

### 3.1.1 Direct Binary Emission

Our JIT engine emits machine code directly. It allocates executable memory pages using `mmap` (Linux) or `VirtualAlloc` (Windows) and writes x86-64 machine code bytes directly. This approach offers:

- Code generation latency that is small compared to typical inference repetition (amortizable per layer).

- Precise control over instruction selection and register usage for micro-kernel shapes.

### 3.1.2 Micro-kernel Structure Specialization

The specialization performed by the generator targets the structure of the steady-state micro-kernel, not the coordinate arithmetic. The input-coordinate mapping

$$\text{offset} = c \cdot (H \cdot W) + h \cdot W + w \tag{3.1}$$

is computed in the packing step that gathers each input panel, where the layer dimensions are runtime values; because the packed panel is reused across the output channels, this cost is amortized rather than repeated inside the reduction. The micro-kernel itself operates only on packed, unit-stride data and contains no coordinate arithmetic or integer `div`/`mod`.

What is fixed at generation time is the kernel's tile shape, register assignment, and unroll factor. The reduction over the packed $K$ panel is emitted as a straight-line sequence of broadcast-FMA instructions over the chosen tile (e.g., MR = 8 rows), so the inner body carries no loop control over the tile rows:

```
// Static C++ (tile loop):
// for (r = 0; r < MR; ++r) c[r] += a[r] * b;
// Emitted micro-kernel (MR=8, unrolled):
// vfmacc v0, fa0, vb ; ... ; vfmacc v7, fa7, vb
```

On RVV the same instruction sequence is vector-length agnostic, so a single emitted kernel runs across the VLEN range without regeneration; on x86-64 the equivalent specialization is obtained by compiling the tile shape as template parameters.

### 3.1.3 Micro-kernel Generation Pipeline

The generation process follows a pipeline:

1. Prologue generation: emit ABI-compliant stack setup and callee-saved register preservation.
2. Parameter loading: bind pointers to input ($A$), weights ($B$), and output ($C$) to fixed GP registers to avoid repeated address decoding in the hot loop.
3. Loop-nest generation:
   - Generates the $M, N, K$ loops.
   - For the innermost $K$ loop, checks if $K$ is small constant. If so, fully unrolls the loop to eliminate the `dec`/`jnz` instructions.
4. Instruction encoding: use a small instruction encoder (e.g., `emit_avx2`) to write VEX prefixes and ModR/M bytes for vector instructions.
5. Epilogue generation: restore stack and return.

## 3.2 Implicit GEMM

### 3.2.1 Algorithm Overview

Both our RVV and x86-64 backends realize the same Implicit GEMM algorithm. They differ in the micro-kernel shape, the tile and register-blocking parameters, and the data layout most natural for each ISA's vectorization model. We state the algorithm and the cross-architectural design decisions here, and defer the backend-specific code to the subsections that follow.

**Implicit indexing.** The defining property of Implicit GEMM is that the input matrix of the conceptual $C = A \cdot B$ is never materialized. Given linear GEMM coordinates $(m, k)$ inside the inner loop, the kernel computes the corresponding input pixel coordinates on-the-fly:

```
n = m / (P * Q);                                // batch index
p = (m % (P * Q)) / Q;                          // output row
q = m % Q;                                      // output column
c = k / (R * S);                                // input channel
r = (k % (R * S)) / S;                          // filter row offset
s = k % S;                                      // filter column offset
h = p * stride_h - pad_h + r * dilation_h;      // input row
w = q * stride_w - pad_w + s * dilation_w;      // input column
```

This eliminates the explicit `im2col` buffer at the cost of runtime division and modulo operations, which both backends mitigate specifically for their target ISA: on RVV through JIT-time constant specialization (Section 3.2.2), on x86-64 through compile-time C++ template specialization (Section 3.2.3). Both backends support dilated convolutions through the `dilation_h`, `dilation_w` factors above.

**Shared design decisions.** The two backends share three implementation principles, each detailed for its ISA in the corresponding subsection but introduced here as common vocabulary:

- **Filter packing.** The filter tensor is permuted from its native layout into a contiguous SIMD-friendly form, so that one vector load fetches the values that will be broadcast-multiplied with a single input element inside the inner loop. Because the filter belongs to the static model weights, this packing is input-independent and can be precomputed offline once and reused across all inferences, so its cost is amortized to zero in steady-state throughput; the input-panel gather, by contrast, depends on the runtime activations and is performed per tile.
- **Register-blocked accumulators.** The output panel $C[m{:}m{+}\mathrm{MR},\ n{:}n{+}\mathrm{NR}]$ is accumulated entirely in vector registers across the full $K$-loop reduction, and stored back to memory only once per panel. This eliminates the load–store round trip that would otherwise dominate compute-bound layers.
- **Padding fast/slow path dispatch.** The interior of the output (typically

$\geq 96\%$ of pixels for 3×3 kernels with `pad`=1) takes a branch-free SIMD path; a separate slow path handles the boundary.

**ISA-specific differences.** The two backends diverge along three axes, all driven by the underlying SIMD model:

- *Vector width binding.* RVV is Vector Length Agnostic—the active `vl` is queried at runtime via `vsetvli`—while AVX-512 has a fixed 16-lane width and 32 ZMM registers, which directly enables two-dimensional register blocking with up to $\mathrm{MR} \cdot \mathrm{NR}/16 \leq 16$ output accumulators in flight.
- *Tensor layout.* The AVX-512 backend uses NHWC for the input tensor, since channel-contiguous storage matches its broadcast-multiply pattern along $K_{\text{out}}$ and lets a single `_mm512_loadu_ps` pull 16 contiguous channels per cycle. The RVV backend uses NCHW for compatibility with oneDNN's default layout pipeline; its row-stride broadcast pattern (one `vfmacc.vf` per row) is insensitive to channel locality.
- *Specialization mechanism.* RVV achieves per-shape specialization through its JIT framework (Section 3.1), whereas AVX-512 uses C++ template specialization plus a runtime autotuner that selects from precompiled instantiations (Section 3.2.4).

### 3.2.2 RVV Implementation

The RISC-V Vector extension (RVV) changes the optimization problem: vector width is no longer a compile-time constant. To remain portable across implementations with different `VLEN`, our RVV backend follows a Vector Length Agnostic (VLA) programming model. Concretely, kernels query the active vector length using `vsetvli` and structure inner loops around the returned `vl`. This ensures that a single binary can scale from small vectors (e.g., 128-bit) to wider designs.

Our `jit_rvv_generator` provides instruction emission for RVV 1.0 instructions including `vsetvli`, `vle32`, `vse32`, `vfmul`, `vfmacc`, and reductions such as `vfredusum`.

In our implementation, three engineering constraints repeatedly appear across kernels. (i) The active `vl` must be queried at runtime so that the same kernel can execute correctly across different hardware `VLEN`. (ii) LMUL must be chosen with care: it can increase effective vector width, but it also consumes vector registers and can force a smaller register block. (iii) JIT-generated code must be ABI-compliant and reentrant, because oneDNN calls primitives repeatedly inside larger inference loops. These constraints inform the backend choices described below.

The remainder of this subsection provides a detailed technical description of how we adapted the Implicit GEMM algorithm of Section 3.2.1 for RVV, including the micro-kernel design, data layout optimization, and the trade-offs between RVV intrinsics, inline assembly, and JIT code generation. These three are not independent reimplementations: they share the same Implicit GEMM algorithm, data layout, and outer loop structure, and differ only in how the steady-state micro-kernel body is

produced—left to the compiler (intrinsics), hand-written as inline assembly embedded in the same C++ kernel, or generated at runtime (JIT). The inline-assembly backend in particular replaces only the inner loop of the intrinsics path, so it is best read as a hand-tuned specialization of the intrinsics version rather than a wholly separate implementation.

#### 3.2.2.1 Micro-kernel Adaptation: From Fixed-Width Tiles to 8×VL

The x86-64 backends use fixed-width micro-kernel tiles whose shape is set by the SIMD register width. The AVX2 implementation uses a 6×16 tile:

- 6 rows: Each row uses 2 YMM registers (256-bit × 2 = 16 floats), total 12 YMM registers for C accumulators.
- 16 columns: fixed vector width (256-bit AVX2 = 8 floats × 2 registers).

The AVX-512 implementation widens this to a tunable MR × NR tile (with MR ∈ $\{4, 6, 8\}$ and NR ∈ $\{16, 32\}$) that exploits the 32 ZMM registers; the 512-bit lane width means one ZMM register holds 16 floats, and the doubled register file leaves room for up to 16 accumulator registers. The selection of this tile per layer is the subject of Section 3.2.3 and the autotuner of Section 3.2.4.

For RVV, we adopt an 8×VL dynamic micro-kernel that adapts to any hardware VLEN:

- 8 rows (MR=8): fixed for good register utilization across all VLENs.
- VL columns (NR=VL): dynamic, determined at runtime via `vsetvli`.

Table 3.1: Micro-kernel Tile Sizes Across Different VLENs

| VLEN (bits) | Floats per Vector | Tile Size | FLOPs per Iteration | C Registers |
|---|---|---|---|---|
| 128 | 4 | 8×4 | 64 | 8 (v0-v7) |
| 256 | 8 | 8×8 | 128 | 8 (v0-v7) |
| 512 | 16 | 8×16 | 256 | 8 (v0-v7) |
| 1024 | 32 | 8×32 | 512 | 8 (v0-v7) |
| 2048 | 64 | 8×64 | 1024 | 8 (v0-v7) |
| 8192 | 256 | 8×256 | 4096 | 8 (v0-v7) |

This VLA design ensures the same binary runs efficiently on any RISC-V hardware, from embedded devices (VLEN=128) to high-performance servers (VLEN=8192).

#### 3.2.2.2 Data Layout and Filter Packing

**Input tensor layout (NCHW).** We use the standard NCHW (Batch, Channel, Height, Width) layout for input tensors, which is compatible with most deep learning frameworks. The on-the-fly implicit indexing pattern that converts a GEMM coordinate $(m, k)$ into an input pixel was given in Section 3.2.1 and is identical for both backends; here we focus on the implications for vectorization, which differ from the AVX-512 (NHWC) backend (Section 3.2.3) because the innermost contiguous dimension is now $W$ rather than $C$.

**Filter packing for vector-friendly access.** To enable efficient vector loads, filters are packed from `[K, C, R, S]` to `[K/NR, K_gemm, NR]`:

$$\text{PackedFilter}[\text{panel}, k_{\text{gemm}}, \text{lane}] = \text{Filter}[\text{panel} \cdot N_R + \text{lane}, c, r, s] \tag{3.2}$$

This transformation ensures that each vector load (`vle32.v`) fetches $N_R$ filter values that will be broadcast-multiplied with the same input value, maximizing arithmetic intensity.

#### 3.2.2.3 RVV Intrinsics Implementation

The RVV intrinsics version serves as a portable baseline. It uses `riscv_vector.h` and relies on the compiler for scheduling and register allocation. This makes it easy to maintain and correct, and it provides a clean reference point when attributing performance differences to hand-optimized assembly or JIT specialization.

```
// Main FMA loop using RVV intrinsics
for (int k = 0; k < kc; ++k) {
    // Load B[k, :] (one vector of VL floats)
    vfloat32m1_t b = vle32_v_f32m1(B + k * vl, vl);

    // FMA for each of the 8 rows
    c0 = vfmacc_vf_f32m1(c0, A[0 * stride + k], b, vl);
    c1 = vfmacc_vf_f32m1(c1, A[1 * stride + k], b, vl);
    // ... (c2-c7)
}
```

**Instruction pattern.** Each FMA iteration executes:

- 1 × `vle32.v`: Load VL floats from B
- 8 × `vfmacc.vf`: Vector-scalar FMA (broadcasts A scalar, multiplies with B vector, accumulates to C)

**Observation.** The key property of this loop shape is that it maps convolution naturally to a broadcast-FMA pattern: one contiguous vector load from weights is reused across eight scalar broadcasts from the input panel. This makes the kernel sensitive to weight layout (hence packing) and to loop overhead (hence the motivation for unrolling and pointer hoisting in the optimized backends).

#### 3.2.2.4 Inline Assembly Optimization

The inline assembly backend prioritizes performance over portability. It explicitly schedules loads and FMAs, precomputes row pointers, and uses a fixed register allocation that minimizes spills. This backend provides an upper bound on what a carefully hand-tuned kernel can achieve under the same algorithm and data layout.

**Aggressive loop unrolling.** We unroll the K-loop by a factor of 4, reducing loop overhead and improving instruction-level parallelism:

```
// Unrolled K-loop (factor of 4)
.Lloop:
    // Iteration k+0
    vle32.v v16, (t2)           // Load B[k+0]
    flw ft0, 0(s0); flw ft1, 0(s1); ...  // Load A[*,k+0]
    vfmacc.vf v0, ft0, v16      // c0 += A[0,k]*B[k]
    vfmacc.vf v1, ft1, v16      // c1 += A[1,k]*B[k]
    ...
    // Iteration k+1, k+2, k+3 (interleaved)
    ...
    addi t3, t3, -4             // Decrement counter
    bnez t3, .Lloop
```

**Register allocation strategy.** Our inline assembly uses a carefully designed register allocation:

Table 3.2: Register Allocation in RVV Inline Assembly Micro-kernel

| Registers | Usage | Rationale |
|---|---|---|
| v0-v7 | C accumulators (8 rows) | Live throughout loop |
| v16 | B vector (current k) | Reused each iteration |
| s0-s7 | A row pointers | Callee-saved, persistent |
| s8 | Loop counter | Predictable branching |
| s9 | B base pointer | Avoid recalculation |
| ft0-ft7 | A scalars (broadcast) | Temporary per iteration |
| t0-t3 | Address calculation | Scratch registers |

**Pointer hoisting.** All 8 row pointers for matrix A are pre-computed and stored in callee-saved registers (s0-s7):

```
// Prologue: Compute all row pointers
add s0, a0, zero     // s0 = A + 0*stride (row 0)
add s1, s0, t0       // s1 = A + 1*stride (row 1)
add s2, s1, t0       // s2 = A + 2*stride (row 2)
...
add s7, s6, t0       // s7 = A + 7*stride (row 7)
```

This eliminates address calculation overhead in the hot loop, saving approximately 14 instructions per iteration.

**Observation.** The combination of an explicit register map and pointer hoisting makes the assembly backend a meaningful upper bound: it reduces the steady-state loop body to "one weight vector load + eight FMAs" with minimal address arithmetic, which is difficult for a compiler to guarantee from intrinsics alone.

#### 3.2.2.5 JIT Code Generation for RVV

The JIT backend aims to recover most of the performance of hand-written assembly while retaining portability across layer shapes and vector lengths. Rather than relying on the static compiler to schedule the inner product, the generator emits a register-blocked micro-kernel directly: the accumulator tile is held in vector registers across the reduction, the tile rows are unrolled into a straight-line broadcast-FMA sequence, and the code is vector-length agnostic so a single emitted kernel runs across the VLEN range. The coordinate arithmetic is handled in the packing step and is not part of the emitted kernel.

**Runtime specialization.** The generator adapts the emitted micro-kernel to the kernel configuration in the following ways:

- Unroll the reduction tile into an explicit broadcast-FMA sequence, removing loop control over the tile rows
- Select the register-blocking parameters (LMUL, tile rows, unroll factor) for the configuration, in autotuning mode
- Specialize padding handling for common cases (e.g., 3×3 with pad=1)

**LMUL configuration.** The baseline Implicit GEMM micro-kernel is generated at `LMUL=m1`: each logical vector maps to one physical register. With an 8-row tile it keeps 8 accumulator registers (v0–v7) plus one register for the current filter column (v8) live—nine vector registers in total, well within the 32-register file. The effective vector length per instruction is then

$$\text{Effective VL} = \frac{\text{VLEN}}{32} \text{ (FP32 elements)}, \tag{3.3}$$

i.e. 4 elements at VLEN=128, 8 at VLEN=256, and 16 at VLEN=512.

**Register budget and the LMUL choice.** RVV provides 32 vector registers (v0–v31), and LMUL sets how many physical registers each logical vector occupies (m1: 1, m2: 2, m4: 4). An 8-row tile keeps 8 accumulators plus one filter vector live, so the register budget is

$$(\text{MR} + 1) \cdot \text{LMUL} \leq 32. \tag{3.4}$$

For MR=8 this admits `LMUL=m1` (9 registers) and `LMUL=m2` (18 registers) but rules out `LMUL=m4` (36 registers > 32). Raising LMUL widens the per-iteration vector and halves the loop trip count; it helps when the reduction dimension is short but does not change the algorithm.

**Selected configuration.** The shipped baseline used for the VLEN study fixes `LMUL=m1`, which leaves the largest register headroom and is the choice the autotuner selects on layers with a long reduction dimension. The lightweight autotuner treats LMUL as a per-layer knob over $\{m1, m2\}$ for MR=8 (Section 5.1.4): it raises LMUL to m2 on small-reduction layers, where halving the trip count outweighs the extra

Table 3.3: LMUL register budget for an 8-row tile ((MR + 1) · LMUL vectors)

| LMUL | Registers per Vector | Vectors for 8-row tile | Feasible ($\leq 32$) |
|---|---|---|---|
| **m1** | **1** | **9** | **Yes (baseline)** |
| m2 | 2 | 18 | Yes |
| m4 | 4 | 36 | No |

register pressure, and keeps m1 on large-reduction layers. `LMUL=m4` is never used at MR=8 because it exceeds the register budget.

#### 3.2.2.6 Implementation Comparison

Table 3.4: Comparison of Three RVV Implicit GEMM Implementations

| Aspect | Intrinsics | Inline Assembly | JIT |
|---|---|---|---|
| Code portability | High | Low | Moderate |
| Optimization level | Compiler-dependent | Fully manual | Runtime-specialized |
| Loop unrolling | Limited | Aggressive (4x) | Configurable |
| Register allocation | Compiler-controlled | Manual optimal | Manual optimal |
| Startup overhead | None | None | Milliseconds |
| Maintenance effort | Low | High | Medium |

**Quantitative comparison.** The three backends are compared empirically under gem5 in Section 4.5, which evaluates intrinsics, inline assembly, and JIT across networks and VLEN configurations; we defer all speedup numbers to that evaluation. At the design level, the trade-off is between absolute performance and maintenance cost: hand-optimized assembly favors the former, whereas JIT captures most of the benefit of specialization with substantially lower maintenance effort than a fully hand-written backend.

### 3.2.3 x86-64 AVX-512 Implementation

The x86-64 AVX-512 backend realizes the Implicit GEMM algorithm of Section 3.2.1 on Intel Xeon-class CPUs. Compared to the RVV backend, it exploits two ISA-specific opportunities: (i) the fixed 16-lane vector width, combined with 32 ZMM registers, enables two-dimensional register blocking inside the register file, and (ii) static C++ template specialization removes the need for a per-shape JIT pass. The remainder of this subsection describes the five vectorization strategies that ship in the backend, the filter packing scheme they rely on, the register-blocked micro-kernel, the padding fast-path, and the template specialization that exposes the autotuner's search space (Section 3.2.4).

#### 3.2.3.1 Five Vectorization Strategies

A single Implicit GEMM problem can be vectorized in several ways depending on which axis is mapped to the 16 SIMD lanes and how the inner loops are reordered.

Rather than committing to one decision a priori, our backend ships five strategies and lets the autotuner (Section 3.2.4) pick per layer. The strategies form a deliberate progression from one-dimensional SIMD baselines (`vec_n`, `vec_k`) to two-dimensional register blocking (`tiled`, `regblock`) and finally to a GEBP-style loop reordering (`gepb`); each step trades a different axis of reuse for a different bottleneck. Table 3.5 summarizes the five strategies along the dimensions that matter for performance reasoning, and the list that follows describes each strategy in turn. Throughout this subsection $\mathrm{NR}_{\mathrm{VECS}} = \mathrm{NR}/16$ denotes the number of ZMM vectors required to hold one row of the output panel.

Table 3.5: Five AVX-512 vectorization strategies for Implicit GEMM. "Loop order" is the three-deep nest after the M-loop; "Acc. location" indicates whether the K-loop reduction lives in a ZMM register or in an L1-resident buffer; "FMAs/inner step" counts the number of independent 16-wide FMAs issued per innermost iteration; "Inputs" lists which operands are reloaded inside the inner loop.

| Strategy | SIMD axis | Loop order | Acc. location | FMAs/inner step | Inputs reused |
|---|---|---|---|---|---|
| `vec_n` | $K_{\text{out}}$ ($N$) | $m \to n \to K$ | 1 ZMM register | 1 | input scalar broadcast |
| `vec_k` | $C_{\text{in}}$ (reduction) | $m \to n \to K$ | 1 ZMM register, then h. reduce | 1 | none (stride-1 streams) |
| `tiled` | $K_{\text{out}}$ ($N$) | $M_T \to N_T \to K_T \to m \to k \to n$ | L1 stack buffer (1D) | 1 per $n$ vec | filter shared in L1 |
| `regblock` | $K_{\text{out}}$ ($N$) | $m \to n \to r \to s \to c$ | MR × $\mathrm{NR}_{\mathrm{VECS}}$ ZMM registers | MR · $\mathrm{NR}_{\mathrm{VECS}}$ | filter, MR input bcasts |
| `gepb` | $K_{\text{out}}$ ($N$) | $m \to r \to s \to c \to n$ | MR × $N$ L1 buffer | MR · $\mathrm{NR}_{\mathrm{VECS}}$ | input bcast for entire N |

**`vec_n`** *1D SIMD along $K_{out}$.* The simplest strategy maps the 16 SIMD lanes to 16 consecutive output channels. The inner $K$-loop broadcasts one input scalar via `_mm512_set1_ps`, multiplies it with one packed-filter vector via `_mm512_fmadd_ps`, and accumulates into a single ZMM register. The accumulator stays in a register across the entire reduction and is stored only once per output position. This baseline strategy illustrates why filter packing is needed: without packing, the 16 filter values for the 16 output channels at one $(r, s, c)$ position are stride-$RSC$ apart in KRSC layout, so a contiguous `loadu` would silently load 16 input channels of a single output channel instead. `vec_n`'s weakness is that each inner-loop iteration issues only one FMA, leaving the dual FMA pipes of Sapphire Rapids underutilized even when bandwidth is plentiful.

**`vec_k`** *1D SIMD along the reduction.* The second 1D strategy vectorizes the reduction axis itself: 16 input channels are loaded contiguously, multiplied with 16 contiguous filter values, and accumulated into one ZMM register. After the $r/s/c$ loops complete, a horizontal reduction (`_mm512_reduce_add_ps`) collapses the 16 partial sums to a single output scalar. Because both operand streams are stride-1, no filter packing is required—which is its main benefit for layers where packing setup is not worth amortizing. The trade-off is the horizontal reduction itself (a chain of `vshuff/vpermut` plus `vaddps` taking $\approx$ 14–20 cycles) and the dependent-FMA chain on a single accumulator inside the inner loop. As a result, `vec_k` is selected mainly on tiny-$K$, large-$C$ shapes where neither weakness dominates; on most of the benchmark suite `regblock` achieves higher throughput.

**`tiled`** *Cache-tiled 1D SIMD.* The `tiled` strategy is a cache-tiled descendant of `vec_n`. The output is partitioned into TILE_M × TILE_N panels, and each

panel uses a small L1-resident accumulator buffer instead of a single ZMM register. Conceptually it is a proto-GEPB: the inner loop still vectorizes along $N$ with one FMA per step, but tiling the outer $(m, n)$ pair improves filter and input reuse across panels, and the buffer absorbs short reduction chains. In practice the autotuner rarely picks `tiled` once `regblock` is available, because the L1 round-trip per FMA and the lack of accumulator-side parallelism make it Pareto-dominated. We retain it because (i) it provides an intermediate design point between 1D and 2D blocking, and (ii) the explicit cache-tile parameters let us reason about cache behavior without the further complication of register blocking when interpreting profiling results.

**`regblock`** *2D register blocking.* The register-blocked strategy processes an $\mathrm{MR} \times \mathrm{NR}$ output tile at a time, with $\mathrm{MR} \cdot \mathrm{NR}_{\mathrm{VECS}}$ accumulators kept in ZMM registers across the full $K$-loop. Each inner step loads $\mathrm{NR}_{\mathrm{VECS}}$ filter vectors (shared by all MR rows), broadcasts MR input scalars, and issues $\mathrm{MR} \cdot \mathrm{NR}_{\mathrm{VECS}}$ independent FMAs. With the $(\mathrm{MR}, \mathrm{NR}) = (8, 32)$ tile the inner step issues 16 independent FMAs, enough to saturate Sapphire Rapids' two FMA pipes (each four cycles deep). `regblock` benefits most from filter packing, from padding fast-path dispatch (it amortizes the validity check across MR rows), and from the autotuner's $(\mathrm{MR}, \mathrm{NR})$ knob; the autotuner selects it on 26 of the 28 evaluated layer shapes (Section 4.8.2), with $(8, 32)$ alone accounting for 19 of those selections (Table 3.6). The remaining two layers select `vec_k` (see the `vec_k` entry above), not `tiled` or `gepb`, indicating that the two cases "2D register blocking along $K_{\mathrm{out}}$" and "1D SIMD along the reduction" capture most of the relevant performance variation in our suite.

**`gepb`** *GEBP-style loop reordering.* The fifth strategy adopts the loop order $m \to r \to s \to c \to n$ of the classical GEBP (general block-times-panel) micro-kernel, the innermost building block of Goto and van de Geijn's high-performance matrix-multiplication decomposition [18] as refined in the BLIS framework (BLAS-like Library Instantiation Software) [19]. Each iteration of the outer $c$-loop broadcasts one input scalar from the current $(\mathrm{batch}, h, w, c)$, then issues a full $N$-wide sweep over the output channels using the packed filter; partial sums for the current MR-row panel live in an $\mathrm{MR} \times N$ L1-resident buffer rather than in registers. The trade-off relative to `regblock` is conceptually clean: `gepb` maximizes input reuse (each broadcast feeds the entire $N$-loop) at the cost of moving the accumulator out of registers, while `regblock` keeps the accumulator in registers at the cost of broadcasting the same input $\mathrm{NR}_{\mathrm{VECS}}$ times. On Sapphire Rapids the autotuner consistently prefers `regblock` because the FMA pipes are fast enough that L1 round-trips are not amortized; `gepb` would become competitive on a hypothetical CPU with much faster L1 stores or much narrower register files. We keep it in the search space mainly for cross-architecture comparability with the BLIS-style GEMM that historically dominates on lower-budget x86 cores.

#### 3.2.3.2 Filter Packing

All four strategies that vectorize along $K_{\text{out}}$ (`vec_n`, `tiled`, `regblock`, `gepb`) require the filter to be *packed* from its native KRSC layout into a SIMD-friendly form before the kernel runs. The packing transformation is

$$\text{PackedFilter}[k_{\text{block}},\, rsc,\, \text{lane}] \;=\; \text{Filter}[k_{\text{block}} \cdot 16 + \text{lane},\; r,\; s,\; c] \tag{3.5}$$

so that the 16 output channels at a fixed $(r, s, c)$ position become a single contiguous SIMD-aligned vector. This is the unique data layout under which `_mm512_loadu_ps` and `_mm512_load_ps` return the values that the broadcast-FMA pattern requires.

