POLITECNICO DI TORINO
Repository ISTITUZIONALE

QUINN: Quantized Inference and Dataflow Characterization for Neural Networks on RISC-V Near-Memory Architectures

Original
QUINN: Quantized Inference and Dataflow Characterization for Neural Networks on RISC-V Near-Memory Architectures / Petrolo, V., Guella, F., Masera, G., Martina, M.. - In: COMPUTERS. - ISSN 2073-431X. - 15:10(2026). [10.3390/computers15100667]

Availability:
This version is available at: 11583/3016323 since: 2026-10-05T10:06:04Z

Publisher:
MDPI

Published
DOI:10.3390/computers15100667

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

27 November 2026

computers

MDPI

Article

QUINN: Quantized Inference and Dataflow Characterization for Neural Networks on RISC-V Near-Memory Architectures

Vincenzo Petrolo *, Flavia Guella *, Guido Masera and Maurizio Martina

Department of Electronics and Telecommunication (DET), Politecnico di Torino, 10129 Turin, Italy; guido.masera@polito.it (G.M.); maurizio.martina@polito.it (M.M.)
* Correspondence: vincenzo.petrolo@polito.it (V.P.); flavia.guella@polito.it (F.G.)

Abstract

Machine learning inference is moving to the edge due to latency, energy, and privacy constraints. The growing complexity of edge workloads places increasing pressure on memory hierarchies, making data movement a major limitation for both performance and energy efficiency. Fixed-function accelerators inherit the memory wall of the von Neumann model and lack the flexibility to follow evolving models. Programmable Vector Processing Unit (VPU)-based near-memory architectures alleviate this bottleneck using standard Complementary Metal-Oxide Semiconductor (CMOS) while remaining programmable, but still lack mature kernel libraries and operator-level abstractions. We address this gap with QUINN, a kernel library for integer-precision inference following quantized Open Neural Network Exchange (ONNX) operator semantics. Operator-to-dataflow mapping is well studied for conventional accelerators; we quantify it for near-memory VPUs whose vector register file is managed by explicit DMA (Direct Memory Access) transfers, using a roofline methodology combining throughput with measured off-chip traffic across dataflows, replication, and bandwidth on a Field-Programmable Gate Array (FPGA)-emulated target. Matrix multiplication enters the compute-bound regime, attaining 61% of the mix-weighted roof at the largest problem size, and one-dimensional convolution scales by 3.05× under four VPUs; the kernels sustain 5.0–51.3× the throughput of a scalar host on the same target. Operational intensity does not follow the same ordering: the scalar schedule retains higher intensity on every operator, so the throughput is bought with additional off-chip traffic in every case, though the gain exceeds the cost by at least 3.8×. QUINN thus provides a characterized operator backend, bit-exact with the ONNX semantics consumed by deployment tools.

Keywords: machine learning; near-memory; RISC-V; vector unit

Check for updates

Academic Editor: Yan Zhang

Received: 1 August 2026
Revised: 15 September 2026
Accepted: 22 September 2026
Published: 1 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Machine learning (ML) inference is increasingly moving from cloud infrastructures to edge devices, driven by bandwidth limitations, communication latency, energy consumption, and privacy concerns [1]. At the same time, the complexity of ML models deployed at the edge continues to grow [2,3]. Modern edge workloads increasingly combine compute-heavy integer-precision convolutional and fully connected layers with floating-point normalization, activation, quantization, and attention-based operators [4]. QUINN targets the former class, for which the choice of dataflow is consequential.

Such workloads place significant pressure on low-power embedded platforms, where performance and energy efficiency are often dominated by data movement across con-

strained memory hierarchies. This limitation, intrinsic to von Neumann architectures, is particularly severe when local storage capacity and off-chip bandwidth are constrained [5].

While specialized accelerators can reduce the computational cost of regular operators such as dense matrix multiplication and convolution, system performance may still remain constrained by memory bandwidth. Furthermore, fixed-function or narrowly optimized accelerators lack the flexibility required to execute a broad variety of ML pipelines. With new model architectures emerging almost daily, hardware must be flexible enough to support a diverse range of operators, tensor shapes, data layouts, and quantization requirements.

Near-Memory Computing (NMC) has emerged as a promising approach to alleviate the memory wall by moving computation closer to the memory subsystem. Compared with Processing-in-Memory (PIM) approaches based on emerging memory technologies, NMC can be implemented using standard Complementary Metal-Oxide Semiconductor (CMOS) technology while still reducing data movement and increasing the effective bandwidth available to computation by exploiting high-bandwidth local storage and avoiding repeated transfers through the processor memory hierarchy. By integrating programmable compute resources near memory, NMC architectures can support memory-intensive workloads while retaining greater flexibility than fixed-function accelerators. Nevertheless, the practical deployment of ML workloads on these architectures remains hindered by the lack of mature programming models, reusable kernel libraries, and operator-level abstractions. Consequently, exploiting an NMC platform often requires microarchitectural-specific kernels and tailored optimizations, spreading effort and limiting portability.

This paper addresses this software gap through QUINN, an ML library identifying a subset of the RISC-V "V" extension for efficient integer inference on programmable RISC-V Vector Processing Unit (VPU)-based NMC architectures. We showcase its effectiveness on NM-Carus [6]. The library follows the numerical semantics of selected quantized Open Neural Network Exchange (ONNX) operators, enabling kernels to be reused and composed across different inference workloads without requiring model-specific redesign. In this way, QUINN provides an operator-level abstraction between quantized ML models and the underlying NMC execution substrate.

In addition to the kernel library, this work characterizes how operator performance depends on the interaction among computational structure, dataflow mapping, compute parallelism, and memory-system behavior. Alternative stationary dataflows are investigated for the supported operator families, and their behavior is evaluated through an operator-centric roofline methodology derived from [7]. The analysis considers attainable throughput together with measured off-chip traffic. The latter quantifies the pressure placed by each dataflow on the external memory system, including the additional transfers introduced by tiling and partial-output spilling.

The main contributions of this paper are summarized as follows:

- We present QUINN, a reusable kernel library for integer-precision edge inference on programmable RISC-V vector-NMC architectures. The library provides tiled kernels for Matrix-Matrix (MM) and Vector-Matrix (VM) variants, and two-dimensional, causal one-dimensional and depthwise convolutions. The kernels implement zero-point correction, widened accumulation, and fixed-point requantization according to the specifications of the corresponding quantized ONNX operators.
- We introduce an operator-centric performance-characterization methodology, based on the roofline model [7], that combines attainable throughput and measured off-chip memory traffic. The methodology identifies compute-bound and memory-bound regimes while quantifying how dataflow selection, memory-system conditions, and the number of cooperating VPUs affect both execution time and external data movement.

https://doi.org/10.3390/computers15100667

- We derive operator-specific optimization guidelines by evaluating alternative stationary dataflows and multi-VPU execution strategies. The results show that compute-intensive kernels primarily benefit from output-resident accumulation and spatial parallelism, whereas memory-bound kernels benefit more from input reuse, buffering, prefetching, and latency hiding than from additional arithmetic throughput. Comparisons against a scalar-host baseline show throughput gains of 5.0–51.3×; the scalar schedule retains higher operational intensity on every operator, but the throughput gained exceeds the intensity forgone by at least 3.8× in every configuration tested.