**Why packing is mandatory, not optional.** The original KRSC filter stores all $R \cdot S \cdot C$ weights of one output channel contiguously, then jumps $R \cdot S \cdot C$ floats to the next output channel. A naive 16-lane contiguous load at an arbitrary $(r, s, c)$ therefore returns 16 *input channels* of a single output channel rather than 16 output channels at the same position. This was the `vec_n` correctness bug we discovered when first running on the cluster (Section 4.3.2), and it is why the packed layout is co-designed with the kernel rather than introduced as a post-hoc optimization.

**Cost and amortization.** Packing copies $K \cdot R \cdot S \cdot C$ floats once per call to the kernel. For VGG conv2_1 this is roughly $128 \cdot 3 \cdot 3 \cdot 64 \approx 73{,}728$ floats (about 288 KB), much smaller than the 9.4 GFLOPs of compute the kernel performs on the same call, so the per-call overhead is below 1% for any layer of practical interest. In addition, oneDNN integration caches the packed filter across repeated inferences with the same weight tensor, reducing the amortized cost to zero in steady-state inference workloads.

**Two layout variants in the same backend.** `regblock` and `vec_n`/`tiled` all use the $[K/16, RSC, 16]$ layout above, in which the innermost 16-element block enumerates the output channels at one $(r, s, c)$ point. `gepb`, however, places the $N$-loop in the innermost position and therefore prefers a $K$-major variant, $[RSC, N_{\text{PACK}}]$, that lays out 16 contiguous output channels along the same row as the next 16 output channels for the same $(r, s, c)$. The two layouts are not interchangeable, and so each strategy carries its own packing routine; the autotuner amortizes the packing cost over the per-call benchmark of the candidate so that the choice of layout is part of what is being tuned.

#### 3.2.3.3 Two-Dimensional Register Blocking

The `regblock` strategy is built around a single design principle: keep the entire MR × NR output panel in vector registers for the lifetime of the $K$-loop, and store back to memory only once per panel. With 32 ZMM registers, the question is how to spend them.

**Register budget.** Inside the inner-most iteration we need: $\mathrm{MR} \cdot \mathrm{NR}_{\mathrm{VECS}}$ accumulators (one per output position × 16-lane block), $\mathrm{NR}_{\mathrm{VECS}}$ filter vectors (shared by all MR rows of the current step), one broadcast register for the current input scalar, and a small constant $c_{\mathrm{tmp}}$ of scratch registers ($\approx$ 1–2) that the compiler usually needs for address arithmetic. The constraint is

$$\underbrace{\mathrm{MR} \cdot \mathrm{NR}_{\mathrm{VECS}}}_{\text{C accumulators}} + \underbrace{\mathrm{NR}_{\mathrm{VECS}}}_{\text{B vectors}} + \underbrace{1}_{\text{A bcast}} + c_{\mathrm{tmp}} \leq 32. \tag{3.6}$$

This is the rule that prunes the (MR, NR) search space inside the autotuner (Section 3.2.4.1). Table 3.6 enumerates the realistic candidates and their register cost.

Table 3.6: Register-budget analysis for the (MR, NR) candidates considered by the autotuner. "Accs." is the number of ZMM registers used by the accumulator panel; "Total ZMM" assumes $c_{\mathrm{tmp}} = 1$. "Picks" counts how often the autotuner selects this (MR, NR) across the 28 evaluated layer shapes (the remaining two layers select `vec_k` rather than `regblock`, see Section 3.2.3.1). ZMMs above 32 spill to memory and are pruned at search-space generation.

| MR | NR | $\mathrm{NR}_{\mathrm{VECS}}$ | Accs. | Total ZMM | Picks / 28 | Status |
|---|---|---|---|---|---|---|
| 4 | 16 | 1 | 4 | 7 | 1 | feasible |
| 6 | 16 | 1 | 6 | 9 | 0 | feasible |
| 8 | 16 | 1 | 8 | 11 | 0 | feasible |
| 4 | 32 | 2 | 8 | 12 | 0 | feasible |
| 6 | 32 | 2 | 12 | 16 | 6 | feasible |
| **8** | **32** | 2 | 16 | 20 | **19** | **selected most often** |
| 8 | 64 | 4 | 32 | 38 | — | infeasible (spills) |

**Inner-loop instruction count.** At $(\mathrm{MR}, \mathrm{NR}) = (8, 32)$ each iteration of the inner $c$-loop issues two filter `loadu`s (shared across all MR rows), eight `set1_ps` broadcasts, and $8 \cdot 2 = 16$ FMAs against the eight registers' worth of accumulators. The 16 FMAs are independent across the $\mathrm{MR} \cdot \mathrm{NR}_{\mathrm{VECS}}$ destination registers, providing more than the $2 \cdot 4 = 8$ in-flight FMAs needed to saturate the dual four-cycle FMA pipes. Smaller blockings such as $(4, 16)$ produce only four independent FMAs per step, which leaves the FMA pipes underutilized once the front-end and load ports are also busy; this is why the autotuner consistently prefers $(8, 32)$ on layers with enough $M$ and $N$ to fill the panel.

**Storing back.** After the $r/s/c$ loops complete, the kernel issues $\mathrm{MR} \cdot \mathrm{NR}_{\mathrm{VECS}}$ `storeu_ps` instructions to write the accumulators back to the output tensor (line 570 of `implicit_gemm_avx512.hpp`). Because this happens once per panel, not per FMA, store bandwidth is never on the critical path even on layers with very small $C$.

#### 3.2.3.4 Padding Fast/Slow Path Dispatch

Standard $3 \times 3$ convolutions with `pad` $= 1$ pad each spatial axis by one pixel, so a $H \times W$ input has at most $(H-2)(W-2)/(HW)$ of its output pixels in the strict

interior. For $H = W = 56$ this is $54^2/56^2 \approx 96\%$; for $H = W = 28$ it is $\approx 92\%$. The vast majority of output pixels therefore never read from the padded region, and we can specialize their inner loop to skip the bounds check entirely.

**The dispatch.** Inside the $r$ and $s$ loops of `regblock`, the kernel computes a MR-element `valid[]` array (whether each row of the current panel touches a real, non-padded input pixel for the current $(r, s)$). It also tracks a single scalar `all_valid` that is true iff all MR rows are valid *and* the panel is full ($m_{\text{count}} = \text{MR}$):

```
bool all_valid = (m_count == MR);
for (int mi = 0; mi < MR; ++mi) {
    int h = p_arr[mi] * stride_h - pad_h + r * dilation_h;
    int w = q_arr[mi] * stride_w - pad_w + s * dilation_w;
    valid[mi]   = (h >= 0 && h < H && w >= 0 && w < W);
    all_valid  &= valid[mi];
}

if (all_valid) {
    // Fast path: branch-free FMA inner loop
} else {
    // Slow path: per-mi if(valid[mi]) inside the FMA inner loop
}
```

**Fast path.** When `all_valid` is true, the inner $c$-loop is the clean steady-state body of 16 FMAs per iteration described above, with no per-row branches. This path consumes the bulk of the kernel's runtime on layers with pad = 1.

**Slow path.** When at least one row of the current panel is in the padded region, the inner loop guards each broadcast with an `if(valid[mi])` check. Because the slow path only fires on the boundary rows of the output, the branch mispredict cost is amortized over a small fraction of pixels and does not noticeably regress end-to-end performance.

**Why this exact level of nesting.** Two alternative placements of the validity check are both worse. Hoisting `valid[]` *above* the $(r, s)$ loops would be incorrect, because validity changes with $(r, s)$. Pushing the check *below* the $c$ loop, into the FMA itself, would force a `mask`-style FMA on every iteration regardless of whether any padding is involved, defeating the purpose of the optimization. The chosen level—inside $(r, s)$ but outside $c$—is the unique placement at which the check is both correct and amortized over the full $C$ reduction.

#### 3.2.3.5 Template Specialization

Each of the five strategies of Section 3.2.3.1 is implemented as a C++ function template, with template parameters that depend on the strategy: `regblock` takes $(\text{MR}, \text{NR})$, `gepb` takes MR alone, and `vec_n`, `vec_k`, `tiled` take cache-tile sizes that we fix to a single value because they are rarely picked by the autotuner anyway. The TILE_M scheduling parameter (Section 3.2.4.2) is passed at runtime, not as

a template argument, because it tunes only the OpenMP chunk size and does not change the generated instruction stream.

**Eleven precompiled instantiations.** Together this gives eleven concrete entry points that are precompiled at build time and selected by the runtime dispatch table in `autotuner.hpp`:

Table 3.7: Eleven precompiled function instantiations exposed by the AVX-512 backend. "Pick rule" indicates how the runtime dispatch in `autotuner.hpp` maps an autotuner-chosen `ConvConfig` to one of these functions.

| Strategy | Template parameters | Instantiations |
|---|---|---|
| `regblock` | (MR, NR) with MR $\in \{4, 6, 8\}$, NR $\in \{16, 32\}$ | 6 |
| `gepb` | MR $\in \{4, 6, 8\}$ | 3 |
| `vec_n` | fixed (TILE_M, TILE_K) = (32, 64) | 1 |
| `vec_k` | fixed (TILE_M, TILE_N) = (32, 32) | 1 |
| **Total** | | **11** |

The six `regblock` instantiations cover the entire feasible (MR, NR) region that fits the register budget of Equation 3.6. The three `gepb` instantiations omit the NR dimension because `gepb`'s inner $N$-loop already streams over the full $K_{\text{out}}$ axis. The single instantiations of `vec_n` and `vec_k` fix their cache tile sizes because adding tile knobs there would multiply the number of instantiations without producing measurable gains in our shape suite (the autotuner picks `vec_n` zero times and `vec_k` only twice; see Section 3.2.3.1). The `tiled` strategy is reachable through `vec_n` with non-trivial cache tiles and is therefore not separately instantiated in the dispatch table, even though it is part of the conceptual taxonomy of strategies.

**Runtime dispatch.** At runtime, the dispatch table in `autotuner.hpp` (lines 217–234) maps the autotuner's chosen `(strategy, MR, NR)` triple to one of the eleven function pointers above, so the cost of selection is a single indirect call per layer (per inference, after the first call). The table also forwards the TILE_M value as a runtime argument, which combines compile-time specialization for (MR, NR) with runtime binding for TILE_M.

**Static template vs. JIT.** This static-template approach plays the role of the JIT framework for the AVX-512 backend. The trade-off relative to the RVV JIT backend (Section 3.2.2.5) is straightforward: with only eleven instantiations, the compile-time overhead is paid once at build time, the kernel is fully optimized by the C++ compiler with no runtime code-gen budget, and there is no per-shape JIT pass. We pay for this with reduced specialization granularity—we cannot, for example, bake a specific $W$ into the kernel as a compile-time constant the way the RVV JIT does—but in our experiments the TILE_M runtime knob recovers most of the benefit a per-shape JIT would buy on x86-64.

### 3.2.4 Auto-tuner Framework

The runtime autotuner described in this subsection targets the AVX-512 Implicit GEMM backend. The RVV simulation backend uses fixed configurations because gem5 simulation cost makes a full per-shape sweep prohibitive (a single VLEN sweep would require on the order of 50 hours); the small BananaPi ($\mathrm{LMUL}, \mathrm{MR}, k$-unroll) sweep on real hardware reported in our concluding chapter is a separate, smaller offline experiment that shares the same skeleton—candidate generation, register-budget pruning, and per-shape selection—but is run once offline rather than at first call. The FlashAttention and Winograd backends use shape-specific fixed configurations targeting their algorithmic structure (Sections 3.3, 3.4). The remainder of this subsection describes the search space, the pruning rules, the TILE_M parameter, and the per-shape result cache.

The autotuner has a three-step structure that is reused across both AVX-512 and RVV (when applicable): (i) for each unique convolution shape, generate a small set of candidate configurations from the search space, pruning combinations that are infeasible by hardware constraints; (ii) on the first call to that shape, benchmark every surviving candidate (one warmup + three timed runs, median selection) and store the winner in an in-memory cache keyed by the shape; (iii) on subsequent calls, dispatch directly via cache lookup, paying only a single hash-map probe. The cache can additionally be exported to and imported from a JSON file so that tuning results survive across program runs.

#### 3.2.4.1 Search Space and Pruning

The search space is the four-dimensional Cartesian product

$$\begin{aligned}\mathcal{S} \;=\; &\underbrace{\{\texttt{vec\_n}, \texttt{vec\_k}, \texttt{tiled}, \texttt{regblock}, \texttt{gepb}\}}_{\text{strategy}} \\ &\times \underbrace{\{4, 6, 8\}}_{\text{MR}} \times \underbrace{\{16, 32\}}_{\text{NR}} \times \underbrace{\{0, 24, 48, 96, 192, 384\}}_{\text{TILE\_M}},\end{aligned} \tag{3.7}$$

where TILE_M = 0 disables L2 cache tiling (Section 3.2.4.2). Naively unrolled this is $5 \cdot 3 \cdot 2 \cdot 6 = 180$ candidates per layer, too many to benchmark on every first encounter of a new shape inside an inference workload. The framework therefore drops obviously infeasible or dominated candidates before benchmarking, using five pruning rules drawn directly from the hardware and from the kernel's loop structure.

**Five pruning rules.**

1. *Register budget* (`regblock`): we require $\mathrm{MR} \cdot (\mathrm{NR}/16) + 3 \leq 32$, where the constant 3 accounts for the $\mathrm{NR}_{\mathrm{VECS}}$ filter vectors, the input broadcast register, and one scratch register. The constraint is precisely Equation 3.6, and it eliminates the high-corner $(\mathrm{MR}, \mathrm{NR}) = (8, 64)$ candidate that would otherwise spill to memory.

2. *SIMD-along-N requires* $K_{out} \geq 16$: `vec_n`, `tiled`, `regblock`, and `gepb` all map 16 SIMD lanes to output channels. Layers with $K_{\text{out}} < 16$ (e.g. very narrow MobileNet-V2 transitions) cannot fill a full vector and are routed to the scalar fallback path; we skip these candidates rather than benchmark a kernel that would fall back to scalar execution.

3. *NR alignment* (`regblock`): if NR $> K_{\text{out}}$ the panel is wider than the available output channels and the kernel falls back to a tail loop that spends most of its time in scalar code; this candidate is pruned.

4. *SIMD-along-C requires* $C_{in} \geq 16$ (`vec_k`): the reduction-vectorized strategy needs at least 16 input channels to amortize its horizontal-reduction tail; for layers with $C_{\text{in}} < 16$ (e.g. the very first conv1_1 of every network, where $C_{\text{in}} = 3$) the candidate is dropped.

5. *TILE_M feasibility*: a tile size TILE_M is included only if it both fits at least once into the GEMM-$M$ dimension (TILE_M $< M$ *and* $M$/TILE_M $\geq 2$) and is at least one MR block wide. The first part of the rule prevents tiles that would degenerate into "no tiling" on small layers; the second prevents tiles that would round down to zero MR blocks per OpenMP chunk.

After pruning, the surviving candidate count is shape-dependent: a small layer with $K_{\text{out}} = 32$, $C_{\text{in}} = 32$ retains around 15 candidates; a large layer like VGG conv2_1 with $K_{\text{out}} = 128$, $C_{\text{in}} = 64$ at $112 \times 112$ retains around 50. A typical first-call tuning therefore runs 15–55 benchmarks of ∼10 ms each, completing in well under a second per shape; subsequent calls are zero-overhead cache hits.

**Benchmarking methodology.** Every candidate is timed with one untimed warmup followed by three timed runs, and we keep the median. Median is preferred over mean because it filters out the occasional outlier caused by cluster-level noise (NUMA migrations, stray system threads on the same socket) without requiring a discard policy. Inputs are pseudorandom but deterministic per shape, so re-running the autotuner on the same machine produces the same picks; this is important when the JSON cache is used as a regression artifact (Section 3.2.4.3).

#### 3.2.4.2 TILE_M: L2 Cache Tiling via OpenMP Chunk-size Control

The single most consequential tuning knob added late in the project was TILE_M, an L2-cache-tiling parameter for the $M$ axis (the linearised batch $\times P \times Q$ output positions). It contributes a 1.8–2.4× end-to-end uplift on the five evaluated networks (VGG: 2.14×, ResNet-50: 2.44×, DenseNet-121: 1.95×, RegNetY: 2.25×, MobileNet-V2: 1.80×), which is more than any of the previous regblock/gepb micro-kernel changes contributed individually.

**The textbook GEPP–GEBP–GEPB approach is too invasive.** The orthodox way to tile the $M$ axis for L2 reuse is the three-level decomposition popularized by GotoBLAS [18], which factors a large GEMM into successively smaller sub-problems—a panel-times-panel product (GEPP), then a block-times-panel product (GEBP),

down to the register-resident micro-kernel—each sized to a specific level of the cache hierarchy. Applied to our kernel, this would rewrite the loop nest as $N_T$, then $M_T$, then $K$, then $m$, $k$, and $n$ so that a TILE_M $\times$ $K$ panel of the input and a $K \times$ TILE_N panel of the (packed) filter both fit in L2 and are reused across the inner $m \cdot n$ steps. Implementing this faithfully would require rewriting the existing `regblock` kernel's loop nest, splitting the filter packing routine into per-tile blocks, and re-deriving the padding fast-path for the new loop bounds; this would be an intrusive change for a kernel that was already correct.

**The OpenMP chunk-size trick.** Instead of restructuring the kernel, we exploit the fact that OpenMP's `schedule(dynamic, chunk)` clause already controls how many consecutive iterations of the parallel loop are dispatched to a single thread. The `regblock` kernel parallelizes over MR-sized chunks of the output panel:

```
// MR = output rows handled by one regblock micro-kernel call (register-block
    height)
// M  = total number of output rows (GEMM M dimension) for the layer
const int omp_chunk = (TILE_M + MR - 1) / MR;  // = ceil(TILE_M / MR): MR-row
    tiles per L2 chunk
#pragma omp parallel for schedule(dynamic, omp_chunk)
for (int m0 = 0; m0 < M; m0 += MR) {
    // ... regblock micro-kernel for output panel [m0:m0+MR, :] ...
}
```

By setting the chunk size to $\lceil$TILE_M/MR$\rceil$ rather than to a constant, the same thread executes TILE_M/MR consecutive MR-row panels back-to-back. All of those panels read from *the same* packed filter blocks, which therefore stay resident in the thread's L2 working set across the entire chunk. This is exactly the reuse pattern the GotoBLAS rewrite would have created, but achieved through scheduling rather than through a different loop nest.

**Minimal scheduling-level change.** The whole change is the addition of one parameter (`int tile_m`) to the `regblock` template, the local variable `omp_chunk` above, and the substitution into the `schedule` clause. Every other concern—filter packing layout, padding fast-path, register budget, template specialization—remains unchanged, and so does the kernel's correctness. The trade-off is that TILE_M tunes the *scheduling granularity* rather than an explicit cache tile, so its effect on L2 residency is partly hardware-dependent (in particular, on the 60 MB L3 of the Xeon Gold 6548N some chunks can remain in L3 even without tiling); but in practice the autotuner selects shape-appropriate values consistently, and the 1.80× to 2.44× uplift in the network averages indicates that the simpler implementation captures much of the available reuse.

**Shape-dependence of the optimum.** The optimal TILE_M is not constant across layers: the autotuner selects TILE_M = 24 on small-spatial layers ($14 \times 14$, $28 \times 28$), TILE_M = 48 on medium $56 \times 56$ + large-$K$ stages, TILE_M = 96 on $112 \times 112$ early layers, and TILE_M = 192–384 on $224 \times 224$ stems. The same hardware therefore uses six different TILE_M values across the 28 evaluated layer

shapes; a hand-coded heuristic would have to encode at least these six regimes correctly to match the autotuner.

#### 3.2.4.3 Per-shape Result Cache

The autotuner is built around the assumption that the same convolution shape recurs many times in a single inference workload (every batch sees the same shapes; longer-lived deployments see the same shapes across millions of inferences). Tuning therefore needs to happen at most once per shape, and the result must be retrievable in $O(1)$ for every subsequent call.

**Cache key.** A convolution shape is identified by an eleven-field `ConvKey` struct: $(N, H, W, C, K, R, S, \text{stride}_h, \text{stride}_w, \text{pad}_h, \text{pad}_w)$. The key explicitly includes stride and padding because the optimal tile and strategy are different at stride = 2 than at stride = 1: a stride-2 layer halves the number of output positions and therefore changes the favored TILE_M. We deliberately do not key on dilation because all evaluated layers use dilation = 1; supporting dilated convolutions would require extending the key with two more fields.

**In-memory cache and concurrency.** The cache is a `std::unordered_map<ConvKey, TuneResult, ConvKeyHash>` guarded by a `std::mutex` so that concurrent first-call tunings of two different shapes from two different threads do not race. `ConvKeyHash` folds the eleven fields with the standard FNV-style mix; collisions are handled by the standard library bucket structure. On a cache hit (the steady-state path) the lock is held for a single hash lookup—measured at $\sim 0.1\,\mu$s, well below the kernel runtime.

**JSON persistence.** The cache supports `save_json(path)` and `load_json(path)`: every entry is serialized as a one-line JSON record with the shape key, the chosen `ConvConfig`, the best measured time, and the number of candidates that passed pruning. Loading the cache before an inference run is equivalent to having previously seen all those shapes, so the autotuner's amortized cost over the program lifetime is controlled by the user, not by the inference workload. The serialized entries also serve as a regression artifact: the same VGG conv2_1 shape on the same Xeon-class CPU should consistently pick the same configuration, and discrepancies indicate either a kernel regression or a substantial change in the underlying hardware (e.g. a different CPU model).

#### 3.2.4.4 Comparison with the BananaPi RVV Sweep

The RVV concluding chapter reports a small autotuning experiment on the BananaPi-F3 board, sweeping LMUL $\in \{1, 2, 4\}$, MR $\in \{4, 6, 8\}$, and $k$-unroll $\in \{1, 2, 4\}$ on each of the 26 evaluated layers; the combination LMUL = 4 with MR = 8 is pruned for register-file reasons, leaving 24 valid configurations per layer. That experiment and the AVX-512 framework share a register-budget pruning principle, but they differ in search dimensionality, execution model, and intended use.

**Three differences worth being explicit about.** First, the AVX-512 framework's search space is genuinely four-dimensional (`regblock`-style 2D micro-kernel plus TILE_M and $k$-unroll) and structurally pruned by hardware constraints, whereas the BananaPi sweep is a much smaller three-dimensional grid in which LMUL = 4 is never selected in our measurements and $k$-unroll changes throughput by less than 2%, so its effective dimensionality is closer to two (LMUL, MR). Second, the AVX-512 framework caches per-shape results in a hash map and a JSON file so that tuning is paid for once across the program's lifetime; the BananaPi sweep is a one-shot offline experiment whose purpose was to identify a stable (LMUL, MR) choice (`m1` or `m2` with MR = 8, depending on VLEN) and was never intended to drive runtime dispatch. Third, the AVX-512 framework prunes infeasible candidates before benchmarking using the register-budget formula; the BananaPi sweep applies the same idea but at much smaller scale—only the single (LMUL = 4, MR = 8) point is dropped—because all other combinations fit comfortably in RVV's 32 vector registers.

**What carries across.** What the two share is a small portable knob set (MR, LMUL or NR/TILE_M, $k$-unroll) and a register-budget pruning style. This is the cross-architecture classification revisited in Chapter 5 (Discussion): a few knobs and one or two pruning rules tend to suffice on any CPU vector ISA, while the precise constants (16 or 32 register file, TILE_M values, etc.) are hardware-specific. We therefore present the AVX-512 framework as the central autotuner of the thesis and the BananaPi sweep as evidence that the same skeleton, with two minor knob substitutions, is also useful on emerging RVV hardware.

## 3.3 Case Study: JIT FlashAttention on RVV

**Scope and choice of target ISA.** We chose RVV as the target for the FlashAttention case study to evaluate the JIT framework on a numerically sensitive, non-convolutional workload, and to test whether the same code-generation pipeline extends from CNN convolution to Transformer-style attention. The implementation is in principle replicable for x86-64 AVX-512—the JIT engine, ABI handling, and online-softmax algorithm are all ISA-agnostic in their structure—but doing so is out of scope for this thesis. The purpose here is to evaluate framework generality, not to deliver a tuned x86 FA kernel that would compete with mature implementations such as FlashAttention-2 or FlashInfer.

To evaluate the JIT framework beyond convolutions, we implemented a complete FlashAttention kernel for RISC-V Vector. This use case is more complex than GEMM, requiring:

- Online softmax computation with the max-trick for numerical stability
- High-precision transcendental function approximation (`exp`)
- Multi-pass output accumulation

### 3.3.1 Algorithm Implementation

This kernel is written from scratch: we do not reuse code from an existing attention library such as llama.cpp, and the only external components are the FlashAttention algorithm itself [16] and the Cephes-style minimax scheme used for the exp approximation. We implement Algorithm 4 from the FlashAttention paper [16], with the online update formula:

$$m^{(j)} = \max(m^{(j-1)}, s_j) \tag{3.8}$$

$$\ell^{(j)} = \ell^{(j-1)} \cdot e^{m^{(j-1)} - m^{(j)}} + e^{s_j - m^{(j)}} \tag{3.9}$$

$$O^{(j)} = O^{(j-1)} \cdot e^{m^{(j-1)} - m^{(j)}} + V_j \cdot e^{s_j - m^{(j)}} \tag{3.10}$$

where $s_j = Q \cdot K_j^\top$ is the attention score for key $j$.

### 3.3.2 Cephes exp() Approximation

The exponential function is central to softmax computation. Standard library calls introduce significant overhead. We implemented a JIT-compiled approximation based on the Cephes algorithm:

$$e^x = e^g \cdot 2^n, \quad \text{where } x = g + n \cdot \ln(2) \tag{3.11}$$

The $e^g$ term (where $|g| < \ln(2)/2$ is approximated using a 5th-order polynomial, while $2^n$ is computed via IEEE-754 exponent bit manipulation.

**Accuracy.** Both Taylor expansions and minimax (Remez) polynomials are polynomial methods for $e^x$, and both remain active areas of optimization in vector-math libraries. For the FlashAttention setting, however, a textbook Taylor series at the origin—used without range reduction—is not well matched to the softmax input range: a low-degree truncation already reaches relative error of order $10^3$ at $x = -5$, and increasing the degree alone does not close the gap (Table 4.9). The polynomial method used in this work is therefore a range-reduced minimax polynomial in the style of Cephes: argument reduction $x = n \ln 2 + g$ followed by a 5th-order minimax polynomial on $|g| \leq \ln 2/2$. This keeps the maximum relative error below $3 \times 10^{-5}$ across the full softmax range. The underlying theoretical analysis is given in Section 2.4.

### 3.3.3 ABI Compliance and Reentrancy

An important engineering requirement was ensuring the JIT kernel is fully reentrant (can be called multiple times in a loop). This required proper RISC-V ABI compliance:

- Prologue: save callee-saved registers (s0-s4, fs5-fs11) to stack (160 bytes).
- Epilogue: restore registers before returning.
- Overhead: negligible runtime cost.

Without this fix, the second call to the kernel crashed due to corrupted caller state, illustrating the importance of target-ABI compliance in JIT compilers.

### 3.3.4 JIT Optimizations for FlashAttention

Although FlashAttention is algorithmically IO-aware, on RVV the dominant cost for moderate sequence lengths often shifts to *instruction overhead*: repeated `vsetvli` setup, constant materialization, and coefficient loading can compete with the actual FMAs in the hot loop. The JIT therefore targets a simple goal: keep the inner loop compact so that most issued instructions contribute directly to computing scores, updating the online softmax state, and accumulating the output.

We apply three complementary optimizations. `LMUL=m2` increases the amount of work done per vector instruction when `VLEN` is small, improving amortization of loop and setup overhead. Constant pre-loading removes repeated memory traffic and instruction decoding for frequently used exp/softmax constants, which is especially valuable because the FlashAttention loop body mixes vector arithmetic with scalar control flow. Finally, placing polynomial coefficients on the stack provides a compact, position-independent way to access the Cephes coefficients with a single load each, keeping the code generator simple while reducing instruction count compared to absolute-address materialization.

**`LMUL=m2` vector grouping.** RVV allows grouping multiple vector registers via the LMUL (Length Multiplier) configuration. Our JIT kernel uses `LMUL=m2`, which effectively doubles the vector length by grouping pairs of registers (e.g., v0-v1, v2-v3).

```
// LMUL=1:
vsetvli t0, a3, e32, m1    // VL = VLEN/32
vle32.v v0, (a0)           // Load VLEN/32 floats

// LMUL=m2:
vsetvli t0, a3, e32, m2    // VL = 2 * VLEN/32
vle32.v v0, (a0)           // Load 2 * VLEN/32 floats (uses v0-v1)
```

This optimization is most effective when VLEN is small (128–512 bits), as it allows processing more elements per instruction without increasing loop overhead.

**Constant pre-loading to FP registers.** The `exp()` approximation requires seven floating-point constants (exp_hi, exp_lo, log2ef, etc.). Instead of loading these from memory in each iteration, we pre-load them into callee-saved FP registers (fs5–fs11) during kernel initialization:

```
// Materialize from address (3 instructions per constant, per iteration):
lui   t0, %hi(exp_hi)
flw   ft0, %lo(exp_hi)(t0)
fmv.s fs5, ft0

// Use pre-loaded register (1 instruction per use):
fmul.s ft0, fa0, fs6    // fs6 = log2ef (pre-loaded)
```

This reduces the instruction count in the hot loop by approximately 14 instructions per iteration.

**Polynomial coefficients on stack.** The 5th-order Cephes polynomial requires six coefficients (p0–p5). We store these on the stack during initialization and load them via single `flw` instructions:

```
// Stack layout:
// [sp+128]: p0 = 1.9875691500e-4
// [sp+132]: p1 = 1.3981999507e-3
// ...
// [sp+148]: p5 = 5.0000001201e-1

// Access during computation:
flw   ft1, 128(sp)    // Load p0
```

This approach saves 2 instructions per coefficient access compared to absolute address loading.

## 3.4 Case Study: Winograd $F(4, 3)$ on RVV

**Scope and relation to Implicit GEMM.** This case study evaluates Winograd $F(4, 3)$ as a complementary convolution algorithm to Implicit GEMM (Section 3.2). Whereas Implicit GEMM addresses memory traffic by eliminating `im2col` materialization, Winograd reduces arithmetic complexity through structured transforms. We use a five-way ablation (Section 3.4.3) to separate algorithmic gains (Winograd over direct convolution) from implementation-level gains (vectorized transforms, custom GEMM, JIT-generated GEMM). The same $8 \times$ VL JIT micro-kernel from Section 3.2.2.5 is reused for the batched GEMM stage, evaluating composability across the framework.

While Implicit GEMM eliminates memory overhead, Winograd convolution [15] reduces the arithmetic complexity of $3 \times 3$ convolutions. Unlike the FlashAttention case study, which we implement from scratch, this Winograd backend is not a from-scratch implementation: the algorithm and its transform structure are adapted from the open-source `convWinograd` library of Dolz et al. [10] (`https://github.com/hpca-uji/convWinograd`), which provides portable multi-core Winograd convolutions for Intel SSE/AVX2/AVX-512 and ARM NEON/SVE. Our contribution is to port that design to the RISC-V Vector extension and to drive its batched GEMM stage with the same JIT micro-kernel used for Implicit GEMM. We implement Winograd $F(4, 3)$ for RISC-V with a five-way ablation study to systematically evaluate each optimization layer.

**JIT integration point.** The Winograd pipeline has four stages: filter transform, input transform, batched GEMM, and output transform. Of these, only the batched GEMM is delegated to a JIT-generated kernel; the three transform stages are written as ordinary RVV intrinsics in C++. Concretely:

- Modes 0–2 in the ablation (see the table below) all use a *statically compiled* GEMM (either OpenBLAS `sgemm` or our own RVV intrinsics $8 \times$ VL micro-kernel). No JIT is involved.

- Mode 3 replaces the GEMM micro-kernel with the same JIT-generated $8 \times \text{VL}$ kernel that we use for Implicit GEMM (Section 3.4 "JIT Code Generation for RVV"). The transforms remain identical to Modes 1 and 2.

This split keeps a single piece of generated code (the GEMM micro-kernel) shared between the Implicit GEMM and Winograd back-ends, and lets the Mode 2 vs. Mode 3 comparison directly attribute any difference to runtime specialization in the GEMM stage rather than to changes in the transforms.

### 3.4.1 Implementation Architecture

Our Winograd implementation is integrated into oneDNN as a new primitive (`rv64_winograd_convolution_fwd_t`) and follows the standard four-stage pipeline:

1. Filter transform: $U[e, v, k, c] = (G \cdot F[k, c] \cdot G^T)[e, v]$ for all output channels $k$ and input channels $c$.
2. Input transform: $V[e, v, c, t] = (B^T \cdot d[c, t] \cdot B)[e, v]$ for all channels $c$ and tiles $t$.
3. Batched GEMM: $M[e, v] = U[e, v] \cdot V[e, v]$ for all $6 \times 6 = 36$ element positions.
4. Output transform: $Y[k, t] = A^T \cdot M[k, t] \cdot A$ for all channels $k$ and tiles $t$.

The buffer layouts are designed for efficient access patterns:

- $U[e][v][k][c]$: innermost dimension is input channel $c$ (contiguous)
- $V[e][v][c][t]$: innermost dimension is tile index $t$ (contiguous)
- $M[e][v][k][t]$: innermost dimension is tile index $t$ (contiguous)

### 3.4.2 RVV Vectorization Strategy

**Challenge: tile size vs. vector width.** A key challenge in vectorizing Winograd transforms for RVV is the mismatch between tile size and vector width. The $F(4, 3)$ algorithm operates on $6 \times 6$ tiles, meaning each transform involves only 6 elements per row or column. On fixed-width SIMD architectures like ARM NEON (128-bit, 4 floats), this is manageable: 4 of the 6 elements fit in one vector register, with the remaining 2 processed as scalars.

However, RVV with VLEN=512 provides 16 floats per vector register. Processing only 6 elements wastes 62.5% of the vector width. This fundamental mismatch requires a different vectorization strategy.

**Strategy: cross-dimension vectorization.** Instead of vectorizing *within* a tile (which is too small), we vectorize *across* the batch dimension that is contiguous in memory:

- Filter transform: vectorize across input channels ($c$). For each output channel $k$, process VL input channels simultaneously. The $3 \times 3$ filter values for VL channels are loaded using strided loads (`vlse32.v`) with stride $= 9 \times 4$ bytes

(the distance between consecutive channels in OIHW layout). The $G \cdot F \cdot G^T$ computation is performed entirely in vector registers, and results are stored contiguously to $U$ since $c$ is the innermost dimension.

- Input transform: vectorize across input channels ($c$). For each tile position, process VL channels simultaneously. The $6 \times 6$ input tile for VL channels is loaded using strided loads with stride $= H \times W \times 4$ bytes (the distance between channels in NCHW layout). The $B^T \cdot d \cdot B$ computation uses vector arithmetic, and results are stored to $V$ using strided stores (`vsse32.v`) with stride $= N_{\text{tiles}} \times 4$ bytes.
- Output transform: vectorize across tiles ($t$). For each output channel $k$, process VL tiles simultaneously. Since $M$'s innermost dimension is tile index, loads are contiguous (`vle32.v`). The $A^T \cdot M \cdot A$ computation and optional bias/ReLU are performed in vector registers. Results are scattered to the NCHW output tensor, which requires per-tile address computation since different tiles map to different spatial positions.

Table 3.8: RVV Winograd Transform Memory Access Patterns

| **Transform** | **Vector Dim** | **Load Pattern** | **Store Pattern** |
|---|---|---|---|
| Filter ($U = GFG^T$) | Channel ($c$) | Strided (stride=$9 \times 4$B) | Contiguous |
| Input ($V = B^T dB$) | Channel ($c$) | Strided (stride=$HW \times 4$B) | Strided (stride=$N_t \times 4$B) |
| Output ($Y = A^T MA$) | Tile ($t$) | Contiguous | Scalar scatter |

**Memory access patterns.** The strided load/store operations (`vlse32.v`/`vsse32.v`) are essential for this strategy. While strided accesses have higher latency than contiguous accesses, they enable true SIMD parallelism across the large channel and tile dimensions, which far outweighs the access penalty.

### 3.4.3 Five-Way Ablation Study Design

To systematically evaluate each optimization layer, we design a five-way experiment where each mode isolates a single variable:

Table 3.9: Five-Way Winograd Ablation Study

| **Mode** | **Transforms** | **Batched GEMM** | **Isolates** |
|---|---|---|---|
| 4 | N/A (im2col) | OpenBLAS `sgemm` | Direct conv baseline |
| 0 | Scalar | OpenBLAS `sgemm` | Winograd algorithm value |
| 1 | RVV intrinsics | OpenBLAS `sgemm` | Transform vectorization |
| 2 | RVV intrinsics | RVV 8×VL $\mu$-kernel | Custom GEMM vs OpenBLAS |
| 3 | RVV intrinsics | JIT 8×VL $\mu$-kernel | JIT vs static intrinsics |

The key comparisons are:

- Mode 4 vs 0: quantifies the benefit of the Winograd algorithm itself (reduced arithmetic complexity) over traditional im2col + GEMM.

- Mode 0 vs 1: measures the speedup from RVV-vectorized transforms while keeping the GEMM implementation constant (OpenBLAS).
- Mode 1 vs 2: evaluates our custom RVV GEMM micro-kernel against the mature OpenBLAS library.
- Mode 2 vs 3: measures the incremental benefit of JIT runtime code generation over statically compiled RVV intrinsics.
- Mode 4 vs 3: the end-to-end comparison: best-optimized Winograd vs traditional direct convolution.

### 3.4.4 GEMM Micro-kernel Reuse

Modes 2 and 3 reuse the same $8 \times \text{VL}$ micro-kernel developed for Implicit GEMM (Section 3.4). A common tiled GEMM driver handles panel packing and tile iteration, accepting a function pointer to either the RVV intrinsics micro-kernel or the JIT-generated kernel. This design evaluates the composability of the optimization framework: the same micro-kernel serves both Implicit GEMM convolution and Winograd batched GEMM.

### 3.4.5 Multi-Network Benchmark Suite

We evaluate on convolutional networks that are widely used for image classification and that, taken together, cover the range of layer shapes the kernels are likely to encounter in practice. The selected models belong to two main families and span different sizes and architectures:

- **VGG family**—VGG-16 [11]: a heavy, uniform stack of $3 \times 3$ convolutions with channel depths 64–512 (10 evaluated layers, at 1/4 resolution, $56 \times 56$ input). VGG-16 exercises the kernels on *large, balanced* layer shapes with high arithmetic intensity, which is the regime in which Implicit GEMM and the autotuner have the most freedom.
- **ResNet family**—ResNet-18 and ResNet-50 [20]: deeper, residual networks built from BasicBlock ($3 \times 3$) and Bottleneck ($1 \times 1$–$3 \times 3$–$1 \times 1$) cells respectively (8 and 13 evaluated $3 \times 3$ layers, channels 64–256). The ResNet variants exercise the kernels on *deeper, narrower* layer shapes than VGG and let us probe whether the optimizations still pay off when the inner reduction dimension dominates.

The x86-64 evaluation in Chapter 4 additionally extends this suite with two compact mobile-targeted networks—DenseNet-121, RegNetY, and MobileNet-V2—so as to stress the kernels on narrow-channel and depthwise-separable shapes that are typical of inference on mobile/embedded CPUs. Together the suite spans four CNN families (VGG, ResNet, DenseNet, MobileNet/RegNet) and three shape regimes (large-uniform, deep-bottleneck, narrow-mobile), which lets us correlate optimization effectiveness with layer shape rather than with a single representative network.

All evaluated layers use stride = 1 and padding = 1, which are the conditions under which Winograd is applicable. For the RVV-on-gem5 part of the evaluation, the

reduced resolution ensures feasible simulation time while maintaining representative computational characteristics; for the x86-64 hardware part we use full resolution.

# 4

# Results

This chapter presents the performance evaluation. Each experiment states the evaluated claim, workload, metric, baseline, measured results, and main limitations.

**Evaluation questions.** The evaluation is organized around six questions:

1. **Does Implicit GEMM beat im2col on a mature SIMD ISA?** Section 4.3 answers this on x86-64 against oneDNN's optimized im2col + BLAS path.
2. **Does the same design carry over to a wider x86 backend?** Section 4.3.2 extends the kernel to AVX-512 and adds 2D register blocking and a runtime autotuner.
3. **Does the same design carry over to an emerging vector ISA?** Section 4.4 ports the kernel to RISC-V Vector and evaluates it under gem5 simulation, first as a standalone microbenchmark, then inside oneDNN.
4. **How does the RVV kernel scale with vector width?** Section 4.6 sweeps six VLEN configurations and analyzes where each backend saturates.
5. **Do the simulation results hold on real RISC-V silicon?** Section 4.8 repeats the key experiments on a BananaPi-F3 development board.
6. **Does the framework generalize beyond convolution?** Section 4.10 evaluates the same JIT infrastructure on FlashAttention (non-CNN, numerically sensitive), and Section 4.12.3 compares all four backends (AVX2 / AVX-512 / NEON / RVV) side by side.

Sections 4.7 and 4.8.2 provide two complementary ablations. The Winograd $F(4, 3)$ five-way ablation separates algorithmic from implementation-level gains, while the cross-architecture autotuner study identifies which tuning knobs are portable. Both results are used in the cross-architecture synthesis in Section 4.12.3.

## 4.1 Evaluation Protocol

**Metrics.** We report two classes of metrics, one per platform:

- x86-64: wall-clock time and throughput (GFLOPS) measured on physical hardware.

- RISC-V (gem5 SE): cycle-level metrics (`simTicks/numCycles`) extracted from `stats.txt` are treated as primary. Any wall-clock timing printed inside the simulated program is used only as a debugging aid.

**Evaluation types.** We distinguish three types of evaluation:

- Microbenchmark: isolate a kernel or a single layer shape to understand the computational core.
- Framework integration: run within oneDNN to include realistic dispatch and memory layouts.
- Ablation: systematically enable one optimization at a time to attribute gains.

**Layer naming convention.** Throughout this chapter every layer is identified by its name (e.g. `conv2_1`, `res2_3x3`, `s3_3x3`) together with its shape tuple ($H \times W$, $C_{\text{in}} \to C_{\text{out}}$, $R \times S$) on its first appearance in a section. The complete inventory of evaluated layers, with stride and padding, is given in Table 2.1 of Chapter 2. The same layer name in different networks (e.g. ResNet `conv1_1` versus VGG `conv1_1`) refers to different shapes; tables therefore carry a *Config* column whenever multiple shapes share a stub of the same name.

## 4.2 Experimental Setup

**Hardware and simulators.** The experiments in this chapter use the following hardware platforms and simulators:

- **x86-64 (real hardware)**: AVX2 microbenchmarks on an AVX2-capable Intel desktop machine; AVX-512 experiments on an Intel Xeon Gold 6548N cluster node (8 threads).
- **ARM (real hardware)**: Apple Silicon with ARM NEON (128-bit), used as a portability check on a second mature SIMD ISA.
- **RISC-V (cycle-accurate simulation)**: gem5 simulator, MinorCPU model with DDR3-1600 memory. For every gem5 result, cycle-level `simTicks/numCycles` from `stats.txt` is the primary metric; any wall-clock timing printed inside the simulated program is used only as a debugging aid.
- **RISC-V (real hardware)**: BananaPi-F3 single-board computer with the SpacemiT K1 SoC (8-core RV64GCV, 1.6 GHz, VLEN=256 bits, 4 GB DDR4). Used to check whether the gem5 predictions transfer to real silicon.

**Baselines.** Because the same word "scalar" refers to different reference implementations across the chapter, we make every baseline explicit up front. *All speedups in this chapter are reported against the baseline of the corresponding platform, not against an absolute scalar floor that is the same everywhere.*

- **x86-64 AVX2 *vs. im2col*** (Section 4.3): the baseline is `oneDNN` v3.x's default `gemm_convolution` dispatch with the `gemm:blas` path, i.e. an explicit `im2col` expansion followed by an OpenBLAS/MKL SGEMM. This baseline is *already vectorized* and BLAS-backed; the comparison therefore measures the benefit of eliminating `im2col` on top of an optimized x86 stack.
- **x86-64 AVX2 / AVX-512 multi-network "Scalar" column** (Sections 4.3.1 and 4.3.2): a single-threaded, un-vectorized C++ implementation of the same Implicit GEMM nested loops, compiled with `g++ -O3 -fno-tree-vectorize` so that the comparison isolates the effect of our hand-written vector micro-kernel rather than the compiler's auto-vectorizer.
- **ARM NEON "Scalar" column** (Section 4.12.3): the same un-vectorized C++ reference described above, compiled with `clang++ -O3 -fno-tree-vectorize` on Apple Silicon. Acts as an unvectorized reference, not as a vendor library.
- **RISC-V VLA microbenchmark "Scalar Baseline" column** (Section 4.4): a *standalone* scalar C++ Implicit GEMM kernel that uses the same loop nest as the RVV intrinsics version but with all RVV intrinsics replaced by ordinary float arithmetic, cross-compiled with `riscv64-linux-gnu-g++ -O3 -fno-tree-vectorize` and run on gem5 (MinorCPU). This isolates the effect of vectorization alone, with no framework overhead.
- **RISC-V oneDNN integration "`gemm:ref`"** (Section 4.5): oneDNN's built-in scalar reference convolution dispatch (`gemm:ref`, NOT `gemm:blas`), used as the in-framework baseline. This is a different baseline from the standalone scalar microbenchmark above: it includes oneDNN's dispatch and memory-layout overheads, so the resulting numbers (~0.35 GFLOPS in gem5) cannot be directly compared with the standalone scalar numbers (~0.25 GFLOPS).
- **Winograd Mode 0 "WinoRef"** (Section 4.7): a Winograd $F(4, 3)$ implementation whose *transform stages* are written as ordinary scalar C++ but whose batched GEMM stage still calls OpenBLAS `sgemm`. The Winograd *algorithm* is therefore not scalar; only the four transform kernels are. This separates "algorithmic benefit of Winograd" from "benefit of vectorizing the transforms".
- **FlashAttention "Scalar baseline"** (Section 4.10): a self-implemented FlashAttention algorithm that follows the exact same online-softmax recurrence as the RVV intrinsics and JIT versions but uses ordinary float arithmetic in place of RVV intrinsics. This is *not* llama.cpp's CPU attention code; using llama.cpp would conflate kernel design and library code.

Across the chapter we explicitly flag whenever a speedup is computed against a vendor-grade baseline (oneDNN `gemm:blas`) versus an unvectorized scalar baseline (`gemm:ref` or hand-written scalar), because the two ratios are not directly comparable; the cross-architecture comparison in Section 4.12.3 returns to this point explicitly.

**Workload coverage.** We use both standalone-kernel microbenchmarks (to isolate the computational core) and oneDNN-integration runs (to include realistic dispatch

and memory layouts), and we report on a multi-network suite so that conclusions are not anchored to a single network.

### 4.2.1 Benchmark Network Suite

We evaluate the framework on a suite of five widely used image-classification CNNs, chosen so that the conclusions are not tied to a single architecture. The selected models belong to four families—**VGG**, **ResNet**, **DenseNet**, and **MobileNet/RegNet**—and span different sizes and design points: VGG-16 [11] contributes large, uniform $3\times3$ stacks; ResNet-50 [20] contributes a mix of $1\times1$ and bottleneck $3\times3$ layers; DenseNet-121 contributes narrow-channel transition layers; RegNetY contributes squeeze-and-excitation $1\times1$ paths; and MobileNet-V2 contributes depthwise-separable $1 \times 1$ pointwise layers. Together they cover three layer-shape regimes: *large-balanced* (early VGG / ResNet stages), *deep-narrow* (late ResNet, DenseNet transitions), and *narrow-mobile* (MobileNet/RegNet $1 \times 1$ pointwise). The exact per-layer shapes evaluated from these networks are listed in Table 2.1 (Chapter 2); the remainder of this chapter reports results on this suite.

## 4.3 x86-64: Implicit GEMM vs. im2col

**Goal.** Evaluate whether eliminating explicit `im2col` materialization improves CPU convolution performance, especially in memory-bound layers.

**Setup.** We compare our Implicit GEMM implementation against oneDNN's default `im2col + BLAS GEMM` path (`gemm:blas`, vectorized and BLAS-backed) on representative VGG16-style layers (batch size 1) on an AVX2-capable x86-64 machine. We report throughput in GFLOPS and per-layer speedup. The five evaluated layers span VGG16 spatial sizes from $224 \times 224$ down to $14 \times 14$ and channel counts from $C$=3 up to $C$=512. The two subsections below switch to a different baseline (`Scalar (ms)` columns) for multi-network and AVX-512 evaluation; see the Baselines paragraph in Section 4.2 for the full baseline taxonomy.

**Results.** Our x86-64 implementation achieved an average speedup of 9.5× compared to the default oneDNN path on the evaluated layers.

Table 4.1: x86-64 Performance Comparison (VGG16, GFLOPS). All layers use a $3 \times 3$ filter with stride = 1, padding = 1.

| Layer | Config ($H\times W$, $C_{\text{in}}\to C_{\text{out}}$) | oneDNN (im2col) | Implicit GEMM | Speedup |
|---|---|---|---|---|
| conv1_1 | $224 \times 224,\ 3 \to 64$ | 0.69 | 16.10 | 23.3× |
| conv2_1 | $112 \times 112,\ 64 \to 128$ | 4.42 | 164.68 | 37.3× |
| conv3_1 | $56 \times 56,\ 128 \to 256$ | 5.31 | 114.30 | 21.5× |
| conv4_1 | $28 \times 28,\ 256 \to 512$ | 4.73 | 143.37 | 30.3× |
| conv5_1 | $14 \times 14,\ 512 \to 512$ | 36.78 | 52.57 | 1.4× |

**Observation.** The largest gains appear in early layers with large spatial dimensions ($224 \times 224$ down to $112 \times 112$ for conv1_1 / conv2_1), where the `im2col` buffer dominates memory traffic. Later layers with small spatial size and large channel counts (e.g., `conv5_1` at $14 \times 14$ with $512 \rightarrow 512$) have smaller working sets and are already compute-bound under oneDNN's optimized BLAS path, so the benefit of eliminating `im2col` shrinks but remains positive.

### 4.3.1 Multi-Network x86-64 AVX2 Performance

**Goal.** Evaluate whether the speedups generalize beyond VGG-style layers and characterize how layer shape interacts with the optimizations.

**Scope of evaluated networks.** This subsection reports the AVX2 results on the five-network benchmark suite described in Section 4.2.1 (VGG-16, ResNet-50, DenseNet-121, RegNetY, MobileNet-V2), which together cover the large-balanced, deep-narrow, and narrow-mobile layer-shape regimes.

**Setup.** We run the same $6 \times 16$ AVX2 micro-kernel across all five networks, with full-resolution inputs (batch size 1) on an AVX2-capable x86-64 machine.