The remainder of this paper is organized as follows. Section 2 reviews optimized kernel libraries, deployment frameworks, and performance-characterization methodologies. Section 3 introduces the quantized ONNX operator semantics, stationary dataflows, and roofline-based characterization framework. Section 4 describes the target VPU-based NMC architecture and its memory hierarchy and presents QUINN, detailing the supported operator mappings, tiling schemes, and multi-VPU execution strategies. Section 5 evaluates the proposed kernels across dataflows, VPU replication, memory-bandwidth conditions, and architectural configurations, including scalar-host comparisons, energy and resource analysis, and an application-level MobileNetV2 projection, to derive operator-specific optimization and scalability guidelines. Finally, Section 6 concludes the paper.

2. Related Work

Efficient edge inference requires support across different levels of the hardware–software stack, ranging from optimized operator kernels to model deployment and accelerator integration. This section positions QUINN with respect to representative kernel libraries, deployment frameworks, and dataflow mapping methodologies for embedded ML inference.

2.1. Optimized Kernel Libraries for Edge Inference

PULP-NN [8] provides low-precision neural network kernels for parallel PULP RISC-V clusters, exploiting multicore execution and architecture-specific Digital Signal Processing (DSP) extensions. More recently, μRISCV-NN [9] investigated manually optimized and compiler-generated kernels for the embedded `Zve32x` profile of the RISC-V Vector Extension. TVM Hybrid-OP [10] instead combines high-level compiler transformations with optimized operators targeting RISC-V packed-SIMD processors.

These libraries demonstrate the benefits of architecture-aware kernel optimization, but assume a processor-centric execution model in which vector load and store instructions transfer operands between the memory hierarchy and the execution datapath. QUINN targets near-memory VPUs whose local storage is directly exposed as the Vector Register File (VRF) and is populated through explicit Direct Memory Access (DMA) transfers. Its kernels are therefore organized around DMA-managed tiling and do not require vector load/store, dot-product, or intra-register reduction instructions.

2.2. Dataflow Mapping and Design-Space Exploration

Operator-to-dataflow mapping has been extensively studied for ML accelerators. Timeloop [11] evaluates mappings over a configurable architecture template, ZigZag [12] enlarges the joint architecture-mapping space with uneven mappings and heuristic search, and LoopTree [13] extends the analysis to fused-layer dataflows. These frameworks estimate cost through analytical models of a parameterized architecture, typically a processing-element array with a multi-level buffer hierarchy. QUINN instead characterizes the mappings by measurement on a target whose local storage is the VRF itself, populated by

explicit DMA transfers, reporting the off-chip traffic each formulation produces rather than an estimate of it.

2.3. Deployment Frameworks and Hardware/Software Co-Design

Several frameworks address ML deployment on heterogeneous embedded systems. Deeploy [14] performs ONNX-to-C lowering and memory-aware code generation for platforms combining RISC-V processors and specialized compute units. HEEPstor [15] supports the integration and exploration of accelerators for quantized inference, whereas WOLT [16] dispatches TensorFlow Lite operators to fixed-function accelerator fabrics.

These frameworks primarily operate at the model, code-generation, or runtime-dispatch level. QUINN instead provides target-specific operator implementations and can consequently serve as a backend for a deployment framework targeting programmable vector near-memory architectures.

Table 1 summarizes the scope and execution model of the considered works. A direct numerical comparison among these works would combine results obtained on different processor organizations, memory hierarchies, instruction sets, numerical precisions, and operator configurations. QUINN is therefore evaluated in Section 5 among dataflows and memory-system configurations on the same target, which instantiates a set of near-memory VPUs [6].

Table 1. Qualitative positioning of representative kernel libraries, deployment frameworks, and dataflow-mapping tools for embedded ML inference.

Work	Scope	Target	Execution Model
PULP-NN [8]	Kernel library	Parallel RISC-V clusters	Multicore with DSP extensions
μRISCV-NN [9]	Kernel library	RISC-V Zve32x	Vector processor
TVM Hybrid-OP [10]	Compiler and kernels	RISC-V packed-SIMD	Processor-centric
Timeloop [11]	Mapping exploration	Configurable accelerator templates	Analytical mapping evaluation
ZigZag [12]	Design-space exploration	DNN accelerators	Joint architecture–mapping search
LoopTree [13]	Dataflow exploration	Fused-layer accelerators	Fused-layer mapping analysis
Deeploy [14]	Deployment framework	Heterogeneous RISC-V systems	Code generation
HEEPstor [15]	Exploration framework	RISC-V with accelerators	Accelerator integration
WOLT [16]	Runtime framework	Accelerator fabrics	Operator dispatch
QUINN	**Kernel library**	**RISC-V vector NMC**	**DMA-managed local storage**

3. Background and Characterization Framework

This section introduces the quantized operator semantics, stationary dataflows, and roofline-based framework used to characterize the QUINN kernels.

3.1. Quantized ONNX Operator Semantics

The kernels considered in this work implement the selected quantized ONNX operators. A quantized value q, associated with scale s and zero point z, represents the real value $r = s(q - z)$. For two input tensors A and B, multiplication is performed on the centered values $\widehat{A} = A - z_A$ and $\widehat{B} = B - z_B$, with products accumulated in an `int32` representation.

The accumulated result C is converted to the output quantization domain as $Y = \text{clip}(\text{round}(\mu C) + z_Y)$, where $\mu = s_A s_B / s_Y$. Since the target VPU supports integer and fixed-point arithmetic, the scaling factor is approximated as $\mu \simeq M2^{-n}$ [17], with round-to-nearest, ties-to-even semantics.

Zero-point correction, widened accumulation, and requantization are shared by the supported operators. The kernels differ in their vector mapping, accumulation order, and local-memory organization. Pointwise convolution is mapped to matrix multiplica-

tion, while MM and VM multiplication provide the main computational primitives used by projection and feed-forward layers in transformer architectures. A specialized kernel is implemented for the VM scenario typical of autoregressive tasks often needed in language models.

3.2. Tiling and Stationary Dataflows

When operands exceed the available VRF capacity, the computation is divided into tiles. Tile dimensions and loop ordering determine operand reuse, memory traffic, and whether partial outputs can remain resident during accumulation [18].

The considered mappings follow three stationary dataflows. Output Stationary (OS) retains partial outputs, avoiding the costly spill and reload of full-integer-precision intermediate values required to preserve numerical accuracy. Input Stationary (IS) retains input tiles to exploit activation reuse, while Weight Stationary (WS) retains filters or matrix weights. The latter two may reduce operand traffic at the cost of additional partial-output transfers. The preferred dataflow depends on the algorithmic complexity of each operator, the activations and weights dimensions, memory layout, memory hierarchy, and available reuse. The operator-specific implementations are presented in Section 4.

3.3. Roofline-Based Characterization

The roofline model [7] was originally developed to drive FLOPS optimization and has more recently proven useful for the design of ML accelerators [19,20]. In this work, we use the roofline model as a framework to compare alternative stationary dataflows and multi-VPU execution strategies for the quantized operators targeted by QUINN. The analysis is based on two key metrics: Operational intensity is defined as $I = N_{\text{op}}/B_{\text{off}}$, where N_{op} is the algorithmic operation count and B_{off} is the measured off-chip byte traffic, which is directly affected by the dataflow choice of each operator. For a fixed operator and tensor shape, N_{op} is identical across dataflows; therefore, differences in operational intensity among OS, IS, and WS arise entirely from the off-chip byte traffic induced by the selected dataflow. The algorithmic operation count comprises the arithmetic prescribed by the quantized operator semantics of Section 3.1:

$$N_{\text{op}} = 2N_{\text{MAC}} + N_{\text{in}} + 4N_{\text{out}}, \tag{1}$$

where N_{MAC} is the number of scalar multiply-accumulate operations required by the operator, counted as two arithmetic operations following the convention commonly adopted in roofline analyses, N_{in} is the total number of activation and weight elements, each requiring one zero-point subtraction, and N_{out} is the number of output elements, each requiring the multiply, shift, zero-point addition, and clipping of the fixed-point requantization. The expression is identical for every operator; only the shape-dependent expansion of the three terms differs. Address generation and loop control are excluded from N_{op}, being artefacts of the implementation rather than of the operator, while their execution time is included in the measured cycle count. This choice ensures that the reported throughput reflects useful algorithmic work: implementation-dependent instructions still contribute to the measured execution time and therefore reduce the attained throughput, without contributing to N_{op}. More generally, when an operation-counting convention is applied consistently to both the operational intensity and the corresponding compute ceiling, the absolute normalization of N_{op} cancels from the comparison with the roofline ridge point. If additional instruction classes were included in N_{op}, their corresponding execution cost would also need to be incorporated into the compute ceiling.