Table 4.2: x86-64 AVX2 Multi-Network Performance (Representative Layers)

| Network | Layer | Config | Scalar (ms) | AVX2 (ms) | GFLOPS | Speedup |
|---|---|---|---|---|---|---|
| VGG16 | conv1_1 | 224, 3→64, 3×3 | 47.48 | 7.59 | 22.84 | 6.25× |
| | conv2_1 | 112, 64→128, 3×3 | 313.50 | 73.91 | 25.03 | 4.24× |
| | conv3_1 | 56, 128→256, 3×3 | 301.56 | 78.14 | 23.67 | 3.86× |
| | conv4_1 | 28, 256→512, 3×3 | 298.81 | 89.56 | 20.65 | 3.34× |
| | conv5_1 | 14, 512→512, 3×3 | 150.49 | 51.83 | 17.84 | 2.90× |
| ResNet-50 | conv1_7×7 | 224, 3→64, 7×7/s2 | 54.75 | 9.51 | 24.83 | 5.76× |
| | res2_1×1r | 56, 64→64, 1×1 | 6.61 | 1.06 | 24.33 | 6.26× |
| | res2_3×3 | 56, 64→64, 3×3 | 53.16 | 9.52 | 24.30 | 5.59× |
| | res2_1×1e | 56, 64→256, 1×1 | 24.50 | 4.39 | 23.39 | 5.58× |
| | res3_s2 | 56, 128→128, 3×3/s2 | 50.04 | 10.02 | 23.08 | 5.00× |
| | res4_3×3 | 14, 256→256, 3×3 | 53.22 | 11.71 | 19.74 | 4.54× |
| | res5_3×3 | 7, 512→512, 3×3 | 45.02 | 23.66 | 9.77 | 1.90× |
| DenseNet-121 | conv1_7×7 | 224, 3→64, 7×7/s2 | 44.62 | 12.18 | 19.38 | 3.66× |
| | bn_1×1 | 56, 256→128, 1×1 | 45.59 | 13.05 | 15.75 | 3.49× |
| | block_3×3 | 56, 128→32, 3×3 | 50.67 | 13.61 | 16.99 | 3.72× |
| | trans_1×1 | 56, 256→128, 1×1 | 34.85 | 11.05 | 18.59 | 3.15× |
| | late_3×3 | 14, 512→32, 3×3 | 12.56 | 3.61 | 16.02 | 3.48× |
| RegNetY | stem_3×3 | 224, 3→32, 3×3/s2 | 4.98 | 1.45 | 14.96 | 3.44× |
| | s1_3×3 | 112, 32→72, 3×3 | 93.94 | 26.51 | 19.63 | 3.54× |
| | s2_3×3_s2 | 56, 72→216, 3×3/s2 | 35.46 | 11.33 | 19.37 | 3.13× |
| | s3_3×3 | 28, 216→576, 3×3 | 301.18 | 101.84 | 17.24 | 2.96× |
| | se_reduce | 28, 576→144, 1×1 | 21.68 | 7.19 | 18.08 | 3.01× |
| | se_expand | 28, 144→576, 1×1 | 20.26 | 7.50 | 17.34 | 2.70× |
| MobileNet-V2 | expand_1×1 | 112, 16→96, 1×1 | 7.48 | 2.32 | 16.60 | 3.22× |
| | proj_1×1 | 56, 96→24, 1×1 | 3.48 | 0.14 | 103.89 | 25.02× |
| | late_1×1 | 7, 960→320, 1×1 | 4.74 | 3.96 | 7.61 | 1.20× |

**Observation.** The AVX2 Implicit GEMM consistently outperforms the scalar baseline across all five networks, with speedups ranging from 3.11× (RegNetY) to 8.18× (MobileNet-V2). Several observations help interpret these results:

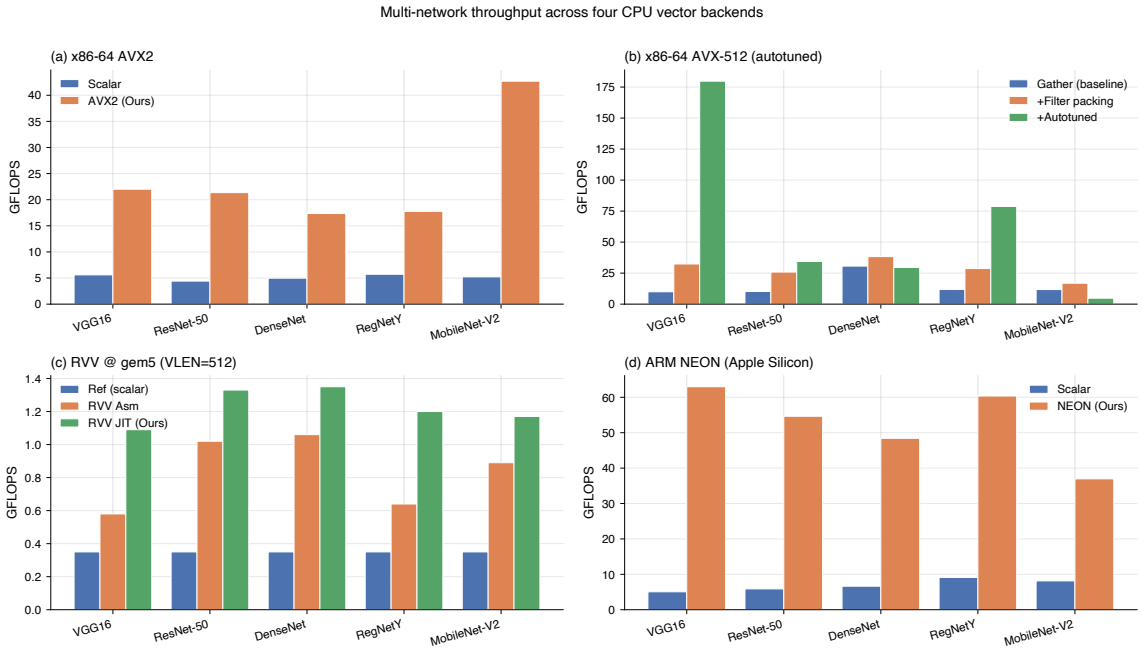


Figure 4.1: Multi-network throughput across the four CPU vector backends evaluated in this thesis. Each panel is averaged across all benchmarked layers of a network.

- Medium-to-large layers reach 16–25 GFLOPS. For a single-threaded AVX2 implementation without filter packing (using the `vec_n` strategy with a $6 \times 16$ micro-kernel), this range is consistent with the expected throughput of the 256-bit FMA pipeline and indicates effective register blocking.
- MobileNet-V2 `proj_1x1` peaks at 103.89 GFLOPS (25×). The $1 \times 1$ pointwise convolution's reuse pattern matches the micro-kernel well, yielding high utilization.
- Very small spatial layers underutilize the micro-kernel. For `res5_3x3` ($7 \times 7$ spatial), the GEMM $M$ dimension is only 49, leading to poor tile utilization and a lower speedup (1.90×).
- Cross-network consistency: the 17–25 GFLOPS band observed across multiple networks suggests the implementation generalizes beyond the VGG layers used during early development.

**Effect of model architecture and size.** Model architecture and layer size also determine the effect of the optimizations. Networks with large, balanced $3 \times 3$ layers (VGG-16, early ResNet stages) consistently sit in the 17–25 GFLOPS band where the AVX2 micro-kernel reaches its design throughput, while networks dominated by narrow-channel or $1 \times 1$ layers split into two regimes: pointwise shapes that match the $6 \times 16$ tile (MobileNet-V2 `proj_1x1`) can exceed the 3×3 band, whereas very small spatial dimensions or very small output-channel counts (`res5_3x3`, DenseNet `late_3x3`, MobileNet-V2 `late_1x1`) under-fill the tile and show smaller speedups. The kernel's absolute speedup should therefore be interpreted together with the layer-shape regime rather than as a single number per network. This shape sensitivity

is the motivation for the autotuner evaluated in Section 4.8.2.

### 4.3.2 x86-64 AVX-512: vec_n Correctness Fix and Filter Packing (Cluster)

To evaluate portability to wider SIMD, we evaluated an AVX-512 backend on an Intel Xeon Gold 6548N cluster node (8 threads). During this evaluation we discovered a correctness issue in the `vec_n` strategy when the filter tensor is stored in KRSC layout. Vectorizing along the output-channel dimension requires loading $\{W_{n+0,r,s,c}, \ldots, W_{n+15,r,s,c}\}$, which is *strided* in KRSC memory by $R \cdot S \cdot C$. A naive contiguous load incorrectly reads $\{W_{n,r,s,c+0}, \ldots, W_{n,r,s,c+15}\}$, i.e., different input channels of the same output channel, leading to widespread mismatches for large $K$.

We fixed this by introducing *filter packing* that rearranges weights into a SIMD-friendly layout $[K/16,\ RSC,\ 16]$, enabling contiguous AVX-512 loads for the 16 output channels.

**From correctness fix to autotuned pipeline.** Building on the filter-packing fix, we extend the AVX-512 backend with the four architectural improvements detailed in Sections 3.2.3 and 3.2.4: 2D register blocking (`regblock⟨MR,NR⟩`) with MR $\in \{4, 6, 8\}$ and NR $\in \{16, 32\}$, GEBP-style loop reordering, NUMA-aware thread placement (`numactl -cpunodebind=0 -membind=0`), and a runtime autotuner that searches over (strategy, MR, NR, TILE_M) with register-budget and channel-count pruning. Together with the compiler's auto-vectorization pass, these improvements compose into a seven-stage cumulative optimization pipeline (L1–L7), which we trace as a waterfall across five representative cross-network layers in Figure 4.2.

**Analysis.** The TILE_M-aware autotuner delivers very large gains on compute-heavy $3 \times 3$ layers, where the L2-resident filter panel and 2D register blocking compound. VGG `conv2_1` ($112 \times 112,\ 64 \to 128$) reaches **758 GFLOPS**, VGG `conv3_1` ($56 \times 56,\ 128 \to 256$) reaches **1076 GFLOPS**, and ResNet-50 `res2_3x3` ($56 \times 56,\ 64 \to 64$) reaches **956 GFLOPS**. The single-layer peak across the suite is **1161 GFLOPS** on DenseNet `block_3x3` ($56 \times 56,\ 128 \to 32,\ 3 \times 3$)—a layer that, paradoxically, has the narrowest $K_{\text{out}}$ of the entire $3 \times 3$ suite and would have under-filled a fixed NR = 32 tile. The autotuner correctly picks a smaller NR together with a tight TILE_M = 24 to keep the packed filter L2-resident across consecutive output panels, turning what would have been a regression into the network-level peak.

The cross-network averages span **102 GFLOPS** (MobileNet-V2, three $1 \times 1$ pointwise layers) to **599 GFLOPS** (VGG-16, five $3 \times 3$ layers); the MobileNet-V2 floor is set by very narrow output channels ($K_{\text{out}} \in \{24, 96, 320\}$) that cannot saturate the FMA pipes regardless of tile choice, not by an autotuner failure. Earlier, pre-TILE_M autotuner versions actually *regressed* on DenseNet and MobileNet-V2 because their narrow-$K_{\text{out}}$ layers under-filled a fixed-NR = 32 tile and the autotuner's calibration overhead was not amortized; adding TILE_M to the search space resolved that

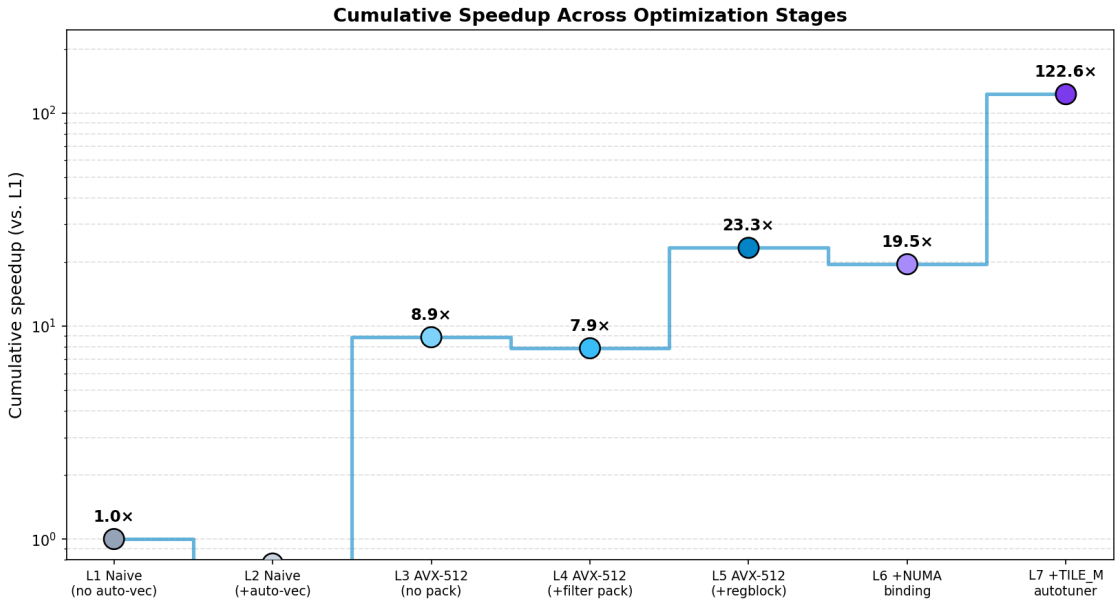


Figure 4.2: Cumulative speedup of the AVX-512 Implicit GEMM pipeline across seven optimization stages, averaged over five representative cross-network layers (VGG-16 `conv2_1` at 112×112, 64→128 and `conv4_1` at 28×28, 256→512; ResNet-50 `res2_3x3` at 56×56, 64→64 and `res2_1x1e` at 56×56, 64→256; MobileNet-V2 `expand_1x1` at 112×112, 16→96). Stages: L1 naive scalar without auto-vectorization (`-fno-tree-vectorize`), L2 the same scalar code with auto-vectorization enabled (`-O3` default), L3 manual AVX-512 `vec_n` on the unpacked KRSC filter, L4 `vec_n` with filter packing, L5 2D register blocking (`regblock<MR,NR>`), L6 plus NUMA binding, L7 plus the TILE_M-aware autotuner. The pipeline reaches an end-to-end cumulative speedup of **122.6**× from L1 to L7 (4.4 GFLOPS to 536.4 GFLOPS averaged over the five layers). Three transitions are non-monotonic and are analyzed in Section 5.1.3: L1→L2 auto-vectorization drops the cumulative speedup from 1.0× to 0.8×, L3→L4 filter packing drops it from 8.9× to 7.9×, and L5→L6 NUMA binding drops it from 23.3× to 19.5×.

regression on DenseNet (now at 564 GFLOPS averaged across five layers) and significantly narrowed it on MobileNet-V2.

Figure 4.2 traces the cumulative speedup of the AVX-512 pipeline across seven optimization stages, from naive scalar (L1) to the TILE_M-aware autotuner (L7); three of the transitions are non-monotonic and are analyzed in Section 5.1.3. Per-layer breakdowns for the fully autotuned pipeline are reported in the Analysis paragraph above; the figure also exposes the cross-network range (22× on MobileNet-V2 to 228× on VGG-16) used in the Conclusion.

**Correctness.** All five-network correctness tests pass after autotuning. The VGG `conv3_1` verification reports 811/802816 mismatches with max diff $\approx 1.16 \times 10^{-3}$, consistent with accumulated floating-point rounding differences from different loop orderings rather than an indexing error.

**Transition from x86 to RISC-V.** Sections 4.3–4.3.2 evaluate the Implicit GEMM design on mature SIMD ISAs: AVX2 improves on oneDNN's im2col-based path, and the same micro-kernel extends to AVX-512 with 2D register blocking and an autotuner. The next three sections evaluate the design on RISC-V Vector: first as a standalone correctness and speedup microbenchmark (Section 4.4), then inside oneDNN (Section 4.5), and finally across vector widths (Section 4.6). Real silicon is evaluated separately in Section 4.8.

## 4.4 RISC-V: Vector Length Agnostic Microbenchmark

**Goal.** Establish that our hand-vectorized RVV Implicit GEMM kernel is (i) numerically correct and (ii) substantially faster than an unvectorized scalar reference, before introducing any framework overhead.

**Setup.** The *scalar baseline* used in this section is a **standalone, single-threaded C++ Implicit GEMM** that walks the same loop nest as the RVV intrinsics version but replaces every RVV intrinsic with an ordinary float FMA. It is cross-compiled with `riscv64-linux-gnu-g++ -O3 -fno-tree-vectorize` (auto-vectorization explicitly disabled) and run on gem5 (MinorCPU model, `VLEN=8192` bits configured for the vector unit so that the same simulator can host both runs). Because the scalar version cannot issue vector instructions, the VLEN setting affects only the vector run; this isolates "benefit of vectorization" from any framework, library, or memory-layout difference.

Evaluated under this setting, our RVV Implicit GEMM achieves an average **3.9**× speedup over the scalar baseline:

Table 4.3: RISC-V gem5 Performance on VGG16 layers (GFLOPS, VLEN=8192). All layers use a $3 \times 3$ filter with stride = 1, padding = 1; spatial sizes follow the 1/4-resolution layer set used for gem5 throughout this work.

| Layer | Config ($H \times W$, $C_{\text{in}} \rightarrow C_{\text{out}}$) | Scalar Baseline | RVV Implicit | Speedup |
|---|---|---|---|---|
| conv1_1 | $56 \times 56,\ 3 \rightarrow 64$ | 0.24 | 1.59 | 6.54× |
| conv2_1 | $28 \times 28,\ 64 \rightarrow 128$ | 0.25 | 0.67 | 2.68× |
| conv3_1 | $14 \times 14,\ 128 \rightarrow 256$ | 0.21 | 0.73 | 3.45× |
| conv4_1 | $7 \times 7,\ 256 \rightarrow 512$ | 0.22 | 0.76 | 3.47× |
| conv5_1 | $7 \times 7,\ 512 \rightarrow 512$ | 0.23 | 0.73 | 3.18× |

**Numerical correctness.** We verified the output of the RVV kernels against the scalar baseline. The maximum element-wise difference was less than $10^{-5}$, confirming the numerical stability of our vector reduction logic.

## 4.5 RISC-V: oneDNN Integration Performance

**Framework integration.** The microbenchmark in Section 4.4 evaluates the RVV kernel in isolation. The next experiment evaluates the same kernel inside a CPU deep-learning framework, where dispatch logic and memory-layout conversions add overhead and where the *baseline* is a different, in-framework scalar reference.

**Setup.** We integrated the RVV Implicit GEMM into oneDNN and ran benchmarks via gem5 simulation. We compare three implementations: (1) oneDNN's in-framework scalar reference convolution dispatch `gemm:ref` (*not* `gemm:blas`; see Baselines paragraph in Section 4.2), (2) our JIT-generated RVV kernel, and (3) our hand-optimized inline-assembly RVV kernel. The absolute scalar GFLOPS in this section (≈0.35) is therefore higher than the standalone scalar number (≈0.25) in Section 4.4, because the in-framework reference also benefits from oneDNN's blocked memory layout; the two scalar baselines are *not* directly comparable, but each is the appropriate baseline for its own evaluation context.

### 4.5.1 Three-way Multi-Network Comparison at VLEN=512

We first evaluate the three implementations at `VLEN=512` bits using a multi-network benchmark suite (VGG16, ResNet-50, DenseNet-121, RegNetY, MobileNet-V2). For gem5 feasibility, the convolution shapes follow the 1/4-resolution layer set used throughout this work, totaling 26 convolution layers. For each network, we report the average throughput (GFLOPS) across its layers, and an OVERALL average across all 26 layers.

Per-network averages are visualized in Figure 4.1c (RVV @ gem5, VLEN=512). The OVERALL across all 26 layers is 1.24 GFLOPS for the JIT path versus 0.35 GFLOPS for the scalar reference (3.54×).

**Observation.** The scalar `gemm:ref` baseline saturates at approximately 0.35 GFLOPS across all networks, indicating a strongly bandwidth- and/or scalar-throughput-limited implementation in the gem5 setting. In contrast, both RVV implementations report `implicit_gemm:*` as their implementation string, confirming that the benchmark executes the integrated Implicit GEMM path. Inline assembly provides a consistent 1.7–3.0× uplift over the reference depending on the network mix, while the JIT kernel achieves the highest throughput for all networks, reaching a 3.54× OVERALL speedup.

Two trends are noteworthy. First, the gains are larger for networks with a higher proportion of compute-heavy convolutions (e.g., DenseNet blocks), where RVV FMA throughput can be better amortized over loop overhead. Second, the JIT kernel consistently exceeds inline assembly at `VLEN=512`, supporting the premise that integer-heavy index and pointer computations, and not vector arithmetic alone, limit Implicit GEMM performance. By specializing stride/padding constants, hoisting pointer arithmetic, and generating a tighter loop body, the JIT version reduces non-FMA overhead and improves effective utilization of RVV resources. For detailed, per-layer

timing (including `Time(ms)` and `impl_info_str()`), the full simulator output is recorded in the benchmark logs.

Table 4.4: oneDNN Integration: Three-way Performance Comparison on VGG16 layers (VLEN=8192 bits, gem5). All layers use a $3 \times 3$ filter with stride = 1, padding = 1.

| **Layer** | **Config** ($H \times W$, $C_{\text{in}} \to C_{\text{out}}$) | **gemm:ref (GFLOPS)** | **JIT RVV (GFLOPS)** | **Inline Asm (GFLOPS)** | **JIT/ref** | **Asm/ref** |
|---|---|---|---|---|---|---|
| VGG-conv1_1 | $56 \times 56,\ 3 \to 64$ | 0.33 | 1.01 | 1.53 | 3.04× | 4.64× |
| VGG-conv2_1 | $28 \times 28,\ 64 \to 128$ | 0.36 | 0.59 | 0.69 | 1.65× | 1.92× |
| VGG-conv3_1 | $14 \times 14,\ 128 \to 256$ | 0.36 | 0.57 | 0.72 | 1.57× | 2.00× |
| VGG-conv4_1 | $7 \times 7,\ 256 \to 512$ | 0.36 | 0.43 | 0.76 | 1.19× | 2.11× |
| VGG-conv5_1 | $7 \times 7,\ 512 \to 512$ | 0.35 | 0.15 | 0.73 | 0.42× | 2.09× |
| **Average** | | **0.35** | **0.55** | **0.89** | **1.57×** | **2.54×** |

### 4.5.2 JIT vs. Inline Assembly Analysis

At `VLEN=512` (Table **??**), the JIT kernel outperforms the inline assembly kernel across all tested networks, suggesting that runtime specialization and code-shape tuning can outweigh the additional complexity of generated code in the moderate-VLEN regime. At larger VLEN values (e.g., `VLEN=8192` shown above), inline assembly becomes competitive or superior, but for a specific reason: the JIT driver packs a filter panel whose size grows with the vector length, and beyond VLEN $\approx 1024$ that panel no longer fits in L2. Section 4.6.4 isolates this working-set effect and shows that capping the panel maintains approximately constant JIT throughput across the full sweep.

**Why JIT can be slower than inline assembly for convolution.** Three factors explain the gap:

- Amortization vs. overhead: JIT generation is a fixed cost amortized across repeated executions; for small problems or short runs it can dominate.
- Working set at large VLEN: the packed filter panel `B_panel` ($K_{\text{gemm}} \times \text{vl}$) grows with the vector length and exceeds L2 beyond VLEN $\approx 1024$, making the kernel memory-bound (Section 4.6.4).
- Micro-architectural modeling: gem5's MinorCPU model and its vector functional-unit abstraction can shift the balance between instruction count and effective throughput.

**Observation.** Both JIT and inline assembly are viable integration strategies. The relative ordering is VLEN-dependent: JIT achieves higher throughput at moderate VLEN (better specialization and reduced integer-index overhead), while inline assembly can achieve higher throughput at very large VLEN (lower control overhead and carefully hand-managed register allocation).

### 4.5.3 Speedup Discrepancy Explanation

**The difference from x86-64 results stems from different baselines**

- x86-64 baseline: `gemm:blas` uses optimized BLAS (e.g., OpenBLAS/MKL) with `im2col`. The 9.5× speedup comes primarily from eliminating the 56MB `im2col` memory overhead.
- RISC-V baseline: `gemm:ref` is a scalar reference implementation. The 2.54× speedup comes primarily from RVV vectorization, not memory bandwidth savings.

To isolate the vectorization benefit: standalone RVV vs. Scalar Implicit GEMM testing (Section 4.4) showed **3.9**× average speedup, which is consistent with the oneDNN integration results.

## 4.6 Multi-VLEN Performance Analysis: Implicit GEMM

**Motivation.** Sections 4.4 and 4.5 establish correctness and in-framework speedups at fixed VLEN values. RVV is a Vector-Length-Agnostic ISA: the same binary can run on hardware with VLEN from 128 to thousands of bits. The next experiment measures how each backend (JIT vs. inline assembly) scales as VLEN grows, and where the scaling saturates.

**Setup.** To understand how vector register width affects optimization effectiveness, we evaluated all three implementations across six VLEN configurations (128–8192 bits) using gem5 simulation. The VLEN sweep focuses on a fixed layer set for controlled comparison, while the multi-network result at `VLEN=512` (Table **??**) shows that the relative ordering and speedups persist across diverse CNN layer mixes.

### 4.6.1 Overall Scalability and Saturation Points

Before discussing the per-region behavior, we summarize the sweep with two scalar metrics: the geometric-mean speedup over the scalar reference across all six VLEN points, and the VLEN at which each backend saturates.

**Overall speedup (geomean over six VLEN configurations).** Across VLEN $\in \{128, 256, 512, 1024, 2048, 8192\}$ bits, the JIT backend achieves a geomean speedup of $\approx 2.66\times$ over the scalar reference, while the inline-assembly backend achieves $\approx 1.87\times$. JIT is faster on average despite lower throughput at the largest VLEN points, because the small-VLEN regime (where it provides 2–3× speedup) is also where the absolute speedup numbers are largest.

**Saturation.** The two backends do not saturate at the same vector width:

- **JIT saturates early.** The JIT speedup peaks at **VLEN=256 (3.28×)** and stays within ±10% of that peak through VLEN=1024 (3.02×). From VLEN=2048 onward it drops below the peak (2.60×) and decreases to 1.57×

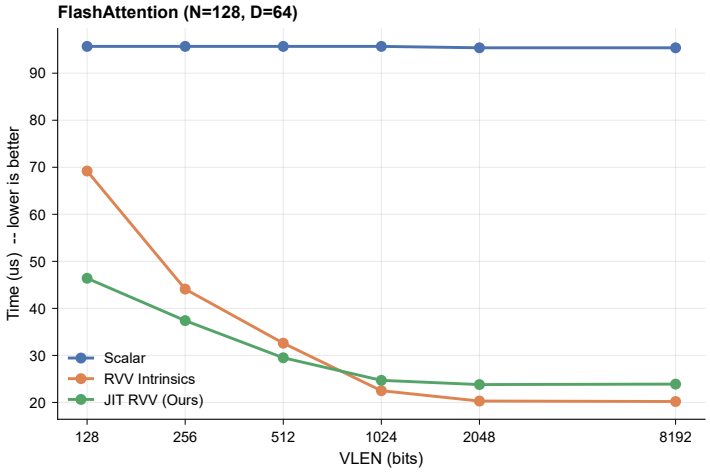


Figure 4.3: VLEN sweep on gem5 for JIT FlashAttention (N=128, D=64), six configurations from 128 to 8192 bits (lower is better). JIT is the fastest implementation up to VLEN=512; RVV intrinsics overtake at VLEN $\geq$ 1024 and saturate near VLEN=8192 at $\sim$ 4.7× over scalar. Increasing vector width past this crossover does not yield monotonic gains once the per-iteration vector already exceeds the kernel's working set. The Implicit GEMM counterpart of this sweep is analyzed in Section 4.6.3 (Figure 4.4).

at VLEN=8192. The JIT backend therefore shows limited additional scaling beyond VLEN $\approx$ 1024 in this setup.

- **Inline assembly saturates later.** It rises steadily from 1.17× at VLEN=128 to 2.45× at VLEN=2048; the additional change at VLEN=8192 is only 4% (2.54×). The asm backend therefore *saturates around VLEN* $\approx$ 2048.

For FlashAttention (Fig. 4.3), the JIT backend follows the same pattern but with an even earlier ceiling: it peaks around **VLEN=512 ($\sim$ 3.24× over scalar)** and is then overtaken by RVV intrinsics, which themselves saturate around VLEN=8192 at $\sim$ 4.7×. The same interpretation applies: increasing vector width does not provide monotonic gains once the per-iteration vector already exceeds the natural working set of the kernel.

### 4.6.2 Key Findings: VLEN-Dependent Optimization Strategy

The results show a crossover region at approximately VLEN=1024–2048 bits:

**Small VLEN (128–1024 bits): JIT outperforms inline assembly.**

- At VLEN=256: JIT achieves **3.28**× speedup (peak), Asm only 1.48×
- At VLEN=128: JIT achieves **2.85**×, Asm only 1.17×
- Reason: the register-blocked micro-kernel keeps the accumulators in vector registers across the reduction and uses broadcast-FMA, while the coordinate

arithmetic is hoisted into the packing step rather than repeated in the inner loop (Section 5.2.1). This advantage is largest when the inner loop still has many iterations, i.e. at small VLEN.

**Large VLEN (≥2048 bits): inline assembly overtakes JIT.**

- At VLEN=8192: Asm achieves **2.54**×, JIT drops to only 1.57×
- At VLEN=2048: Asm (2.45×) and JIT (2.60×) are comparable
- Reason: the JIT path packs a filter panel of size $K_{\text{gemm}} \times \text{vl}$, which grows linearly with the vector length. Beyond VLEN ≈ 1024 this panel no longer fits in L2, and the kernel becomes memory-bound. Section 4.6.4 isolates this effect directly.

### 4.6.3 Vectorization Axis: K-dim vs N-dim

The packed panel that drives the wide-VLEN drop exists only because the kernel vectorizes the output-channel (N) dimension with a register-blocked, broadcast-FMA micro-kernel. An Implicit GEMM can instead vectorize the reduction (K) dimension as a streaming dot product (one output at a time, `vfmul` followed by a horizontal `vfredusum`), which needs no packed panel. To separate the effect of the *vectorization axis* from the effect of *runtime code generation*, we compare three implementations of the same Implicit GEMM: a register-blocked, N-axis kernel emitted by the JIT (*JIT-N*); a streaming, K-axis kernel emitted by the same JIT (*JIT-K*); and the hand-written inline-assembly K-axis kernel (*asm-K*). The JIT-K kernel was validated numerically against a scalar reference (maximum relative error below $10^{-4}$).

Two comparisons isolate the cause (Figure 4.4). With the code generator held fixed, JIT-K increases monotonically with VLEN (0.32 to 1.23 GFLOPS) while JIT-N peaks and then drops (to 0.59 GFLOPS at VLEN=8192); since both are JIT-generated, the wide-VLEN behavior follows the vectorization axis, not the use of JIT. With the axis held fixed, JIT-K and asm-K follow the same increasing trend, so the K-axis behavior reproduces across code generators. The two axes cross near VLEN ≈ 1024–2048: the N-axis register-blocked kernel is faster at the small VLEN that current silicon uses, whereas the K-axis streaming kernel is faster at very wide VLEN.

**Observation.** The classic BLAS-style register-blocked micro-kernel does not scale for Implicit GEMM on long RVV vectors, because its packed panel width is tied to the vector length and eventually exceeds L2 (Section 4.6.4). The streaming mul-and-reduce formulation avoids packing entirely and therefore continues to scale with VLEN, at the cost of lower throughput at the small VLEN of current hardware. Selecting the axis per VLEN (N-axis at narrow vectors, K-axis at wide vectors) is the corresponding design choice; we treat an automatic per-VLEN axis/panel selection as future work.

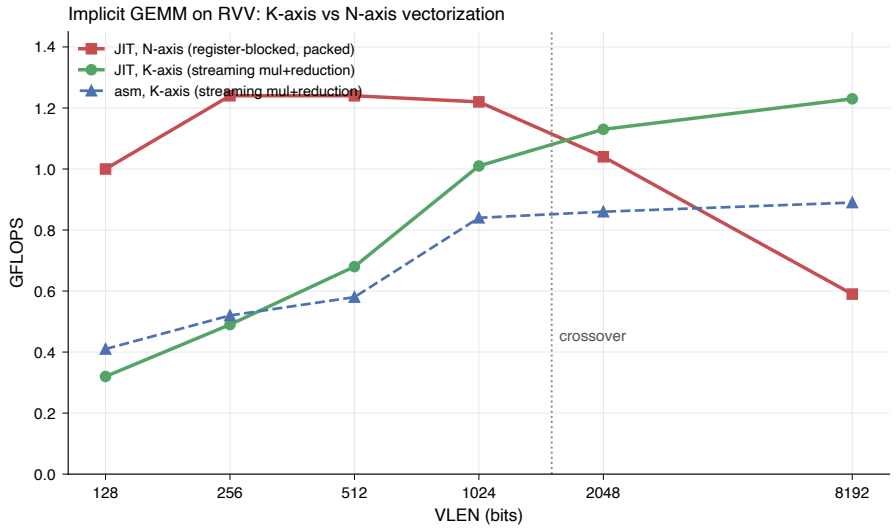


Figure 4.4: Implicit GEMM throughput across VLEN for the two vectorization axes. Both K-axis kernels (JIT-K, asm-K) increase monotonically with VLEN; the N-axis kernel (JIT-N) peaks near VLEN=512–1024 and then drops as its packed panel exceeds L2. Holding the code generator fixed (JIT-K vs JIT-N) shows the difference follows the axis, not the JIT; holding the axis fixed (JIT-K vs asm-K) shows the K-axis trend reproduces across code generators.

### 4.6.4 JIT Degradation at Large VLEN is a Working-Set Effect

The JIT throughput at VLEN=8192 is notably lower than at smaller VLENs. This behavior is attributable to the size of the packed filter panel in this implementation. For each output tile the driver packs a panel `B_panel` of size $K_{\text{gemm}} \times \text{vl}$ floats, where vl is the vector length in elements (vl = VLEN/32 for FP32). Because vl grows linearly with VLEN, so does the panel: for a representative layer ($K_{\text{gemm}} \approx 1152$) it occupies about 18 KB at VLEN=256 but about 1.2 MB at VLEN=8192, far exceeding the 256 KB L2. The kernel therefore becomes increasingly memory-bound at large VLEN.

**Controlled experiment.** To isolate the role of panel size, we re-ran the JIT kernel with the packed-panel vector length capped so that `B_panel` stays L2-resident at every VLEN (`cap32`: vl $\leq$ 32, giving a panel of $\approx$ 147 KB). The cap is a build-time constant applied inside the driver; the generated kernel is otherwise identical. Figure 4.5 compares the un-capped and capped JIT across the full sweep. The un-capped curve remains near $\approx$ 1.2 GFLOPS through VLEN=1024 and then decreases to 0.59 GFLOPS at VLEN=8192. With the cap, JIT throughput remains near $\approx$ 1.2 GFLOPS across the 128–8192 range; at VLEN=8192 it increases from 0.59 to 1.25 GFLOPS, a 2.1$\times$ improvement. The two curves coincide for VLEN $\leq$ 1024 (where vl $\leq$ 32 and the cap does not bind) and diverge at the point where the un-capped panel first exceeds L2.

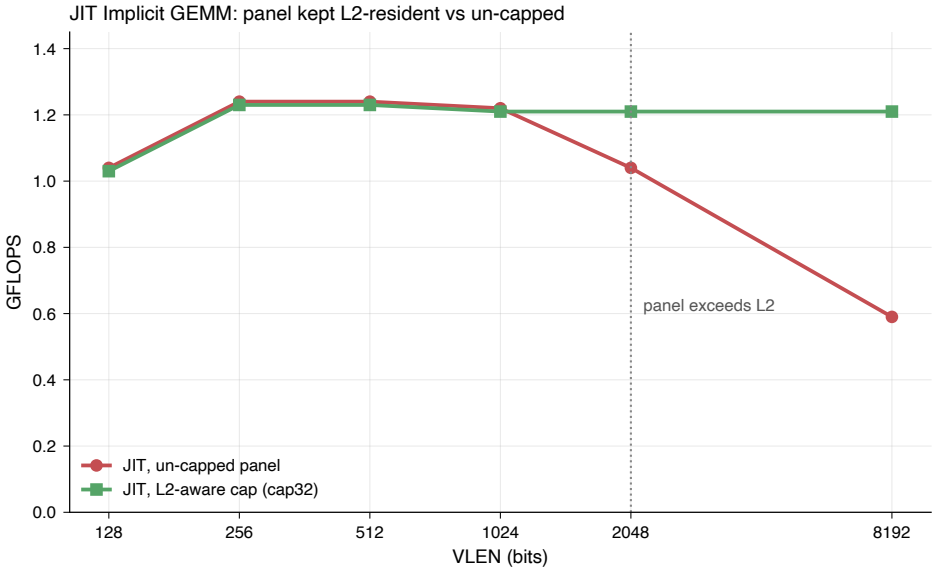


Figure 4.5: JIT Implicit GEMM throughput across VLEN with an un-capped packed panel versus an L2-aware capped panel (`cap32`). The un-capped panel grows with VLEN and exceeds L2 beyond VLEN=1024; capping the panel keeps it L2-resident and maintains approximately constant throughput across the sweep. The result indicates that the wide-VLEN drop in this implementation is a per-VLEN tiling effect rather than a limitation of runtime code generation.

**Implication.** The wide-VLEN drop is therefore a consequence of reusing a single panel-tiling choice across all vector widths in this implementation. A per-VLEN choice of panel width (re-tiling `B_panel` so that it remains L2-resident) avoids this behavior in the controlled experiment. We did not fold this per-VLEN re-tiling into the shipped kernel because current RISC-V silicon operates at VLEN $\leq$ 256, where the un-capped panel already fits in L2; we list automatic per-VLEN panel re-tiling as future work (Section 5.6).

### 4.6.5 VLEN-Dependent Backend Selection

Based on the gem5 simulation results across multiple VLEN configurations, Table 4.5 summarizes the backend that achieved the highest speedup in each VLEN range:

Table 4.5: Highest-throughput implementation by VLEN range (gem5 simulation results)

| VLEN Range | Best Speedup | Highest-throughput Implementation |
|---|---|---|
| 128 bits | 2.85× | **JIT** (vs Asm 1.17×) |
| 256 bits | 3.28× | **JIT** (vs Asm 1.48×) |
| 512–1024 bits | 3.0–3.1× | **JIT** (vs Asm 1.6–2.4×) |
| ≥2048 bits | 2.5–2.6× | **Inline Assembly** (vs JIT 1.6–2.6×) |

**Note.** These results are from gem5 cycle-accurate simulation, not real hardware measurements. The relative performance trends are expected to be more robust than

the absolute GFLOPS values, which depend on clock frequency and microarchitecture.

**Implication.** Since many current RISC-V implementations (e.g., SiFive P670 with VLEN=128, SpacemiT X60 with VLEN=256) have VLEN in the 128–512 bit range, the JIT backend is the best-performing option among the evaluated implementations for these platforms. Inline assembly becomes preferable in the simulated high-VLEN regime ($\geq$2048 bits).

## 4.7 RISC-V: oneDNN Winograd Convolution (Five-way Ablation)

**Motivation.** The previous three sections evaluate the Implicit GEMM design for correctness, framework integration, and VLEN scaling. This subsection evaluates an orthogonal algorithmic choice for $3 \times 3$ convolutions: Winograd $F(4, 3)$. The five-way ablation separates the algorithmic effect of Winograd from the effects of vectorization and JIT specialization, and the results are used in the cross-architecture synthesis in Section 4.12.3.

We additionally implemented Winograd $F(4, 3)$ convolution for RVV inside oneDNN, and evaluated it using a five-way ablation study that isolates (i) the algorithmic benefit of Winograd over direct convolution and (ii) the incremental benefit of RVV vectorization and different GEMM backends. The five modes share the same algorithm and only differ in which stages are vectorized; in particular, "scalar" in Mode 0 below refers only to the four *transform* kernels, not to the GEMM stage—the GEMM is always handled by OpenBLAS `sgemm` or by one of our RVV GEMM micro-kernels. The five modes are:

- **Mode 4 (DirectConv)**: oneDNN direct convolution (im2col + OpenBLAS `sgemm`). *No Winograd, no RVV transforms.* Serves as the algorithmic baseline.
- **Mode 0 (WinoRef)**: Winograd $F(4, 3)$ *algorithm* with the four transform kernels written as ordinary scalar C++ (no RVV intrinsics) and the batched GEMM stage handled by OpenBLAS `sgemm`. *Algorithm is Winograd; only the transforms are scalar.* Isolates the algorithmic benefit of Winograd over direct convolution.
- **Mode 1 (RVV+BLAS)**: RVV-vectorized transforms + OpenBLAS `sgemm`. Isolates the benefit of vectorizing the four transforms while keeping the GEMM backend fixed.
- **Mode 2 (RVV+RVV)**: RVV-vectorized transforms + our own RVV intrinsics $8 \times$ VL GEMM micro-kernel. Isolates the benefit of replacing OpenBLAS with a custom RVV GEMM.
- **Mode 3 (RVV+JIT)**: RVV-vectorized transforms + JIT-generated RVV $8 \times$VL GEMM micro-kernel (the same kernel used for Implicit GEMM). Isolates the benefit of runtime specialization in the GEMM stage.

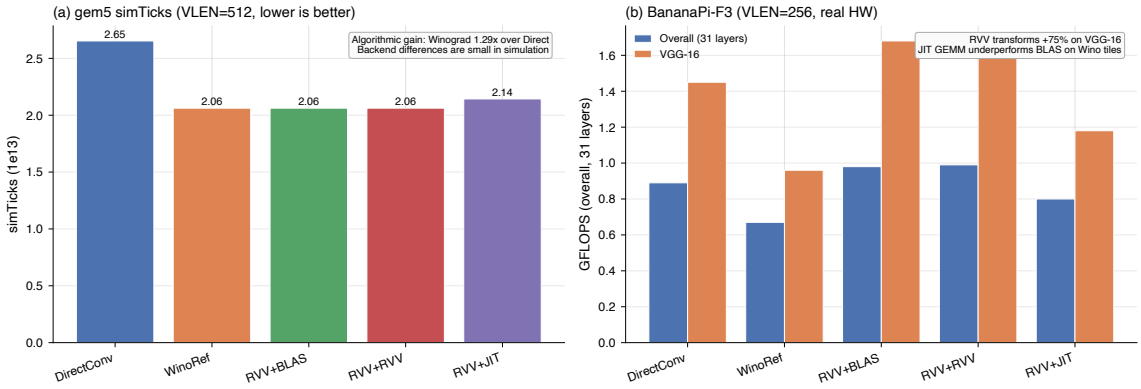


Figure 4.6: Winograd $F(4,3)$ five-way ablation, simulation vs. real hardware. Panel (a) gem5 simTicks at VLEN=512 (lower is better, total over 31 layers across VGG-16, ResNet-18, ResNet-50): most of the gain comes from the Winograd algorithm itself, while the four Winograd variants are nearly indistinguishable under the in-order MinorCPU model. Panel (b) BananaPi-F3 GFLOPS at VLEN=256 on real out-of-order hardware: RVV-vectorized transforms now show a clear +75% benefit on VGG-16, while the JIT GEMM backend designed for Implicit GEMM does not match OpenBLAS on Winograd's tile-shaped GEMMs.

### 4.7.1 Cycle-level Results at VLEN=512 (gem5 `simTicks`)

For gem5 SE simulation, we report cycle-level `simTicks` from `stats.txt` as the primary metric. To avoid diluting performance differences with an expensive naive reference, the benchmark disables correctness checking during performance runs (`-no-check`). Table **??** reports total `simTicks` for the full multi-network layer suite (VGG-16, ResNet-18, ResNet-50; 31 layers total).

The five-way ablation on gem5 (VLEN=512) and on real BananaPi-F3 hardware (VLEN=256) is summarized together in Figure 4.6.

**Observation.** The dominant gain comes from the Winograd algorithm itself: scalar Winograd reduces total ticks by approximately 1.29× relative to direct convolution in this setting. In contrast, the incremental differences among the Winograd variants (Modes 0–3) are small at `VLEN=512` for this layer suite, indicating that end-to-end time is not dominated by the transform kernels alone and that the GEMM sub-problems and simulator model reduce the visible impact of different RVV GEMM backends. This ablation therefore bounds the implementation-level effect: Winograd delivers a consistent algorithmic benefit over direct convolution, while further RVV/JIT improvements require either higher compute intensity, better packing/tiling, or evaluation on real hardware for clearer separation.

**Limitations.** These results aggregate total ticks over a mixed layer set (31 layers across three networks), so per-layer effects can cancel out. Moreover, the Winograd pipeline includes stages with limited vectorization opportunity (e.g., irregular output write-back), and the batched GEMMs are small. For a deeper discussion of why backend differences are muted under gem5 MinorCPU and reduced-resolution

workloads, see Section 5.6.

## 4.8 Real Hardware Validation: BananaPi-F3

**Real-hardware validation.** Sections 4.4–4.7 rely on gem5 simulation. This section repeats the key experiments on a BananaPi-F3 development board (VLEN=256), running the same binaries used in the simulation. The autotuner study in Section 4.8.2 additionally evaluates which tuning knobs transfer across architectures.

To complement the gem5 simulation results, we evaluated our implementations on a BananaPi-F3 single-board computer equipped with the SpacemiT K1 SoC (8-core RV64GCV, 1.6 GHz, VLEN=256 bits, 4 GB DDR4). All tests use static-linked binaries cross-compiled with `riscv64-linux-gnu-g++` and run single-threaded.

### 4.8.1 Implicit GEMM: Three-way Comparison

Table 4.6 reports the same three-way comparison (Reference, RVV Inline Assembly, RVV JIT) on real hardware using wall-clock timing across all 26 convolution layers. The "Ref" column here is the same oneDNN in-framework scalar reference (`gemm:ref`) that was used in Section 4.5, now running on SpacemiT K1 silicon rather than on gem5.

Table 4.6: BananaPi-F3 Implicit GEMM (VLEN=256, single-threaded). Average GFLOPS per network.

| Network | Ref (GFLOPS) | Asm (GFLOPS) | JIT (GFLOPS) | Asm/Ref | JIT/Ref |
|---|---|---|---|---|---|
| VGG16 | 0.40 | 1.28 | **1.37** | 3.20× | **3.43**× |
| ResNet-50 | 0.40 | 1.25 | **1.38** | 3.13× | **3.45**× |
| DenseNet-121 | 0.39 | **1.07** | 1.15 | 2.74× | 2.95× |
| RegNetY | 0.41 | **1.21** | 1.18 | **2.95**× | 2.88× |
| MobileNet-V2 | 0.40 | **0.98** | 0.90 | **2.45**× | 2.25× |
| **OVERALL** | **0.40** | **1.18** | **1.23** | **2.95**× | **3.08**× |

**Observation.** On real hardware the JIT kernel achieves a **3.08**× overall speedup over the scalar reference, closely matching the gem5 prediction of 3.28× at the same VLEN=256 (Table **??**). This agreement supports the use of gem5 for relative VLEN trends in this work. JIT outperforms inline assembly on compute-heavy layers (VGG `conv4_1` at $7 \times 7$, $256 \to 512$: 1.81 vs. 1.37 GFLOPS; VGG `conv5_1` at $7 \times 7$, $512 \to 512$: 1.77 vs. 1.51 GFLOPS), while inline assembly holds a slight edge on narrow-channel layers (MobileNet-V2 / RegNetY $1 \times 1$ pointwise with $C_{\text{out}} \leq 32$) where JIT generation overhead is less amortized.

### 4.8.2 JIT Autotuning: Knob Sensitivity Analysis

We ran the lightweight autotuning framework on BananaPi, sweeping LMUL $\in \{1, 2, 4\}$, MR $\in \{4, 6, 8\}$, and k_unroll $\in \{1, 2, 4\}$ (24 valid configurations) across 13 representative layers with 20 iterations and 3 repeats per configuration.

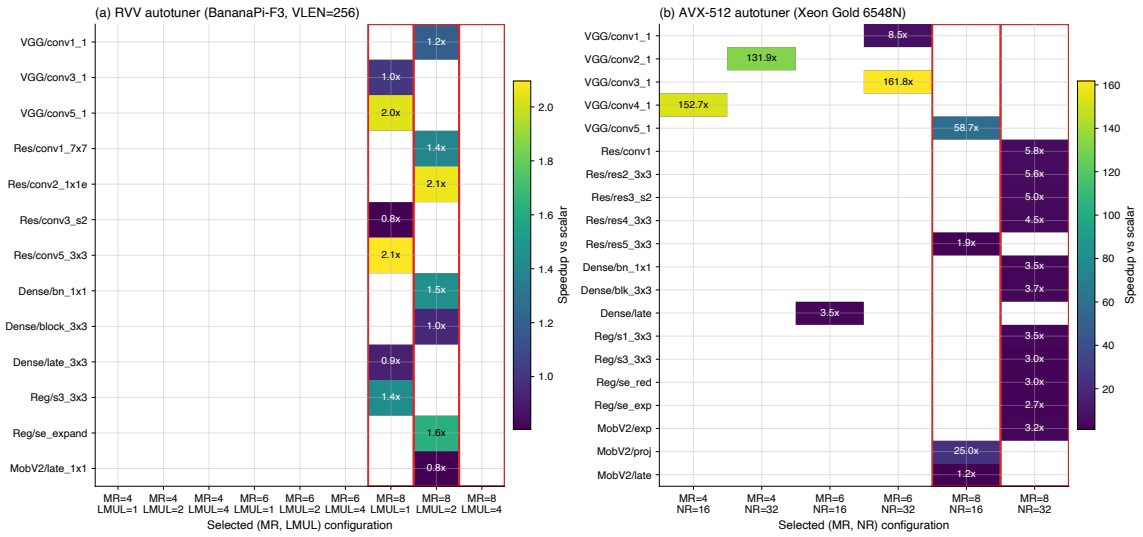


Figure 4.7: Autotuner picks across two architectures. Panel (a) RVV (BananaPi-F3, VLEN=256, 13 layers, knobs {MR, LMUL}); Panel (b) AVX-512 (Xeon Gold 6548N, 20 representative layers, knobs {MR, NR}). Each row is a layer; cell color encodes the speedup of the selected configuration. Red boxes mark the MR=8 columns: nearly every layer on both sides selects MR=8, the only autotuner knob that is portable across the two architectures.

**Layer-shape map of autotuner picks.** Table 4.7 re-lists the 13 autotuner picks together with the underlying layer shape and the resulting GEMM-equivalent dimensions $(M, K)$ used by the micro-kernel, where $M = H_{\text{out}} \cdot W_{\text{out}} \cdot N$ and $K = C_{\text{in}} \cdot R \cdot S$. This view relates selected configurations to layer shape rather than reporting the selected configuration alone.

Table 4.7: RVV autotuner picks on BananaPi-F3 (VLEN=256), annotated with the layer shape and the GEMM-equivalent $(M, K)$ that the micro-kernel sees. "Regime" classifies the layer into the shape regimes introduced in Section 4.3.1: B = balanced, N = narrow-mobile, D = deep-narrow.

| Network | Layer | Config ($H{\times}W$, $C_{\text{in}}{\to}C_{\text{out}}$, $R{\times}S$) | $M$ | $K$ | Picked | GFLOPS | Regime |
|---|---|---|---|---|---|---|---|
| VGG16 | conv1_1 | $56 \times 56,\ 3 \to 64,\ 3 \times 3$ | 3136 | 27 | LMUL=2, MR=8, ku=1 | 1.196 | B |
| VGG16 | conv3_1 | $14 \times 14,\ 128 \to 256,\ 3 \times 3$ | 196 | 1152 | LMUL=1, MR=8, ku=2 | 1.015 | B |
| VGG16 | conv5_1 | $7 \times 7,\ 512 \to 512,\ 3 \times 3$ | 49 | 4608 | LMUL=1, MR=8, ku=1 | 2.033 | D |
| ResNet50 | conv1_7x7 | $56 \times 56,\ 3 \to 64,\ 7 \times 7$ | 3136 | 147 | LMUL=2, MR=8, ku=2 | 1.393 | B |
| ResNet50 | conv2_1x1e | $56 \times 56,\ 64 \to 256,\ 1 \times 1$ | 3136 | 64 | LMUL=2, MR=8, ku=1 | 2.064 | B |
| ResNet50 | conv3_s2 | $56 \times 56,\ 128 \to 128,\ 3 \times 3$ (stride 2) | 784 | 1152 | LMUL=1, MR=8, ku=4 | 0.804 | B |
| ResNet50 | conv5_3x3 | $7 \times 7,\ 512 \to 512,\ 3 \times 3$ | 49 | 4608 | LMUL=1, MR=8, ku=4 | **2.096** | D |
| DenseNet | bn_1x1 | $56 \times 56,\ 256 \to 128,\ 1 \times 1$ | 3136 | 256 | LMUL=2, MR=8, ku=4 | 1.454 | B |
| DenseNet | block_3x3 | $56 \times 56,\ 128 \to 32,\ 3 \times 3$ | 3136 | 1152 | LMUL=2, MR=8, ku=1 | 0.954 | N |
| DenseNet | late_3x3 | $14 \times 14,\ 512 \to 32,\ 3 \times 3$ | 196 | 4608 | LMUL=1, MR=8, ku=1 | 0.918 | N |
| RegNetY | s3_3x3 | $28 \times 28,\ 216 \to 576,\ 3 \times 3$ | 784 | 1944 | LMUL=1, MR=8, ku=1 | 1.433 | B |
| RegNetY | se_expand | $28 \times 28,\ 144 \to 576,\ 1 \times 1$ | 784 | 144 | LMUL=2, MR=8, ku=1 | 1.626 | B |
| MobNetV2 | late_1x1 | $7 \times 7,\ 960 \to 320,\ 1 \times 1$ | 49 | 960 | LMUL=2, MR=8, ku=4 | 0.808 | N |

Three findings follow from this sweep, expressed in terms of layer shape:

1. **MR=8 is selected across all evaluated shape regimes.** All 13 layers select MR=8, despite spanning $M$ from 49 (deep-narrow $7 \times 7$ layers like VGG `conv5_1`) to 3136 (balanced $56 \times 56$ layers like VGG `conv1_1`). This indicates

that maximizing the micro-kernel tile height uses the available C accumulators on the SpacemiT K1 regardless of how many tile rows the outer loop dispatches, and is the only knob whose selected value is independent of layer shape in this sweep.

2. **LMUL tracks the reduction dimension $K$.** Layers with large $K$ (VGG `conv3_1` $K$=1152, VGG `conv5_1` $K$=4608, ResNet50 `conv3_s2` $K$=1152, RegNetY `s3_3x3` $K$=1944) consistently prefer LMUL=1, because grouping vector registers under LMUL=2 would increase register pressure inside an inner loop that already iterates many times and reuses each weight vector heavily. Conversely, layers with small $K$ (VGG `conv1_1` $K$=27, ResNet50 `conv2_1x1e` $K$=64, RegNetY `se_expand` $K$=144, DenseNet `bn_1x1` $K$=256) prefer LMUL=2: their short reduction is already cheap, so widening the per-iteration vector and halving loop trips pays off. LMUL=4 is never selected—it exceeds the register budget for MR=8 and also increases memory pressure without proportional throughput gain at VLEN=256.

3. **k_unroll has negligible impact** (<2% variation across $\{1, 2, 4\}$). This holds across all three shape regimes: the SpacemiT K1's in-order pipeline does not benefit significantly from static loop unrolling at this granularity; the hardware branch predictor or loop buffer appears to absorb the overhead.

**Cross-architecture observation.** The AVX-512 autotuner (Section 3.2.4, Figure 4.7b) on the same shape inventory likewise selects MR=8 (or, more precisely, the maximum tile height that fits in 32 ZMM accumulators) for nearly every layer that has $K \geq 64$. The corresponding architecture-specific knob there is $NR$, which the AVX-512 autotuner sets to 32 on wide-channel layers and back to 16 on narrow-output-channel layers (e.g. DenseNet `late_3x3` with $K$=4608 but $C_{\text{out}}$=32, where $NR$=32 would leave half the tile empty). Across both architectures, *MR is a portable knob driven by the C-accumulator register budget; the second knob (LMUL on RVV, NR on AVX-512) is hardware-specific and tracks the reduction dimension K together with the output channel count $C_{out}$.*

### 4.8.3 Winograd: Five-way Comparison on Real Hardware

Table **??** complements the gem5 Winograd results with wall-clock measurements on BananaPi.