Given peak throughput P_{peak} achievable by the hardware and sustained memory bandwidth β_{mem}, the performance bound is $P_{\text{roof}}(I) = \min(P_{\text{peak}}, I\beta_{\text{mem}})$, with ridge point $I_{\text{ridge}} = P_{\text{peak}}/\beta_{\text{mem}}$. Our library characterization compares operator families, tensor dimensions, stationary dataflows, numbers of cooperating VPUs, and memory-system conditions. For each configuration, it reports operational intensity and attainable throughput. As done in [7], we measure the off-chip traffic through the cache levels to capture the additional transfers introduced by tiling and partial-output spilling.

4. QUINN: Near-Memory Kernel Library

To support a broad variety of edge-ML workloads on programmable near-memory platforms, QUINN provides integer-precision kernels spanning different computational and memory-access regimes. The supported operators include `QLinearMatMul`, with specialized kernels for MM and VM multiplication, and two-dimensional, causal one-dimensional, and depthwise `QLinearConv`.

4.1. Target NMC Architecture

Near-memory computing encompasses architectures that place programmable or specialized processing resources close to the memory subsystem, reducing data movement between storage and computation. Representative implementations include vector and streaming processors coupled to memory controllers, reconfigurable vector units in the logic layer of 3D-stacked memories, and computational Static Random-Access Memories operating as vector coprocessors [21–24]. Although these approaches differ in memory organization and programmability, they share the objective of exploiting the high bandwidth available close to memory while avoiding repeated transfers through the processor memory hierarchy.

Among these approaches, this work considers programmable vector-based NMC architectures, whose processing units operate close to the memory system. Figure 1 presents the reference system used to evaluate QUINN. The reference system relies on a `cv32e40x` OpenHW RISC-V Central Processing Unit (CPU) supporting the base RISC-V Instruction Set Architecture (ISA) `rv32imac`, as well as the NMC `xvnmc` ISA first described in [6]. The scalar CPU dispatches `xvnmc` instructions to the near-memory VPUs through the standard OpenHW Core-V eXtension Interface (CV-X-IF). Following the NMC execution model, operands are accessed directly from the local VRF; vector load and store instructions are therefore not required. Data transfers between the local VRFs and off-chip memory are explicitly managed by a private single-channel DMA engine, programmed by the host CPU and supporting burst transactions.

The near-memory engines are based on the VPU architecture presented in [6].

Each VPU operates directly on data stored in its private local VRF. A 128 kiB, 8-way set-associative, write-back cache with a 64-byte line size is placed between the DMA and off-chip memory. The cache is dedicated exclusively to data transferred for VPU execution and is neither shared with nor accessed by the host CPU. The CPU and the VPUs operate on disjoint memory regions, avoiding the need for cache-coherence mechanisms. During kernel execution, the VPUs access only their local VRFs, while the cache serves the background transfers between the VRFs and off-chip memory.

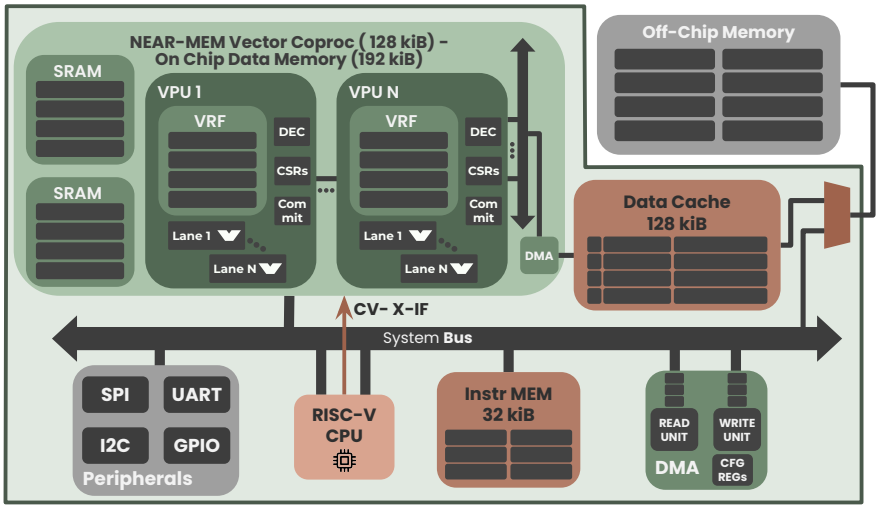

Figure 1. Block diagram of the system configuration used for the experiments. A 128 kiB, 8-way set-associative cache is placed on the DMA path to mitigate the off-chip memory delay when fetching data to be used in computing mode by the vector coprocessor. The 192 kiB on-chip data memory consists of four 32 kiB VRFs plus 64 kiB of additional Static Random-Access Memory (SRAM) memory.

4.2. QUINN Execution Model

Building on the NMC organization described in Section 4.1, QUINN targets architectures in which VPUs operate on explicitly managed local storage. In the reference system, this storage is exposed directly as the VRF, while operands are transferred between the external memory hierarchy and the VPU through contiguous or strided DMA transactions. Accordingly, QUINN adopts a DMA-centric, software-managed execution model. Tiling, loop ordering, operand residency, and data transfers are explicitly coordinated by software, while arithmetic instructions operate on data already resident in the VRF. This organization also allows the stationary dataflows of Section 3.2 to directly control which tiles remain local, which are transferred, and when data movement can be overlapped with computation.

To decouple this execution model from a specific NMC implementation, architecture-dependent operations, including DMA configuration, contiguous and strided transfers, synchronization, and target-specific VPU operations, are exposed through lightweight wrapper interfaces. Porting QUINN to another programmable NMC architecture therefore mainly requires adapting these interfaces to the target drivers and execution primitives, while preserving the operator-level tiling, dataflow, and quantization structure. This modular organization makes QUINN potentially portable across NMC architectures exposing different low-level interfaces.

Finally, since near-memory processing units are typically designed to remain lightweight, QUINN does not rely on hardware-expensive operations such as dot products or intra-register reductions. Its minimum required instruction set is compatible with a subset of the `Zve32x` RISC-V Vector Extension, including integer arithmetic and logic operations, vector–scalar and vector–vector Multiply-accumulate (MAC) instructions, vector moves, and slide-up and slide-down operations. In the remainder of the paper, local on-chip memory and VRF refer to this same storage resource. QUINN can also target systems containing multiple cooperating VPUs, provided that vector instructions can be dispatched to one or more units and that each VPU exposes independent local storage. Depending on the operator, the additional units are used either for parallel computation or as buffers for overlapping data movement and execution.

4.3. Common Kernel Execution Flow

The supported operators follow the numerical semantics described in Section 3.1. Each kernel manages DMA transfers and performs zero-point correction, widened `int32` accumulation, and fixed-point requantization using the multiplier-and-shift representation from [17]. The following subsections describe how this common flow is specialized for each operator, focusing on the reduction-free vector mapping, tile organization, stationary dataflows, and multi-VPU execution strategy. Table 2 reports the resulting VRF allocation for each kernel and dataflow.

Table 2. Vector register file allocation per kernel and dataflow. In convolutions, a 3×3 filter is assumed. Each VPU exposes 32 vector registers (V0–V31). For 2D and depthwise convolution the windows are identical across dataflows, which differ in loop order and input reuse rather than in register assignment, so a single row is given. Each partition was obtained by grid search over the feasible register allocations, conducted identically for every dataflow, retaining the configuration attaining the highest throughput.

Kernel	Dataflow	Input	Weights	Output (`int32`)
Matrix–Matrix	OS	V31	V0–V14	V15–V29
	IS	V31	V0–V21	V22–V29
	WS	V31	V0–V21	V22–V29
Vector–Matrix	— [a]	V31	V0–V29	V30
2D Convolution	OS/IS/WS	V0–V20 [d]	V31	V21–V28
1D Convolution	OS	V0–V10 [e]	V31	V11–V30
	IS	V0–V7 [b]	V31	V8–V28
	WS	V0–V13	V31	V14–V26
Depthwise Convolution [c]	OS/IS/WS	V0–V8	V16–V24	V15

[a] The vector–matrix variant has a single implementation ($M = 1$): the output occupies one register, so the remainder of the file backs the weight tile. [b] Input Stationary holds all C_{in} input channels resident; the window shown corresponds to $C_{\text{in}} = 8$ and extends with the channel count, falling back to streaming above 26 channels. [c] The depthwise convolution uses the HWC memory layout, meaning that for each row in the VRF we have one pixel for the input patch; such an algorithm can be applied only up to a $K \times K$ filter size of $K = 3$. A $K = 4$ filter would already require 16 VRF rows leaving no space for both the input and output accumulator. $K > 3$ is still possible but forces many off-chip transfers with one input pixel at a time strongly degrading throughput. [d] In the 2D Convolution roughly half of this range is used to store the original input, while the remaining half is used to hold shifted replicas of it. [e] The OS dataflow for 1D Convolution supports sequences of length L up to $L = 11 \times VLEN$.

4.3.1. Matrix Multiplication

Matrix multiplication is the most common operator in modern transformer-based architectures. We target `QLinearMatMul`, which consumes two quantized input tensors, **A** and **B**, of size $M \times K$ and $K \times N$, respectively, with their scales, zero points, and biases. The operator produces a quantized output tensor, **C**, of shape $M \times N$ in a different domain via the requantization scheme of Section 3.1. When the operands do not fit the available VRF, QUINN partitions **A**, **B**, and **C** into tiles of size $m \times k$, $k \times n$, and $m \times n$, respectively. We denote by t_m, t_k, and t_n the corresponding tile positions along the M, K, and N dimensions. The three stationary dataflows, described in Section 3.2, execute the same arithmetic kernel and differ only in the ordering of these outer tile loops and, consequently, in which operand remains resident in the VRF.

As stated in Section 4.2, this implementation uses the outer-product strategy: for a fixed output row, one scalar element of the $m \times k$ tile of **A** is broadcast and multiplied with an n-element row of the corresponding **B** tile. The resulting vector MAC updates an

n-element slice of the output accumulator. Repeating this operation over the k reduction dimension completes the contribution of one pair of input tiles to the $m \times n$ output tile. Hence, vector reduction instructions are not required.

Figure 2 shows a simplified example of the three tiling strategies. OS keeps the K axis innermost, so the `int32` accumulator stays resident in the VRF and no partial sum is spilled. IS reorders the loop to m, k, n so that each **A** sub-tile is loaded once and reused across the whole N sweep. WS instead adopts the n, k, m ordering, allowing each **B** sub-tile to be loaded once and reused across the whole M sweep. In IS and WS the K axis is no longer innermost, so partial sums must be spilled to main memory after each non-final K-tile and later be reloaded; this is the classic trade-off of the stationary schemes, which exchange extra output-partial traffic for minimal input or weight traffic. As a simple example, consider a matrix multiplication requiring two tiles along each of the M, K, and N dimensions. In the OS formulation, the loop order is (t_m, t_n, t_k). For a fixed output tile, e.g., $\mathbf{C}_{0,0}$, the contributions associated with $t_k = 0$ and $t_k = 1$ are therefore executed consecutively while $\mathbf{C}_{0,0}$ remains resident in the VRF. The completed tile is written back only after the reduction over K terminates, as shown in Figure 2. In IS, the order becomes (t_m, t_k, t_n). The same $\mathbf{A}_{t_m,t_k}$ tile can then be reused while sweeping different t_n values. However, the complete K-reduction of an output tile is no longer contiguous in time, and its int32 partial result must be stored and subsequently reloaded. Conversely, WS adopts (t_n, t_k, t_m), keeping the corresponding $\mathbf{B}_{t_k,t_n}$ tile resident while sweeping the M dimension. As for IS, this reuse is obtained at the cost of spilling intermediate output tiles. Thus, the loop permutation changes data residency and memory traffic, but not the arithmetic performed by the inner vector kernel.

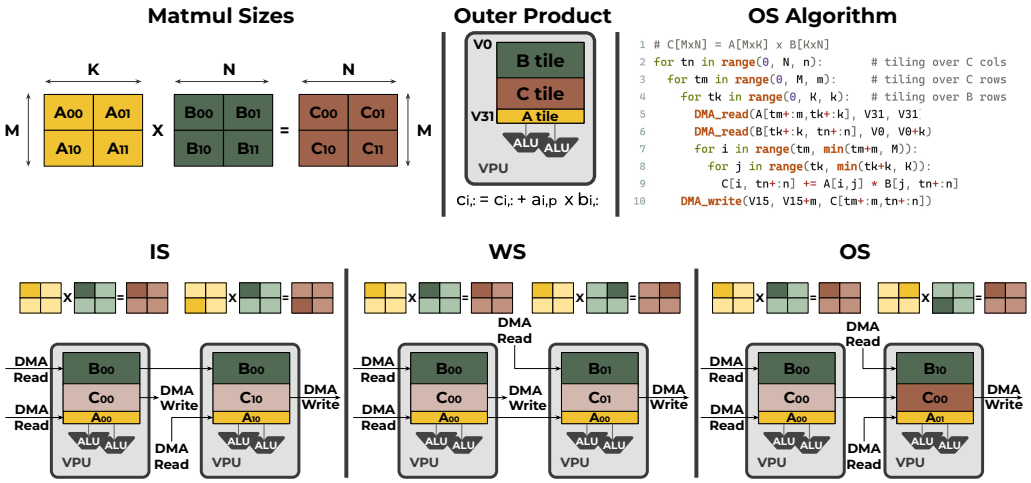

Figure 2. Matrix multiplication tiling algorithms on a single VPU. Yellow, green and brown mark the input, weight and output matrices, respectively. Light colors indicate non-active tiles for the current iteration. Scheduling of two consecutive iterations is shown for all three tiling algorithms. Light brown outputs in the VPU are partial outputs. At the top-center, we show the mapping of matrix multiplication onto the VPU VRF and the outer product algorithm. At the top-right, the pseudocode of the OS algorithm is shown.