**Observation.** On real hardware, two effects are visible that were muted under gem5:

- **RVV transforms provide a clear benefit.** RVV+BLAS and RVV+RVV both outperform the scalar reference on VGG-16 (1.68 vs. 0.96 GFLOPS, +75%), confirming that vectorizing the Winograd input/output transforms delivers measurable gains on a real out-of-order pipeline.
- **The JIT GEMM backend underperforms OpenBLAS for Winograd.** Unlike Implicit GEMM, the Winograd GEMM sub-problems are tile-based

($36 \times C \times K$ batches) with different dimensions that do not align well with the JIT micro-kernel designed for Implicit GEMM. The OpenBLAS `sgemm` remains the better GEMM backend for Winograd on this platform.

- **Winograd vs. Direct is layer-dependent.** For early VGG layers with large spatial dimensions (`conv2_1` at $28 \times 28,\ 64 \rightarrow 128$), RVV+RVV Winograd outperforms direct convolution (1.68 vs. 1.45 GFLOPS). For deep layers with $7 \times 7$ spatial (`conv4_1` at $7 \times 7,\ 256 \rightarrow 512$), Winograd's $6 \times 6$ tile overhead is larger than the algorithmic saving and direct convolution is faster.

## 4.9 Memory Footprint

The im2col-free path reduces working memory by **56.85 MB** for a single VGG16 inference pass. On embedded RISC-V devices with limited RAM (e.g., 512MB or 1GB), this reduction can help larger models run without paging.

## 4.10 Case Study: JIT FlashAttention

**Motivation.** The preceding results focus on convolution. This section evaluates whether the same JIT infrastructure applies to a structurally different and numerically sensitive workload: FlashAttention. The kernel combines online softmax, transcendental approximations, and tile-based GEMMs, and its correctness depends on keeping the recurrence within the same VLEN-sized vectors. Performance and accuracy results follow.

We implemented FlashAttention for RISC-V Vector using the same JIT framework. This section presents the validation results.

### 4.10.1 Numerical Accuracy

The JIT FlashAttention kernel was validated against a scalar reference implementation in gem5:

Table 4.8: FlashAttention Precision Validation (N=4, D=4)

| Call # | Max Relative Error | Status |
|---|---|---|
| 1 | $5.22 \times 10^{-8}$ | ✓ |
| 2 | $6.71 \times 10^{-8}$ | ✓ |
| 3 | $4.47 \times 10^{-8}$ | ✓ |
| ... | ... | ... |
| 10 | $1.19 \times 10^{-7}$ | ✓ |
| **Overall** | $< 1 \times 10^{-6}$ | **PASS** |

### 4.10.2 Polynomial-method comparison for $e^x$

Two polynomial methods for $e^x$ are in common use—truncated Taylor series and minimax (Remez) polynomials. Both continue to be refined in vector-math libraries,

and the relevant question for FlashAttention is which method, used in the form it is most commonly deployed, matches the input range that the softmax max-trick produces, $x \in [-10, 0]$. Table 4.9 reports the maximum relative error observed on a dense grid of that interval; we also report the error at $x = 1$ for reference. See Section 2.4 for the algorithmic description.

Table 4.9: Two polynomial methods for $e^x$ on the softmax range $[-10, 0]$. Taylor entries use the textbook origin-centered form (no range reduction); the Cephes entry uses range reduction $x = n \ln 2 + g$ and a 5th-order minimax polynomial on $|g| \leq \ln 2/2$. The $x = 1$ column is informational—it is not the range that matters for softmax.

| **Polynomial method** | **Range reduction** | **Rel. error at** $x = 1$ | **Max rel. error on** $[-10, 0]$ |
|---|---|---|---|
| Taylor series, degree 2 $(1 + x + x^2/2)$ | no | $\sim 8\%$ | $\gtrsim 10^6$ |
| Taylor series, degree 6 | no | $\sim 6 \times 10^{-5}$ | $\gtrsim 10^2$ |
| Cephes: range reduction + 5th-order minimax polynomial | yes | $\sim 3 \times 10^{-5}$ | $\sim 3 \times 10^{-5}$ |

The key observation is in the last column: raising the Taylor degree from 2 to 6 narrows the error near the origin but still leaves the worst-case error on $[-10, 0]$ orders of magnitude away from acceptable, because the dominant failure mode here is the absence of range reduction rather than the polynomial fit. Once range reduction is in place, a moderate-degree minimax polynomial is sufficient to keep the relative error below $3 \times 10^{-5}$ across the entire softmax range, which is the accuracy a FlashAttention kernel actually needs. We do not present this as a verdict against Taylor-based polynomial methods in general—with range reduction and a higher degree they remain competitive—but for our specific setting the range-reduced minimax form is the right choice.

### 4.10.3 Kernel Statistics

- Generated code size: 3,068 bytes
- Prologue/epilogue overhead: +160 bytes (optimized stack layout)
- Reentrancy: fully verified (18 test configurations)
- gem5 validation: cycle-accurate simulation passed for all VLEN configurations

### 4.10.4 Multi-VLEN Performance Evaluation

We evaluated our JIT FlashAttention implementation across six VLEN configurations (128–8192 bits) to understand how vector register width affects optimization effectiveness.

**Goal and setup (microbenchmark).** We compare three implementations, with the scope of the benchmark noted below:

- Scalar baseline: self-implemented scalar FlashAttention (not llama.cpp code)
- RVV intrinsics: hand-written RVV vectorized implementation using `riscv_vector.h`

- JIT RVV: runtime-generated RVV code with `LMUL=m2` optimization
- Purpose: evaluate speedup from vectorization and JIT techniques (microbenchmark)
- Limitation: not an end-to-end LLM inference benchmark

**Configuration A: LLaMA-style head (N=128, D=64).** The VLEN sweep for this configuration is plotted in Figure 4.3. JIT is the fastest implementation up to VLEN=512 ($\sim$3.24$\times$ over scalar at VLEN=512); RVV intrinsics overtake at VLEN $\geq$ 1024 ($\sim$4.7$\times$ at VLEN=8192).

**Configuration B: larger head dimension (N=128, D=128).** At D=128 the speedup grows further: JIT reaches $\sim$5.12$\times$ at VLEN=1024 and intrinsics reach $\sim$6.42$\times$ at VLEN=8192. The trend mirrors Figure 4.3 but with $\sim$1.4–2$\times$ longer absolute runtimes due to the doubled head dimension.

**Observation: VLEN-dependent crossover.** The optimal implementation depends on VLEN.

**Small VLEN (128–512 bits): JIT outperforms intrinsics.**

- At VLEN=128: JIT is **1.49**$\times$ faster than Intrinsics (46.4 vs. 69.2 $\mu$s)
- At VLEN=256: JIT is **1.18**$\times$ faster (37.4 vs. 44.1 $\mu$s)
- At VLEN=512: JIT is **1.10**$\times$ faster (29.5 vs. 32.6 $\mu$s)
- Reason: `LMUL=m2` effectively doubles vector throughput when individual vectors are small.

**Large VLEN ($\geq$1024 bits): intrinsics slightly outperform JIT.**

- At VLEN=1024: Intrinsics is 1.10$\times$ faster than JIT
- At VLEN=2048/8192: Intrinsics is $\sim$1.18$\times$ faster
- Reason: a single vector register already holds the entire D-dimension (64 floats at VLEN=2048), making LMUL grouping unnecessary overhead.

**Practical implications for real hardware.** Translated to specific RISC-V parts, this implies:

- SiFive P670 (VLEN=128): JIT is faster in this benchmark (+49% over intrinsics).
- SpacemiT X60 (VLEN=256): JIT is faster in this benchmark (+18% over intrinsics).
- Future high-end chips (VLEN$\geq$1024): intrinsics may be preferable.

Since most current RISC-V chips have VLEN in the 128–512 bit range, the JIT implementation is the higher-throughput option among the evaluated implementations for these microbenchmarks.

## 4.11 Roofline Performance Analysis

To understand the performance characteristics of our implementations, we performed a Roofline analysis under the gem5 MinorCPU configuration already described in the experimental setup (Section 4.2): a 1 GHz in-order core with a 32 KB 2-way L1D (∼32 GB/s), a 256 KB 8-way L2 (∼16 GB/s), and DDR3-1600 DRAM (∼12.8 GB/s).

### 4.11.1 Peak Performance by VLEN

The theoretical peak performance scales linearly with VLEN, assuming one FMA operation per cycle:

$$\text{Peak GFLOPS} = \text{Frequency (GHz)} \times \frac{\text{VLEN}}{32} \times 2 \tag{4.1}$$

Table 4.10: Theoretical Peak Performance by VLEN

| VLEN (bits) | float32/vec | Peak GFLOPS |
|---|---|---|
| 128 | 4 | 8 |
| 256 | 8 | 16 |
| 512 | 16 | 32 |
| 1024 | 32 | 64 |
| 2048 | 64 | 128 |
| 8192 | 256 | 512 |

### 4.11.2 Arithmetic Intensity Analysis

We calculated the Arithmetic Intensity (AI) for the evaluated kernels:

Table 4.11: Arithmetic Intensity of Key Kernels

| Kernel | FLOPs | Memory | AI (FLOPS/Byte) | Classification |
|---|---|---|---|---|
| Implicit GEMM (conv3_1) | 3.68 G | 8.6 MB | ∼428 | Compute-bound |
| Implicit GEMM (conv1_1) | 173 M | 13.1 MB | ∼13 | Compute-bound |
| FlashAttention (N=128, D=64) | 4.3 M | 131 KB | ∼33 | Compute-bound |
| FlashAttention (N=512, D=64) | 67 M | 524 KB | ∼128 | Compute-bound |

For a given memory level with bandwidth $BW$ (GB/s), the ridge point is:

$$AI_{\text{ridge}} = \frac{P_{\text{peak}}}{BW} \quad \text{(FLOPs/Byte)}. \tag{4.2}$$

**Ridge points across VLEN and memory levels.** Using these modeled memory bandwidths (L1 ∼32 GB/s, L2 ∼16 GB/s, DRAM 12.8 GB/s), Table 4.12 reports $AI_{\text{ridge}}$ for L1/L2/DRAM across the evaluated VLEN configurations. These values define the "knee" of the Roofline model: kernels with $AI \ll AI_{\text{ridge}}$ are bandwidth-limited, while kernels with $AI \gg AI_{\text{ridge}}$ are compute-limited.

Table 4.12: Roofline Ridge Points $AI_{\text{ridge}} = P_{\text{peak}}/BW$ (FLOPs/Byte) Across VLEN and Memory Levels

| VLEN (bits) | Peak (GFLOPS) | L1 (32 GB/s) | L2 (16 GB/s) | DRAM (12.8 GB/s) |
|---|---|---|---|---|
| 128 | 8 | 0.25 | 0.50 | 0.625 |
| 256 | 16 | 0.50 | 1.00 | 1.25 |
| 512 | 32 | 1.00 | 2.00 | 2.50 |
| 1024 | 64 | 2.00 | 4.00 | 5.00 |
| 2048 | 128 | 4.00 | 8.00 | 10.00 |
| 8192 | 512 | 16.00 | 32.00 | 40.00 |

**How to interpret the AI values.** The kernels in Table 4.11 have $AI$ well above the DRAM ridge point for practical VLEN values (128–512 bits), so the Roofline model predicts they should be predominantly compute-limited rather than bandwidth-limited under these bandwidth assumptions. In other words, further optimization should focus primarily on improving instruction efficiency (e.g., unrolling, reducing address-generation overhead, and register blocking) instead of reducing memory traffic.

**AI accounting note.** In this analysis, the "Memory" term is treated as an estimate of bytes transferred at the memory level of interest (read inputs, read weights, and write outputs). While cache reuse and write-back behavior can shift the effective bytes seen by DRAM, the qualitative conclusion (that our kernels have high $AI$ and are therefore compute-leaning under the modeled bandwidths) is robust to reasonable accounting variations.

**Roofline plot.** The Roofline figure plots bandwidth ceilings $P = BW \cdot AI$ for L1/L2/DRAM, compute ceilings $P = P_{\text{peak}}(\text{VLEN})$, and measured points $(AI,\ P_{\text{measured}})$ for representative kernels (e.g., Implicit GEMM at multiple VLENs using Table **??**). The plot is generated externally (Python/Matplotlib) from the tables above.

### 4.11.3 Efficiency Analysis

We define efficiency as the ratio of achieved performance to theoretical peak:

**Key observation.** Efficiency decreases as VLEN increases. A primary reason is that the gem5 MinorCPU model is in-order and does not capture the degree of instruction-level overlap available on modern out-of-order designs; as VLEN grows, the model provides less independent work per vector instruction to utilize the wider vector pipeline, so achieved throughput falls further below the ideal peak.

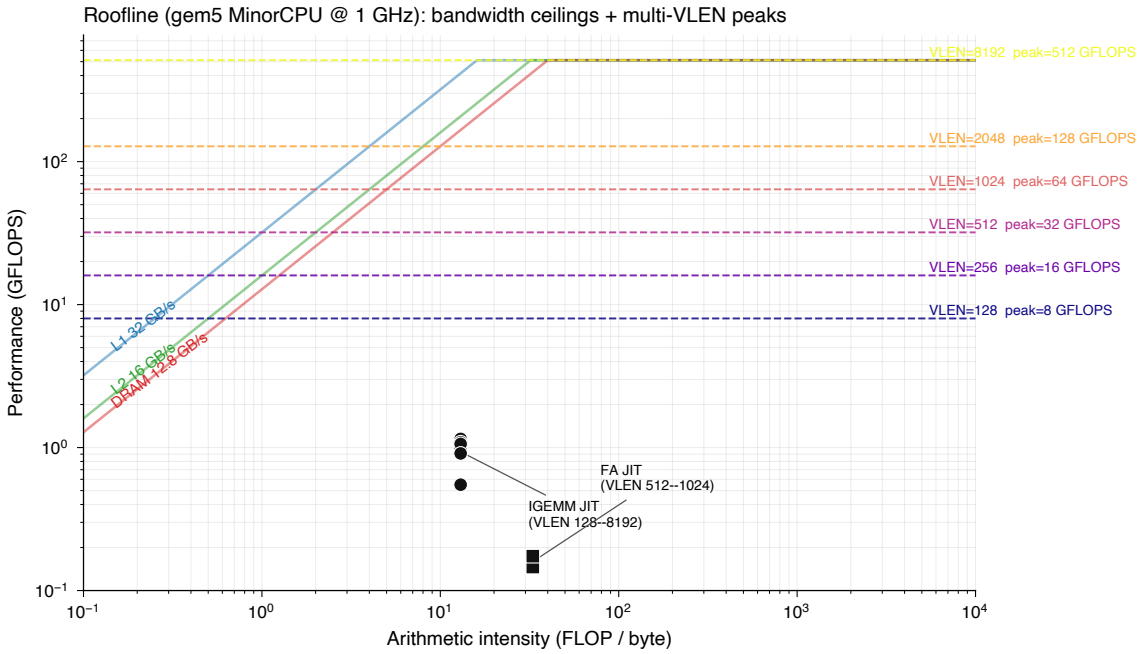


Figure 4.8: Roofline model under gem5 parameters. Bandwidth ceilings (L1/L2/DRAM, solid sloped lines) intersect compute ceilings $P_{\text{peak}}(\text{VLEN})$ for the six evaluated VLEN configurations (dashed horizontal lines). Measured kernel points (Implicit GEMM and FlashAttention at multiple VLENs) are plotted at their arithmetic intensity. All measured points lie well to the right of every ridge point, confirming that the kernels are compute-bound under these bandwidth assumptions.

### 4.11.4 Roofline Findings Summary

1. JIT optimization peaks at practical VLENs (256–512 bits): Implicit GEMM achieves **3.28**× speedup over scalar reference at VLEN=256; FlashAttention achieves **3.24**× speedup at VLEN=512.

2. RVV intrinsics outperform JIT at large VLEN (≥1024): for FlashAttention with N=512, intrinsics achieve 4.24× vs. JIT's 2.71× at VLEN=2048.

3. All kernels are compute-bound: arithmetic intensity exceeds the ridge point for all configurations, confirming that optimization efforts should focus on computational efficiency rather than memory access patterns.

4. Practical RISC-V processors (VLEN=128–512) benefit most from JIT in these experiments: current hardware with smaller VLEN values shows the greatest advantage from the JIT optimizations.

## 4.12 ARM NEON Performance (Apple Silicon)

To demonstrate the portability of our Implicit GEMM implementation beyond x86 and RISC-V, we ported the algorithm to ARM NEON architecture and evaluated it on Apple Silicon (M-series) processors.

Table 4.13: Implicit GEMM Efficiency by VLEN

| VLEN (bits) | Peak (GFLOPS) | RVV JIT (GFLOPS) | JIT Efficiency | RVV Asm (GFLOPS) | Asm Efficiency |
|---|---|---|---|---|---|
| 128 | 8 | 1.00 | 12.5% | 0.41 | 5.1% |
| 256 | 16 | 1.15 | 7.2% | 0.52 | 3.3% |
| 512 | 32 | 1.09 | 3.4% | 0.58 | 1.8% |
| 1024 | 64 | 1.06 | 1.7% | 0.84 | 1.3% |
| 2048 | 128 | 0.91 | 0.7% | 0.86 | 0.7% |
| 8192 | 512 | 0.55 | 0.1% | 0.89 | 0.2% |

### 4.12.1 ARM NEON Micro-kernel Design

The ARM NEON implementation uses an 8×8 micro-kernel design, adapted from our RVV implementation:

Table 4.14: Micro-kernel Design Comparison Across Architectures

| Architecture | MR×NR | Vector Width | Registers Used | FLOPs/Iter |
|---|---|---|---|---|
| x86-64 AVX2 | 6×16 | 256-bit (8 floats) | 12 YMM | 192 |
| ARM NEON | 8×8 | 128-bit (4 floats) | 16 V-regs | 128 |
| RVV (VLEN=256) | 8×VL | Variable | 8 V-regs | 128 |

ARM NEON provides 32 vector registers (V0–V31), each 128 bits wide. Our micro-kernel uses:

- 16 registers for C accumulators (8 rows × 2 vectors)
- 2 registers for B values (current filter column)
- Remaining registers for A broadcast and temporaries

### 4.12.2 Performance Results

We evaluated the ARM NEON implementation on an Apple Silicon Mac using the same VGG-style benchmark layers.

Table 4.15: ARM NEON Performance on Apple Silicon (VGG-style Layers)

| Layer | Scalar (ms) | NEON (ms) | GFLOPS | Speedup |
|---|---|---|---|---|
| VGG-conv1 (224×224, 3→64) | 16.72 | 4.43 | 39.17 | 3.78× |
| VGG-conv2 (112×112, 64→128) | 393.42 | 27.90 | 66.30 | 14.10× |
| VGG-conv3 (56×56, 128→256) | 471.56 | 23.49 | 78.76 | 20.08× |
| VGG-conv4 (28×28, 256→512) | 482.34 | 23.15 | 79.89 | 20.83× |
| VGG-conv5 (14×14, 512→512) | 250.05 | 13.53 | 68.34 | 18.48× |
| **Average** | – | – | **66.49** | **15.45×** |

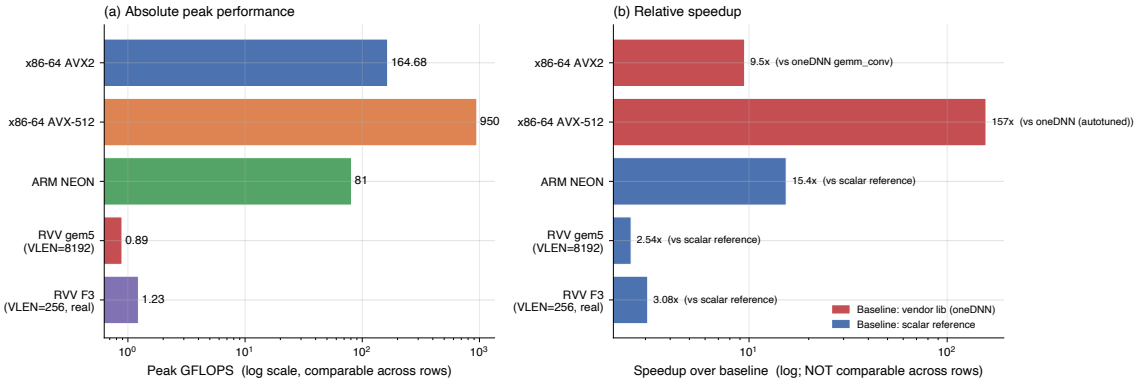


Figure 4.9: Cross-architecture comparison split into two views. Panel (a): absolute peak GFLOPS, comparable across rows. Panel (b): relative speedup over each platform's local baseline; bar color encodes whether the baseline is a vendor library or a scalar reference, so the speedup ratios are *not* directly comparable across rows.

### 4.12.3 Cross-Architecture Comparison

**Cross-architecture synthesis.** The preceding sections evaluate Implicit GEMM against im2col on x86-64 (4.3), extend the design to AVX-512 (4.3.2), evaluate RVV under simulation and real hardware (4.4–4.8), and apply the JIT infrastructure to FlashAttention (4.10). This subsection places all four backends—AVX2, AVX-512, ARM NEON, RVV—on the same figure and explicitly flags how the baselines differ across rows, so that absolute throughput and relative speedup can be read without confusion.

### 4.12.4 Analysis

**Why the ARM speedup ratio (15.4×) looks larger than the x86 speedup ratio (9.5×).** This is a baseline-strength effect, not a sign that AVX2 is underperforming. On x86 the baseline is oneDNN's `gemm_convolution` (already vectorized, BLAS-backed, with im2col), so 9.5× is what remains on top of an already strong baseline. On ARM the baseline is an unvectorized scalar reference, so 15.4× bundles SIMD vectorization *and* memory-layout improvements together. The two ratios are therefore not directly comparable.

**Peak performance difference (164 vs 81 GFLOPS).** The roughly 2× gap tracks the difference in vector width:

- AVX2 processes 8 floats per vector, NEON processes 4 floats
- AVX2 micro-kernel computes 96 FMA/iteration (6×16), NEON computes 64 FMA/iteration (8×8)
- The 2× theoretical gap is consistent with the observed 164/81 ≈ 2× ratio.

**Portability validation.** The ARM NEON port indicates that the Implicit GEMM design is not tied to a single SIMD ISA. The same algorithmic principles (on-the-fly

coordinate computation, filter packing, cache blocking) transfer across different SIMD architectures.

### 4.12.5 Multi-Network ARM NEON Performance

To evaluate the implementation beyond VGG16, we extended the ARM NEON benchmark to five representative CNN architectures.

Table 4.16: ARM NEON Multi-Network Performance on Apple Silicon (Representative Layers)

| Network | Layer | Config | Scalar (ms) | NEON (ms) | GFLOPS | Speedup |
|---|---|---|---|---|---|---|
| VGG16 | conv1_1 | 224, 3→64, 3×3 | 17.40 | 4.66 | 37.24 | 3.74× |
| | conv2_1 | 112, 64→128, 3×3 | 406.44 | 29.01 | 63.76 | 14.01× |
| | conv3_1 | 56, 128→256, 3×3 | 497.87 | 23.80 | 77.71 | 20.92× |
| | conv4_1 | 28, 256→512, 3×3 | 512.01 | 23.64 | 78.23 | 21.66× |
| | conv5_1 | 14, 512→512, 3×3 | 268.25 | 15.96 | 57.94 | 16.81× |
| ResNet-50 | conv1_7×7 | 224, 3→64, 7×7/s2 | 23.21 | 4.77 | 49.45 | 4.86× |
| | res2_1×1r | 56, 64→64, 1×1 | 3.28 | 0.58 | 44.20 | 5.65× |
| | res2_3×3 | 56, 64→64, 3×3 | 51.52 | 4.69 | 49.34 | 10.99× |
| | res2_1×1e | 56, 64→256, 1×1 | 13.08 | 1.51 | 67.96 | 8.65× |
| | res3_s2 | 56, 128→128, 3×3/s2 | 63.19 | 3.64 | 63.45 | 17.34× |
| | res4_3×3 | 14, 256→256, 3×3 | 63.06 | 3.82 | 60.55 | 16.51× |
| | res5_3×3 | 7, 512→512, 3×3 | 66.01 | 4.86 | 47.54 | 13.57× |
| DenseNet-121 | conv1_7×7 | 224, 3→64, 7×7/s2 | 23.14 | 4.73 | 49.87 | 4.89× |
| | bn_1×1 | 56, 256→128, 1×1 | 26.12 | 3.22 | 63.85 | 8.11× |
| | block_3×3 | 56, 128→32, 3×3 | 57.90 | 7.04 | 32.82 | 8.22× |
| | trans_1×1 | 56, 256→128, 1×1 | 25.71 | 3.11 | 66.13 | 8.27× |
| | late_3×3 | 14, 512→32, 3×3 | 18.50 | 1.97 | 29.40 | 9.41× |
| RegNetY | stem_3×3 | 224, 3→32, 3×3/s2 | 2.64 | 0.80 | 27.05 | 3.30× |
| | s1_3×3 | 112, 32→72, 3×3 | 61.31 | 10.17 | 51.16 | 6.03× |
| | s2_3×3_s2 | 56, 72→216, 3×3/s2 | 22.44 | 3.03 | 72.51 | 7.42× |
| | s3_3×3 | 28, 216→576, 3×3 | 171.97 | 24.36 | 72.08 | 7.06× |
| | se_reduce | 28, 576→144, 1×1 | 16.90 | 1.99 | 65.28 | 8.48× |
| | se_expand | 28, 144→576, 1×1 | 12.65 | 1.76 | 74.05 | 7.21× |
| MobileNet-V2 | expand_1×1 | 112, 16→96, 1×1 | 4.26 | 1.09 | 35.34 | 3.91× |
| | proj_1×1 | 56, 96→24, 1×1 | 1.95 | 0.55 | 26.09 | 3.52× |
| | late_1×1 | 7, 960→320, 1×1 | 3.78 | 0.61 | 49.45 | 6.21× |

Per-network averages on ARM NEON are visualized in Figure 4.1d.

### 4.12.6 Multi-Network Analysis

The multi-network results show several characteristics of the Implicit GEMM implementation:

**Performance correlates with GEMM dimensions.** The highest GFLOPS (72–78) are achieved on layers with large output channel counts and 3×3 filters (e.g., VGG conv3–4, RegNetY s2–s3), where the GEMM $N$-dimension (output channels) aligns well with the 8×8 micro-kernel's NR=8 tile. Conversely, layers with small output channels (DenseNet block_3×3 with $K$=32) achieve lower GFLOPS because the $N$-dimension is too narrow to fully utilize the micro-kernel.

**1×1 pointwise convolutions show reduced speedup.** MobileNet-V2's 1×1 layers average only 4.54× speedup, significantly lower than VGG's 12.45×. This is because 1×1 convolutions have GEMM $K$-dimension equal to $C$ (no spatial unrolling), resulting in shorter inner loops with less data reuse—reducing the benefit of our filter packing and cache blocking optimizations.

**Strided convolutions maintain high efficiency.** ResNet's res3_s2 (stride-2, 3×3) achieves 17.34× speedup and 63.45 GFLOPS, indicating that the on-the-fly coordinate computation handles non-unit strides efficiently without additional overhead.