The VRF is accordingly partitioned between the B window and the int32 accumulator window: OS uses a balanced split, whereas IS and WS use a K-heavy split that deepens the K-tile and reduces the number of spills. The dataflow that minimizes the overall memory traffic depends on the tensor shapes, so all three are characterized in Section 5.5. When multiple VPUs are available, tiling still remains applicable: the tile is simply enlarged.

Single Instruction, Multiple Data (SIMD) parallelism is effective in this scenario because it raises the compute ceiling and allows P cooperating VPUs to aggregate the MAC

throughput. The cooperating VPUs partition the M axis, and the broadcast scalar from A is shared so that each VPU computes a distinct slice of the output. Given that the VM variant is inherently memory-bound, computation is serialized to maximize the compute window. While one VPU computes, the next one is prefetched, and at the end of each VPU's computation control is transferred to the next VPU. Prefetching overlaps the memory latency of each transfer with the computation of its predecessor, thereby avoiding pipeline stalls.

4.3.2. 2D Convolution

Convolution has an arithmetic complexity of $K_h K_w C_{\text{in}} C_{\text{out}} H_{\text{out}} W_{\text{out}}$ MACs, because it correlates spatial and channel information within a single operator. The high reuse gives this operator a higher operational intensity than the VM and depthwise cases and allows it to become compute-bound for the evaluated configurations. We target `QLinearConv`, which consumes a quantized input tensor A of shape $C_{\text{in}} \times H \times W$ and a quantized filter F of shape $C_{\text{out}} \times C_{\text{in}} \times K_h \times K_w$, and produces a quantized output R of shape $C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}$.

The inner kernel maps the convolution to a sequence of broadcast MAC operations over the $K_h \cdot K_w$ filter taps. One input row is held in a vector register, so that a single MAC covers an entire output row in one instruction. The receptive field is covered by sliding the input row across the vector register, so that element c contributes input column $c + k_w$ for filter column k_w. Successive input rows are then processed for each filter row K_h, with the corresponding weight broadcast as a scalar. Each output row therefore costs $K_h \cdot K_w$ multiply-accumulates, and the C_{in} reduction is folded in by the caller, which re-invokes the inner kernel for each input channel and accumulates into the same output registers.

The OS dataflow keeps C_{in} innermost, so the `int32` accumulator stays resident in the VRF and no partial sum is spilled. In the IS dataflow, for each H-tile and input channel, the input sub-tile is loaded once and reused across the whole C_{out} sweep, cutting input traffic by the C_{out}-tile count. WS reorders so that, for each C_{out}-tile and input channel, the filter sub-tile is loaded once and reused across the whole H sweep, cutting weight traffic by the H-tile count. In IS and WS the C_{in} axis is no longer innermost, so `int32` partial sums are spilled to main memory and later reloaded. Figure 3 shows a simplified scheme of the three tiling strategies. Consider one spatial output tile with two input-channel tiles and two output-channel tiles: OS fixes one output tile and processes the two C_{in} tiles consecutively, so that its int32 accumulator remains resident until the channel reduction is complete. IS instead fixes one input tile and applies it to both output-channel tiles before loading the next input tile; the input transfer is therefore amortized across the C_{out} sweep, but both output tiles must be preserved as partial results until the next C_{in} tile is processed. WS follows the analogous strategy for the filter, keeping one weight tile resident while sweeping spatial output tiles.

For higher throughput, the cooperating VPUs partition the output-height dimension, since the resulting output stripes are independent and do not require cross-VPU reduction. Each VPU loads the input region required for its stripe, including the spatial overlap introduced by the filter. No inter-VPU reduction or explicit halo exchange is required during the computation.

4.3.3. 1D Convolution

Conv1d is a fundamental operator in temporal and sequence models such as Temporal Convolutional Networks (TCNs) and autoregressive transformers. The 1D variant of `QLinearConv` consumes a quantized input tensor A of shape $C_{\text{in}} \times 1 \times L$, a quantized filter F of shape $C_{\text{out}} \times C_{\text{in}} \times 1 \times K$, and produces a quantized output R of shape $C_{\text{out}} \times 1 \times L$. The kernel is causal, meaning each output $R[t]$ depends only on the inputs $A[t-K+1], \ldots, A[t]$, the layout required by autoregressive sequence models. Its arithmetic complexity is lower

than that of the 2D variant as the spatial window collapses to one dimension. Depending on its working-set size, the operator can reach the compute-bound region when its data remain within the on-chip hierarchy.

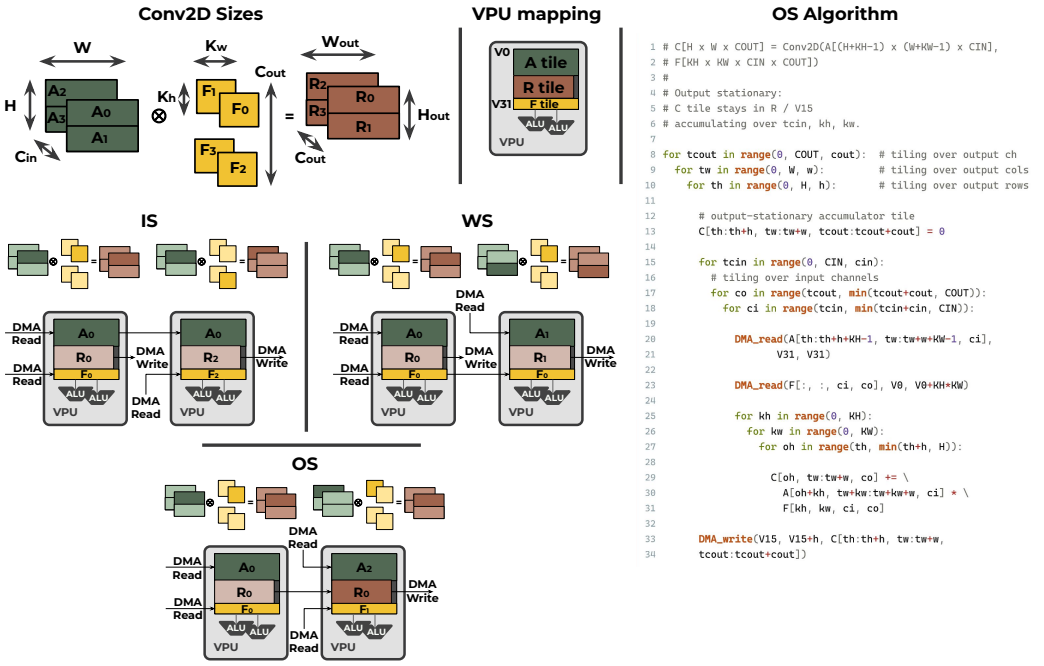

Figure 3. 2D convolution tiling algorithms. At the top-center, the mapping of inputs and outputs onto the VPU VRF. Yellow and green indicate the input and weight matrices, respectively, while brown marks the output matrix. Light colors indicate non-active tiles for the current iteration. Scheduling of two consecutive iterations is shown for all three tiling algorithms. Light brown outputs in the VPU are partial outputs. On the right, the pseudocode for the OS scheduling is shown.

The inner kernel is the 1D analogue of the 2D shift-and-MAC: the spatial window has K taps and a block of N_B output channels is held resident in the VRF as `int32` accumulators while the $C_{\text{in}} \cdot K$ reduction is performed in place.

OS keeps several output channels resident and loads the input once per chunk; IS holds the input channels resident across the C_{out} sweep, processing them in groups when C_{in} exceeds the VRF capacity. WS differs from the 2D case because the filter is applied in one pass over the whole sequence and then discarded. Unlike in the 2D convolution, where the H axis provides a reuse dimension for the filter, holding the filter resident buys no reuse; the VRF is instead split between a resident fraction of the input channels and a resident block of output accumulators, combining the IS and OS strategies. Consider a sequence chunk processed over multiple input- and output-channel groups. OS reserves most of the VRF for a block of int32 output accumulators and completes their $C_{\text{in}}K$ reduction before eviction. IS instead keeps a group of input channels resident while sweeping multiple output-channel groups. The implemented WS variant uses the available VRF capacity jointly for resident input channels and output accumulators, effectively combining input and output stationarity. Similarly to Section 4.3.2, cooperating VPUs partition the sequence length L, and the filter taps are extracted once from one VPU and broadcast to every cooperating VPU. Each VPU loads only its own input band together with the $K - 1$ causal-overlap samples; no inter-VPU synchronization or halo exchange is required.

4.3.4. Depthwise Convolution

Depthwise convolution captures the spatial feature mixing of a standard convolution at a fraction of the cost, and is paired with a pointwise convolution that mixes channels;

the two together substitute the standard convolution in MobileNet-class networks [25]. Unlike 2D convolution, there is no reduction across channels. The arithmetic complexity decreases to $K_h K_w C_{\text{in}} H_{\text{out}} W_{\text{out}}$ MACs. Depthwise convolution belongs to the `QLinearConv` class, distinguished by the *group* attribute.

In modern convolutional networks such as the MobileNet family, spatial convolution is performed progressively on small images up to 7×7 while the channel count grows to over a thousand. For large spatial dimensions, the kernel presented in Section 4.3.2 is sufficient to exploit the vector parallelism. When the spatial tile becomes smaller than the channel dimension, the Height, Width, Channels (HWC) layout is preferred. In this channel-last organization, a single vector instruction operates on an entire channel slice. Since each channel is independent, no cross-channel reduction is required. Accumulation over the $K \times K$ taps is performed across vector registers via `vmacc.vv`.

The inner kernel loads a $K_h \times K_w$ patch of input pixels for one channel slice and multi-accumulates it with the resident filter. For $K \leq 3$ the activation and filter fit the VRF, while for $K > 3$ the operands are streamed while the output stays resident.

Unlike the standard convolution, it is already visible from the data layout that an OS approach is not preferable overall: the HWC layout allows selectively keeping some pixels resident while streaming in new ones, which is the IS strategy. Nevertheless, we still provide a full (albeit naive) OS dataflow where we reload the input pixels at each new output pixel. IS, as shown in Figure 4, preserves the inputs for as long as possible. It holds the patch resident and slides it one output column at a time via a rotating ring. Only the new rightmost column is loaded, reducing the input reads to approximately K_h per pixel [26]. WS is free in the depthwise case when $K \leq 3$: the filter is a single $K \times K$ tap-set per channel, small enough to stay resident for the whole tile, and the output is a single accumulator per pixel, so weight- and output-stationarity are both free. WS therefore converges to the same optimum as IS, taking the IS ring because input reuse is the only axis that actually differs. Unlike other kernels, this dataflow does not spill `int32` partial sums to main memory. Although parallelizing across independent channels might seem appealing, it is not the most effective strategy as it would decrease the already low share of compute time relative to memory-access time. Software prefetching and double buffering are preferred under multiple cooperating VPUs.

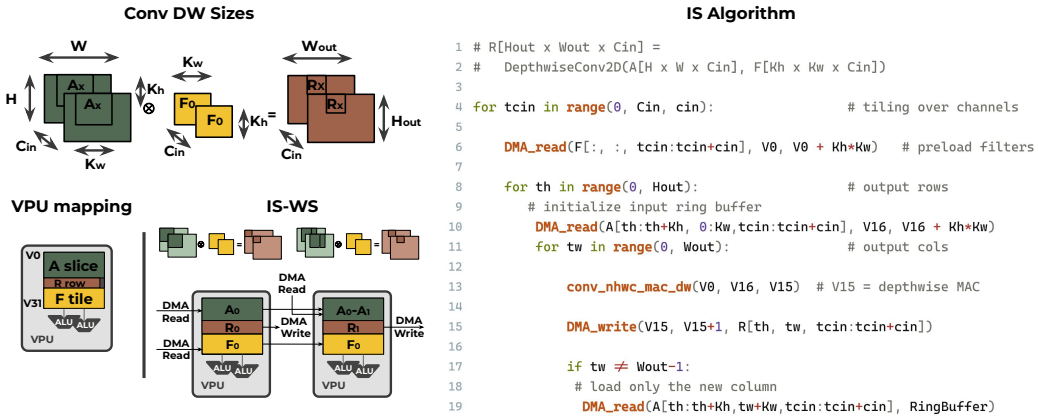

Figure 4. Depthwise convolution with IS tiling on a single VPU, implementation scheme (**left**) and pseudocode (**right**). IS is shown as the reference dataflow as it offers the greatest opportunities for operand reuse through the ring-buffer mechanism. WS tiling is equivalent to IS for filter sizes fitting the VRF. Light colors indicate non-active tiles for the current iteration. Scheduling of two consecutive iterations is shown for all three tiling algorithms. Light brown outputs in the VPU are partial outputs.

5. Experimental Evaluation

This section evaluates QUINN on a real target architecture. We first describe the hardware emulation setup, the correctness checks applied to every kernel, and the roofline methodology used throughout. The remainder of the section reports per-operator results, comparing the dataflow formulations of Section 4 at iso-problem size.

5.1. Emulation Setup

The target architecture is emulated on a `zcu104` board (`xczu7ev-2ffvc1156`) clocked at 80 MHz and synthesized with Vivado 2020.2. The experiments vary the number of VPUs operating in parallel and the response delay of the off-chip memory. The number of lanes is fixed to eight in all experiments, except for those in Section 5.10, to isolate the effects of dataflow selection, VPU replication, and memory bandwidth. Lane-level scaling has been investigated separately in [27] and is outside the scope of this work. Kernel code is compiled with GNU Compiler Collection (GCC) 12.3.1 at `-O3`. All performance measurements are collected from execution on the Field-Programmable Gate Array (FPGA)-emulated target. Execution cycles are obtained from the `mcycle` Control and Status Register (CSR), while memory-traffic measurements are derived from the hardware counters described in Section 5.3. A tunable latency module intercepts off-chip traffic and introduces additional memory-response latency, allowing lower sustained off-chip bandwidth conditions to be emulated without modifying the physical memory interface.

5.2. Benchmark Workloads and Baselines

The experimental evaluation is performed at the operator level and covers the complete set of supported kernel families, including `QLinearMatMul` in both matrix–matrix and vector–matrix form, two-dimensional `QLinearConv`, causal one-dimensional convolution, and depthwise convolution. For each operator, representative tensor shapes are selected to cover different computational intensities and application regimes, including vision and autoregressive workloads. The same operator definitions and problem sizes are also executed on the scalar `rv32imac cv32e40x` host using the scalar fallback kernels of μRISCV-NN, providing a common baseline on the same target system.

5.3. Functional Validation

Correctness is established by unit tests comparing each QUINN kernel against reference outputs produced by ONNX Runtime [28] (version 1.21.0) for the same quantized operators. Reference inputs and outputs are exported and compiled into the on-target test suite. Test vectors cover non-zero zero points, distinct per-tensor scale factors, biases, accumulator saturation, tensor shapes that are not multiples of the vector length, and scale factors chosen to exercise the rounding behaviour of the fixed-point approximation of Section 3.1, where a deviation of one Least Significant Bit (LSB) would arise if the round-to-nearest result differed from the floating-point computation at a tie. All outputs agree exactly. Together with the hardware-based cycle and memory-traffic measurements described in Section 5.4, this validation ensures that the reported performance results correspond to functionally correct executions of the target operators.