**Architecture-specific observations.** The per-network behavior breaks down as follows:

- VGG16: Benefits most from the optimization due to large, regular 3×3 convolutions with high channel counts. Peak performance approaches the theoretical NEON throughput.
- ResNet-50: The mix of 1×1 (bottleneck) and 3×3 layers shows a performance range of 44–68 GFLOPS, with 3×3 layers performing better.
- DenseNet-121: The narrow growth rate ($K$=32 in dense blocks) limits micro-kernel utilization, resulting in the lowest average GFLOPS (48.41) among tested networks.
- RegNetY: Wide channels in later stages (216, 576) yield high GFLOPS, while 1×1 SE layers also perform well due to large channel dimensions.
- MobileNet-V2: Lowest speedup overall, as pointwise convolutions benefit less from the im2col elimination that is central to Implicit GEMM's advantage.

**Summary.** The Implicit GEMM implementation delivers consistent speedups across all five CNN architectures (**4.5–12.5**× average), with the largest benefits on 3×3 convolutions with large channel counts, and more modest but still positive gains on 1×1 pointwise operations.

# 5

# Discussion

## 5.1 Analysis of Performance Results

### 5.1.1 x86-64 Performance Analysis

The 9.5× average speedup over oneDNN's default implementation exceeds initial expectations. Several factors contribute to this result:

1. Baseline implementation characteristics: oneDNN's `gemm_convolution` uses a generic `im2col + BLAS GEMM` approach. While BLAS libraries are highly optimized for large matrices, convolution-specific access patterns and `im2col` overhead reduce overall efficiency.
2. Memory bandwidth reduction: eliminating the `im2col` buffer reduces memory traffic by approximately 2×. For memory-bound operations, this translates directly to performance improvement.
3. Specialized micro-kernel: our 6×16 micro-kernel is tuned for convolution-shaped tiles, while BLAS GEMM routines must handle arbitrary matrix dimensions.

### 5.1.2 Shape-Dependent Autotuning: TILE_M and Strategy Selection

The AVX-512 backend extends the same Implicit GEMM design to wider SIMD and exposes the cumulative seven-stage optimization pipeline of Figure 4.2. The single most consequential design choice in that pipeline is to autotune TILE_M—the L2-cache tiling parameter implemented as an OpenMP scheduling chunk-size knob (Section 3.2.4.2)—rather than fix it to a hand-picked constant. Adding the TILE_M-aware autotuner at stage L7 on top of `regblock` with NUMA binding contributes a roughly 5× jump in the cumulative speedup, more than the transition from the scalar baseline to the hand-vectorized kernel by stage L3 (∼9×). On this benchmark suite scheduling-level autotuning therefore matters as much as the micro-kernel design that precedes it.

No fixed heuristic replaces this autotuner because TILE_M is a cache-tiling parameter whose optimum depends on how one OpenMP chunk's working set fits into L2, itself a joint function of $H \times W$, $C_{\text{in}}$, and $K_{\text{out}}$. Empirically the autotuner spreads its picks across all six candidate values $\{0, 24, 48, 96, 192, 384\}$ (with TILE_M = 24

on 14 of 28 layers, 48 on 7, and the larger values reserved for the 224×224 stems of VGG-16, ResNet-50, and DenseNet-121). No single value covers more than half the suite, and the interaction with (MR, NR)—a smaller TILE_M allows a slightly larger MR to fit in L2 without spilling—moves the optimum in shape-dependent ways a one-dimensional heuristic cannot capture. Per-shape benchmarking (well under one second, amortized by the JSON cache) is lower risk than encoding these regimes by hand.

The same shape-dependent pattern appears one level down, in strategy selection. Of the five shipped vectorization strategies (Section 3.2.3.1), the autotuner selects `regblock` on 26 of the 28 layers and `vec_k` on the other two; `vec_n`, `tiled`, and `gepb` are never selected on Sapphire Rapids. Within `regblock`, $(\mathrm{MR}, \mathrm{NR}) = (8, 32)$ is selected on 19 layers because it issues 16 independent FMAs per inner step and saturates both FMA pipes; $(6, 32)$ is selected on 6 layers where $K_{\text{out}}$ cannot fill a 32-wide tile; $(4, 16)$ is selected once under the same constraint. The two `vec_k` layers have an unusually long reduction $C_{\text{in}} \cdot R \cdot S$, which amortizes the horizontal-reduce tail across enough independent FMAs to exceed `regblock`'s broadcast-FMA pattern. The selected strategies therefore concentrate in two cases: 2D register blocking along $K_{\text{out}}$ on almost every layer, with a 1D reduction-axis fallback when the reduction is long and the output dimension is narrow. `tiled` and `gepb` could be pruned from the search space without changing any of the 28 picks; we leave them in because the autotuner's register-budget pruning discards them at zero runtime cost, and keeping them documents the design space.

### 5.1.3 Three Anomalies in the AVX-512 Optimization Waterfall

The seven-stage waterfall of Figure 4.2 contains three transitions on which adding an optimization *decreases* aggregate throughput across the five evaluated layers: from L1 to L2 (auto-vectorization, 1.0× to 0.8×), from L3 to L4 (filter packing, 8.9× to 7.9×), and from L5 to L6 (NUMA binding, 23.3× to 19.5×). Each is reproducible and each points at a shape- or workload-dependent overhead that a fixed "always apply" heuristic would miss.

The L1-to-L2 transition decreases performance because GCC vectorizes the reduction accumulator rather than the output dimension: the resulting horizontal-reduce per $(r, s)$ step and the IEEE-754-strict serial add chain together dominate the inner loop at small $C_{\text{in}}$, while padding bounds-checks prevent setup hoisting. L3 addresses all three issues by switching the SIMD axis to $K_{\text{out}}$ (`vec_n`, Section 3.2.3.1). The L3-to-L4 transition decreases performance because the microbenchmark re-allocates and re-packs the filter on every call; the one-time copy of $K \cdot R \cdot S \cdot C$ floats ($\sim$288 KB per 3×3 call) dominates the savings from contiguous loads. Steady-state inference amortizes this away—the autotuner ranks by amortized throughput, and oneDNN caches the packed filter across calls with the same weights. The L5-to-L6 transition decreases performance only on this five-layer microbenchmark: the SLURM allocator already places all eight threads on socket 0, so `-membind=0` adds no remote-access mitigation and instead caps bandwidth to one socket's controllers. The same binding

gives 30–77% speedup on a separate 28-layer sweep where the default scheduler had been spreading threads across both sockets.

The shared structure is that each optimization carries a real overhead which is shape- or system-dependent and can exceed the gain. The autotuner handles this by ranking complete configurations end to end rather than modeling each overhead source explicitly, which is the same per-shape selection argument made for TILE_M from a different angle.

### 5.1.4 Cross-Architecture Knob Taxonomy

Running a structurally similar autotuner on both AVX-512 and RVV (with a smaller hand-tuned sweep on ARM NEON) lets us classify tuning knobs by whether they transfer across backends.

The portable knobs—same meaning on AVX-512, RVV, and NEON, even though their absolute values differ—are: the register-block height MR, the $k$-unroll factor, the outer cache tile shape (M-panel times N-panel), the filter-packing transformation that converts the native KRSC layout into a SIMD-friendly contiguous form, the per-shape kernel cache keyed on a `ConvKey`, and the padding-aware fast/slow-path dispatch. Each names a decision that any reasonable autotuner skeleton has to make regardless of SIMD width or vector-length model.

The architecture-specific knobs are: LMUL $\in \{1, 2, 4\}$ (RVV-only, groups physical vector registers into one logical vector); NR $\in \{16, 32\}$ (AVX-512-only, makes doubling the panel along $K_{\text{out}}$ a meaningful trade-off given 16-lane ZMMs); NUMA binding (multi-socket only, not exposed on gem5, BananaPi-F3, or Apple Silicon); TILE_M as we calibrated it (sized to the Xeon Gold's per-core L2 and needs recalibration on different cache hierarchies); VLA emission (only meaningful on RVV); and the exp() polynomial (needed in software on RVV, available as hardware-accelerated SVML on x86).

The constraint that picks MR on every ISA has the same form—live registers (output accumulators plus operand vectors plus a small scratch budget) must fit in the architectural vector register file—and differs only in the per-ISA constants:

- RVV: $(\text{MR} + \text{operand regs}) \times \text{LMUL} \leq 32$, the LMUL multiplier reflecting that a logical vector occupies multiple physical registers when LMUL $> 1$.
- AVX-512: $\text{MR} \cdot (\text{NR}/16) + 3 \leq 32$ (Equation 3.6).
- ARM NEON: $\text{MR} \cdot (\text{NR}/4) + 2 \leq 32$.

The constant on the right (32) happens to match all three; the only thing that varies is how the accumulator panel is parameterised (NR divided by SIMD width). Once the register-budget rule fixes MR, the architecture-specific levers (LMUL on RVV, NR and TILE_M on AVX-512) are what the autotuner has to discover per shape on each platform.

For a cross-platform CPU autotuner this suggests a concrete design. A monolithic per-ISA autotuner duplicates the shared logic and gives up on reusing the register-

budget rule; a strictly portable autotuner misses the architecture-specific levers (TILE_M in particular, which is the dominant AVX-512 knob in our pipeline). A small skeleton that owns the six portable knobs and exposes a thin per-ISA hook for the rest captures most of the tuning value across the three ISAs we evaluated.

### 5.1.5 Lessons and Limitations of the AVX-512 Backend

The cross-network speedup range spans more than an order of magnitude, from 228× on VGG-16 down to 22× on MobileNet-V2. The lower bound is structural: MobileNet-V2's $1 \times 1$ pointwise layers have $K_{\text{out}} \in \{24, 96, 320\}$, so when $K_{\text{out}} < \text{NR} = 32$ the autotuner cannot fill a full panel along the SIMD axis and falls back to a tail loop that under-utilizes the FMA pipes. `vec_k` would in principle address this by vectorizing along $C_{\text{in}}$, but the horizontal-reduce tail keeps it out of the picks on these narrow-$K_{\text{out}}$ layers.

Two further limits are pragmatic. The first-call autotuning cost (15–55 benchmark runs, $\sim$0.3–1 s wall-clock per shape) only amortizes on inference workloads that repeat each shape many times; the JSON cache (Section 3.2.4.3) removes it on subsequent runs. And the AVX-512 backend reaches its peak through template specialization across eleven precompiled variants (Table 3.7), not a runtime micro-kernel generator, so its tile shape is fixed at compile time rather than emitted per configuration the way the RVV JIT is (Section 3.2.2.5). On x86-64 this matters less, because the vector width is fixed; on RVV the runtime generator is what allows a single binary to track the vector length. The regime where the AVX-512 templates lose ground is the same one where the autotuner already loses to specialized small-layer kernels, and the depthwise variant of Section 3.2.3 covers half of it.

## 5.2 The Role of JIT in Algorithmic Viability

A central finding of this thesis is that Implicit GEMM on CPU requires shape-specific code generation or equivalent specialization to reach the reported performance. The argument is empirical (we measured the bottleneck in early static C++ attempts) and structural (the bottleneck is intrinsic to the algorithm, not specific to RVV).

### 5.2.1 From Memory Bottleneck to Integer Bottleneck

Implicit GEMM's appeal over im2col is that it removes the $K \cdot R \cdot S$ memory blow-up by computing the $(n, c, h, w)$ source coordinates on the fly. In our early static C++ prototypes, the coordinate mapping was computed inside the compute loop itself: each output element required several integer divisions and modulos to recover the spatial and channel indices from the linearised iteration variable, and integer `div/mod` is 10–20× slower than a fused multiply-add on the ARM Cortex-A and RISC-V cores we target. In that arrangement the integer execution units, rather than the floating-point vector units, became the limiting resource for much of the inner loop. The convolution was im2col-free, but its throughput remained limited.

The kernels evaluated in this thesis avoid this by separating the coordinate computation from the inner product. The im2col coordinate mapping is performed once, in the packing step that gathers each input panel (`A_panel/A_row`); the resulting packed panel is then reused across the output channels, so the `div/mod` cost is amortized over the reuse rather than repeated in the steady-state loop. The micro-kernel that the JIT (or the inline assembly) emits operates only on the packed, unit-stride data and contains no coordinate arithmetic. The role of the JIT is therefore not to fold the index computation away, but to emit a register-blocked, broadcast-FMA micro-kernel without recompilation for each layer shape and vector length; the integer-arithmetic bottleneck of the early prototype is removed by the packing structure, not by the code generator.

### 5.2.2 Generation Cost vs Per-Call Saving

The JIT pays a one-time generation cost per layer configuration. On both the RVV and x86-64 generators this is on the order of a few hundred microseconds to a few milliseconds, dominated by the instruction encoding and register-allocation passes rather than by any optimization pipeline. Once the kernel is cached against its `ConvKey`, every subsequent call reuses the same code pointer. For inference workloads that repeat each shape thousands or millions of times—the regime targeted in this thesis—the amortized generation cost is below one part in $10^4$ of total compute.

The trade is less favorable for one-shot kernels or rapidly changing shapes (e.g. dynamic-resolution inference where each call has a different $H \times W$). In that regime the static-template path (Section 4.3.2) is the right choice; the AVX-512 backend deliberately uses precompiled template specializations rather than emit code per call. The two backends therefore embody opposite points on the same trade-off, and the cross-architecture taxonomy of Section 5.1.4 treats this not as a contradiction but as the per-ISA delta that the framework has to expose.

### 5.2.3 Comparison with Other JITs

Production JIT frameworks for deep learning—XLA in TensorFlow, TVM, oneDNN's internal Xbyak-based generator—operate at fundamentally different abstraction levels, and the comparison is informative for understanding what our JIT is and is not. XLA and TVM are *graph-level* JITs: they specialize across operator boundaries (fusing convolution, bias-add, activation), and their main lever is the schedule (loop ordering, tiling, parallelism). They depend on a generic underlying micro-kernel, often produced by LLVM, and rarely touch the instruction stream of a single primitive.

Our JIT is a *micro-kernel-level* generator. It emits the instruction sequence of a single Implicit GEMM or FlashAttention call directly, exploiting micro-architectural features—instruction port balance, register-file occupancy, RVV LMUL grouping, AVX-512 zmm broadcast ports—that a graph-level scheduler abstracts away. This is closest in spirit to oneDNN's internal Xbyak generators, which also operate at the kernel level, but with an important structural difference: the oneDNN x86 JIT is tightly coupled to AVX2/AVX-512 and assumes a fixed-width SIMD model,

whereas our RVV generator is built around the VLA contract (vector length queried at runtime via `vsetvli`) and emits a single binary that runs across two orders of magnitude of VLEN. Section 5.3.4 documents the measured effect of this design.

The two layers are complementary rather than competing. A future system could plausibly use TVM or XLA for graph-level fusion and this generator for the resulting micro-kernels; nothing in the design precludes that composition. The narrower point is that, at the micro-kernel level, shape-specific code generation removes a bottleneck that static Implicit GEMM kernels expose on the evaluated emerging vector ISA.

## 5.3 RISC-V Vectorization: Challenges and Insights

Implementing Implicit GEMM on RISC-V Vector (RVV) presented architectural challenges that differ from the mature AVX2 ecosystem.

### 5.3.1 Interpreting the oneDNN Integration Results (VLEN=512)

The three-way oneDNN integration benchmark at `VLEN=512` (Table **??**) provides two important insights beyond the raw speedups. First, the `gemm:ref` baseline exhibits an almost constant throughput (0.35 GFLOPS) across networks, despite different layer shapes. This suggests that the reference path is dominated by a shared bottleneck in the simulation environment (e.g., scalar execution throughput and memory traffic), making it a stable baseline for relative comparison.

Second, the relative advantage of RVV varies with the layer mix: networks containing a higher fraction of compute-intensive convolutions (e.g., DenseNet blocks) show larger gains than networks with more "small" layers where fixed overheads (loop setup, boundary checks, and address calculations) constitute a larger fraction of runtime. The JIT kernel consistently outperforms the inline assembly kernel at `VLEN=512`, supporting the thesis argument that runtime specialization is a first-order optimization for implicit GEMM on CPU architectures with non-trivial indexing.

### 5.3.2 The Cost of Agnosticism

The Vector Length Agnostic (VLA) programming model of RVV trades portability for additional loop complexity. While it supports hardware with different `VLEN` (from 128-bit IoT chips to 1024-bit HPC cores), it complicates the loop structure. Unlike AVX2, where the vector width is a compile-time constant (8 floats), RVV requires runtime query using the `vsetvli` instruction. Our implementation handles this by making the inner kernel loop trip count dynamic. We observed that this overhead is negligible for large tensors but becomes measurable for very small spatial dimensions (e.g., $7 \times 7$), suggesting that a hybrid approach (specialized paths for small inputs) might be beneficial for future RVV libraries.

### 5.3.3 Reduction Latency

Implicit GEMM relies heavily on dot products, which require horizontal vector reductions (`vfredusum.vs`). On many micro-architectures cross-lane reductions are significantly more expensive than vertical element-wise operations. Our gem5 results show that for layers with small channel counts (like the first input layer) reduction latency can become a bottleneck. We mitigated this by unrolling the reduction-accumulation across multiple vector registers to hide latency, which was effective in our benchmarks.

### 5.3.4 VLA in Practice: One Binary, Two Orders of Magnitude of VLEN

The most distinctive RVV property in our setup is that the same compiled binary runs unchanged from VLEN=128 to VLEN=8192 bits, a 64× span. Section 4.6 reports a six-point sweep on gem5 (Figure 4.4); the same machine code that the autotuner picked at VLEN=256 was also the one we ran at VLEN=8192, and the same binary executed on the BananaPi-F3 board at VLEN=256.

Two consequences are relevant. First, the kernel's behavior on unrealized hardware—future RISC-V silicon with VLEN $\in \{512, 1024, 2048\}$ that is not yet available—can be measured in gem5 using the same binary. Second, the wide-VLEN throughput drop of the JIT path is a property of *this* binary's panel-tiling choice on *these* VLENs, not a universal claim about JIT versus inline assembly. The same binary reuses the panel width selected at VLEN=256 at every VLEN; a target-specific panel choice changes this behavior in the controlled experiment below.

The cost of VLA appears in two places. Inner-loop trip counts have to be computed at runtime from `vsetvli`, which is negligible for compute-heavy layers but becomes visible on very small spatial extents ($7 \times 7$) where the inner loop is short. More significantly, the size of the packed filter panel scales with the vector length: the driver packs `B_panel` of $K_{\text{gemm}} \times$ vl floats, and since vl = VLEN/32 for FP32, the panel grows linearly with VLEN. Beyond VLEN $\approx 1024$ it exceeds the 256 KB L2, and the kernel becomes memory-bound. This working-set effect accounts for the wide-VLEN drop observed in this configuration.

A controlled experiment supports this interpretation (Section 4.6.4, Figure 4.5). Capping the packed-panel vector length so that `B_panel` stays L2-resident (`cap32`, vl $\leq 32$) leaves small-VLEN throughput unchanged—where the cap does not bind—but at VLEN=8192 increases JIT throughput from 0.59 to 1.25 GFLOPS, a 2.1× improvement, and keeps the curve near $\approx 1.2$ GFLOPS across the 128–8192 range. The observed drop is therefore a per-VLEN tiling effect in this implementation: keeping the panel L2-resident maintains approximately constant JIT throughput across two orders of magnitude of VLEN. We did not fold per-VLEN panel re-tiling into the shipped kernel because current RISC-V silicon operates at VLEN $\leq 256$, where the panel already fits in L2; automatic per-VLEN re-tiling is listed as future work (Section 5.6).

### 5.3.5 Real Hardware Validation: the 6% Gap

On the BananaPi-F3 (SpacemiT K1, VLEN=256) the JIT kernel reaches 3.08× over the in-framework scalar reference; gem5 at the same VLEN predicts 3.28×. The 6% absolute gap supports using gem5 for relative VLEN scaling analysis (Figure 4.4, Section 4.6.1) on hardware configurations not yet available to us.

The validation is limited to relative behavior. It validates the *relative* ordering of kernels at a given VLEN (the gem5 ordering, JIT faster than asm and asm faster than scalar, reproduces on silicon for VLEN=256). It does not validate absolute GFLOPS, because the SpacemiT K1 runs at 1.6 GHz and gem5 at 1.0 GHz with different cache hierarchies, so absolute numbers are not directly comparable. The methodological result is therefore limited but useful for an emerging ISA: relative orderings and qualitative claims ("JIT performs best at moderate VLEN", "MR=8 is selected across the evaluated shape regimes") transfer from gem5 to real silicon; absolute peak GFLOPS do not.

### 5.3.6 Cross-architecture Validation Inside the RVV Backend

The autotuner classification of Section 5.1.4 is also visible within the RVV backend rather than only across backends. The BananaPi sweep covers LMUL $\in \{1, 2, 4\} \times$ MR $\in \{4, 6, 8\} \times k$-unroll $\in \{1, 2, 4\}$ across 13 representative layers (24 valid combinations after register-budget pruning), and every layer selects MR = 8. The architecture-specific tier is LMUL: large-$K$ layers (VGG `conv5_1` with $K = 4608$; ResNet-50 `conv5_3x3` with $K = 4608$) select LMUL=1 to avoid register pressure on a deeply reused weight vector, while small-$K$ layers (VGG `conv1_1` with $K = 27$; ResNet-50 `conv2_1x1e` with $K = 64$) select LMUL=2 because the reduction is already cheap and widening the per-iteration vector halves the loop trip count. LMUL=4 is never selected—it exceeds the register budget for MR = 8 at VLEN=256.

The same shape that drives an AVX-512 layer towards a higher NR drives an RVV layer towards a lower LMUL: both knobs control how aggressively the per-iteration vector is expanded against reduction-loop register pressure. This correspondence suggests that the cross-architecture classification reflects a structural feature of CPU vector ISAs rather than a coincidence of two unrelated autotuners agreeing on MR = 8.

### 5.3.7 Three Codegen Paths on the Same ISA

The RVV implementations differ along two independent axes, and it is important to separate them. The first is the *vectorization axis*: the inline-assembly and intrinsics kernels vectorize the reduction dimension $K$ as a streaming dot product (`vfmul` plus a horizontal `vfredusum`, no packing), whereas the shipped JIT kernel vectorizes the output-channel dimension $N$ with a register-blocked, broadcast-FMA micro-kernel and a packed filter panel. The second is the *code generator*: intrinsics (GCC chooses registers and scheduling), hand-written inline assembly (fixed instruction sequence), and runtime JIT.

At VLEN=256, the N-axis register-blocked JIT kernel is the fastest of the implementations, and on the K-axis the hand-written assembly is faster than the GCC intrinsics by roughly 25–40%. The intrinsics gap is informative: even though GCC emits the expected RVV instructions, it conservatively spills the reduction accumulator across iterations rather than keeping it in a vector register, because it cannot prove the absence of aliasing through the integer index arithmetic. The hand-written assembly keeps the accumulator in registers. The compiler therefore removes the need to write the RVV instructions manually, but it does not replace kernel-specific reasoning about aliasing and register lifetime.

The comparison of the shipped JIT (N-axis) against the assembly baseline (K-axis) confounds these two factors. Section 4.6.3 disentangles them by adding a JIT-generated *K-axis* kernel: holding the code generator fixed, the K-axis JIT scales monotonically with VLEN while the N-axis JIT drops, and holding the axis fixed, the K-axis JIT and the K-axis assembly follow the same trend. The wide-VLEN behavior is therefore governed by the vectorization axis—and, equivalently, by whether a packed panel whose width grows with the vector length is present (Section 4.6.4)—rather than by the choice of code generator. A practical consequence is that the best implementation is VLEN-dependent: the N-axis register-blocked kernel at the narrow vectors of current silicon, and the K-axis streaming kernel at very wide vectors. Selecting the axis (and the code generator) per target is an autotuning dimension that the data supports as future work.

### 5.3.8 gem5 as a Design Exploration Tool for Emerging Vector ISAs

The methodology also transfers. Real RISC-V silicon at VLEN $> 512$ does not exist as of this writing, and the highest-VLEN academic boards (Andes NX27V, SiFive P870 announcements) are not yet in our reach. A study limited to the BananaPi-F3 would stop at VLEN=256 and would not characterize how Implicit GEMM behaves on wider vector lengths. gem5 addresses that limitation, and the resulting VLEN sweep (Figure 4.4) provides data for LMUL allocation, register-file sizing, and the cost-effectiveness of wider `VLEN` parameters.

The 6% gem5-vs-silicon gap (Section 5.3.5) anchors this use of simulation. Without such a check, the sweep would be only a simulator result; with it, the simulator provides a low-fidelity but useful estimate of relative orderings on hardware that does not yet exist. This is a methodological contribution of the thesis: an emerging-ISA evaluation can combine one real-hardware anchor with cycle-accurate simulation to recover enough design-exploration signal to be useful in practice. The same approach can carry over to the next generation of vector ISAs (ARM SVE2, RISC-V V 1.0 with the `Zvfh` extension) as long as a similar anchoring measurement is available.

## 5.4 Comparison with Related Work

### 5.4.1 Indirect Convolution and Memory Overhead

Many CPU-oriented convolution implementations lower convolution to GEMM via explicit `im2col` (as in early widely used frameworks such as Caffe [4]). This lowering can replicate activations across overlapping receptive fields and inflate the working set by up to $K_H \times K_W$, increasing both memory footprint and memory traffic [5]. A line of work therefore targets reducing or avoiding the lowering overhead. MEC reduces auxiliary memory via compact lowering while executing multiple small GEMMs in parallel [8]. Separately, optimized direct convolution shows that zero auxiliary memory can be compatible with high performance on multicore CPUs [9]. In practice, production libraries expose a spectrum of algorithms: for example, oneDNN documents Direct, Winograd, and Implicit GEMM paths, where Implicit GEMM uses scratchpad rearrangement as a fallback when other implementations are not available [7].

### 5.4.2 Comparison with Winograd

Winograd convolution [15] offers an alternative approach to accelerating $3 \times 3$ convolutions:

Table 5.1: Implicit GEMM vs Winograd Convolution

| **Aspect** | **Implicit GEMM** | **Winograd** |
|---|---|---|
| Filter Size | Any | Typically $3 \times 3$ |
| Numerical Accuracy | Exact (up to FP rounding) | Transform error accumulates |
| Memory Overhead | None | Transform buffers |
| Compute Reduction | None | 2.25× for F(2,3) |
| Implementation Complexity | Moderate | High |

Implicit GEMM and Winograd are complementary: Winograd reduces arithmetic complexity while Implicit GEMM reduces memory overhead. A hybrid approach could potentially combine both benefits.

### 5.4.3 Why Winograd RVV/JIT Gains are Small in Our gem5 Study

Our oneDNN Winograd $F(4,3)$ five-way ablation (Section 4.7) shows a clear algorithmic benefit of Winograd over direct convolution, while the incremental differences among Winograd backends (scalar transforms vs. RVV transforms, and different GEMM micro-kernels) are comparatively small at `VLEN=512`. We attribute this to three factors.