5.4. Roofline Methodology

We evaluate the kernels with the roofline model summarized in Section 3.3. Traffic is used as a direct data-movement metric: it captures the effects of tiling, cache misses, evictions, and partial-output spilling, whereas total energy would additionally depend on the implementation technology, local-memory activity, datapath utilization, clocking, and static power.

Our platform intentionally integrates a cache (Figure 1). In a cacheless system, traffic would be fully determined by the schedule and the comparison settled by construction; we instead ask which formulation performs best in the presence of a cache. The evaluation is controlled by keeping the system configuration fixed and varying only the dataflow, degree of VPU replication, and memory bandwidth. The resulting differences can therefore be attributed to the formulation itself. This separation would not be possible in a comparison across different systems.

The horizontal axis reports the operational intensity I (Section 3.3), whose denominator is the sum of cache miss and eviction counts multiplied by 64 B; the memory roof is drawn at 1.2 B/cycle, measured with long unit-stride DMA transfers. The vertical axis reports the attainable throughput $P = \text{Ops/Cycles}$, obtained from the `mcycle` CSR. The operations counted by Equation (1) do not all issue at the same rate: [6] reports 0.33 `int32` MAC/Cycle/lane, so counting one multiplication and one addition as two operations gives $r_{\text{MAC}} = 5.28$ Ops/Cycle over our eight lanes, whereas the adder sustains one 32-bit word every two cycles per lane, giving $r_{\text{add}} = 4.00$ Ops/Cycle for the zero-point and requantization terms, which QUINN issues at `SEW` $= 32$. A MAC-only roof would overestimate the ceiling of the memory-bound kernels, so we draw instead the harmonic mean of the class rates, weighted by the operation mix each shape prescribes,

$$P_{\text{peak}} = \frac{N_{\text{op}}}{T}, \qquad T = \sum_c \frac{n_c}{r_c}, \tag{2}$$

where n_c is the number of operations of Equation (1) issued at rate r_c, with $\sum_c n_c = N_{\text{op}}$. Where a panel holds more than one operator or problem shape, the highest resulting roof is drawn. Values span 4.77 to 5.28 Ops/Cycle across the evaluated configurations.

5.5. QLinearMatMul

We analyze the `QLinearMatMul` dataflows over a range of problem sizes. For MM, we increase the dimensions according to $M = K = N = n$, whereas for VM we set $M = 1$ and $K = N = n$. Since the objective is to compare the dataflows, we use a single near-memory VPU. Figure 5 shows for the MM regime that the advantage of OS becomes increasingly pronounced as the matrix size grows, reaching 1.49× higher throughput than the alternative dataflows at the largest tested size, MM256. This follows from retaining partial outputs in the VRF and avoiding the spill/reload traffic incurred by IS and WS. Its peak is reached at 3.21 Ops/Cycle (61% of the ceiling) but cannot scale further with problem size as the maximum VLEN is reached. The residual gap to the 5.28 Ops/Cycle ceiling is attributable to control-flow instructions, interleaved DMA-descriptor programming, and DMA-transfer stalls. Within the `MM` cluster, operational intensity increases rapidly with n: the $O(MKN)$ complexity term grows cubically in n while matrix traffic grows only quadratically, so each byte fetched amortizes over more MACs as the tile grows. This scaling behavior also accentuates the differences among the three dataflows, whose performance diverges progressively as the problem size increases.

The VM instances follow the opposite trend: their arithmetic complexity grows only quadratically in n, so the operational intensity stays substantially constant and the `VM` kernels of different problem sizes collapse onto a single cluster.

The horizontal-axis spread between OS and its peers widens with n and narrows below `MM64`, where all three dataflows converge toward the same operational intensity, indicating that IS and WS thrash the cache less with their partial-output spills at those sizes. Despite delivering lower throughput, OS remains more data-efficient at `MM16` where WS obtains higher ops-per-cycle.

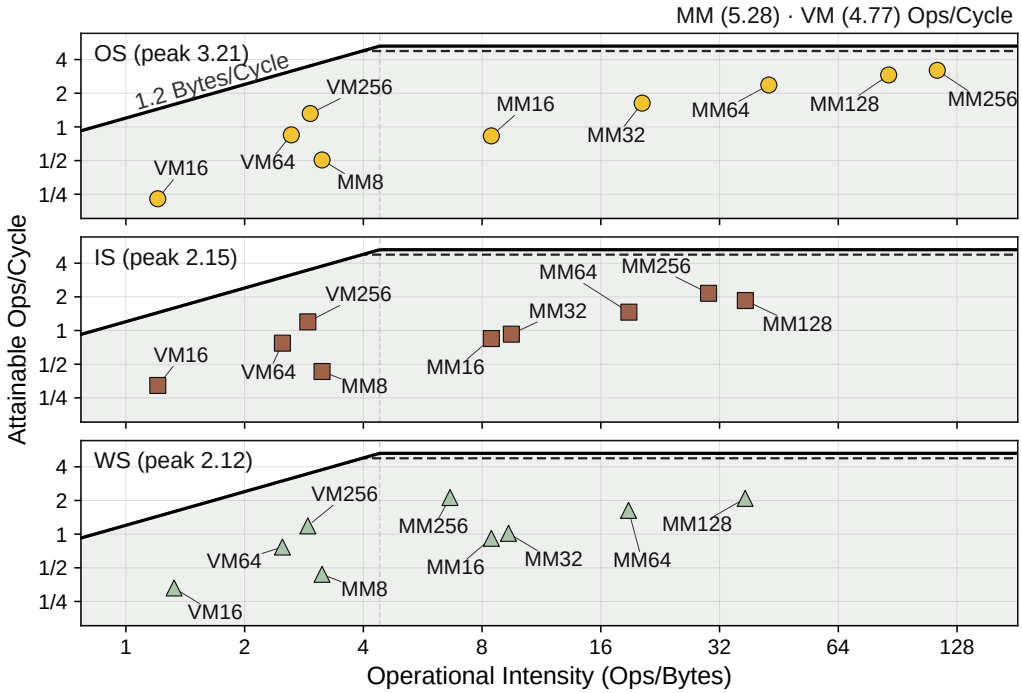

Figure 5. Roofline characterization of the `QLinearMatMul` kernel. Each panel reports one dataflow: OS, IS, and WS, from top to bottom. The MM instances follow an $N = K = M = n$ sweep and VM instances follow a $K = N = n, M = 1$. The straight line is the MM Ops/Cycle peak, while the dotted line indicates the VM peak. Only the maximum ceiling is reported as stated in Section 5.4. The maximum deviation between the displayed and shape-specific ceilings is 7.7% for MM and 1.7% for VM.

The VM cluster, by contrast, remains tightly grouped at low operational intensity across the full range tested, with only representative points reported. This behavior is consistent with the $O(NK)$ complexity, as memory traffic scales proportionally with the operation count, keeping VM memory-bound at every size probed. The points group by problem size rather than by dataflow, indicating that no implementation is more data-efficient than the others; the operational intensity ranges from 1.5 at `VM16` to 3 for the larger workloads. A modest but consistent separation nonetheless emerges as n grows, with all three dataflows attaining higher throughput as they exploit vector register parallelism more effectively.