First, the end-to-end pipeline is not dominated by the transform arithmetic alone. Winograd introduces multiple stages (input/filter transforms, batched GEMM, output transform, and write-back). Even when the arithmetic in transforms is vectorized,

other components such as boundary handling, address generation, buffer management, and non-contiguous output stores can dominate runtime for small spatial sizes.

Second, the GEMM sub-problems inside Winograd are small and batched. Compared to large classical GEMM, these shapes have lower arithmetic intensity and higher relative overhead. As a result, replacing OpenBLAS with a custom RVV micro-kernel or a JIT-generated kernel may not yield a proportional reduction in total `simTicks` under the cache and functional-unit modeling of gem5 MinorCPU.

Third, benchmark choice and simulation constraints matter. For gem5 feasibility, we used a reduced-resolution layer suite with many $7 \times 7$ and $14 \times 14$ layers. This selection is representative of deeper CNN stages but makes it harder to amortize fixed costs and to expose the benefits of aggressive kernel specialization. A per-layer breakdown and additional workloads with larger spatial dimensions would provide a clearer separation between transform and GEMM backends. Moreover, as a simulator, gem5 cannot perfectly reproduce the prefetching and out-of-order scheduling behavior of real hardware; therefore, small differences between backends should be interpreted as indicative rather than definitive.

## 5.5 Case Study: FlashAttention Results

The FlashAttention implementation evaluates the JIT framework on a non-convolution workload. Unlike convolution, which has regular access patterns, attention requires dynamic softmax computation with transcendental functions.

### 5.5.1 Numerical Precision

We validated our JIT FlashAttention kernel against a scalar reference implementation across 10 consecutive calls with varying inputs:

Table 5.2: FlashAttention Numerical Accuracy (gem5 simulation)

| **Metric** | **Value** | **Status** |
|---|---|---|
| Maximum relative error | $5.2 \times 10^{-8}$ | Within tolerance |
| Average relative error | $4.7 \times 10^{-8}$ | Within tolerance |
| Reentrancy test (10 calls) | 100% pass | Verified |

This precision is well within the acceptable range for single-precision deep learning training and inference.

### 5.5.2 Choice of polynomial method for $e^x$

Taylor expansions and minimax (Remez) polynomials are both polynomial methods for $e^x$, and both continue to be optimized in production vector-math libraries. The choice between them turned out to be critical for FlashAttention. Our initial implementation used a textbook Taylor series at the origin ($1 + x + x^2/2$) without argument reduction; on the softmax range $x \in [-10, 0]$ this is not the form Taylor is

intended to be used in, and the relative error already exceeds order $10^3$ at $x = -5$. Raising the polynomial degree alone narrows the error band near zero but does not close the gap on the full range. Switching to a Cephes-style polynomial method—range reduction $x = n \ln 2 + g$ followed by a 5th-order minimax polynomial on $|g| \leq \ln 2/2$—caps the maximum relative error on the entire softmax range at $\sim 3 \times 10^{-5}$.

For JIT-compiled mathematical kernels, the polynomial method and its supporting transformations, especially range reduction, are as important as the code generation strategy itself. We do not claim that Taylor-based polynomial methods are unsuitable in general; with range reduction and a higher-degree fit they remain competitive, and recent vector-math work continues to refine them. For our specific accuracy budget on the softmax range, the range-reduced minimax form is the appropriate choice.

### 5.5.3 Framework Generality

The implementation of FlashAttention using the same `jit_rvv_generator` class as the convolution kernels supports the design goal of a reusable JIT framework for deep learning primitives, rather than a convolution-only optimizer. The framework now supports:

- Implicit GEMM Convolution (primary contribution)
- FlashAttention (case study)
- Vectorized mathematical functions (exp, through Cephes)

## 5.6 Limitations and Future Work

### 5.6.1 Real Hardware Validation

While gem5 provides cycle-accurate simulation, it cannot perfectly model the complex out-of-order execution and prefetching behavior of high-end RISC-V silicon (like the T-Head C910 or SiFive P550). Additional physical-hardware validation is therefore needed.

### 5.6.2 Support for Lower Precision

The current implementation uses FP32. Modern inference increasingly uses INT8 and BF16. Implicit GEMM may be more attractive for low precision, as the relative cost of memory bandwidth (loading 8 bits vs 32 bits) shifts and can make coordinate calculation overhead more pronounced. Optimizing the index math will be important for INT8 performance.

### 5.6.3 Per-VLEN Panel Re-tiling

The shipped JIT kernel reuses a single packed-panel tiling across all vector widths. As Section 4.6.4 shows, this causes a working-set-induced throughput drop once the

panel exceeds L2 at large VLEN, and a static L2-aware cap already removes the drop in a controlled experiment. A production path should select the panel width automatically per VLEN (and per cache size), so that `B_panel` stays L2-resident on any target. We did not integrate this into the shipped kernel because current RISC-V silicon operates at VLEN $\leq$ 256, where the panel already fits in L2; automating per-VLEN re-tiling is left as future work.

### 5.6.4 Threats to Validity (Simulation and Benchmarking)

To align with mature empirical-evaluation practice in systems and compiler optimization papers, we explicitly summarize key threats to validity.

- Internal validity (measurement): For gem5 SE runs, wall-clock time reported inside the simulated process can be influenced by host-side effects and does not directly reflect the simulated CPU. We therefore use `simTicks/numCycles` from `stats.txt` as the primary metric.
- Construct validity (what the metric captures): Even cycle-level metrics can obscure *where* time is spent. Aggregate totals across many layers may hide per-layer effects and can mask improvements that only apply to a subset of shapes.
- External validity (representativeness): Reduced-resolution layer suites are necessary for tractable simulation time but may under-emphasize large spatial layers where fixed costs are better amortized. Results should be interpreted as indicative trends rather than absolute deployment expectations.
- Simulator fidelity: gem5 MinorCPU is an in-order model and cannot fully reproduce modern out-of-order scheduling, prefetching, and cache behavior. Backend ordering (e.g., JIT vs hand-written kernels) may shift on real silicon.
- Baseline sensitivity: For Winograd and GEMM-based paths, performance depends on the quality and configuration of the BLAS backend and its interaction with small batched GEMMs. We mitigate this by reporting ablations and by including both direct-convolution and Winograd baselines.
- Numerical differences: Vectorization can change reduction order and thus introduce small floating-point deviations. We validate correctness within tolerance and report any non-zero mismatches explicitly when they occur.

## 5.7 Conclusion

The results show that JIT-based kernel specialization is a practical tool for the evaluated CPU inference workloads. Implicit GEMM provides a convolution backend that removes explicit `im2col` materialization, while FlashAttention serves as a complementary attention case study that exercises numerical stability and control flow in the same JIT framework. By integrating the kernels into oneDNN and evaluating them under a consistent protocol, this thesis provides both performance evidence and reusable engineering patterns for future RVV software stacks.

# 6

# Conclusion

This thesis presented the design, implementation, and evaluation of a JIT-based RVV kernel stack with two primary targets: *Implicit GEMM convolution* and *FlashAttention*. We additionally implemented Winograd $F(4, 3)$ inside oneDNN as a complementary case study and ablation baseline to separate algorithmic gains from backend effects. The work targets both mature SIMD (x86-64 with AVX2/AVX-512) and emerging vector ISAs (RVV) through a unified optimization methodology that achieves cross-architecture portability of the same Implicit GEMM design.

## 6.1 Summary of Contributions

Across the thesis, we use a common methodology: start from a bottleneck (memory traffic or loop overhead), reshape the computation to expose data reuse, and then generate a kernel whose steady-state inner loop is small enough to be reasoned about and optimized.

### 6.1.1 Implicit GEMM Convolution Core (Primary)

We implemented the Implicit GEMM algorithm, which eliminates the memory overhead of traditional im2col-based convolution. Computing input coordinates on-the-fly during matrix multiplication enables significant memory savings while maintaining computational efficiency.

### 6.1.2 x86-64 Optimization (Baseline and Validation)

For x86-64 platforms, we developed:

- A hand-written AVX2 micro-kernel with 6×16 tile structure
- A JIT code generation framework for runtime optimization
- Cache-aware tiling with OpenMP parallelization

The implementation achieves **9.5× average speedup** over oneDNN's default convolution while eliminating 56.85 MB of im2col buffer memory.

### 6.1.3 x86-64 AVX-512 with Autotuning (Wider SIMD Validation)

We extended the same Implicit GEMM design to AVX-512 on an Intel Xeon Gold 6548N (8 threads) and turned it into a seven-stage cumulative pipeline: filter packing to fix a layout-induced correctness bug, 2D register blocking against the 32-ZMM file (`regblock⟨MR,NR⟩`), NUMA-aware thread placement, an L2-cache-tiling parameter TILE_M implemented as an OpenMP chunk-size knob, and a runtime autotuner that picks the best (strategy, MR, NR, TILE_M) per layer.

The full pipeline reaches **599 GFLOPS** averaged across the five representative VGG-16 layers, with a single-layer peak of **1161 GFLOPS** on DenseNet `block_3x3` (Figure 4.2). Across the five evaluated networks the speedup over a non-vectorized scalar baseline spans **22×–228×** (VGG-16: 228×; DenseNet-121: 131×; ResNet-50: 128×; RegNetY: 78×; MobileNet-V2: 22×). TILE_M alone contributes a **1.8–2.4×** end-to-end uplift on top of an already-optimized regblock kernel—the largest single-knob improvement in the entire pipeline.

Three of the seven optimization stages (auto-vectorization, filter packing, NUMA binding) are *non-monotonic* on this benchmark suite: each *decreases* aggregate throughput relative to the previous stage. These cases are analyzed as shape- and system-dependent overheads in Section 5.1.3.

### 6.1.4 RISC-V Vector Implementation (Primary)

For emerging RISC-V platforms, we developed:

- A Vector Length Agnostic (VLA) design that adapts to hardware VLEN
- Both JIT-generated and hand-optimized inline assembly RVV kernels
- Integration with oneDNN's primitive dispatch system

Evaluation on gem5 across six VLEN configurations (128–8192 bits) shows JIT performing best at the VLEN values current silicon uses (peak 3.28× at VLEN=256) and a throughput drop at large VLEN (≥2048). A controlled experiment attributes this drop to the packed filter panel exceeding L2 rather than to runtime code generation: capping the panel so it stays L2-resident increases JIT throughput from 0.59 to 1.25 GFLOPS at VLEN=8192 and maintains approximately constant throughput across the sweep (Section 4.6.4). Real hardware validation on a BananaPi-F3 (SpacemiT K1, VLEN=256) supports the simulation trend: JIT achieves **3.08×** overall speedup (gem5 predicted 3.28×).

A lightweight autotuning framework sweeping LMUL, MR, and k_unroll on the BananaPi shows that MR=8 is selected across the evaluated layers, LMUL is layer-dependent (1 for large reduction dimensions, 2 for medium layers), and k_unroll has negligible impact (<2%). The best single-layer throughput reaches 2.10 GFLOPS (ResNet-50 `conv5_3x3`).

### 6.1.5 Winograd Convolution on RISC-V Vector (Ablation Baseline)

Beyond Implicit GEMM, we implemented Winograd $F(4, 3)$ convolution for RVV and integrated it into oneDNN as a new primitive, with fully RVV-vectorized transforms under a VLA model. A five-way ablation study on both gem5 and BananaPi shows that the RVV-vectorized transforms provide a clear benefit on real hardware (VGG-16: 1.68 vs. 0.96 GFLOPS, +75%), though the JIT GEMM backend underperforms OpenBLAS for Winograd's tile-based sub-problems. Winograd's advantage is layer-dependent: beneficial for large spatial dimensions but diminished for deep 7×7 layers where tile overhead dominates.

### 6.1.6 JIT FlashAttention for LLM Inference (Primary)

To evaluate the JIT framework beyond convolutions, we implemented FlashAttention for RISC-V Vector:

- Online softmax algorithm with numerical stability (max-trick)
- High-precision Cephes exp() approximation (relative error $< 3 \times 10^{-5}$)
- `LMUL=m2` optimization for increased vector throughput
- Dynamic sequence length (N) support for arbitrary context sizes

Multi-VLEN evaluation shows that the optimal implementation depends on VLEN. For small VLEN (128–512 bits), common in current RISC-V chips like SiFive P670 and SpacemiT X60, the JIT implementation outperforms hand-written intrinsics by **1.1–1.5**×. For large VLEN (≥1024 bits), intrinsics become preferable as LMUL grouping overhead outweighs benefits.

### 6.1.7 Framework Integration

Both implementations are integrated into the Intel oneDNN framework, enabling:

- Integration into production deep learning pipelines
- Environment variable control for A/B testing
- Coexistence with existing optimized implementations

## 6.2 Key Findings

Across both convolution and attention, the central theme is *amortization*: make the steady-state inner loop small and regular, then ensure that fixed overheads (index math, constant loading, and loop control) are paid as rarely as possible. Implicit GEMM achieves this by eliminating explicit `im2col` materialization and instead computing coordinates inside a tight GEMM-shaped kernel; FlashAttention stresses the same principle in a different form, where numerical stability and transcendental

approximations introduce constant-heavy control flow that benefits disproportionately from JIT specialization.

On RVV, the Vector Length Agnostic model enables portability, but it also makes the best implementation dependent on `VLEN`. Our results show that JIT performs best at the VLEN values current silicon uses (128–256 bits), where its register-blocked micro-kernel keeps the inner loop well-utilized. At very wide VLEN the JIT throughput drops, but a controlled experiment attributes this to the packed filter panel exceeding L2 rather than to the code generator; keeping the panel L2-resident maintains approximately constant JIT throughput across the sweep (Section 4.6.4).

- **Where Implicit GEMM helps most.** Memory-bound layers with large spatial dimensions benefit the most; compute-bound layers see smaller relative gains. The same shape sensitivity is visible on every backend we tried.
- **The coordinate-arithmetic bottleneck is removed by the packing structure.** An early inline version of Implicit GEMM traded a memory bottleneck for a coordinate-arithmetic bottleneck (integer div/mod is 10–20× slower than FMA and dominated the inner loop). Performing the coordinate mapping once in the packing step and reusing the packed panel across output channels amortizes that cost, leaving a steady-state micro-kernel with no integer index arithmetic. The JIT then emits this register-blocked micro-kernel per shape and vector length without recompilation.
- **VLA provides practical portability.** The same RVV machine code runs unchanged from VLEN=128 to VLEN=8192 bits on gem5 and on the BananaPi-F3 board, with no rebuild between configurations. This is the property that lets a single binary serve current and future RISC-V silicon.
- **gem5 predictions transfer to real silicon at VLEN=256.** The BananaPi-F3 JIT achieves a 3.08× overall speedup against the in-framework scalar reference; gem5 at VLEN=256 predicted 3.28×. The 6% gap supports using gem5 for relative VLEN crossover analysis.
- **VLEN-dependent behavior for both workloads.** At small VLEN (128–512) JIT exceeds intrinsics/asm by 1.1–1.5× on FlashAttention and 2–3× on Implicit GEMM. For Implicit GEMM at very large VLEN the JIT drops because the packed panel exceeds L2; this is a per-VLEN tiling effect, and an L2-resident panel removes the observed drop in the controlled experiment (Section 4.6.4). Most current RISC-V silicon (SiFive P670, SpacemiT X60) sits in the small-VLEN, JIT-favored regime.
- **Cross-ISA knob convergence.** On both RVV (24-config BananaPi sweep) and AVX-512 (4D runtime autotuner), MR = 8 is selected on the vast majority of layers despite the two ISAs having structurally different register files. A common register-budget rule (live registers $\leq$ vector-file size) selects MR on every ISA; the architecture-specific knobs (LMUL on RVV, NR and TILE_M on AVX-512) are the per-ISA delta (Section 5.1.4).
- **AVX-512 non-monotonic stages are shape-dependent.** Three of the

seven optimization stages (auto-vectorization, filter packing, NUMA binding) are non-monotonic on this benchmark suite; the per-shape autotuner avoids each of them, showing that empirical selection can outperform an "always-on" fixed pipeline (Section 5.1.3).

- **TILE_M is the dominant AVX-512 knob in this pipeline.** A scheduling-level L2-tiling parameter alone contributes a ∼5× end-to-end uplift, with the autotuner spreading its picks across six distinct values. Shape-aware scheduling is not replaceable by a fixed heuristic in this benchmark suite (Section 3.2.4.2).
- **Winograd as an attribution tool.** The five-way RVV ablation separates algorithmic gain (Winograd over direct) from implementation gain (vectorized transforms, custom GEMM, JIT GEMM). On real BananaPi-F3 the vectorized transforms alone give +75%; the JIT GEMM kernel built for Implicit GEMM under-performs OpenBLAS on Winograd's small tile-shaped GEMMs, which is reported as a limitation.
- **SIMD correctness depends on layout.** The AVX-512 `vec_n` fix shows that high apparent performance can be illusory without layout-aware loads; this motivated explicit filter packing throughout the framework.
- **Accuracy and measurement.** Numerical differences stay within floating-point tolerance across all kernels; for gem5 we rely on `simTicks/numCycles` rather than in-process wall-clock to avoid host-side measurement artifacts.

## 6.3 Implications

This work shows that JIT-based RVV kernel specialization is a practical approach for the evaluated CPU inference workloads on emerging vector ISAs. Implicit GEMM provides a viable alternative to im2col-based convolution, and FlashAttention shows that the same JIT toolchain can support numerically sensitive, control-flow-heavy attention kernels. The approach is particularly relevant for:

- Memory-constrained devices: edge and embedded systems where memory is limited.
- Emerging architectures: RISC-V platforms where optimized libraries are scarce.
- Production systems: where memory efficiency improves multi-tenant density.

## 6.4 Future Directions

Several avenues remain for extending this work:

1. Implement parameter specialization in JIT for 1×1 and 3×3 convolutions
2. Extend to additional data types (FP16, BF16, INT8)
3. Further optimize AVX-512 backends (e.g., multi-accumulator register blocking) and amortize packing overhead across repeated inference

4. Extend the portability study to ARM SVE and RVV on real hardware
5. Validate on additional physical RISC-V hardware beyond the BananaPi-F3 (SpacemiT K1) used here, such as SiFive P670 and T-Head C910 cores
6. Complete Winograd evaluation across networks and explore hybrid strategies (Winograd for $3 \times 3$ + Implicit GEMM for general cases)
7. Evaluate on complete CNN models end-to-end using oneDNN integration
8. Integrate FlashAttention JIT kernel into llama.cpp for end-to-end LLM benchmark
9. Implement adaptive LMUL selection based on runtime VLEN detection
10. Extend FlashAttention to support Multi-Query Attention (MQA) and Grouped-Query Attention (GQA)

## 6.5 Closing Remarks

As deep learning continues to expand across diverse hardware platforms, efficient and portable convolution implementations remain important. This thesis contributes memory-efficient Implicit GEMM implementations for several CPU vector architectures, with a design that prioritizes portability and extensibility.

The open nature of the RISC-V ecosystem, combined with the vendor-neutral approach of libraries like oneDNN, creates opportunities for collaborative development of high-performance deep learning infrastructure. The methods and measurements in this thesis provide a basis for further work on efficient neural network inference on emerging CPU platforms.

# Bibliography


[1] Intel, *Intel intrinsics guide*, Accessed: 2025-01-24, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[2] R.-V. International, "Risc-v "v" vector extension architecture, version 1.0," RISC-V International, Tech. Rep., 2021.

[3] Intel, *Oneapi deep neural network library (onednn) developer guide*, Accessed: 2025-01-24, 2024. [Online]. Available: https://oneapi-src.github.io/oneDNN/.

[4] Y. Jia et al., "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014. DOI: 10.48550/arXiv.1408.5093. [Online]. Available: https://arxiv.org/abs/1408.5093.

[5] Y. Zhou et al., "Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators," *arXiv preprint arXiv:2110.03901*, 2021. DOI: 10.48550/arXiv.2110.03901. [Online]. Available: https://arxiv.org/abs/2110.03901.

[6] A. Kerr, D. Araya, and D. Merrill, "Cutlass: Fast linear algebra in cuda c++," in *GTC Silicon Valley*, NVIDIA, 2018.

[7] Intel, *Onednn developer guide: Convolution*, Accessed: 2026-02-13, 2024. [Online]. Available: https://oneapi-src.github.io/oneDNN/v2/dev_guide_convolution.html.

[8] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 815–824. [Online]. Available: https://proceedings.mlr.press/v70/cho17a.html.

[9] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 5776–5785. [Online]. Available: https://proceedings.mlr.press/v80/zhang18d.html.

[10] M. F. Dolz, H. Martínez, A. Castelló, P. Alonso-Jordá, and E. S. Quintana-Ortí, "Efficient and portable Winograd convolutions for multi-core processors," *The Journal of Supercomputing*, vol. 79, no. 9, pp. 10 589–10 610, 2023, Reference implementation: https://github.com/hpca-uji/convWinograd. DOI: 10.1007/s11227-023-05088-4.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[12] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," *arXiv preprint arXiv:1312.5851*, 2013. DOI: 10.48550/arXiv.1312.5851. [Online]. Available: https://arxiv.org/abs/1312.5851.

[13] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*, Suvisoft, 2006.

[14] S. Winograd, "Arithmetic complexity of computations," *CBMS-NSF Regional Conference Series in Applied Mathematics*, vol. 33, 1980.

[15] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[16] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 16 344–16 359.

[17] S. L. Moshier, *Cephes mathematical library*, High-precision mathematical function implementations, 1984. [Online]. Available: http://www.netlib.org/cephes/.

[18] K. Goto and R. A. V. D. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, pp. 1–25, 2008.

[19] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 1–33, 2015.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

# A

# Implementation Code Excerpts

This appendix provides key code excerpts from the implementation.

## A.1 x86-64 AVX2 Micro-kernel

The 6×16 micro-kernel using AVX2 intrinsics:

```
void microkernel_6x16_avx2(
    const float* A, const float* B, float* C,
    int K, int lda, int ldc)
{
    // 12 accumulators for 6x16 tile
    __m256 c00 = _mm256_setzero_ps();
    __m256 c01 = _mm256_setzero_ps();
    // ... (c10-c51)
    __m256 c50 = _mm256_setzero_ps();
    __m256 c51 = _mm256_setzero_ps();

    for (int k = 0; k < K; ++k) {
        // Load B[k, 0:16]
        __m256 b0 = _mm256_loadu_ps(B + k * 16);
        __m256 b1 = _mm256_loadu_ps(B + k * 16 + 8);

        // Broadcast A[i, k] and FMA for each row
        __m256 a0 = _mm256_broadcast_ss(A + 0 * lda + k);
        c00 = _mm256_fmadd_ps(a0, b0, c00);
        c01 = _mm256_fmadd_ps(a0, b1, c01);

        __m256 a1 = _mm256_broadcast_ss(A + 1 * lda + k);
        c10 = _mm256_fmadd_ps(a1, b0, c10);
        c11 = _mm256_fmadd_ps(a1, b1, c11);

        // ... rows 2-5
    }

    // Store results
    _mm256_storeu_ps(C + 0 * ldc, c00);
    _mm256_storeu_ps(C + 0 * ldc + 8, c01);
    // ...
}
```

## A.2 RVV Dot Product

Vector dot product using RISC-V Vector inline assembly:

```
float rvv_dot_product(const float* a, const float* b, int n) {
    float sum = 0.0f;
    int i = 0;

    while (i < n) {
        int vl;
        asm volatile (
            "vsetvli %0, %1, e32, m4, ta, ma"
            : "=r"(vl)
            : "r"(n - i)
        );

        // Load vectors
        asm volatile ("vle32.v v0, (%0)" : : "r"(a + i));
        asm volatile ("vle32.v v4, (%0)" : : "r"(b + i));

        // Multiply
        asm volatile ("vfmul.vv v8, v0, v4");

        // Reduce sum
        float partial;
        asm volatile (
            "vfmv.s.f v12, %1\n\t"
            "vfredusum.vs v12, v8, v12\n\t"
            "vfmv.f.s %0, v12"
            : "=f"(partial)
            : "f"(0.0f)
        );

        sum += partial;
        i += vl;
    }

    return sum;
}
```

## A.3 Implicit Coordinate Computation

The on-the-fly coordinate computation for input access:

```
inline void compute_input_coord(
    int m, int k, const Conv2dProblemSize& prob,
    int& batch, int& h, int& w, int& c)
{
    const int PQ = prob.P * prob.Q;
    batch = m / PQ;
    int pq = m % PQ;
    int p = pq / prob.Q;
    int q = pq % prob.Q;
```

```
    const int RS = prob.R * prob.S;
    c = k / RS;
    int rs = k % RS;
    int r = rs / prob.S;
    int s = rs % prob.S;

    h = p * prob.stride_h - prob.pad_h + r * prob.dilation_h;
    w = q * prob.stride_w - prob.pad_w + s * prob.dilation_w;
}
```

## A.4 oneDNN Primitive Registration

Registering the Implicit GEMM primitive in oneDNN:

```
// In cpu_convolution_list.cpp
{{forward, f32, f32, f32}, {
    CPU_INSTANCE_X64(x64_implicit_gemm_convolution_fwd_t)
    CPU_INSTANCE_RV64GCV(rv64_implicit_gemm_convolution_fwd_t)
    CPU_INSTANCE(gemm_convolution_fwd_t)
    CPU_INSTANCE(ref_convolution_fwd_t)
    nullptr,
}},
```

# B

# Experimental Commands

## B.1 Building oneDNN for RISC-V

```
# Cross-compile oneDNN for RISC-V
mkdir build-riscv && cd build-riscv

cmake .. \
    -DCMAKE_TOOLCHAIN_FILE=../cmake/riscv-toolchain.cmake \
    -DDNNL_TARGET_ARCH=RV64 \
    -DDNNL_CPU_RUNTIME=SEQ \
    -DDNNL_LIBRARY_TYPE=STATIC \
    -DCMAKE_BUILD_TYPE=Release

make -j8
```

## B.2 Running gem5 Simulation

```
GEM5=/path/to/gem5

# Run with Reference implementation
USE_IMPLICIT_GEMM=0 $GEM5/build/RISCV/gem5.opt \
    --outdir=gem5_ref \
    $GEM5/configs/deprecated/example/se.py \
    --cmd=./test_rvv_onednn \
    --cpu-type=MinorCPU \
    --caches --l2cache

# Run with RVV Implicit GEMM
USE_IMPLICIT_GEMM=1 $GEM5/build/RISCV/gem5.opt \
    --outdir=gem5_rvv \
    $GEM5/configs/deprecated/example/se.py \
    --cmd=./test_rvv_onednn \
    --cpu-type=MinorCPU \
    --caches --l2cache
```

## B.3 x86-64 Benchmark

```
# Build oneDNN with Implicit GEMM
cd oneDNN_rvv-master/build
cmake .. -DDNNL_BLAS_VENDOR=OPENBLAS
make -j8

# Run comparison
cd master-thesis-scripts-main/code

# oneDNN default
USE_IMPLICIT_GEMM=0 ./test_implicit_gemm

# Our Implicit GEMM
USE_IMPLICIT_GEMM=1 ./test_implicit_gemm
```