5.6. QLinearConv

The operational intensity for the 2D Conv increases linearly with the number of input channels c and quadratically with the k filter size. We fix the image size to 224×224 and the number of output channels to 32, a representative configuration for modern vision Neural Networks (NNs) such as DINOv2 [4] and MobileNetV2 [25], while sweeping the number of input channels and the filter size. In Figure 6, we observe that data points cluster by the filter size k (especially visible in the WS dataflow), confirming the expected quadratic dependence of operational intensity on filter size. OS reaches the highest throughput across all configurations because IS and WS spill partial outputs, increasing cache pressure and limiting the achievable throughput. Within each IS cluster, configurations with larger input-channel counts are more data-efficient because they provide greater reuse; this separation becomes more pronounced as the filter size increases.

For the OS we observe that, as in the IS scenario, the Pareto-optimal data point is the c32/k7 because of the long, unit-stride memory accesses used to load the 7×224 input

patch. The OS dataflow emerges as Pareto-optimal because retaining intermediate outputs locally avoids costly memory spills.

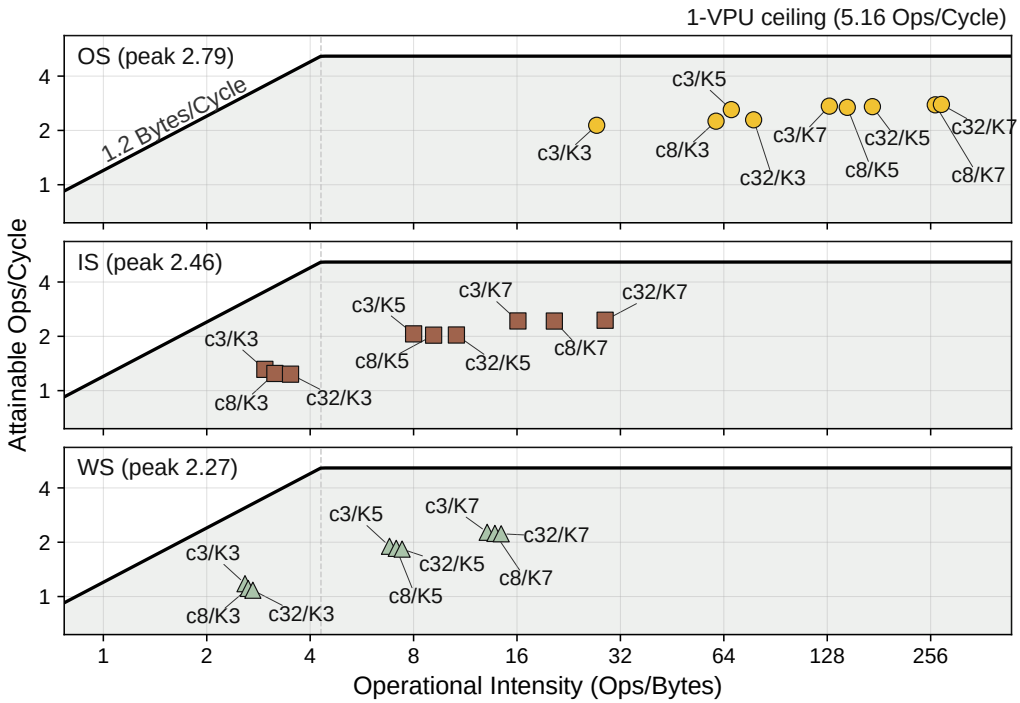

Figure 6. Roofline characterization of the `QLinearConv` (2D) kernel. Each panel reports one dataflow: OS, IS, and WS, from top to bottom. The image size is fixed at 224×224, with the number of output channels fixed at $Cout = 32$. Each data point is labeled by the number of input channels c and the filter size k. Only the maximum ceiling is reported as stated in Section 5.4. The maximum deviation between the displayed and shape-specific ceilings is 2.2%.

5.7. QLinearConv1D

One-dimensional convolution has a single spatial axis, so its arithmetic complexity per output element is small and grows only with the filter size K. Its working set is correspondingly compact and fits the near-memory subsystem and Last-Level Cache (LLC), which keeps off-chip traffic negligible and places the measured operational intensity far to the right of the knee and close to the theoretical limit. To exercise the memory system and the available parallelism, we fix the sequence length at $L = 256$ and drive the channel count high, sweeping $C_{\text{in}} = C_{\text{out}}$ over the range labeled in Figure 7. The filter size sets the operational intensity, which grows with K; at low channel counts this separates the `K3` and `K7` variants by a wide margin, as `c8/K3` and `c8/K7` show, whereas the gap narrows as C_{in} rises and channel traffic comes to dominate the filter contribution.

Figure 7 shows the three dataflows landing within a narrow band, at per-dataflow peaks of 2.08 Ops/Cycle for IS (the best), 1.96 Ops/Cycle for WS, and 1.86 Ops/Cycle for OS. The near-tie follows from the on-chip residency of the data, which limits the opportunity for one dataflow to significantly outperform the others, as the VRF buffers many channels of the resident length-L sequence. A set of output channel accumulators is held in the VRF, effectively acting as a blocked output stationary dataflow that decreases the output spills. Both IS and WS lead in throughput because of their inherent data reuse combined with the blocked OS scheme. OS trails as expected because IS and WS already retain blocked output accumulators while additionally exploiting input or weight reuse.

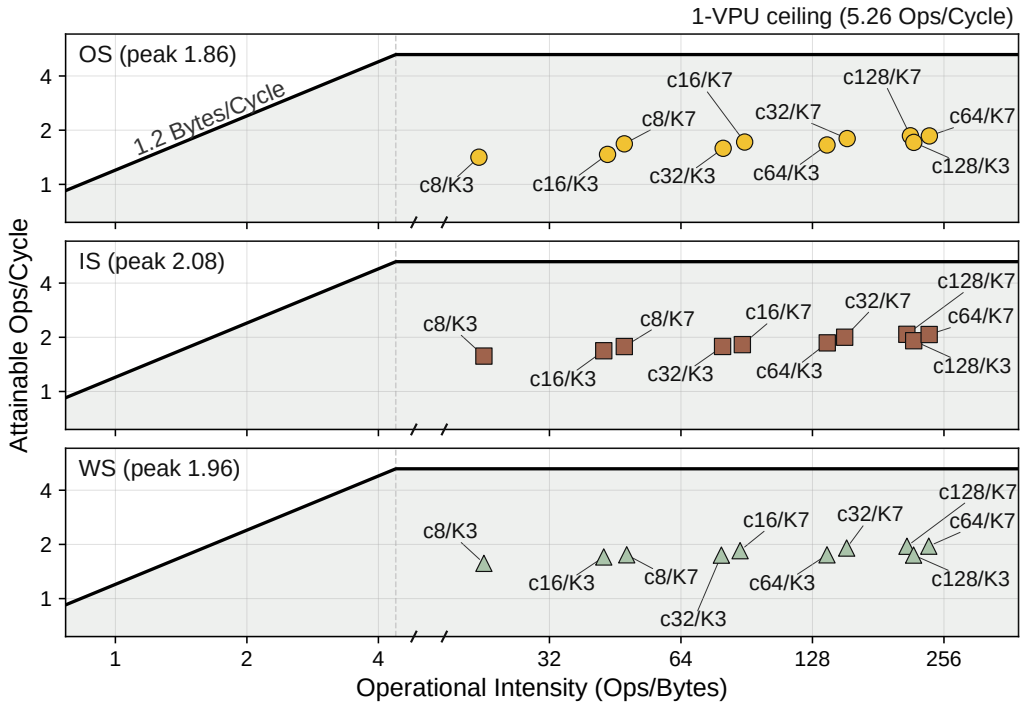

Figure 7. Roofline characterization of the `QLinearConv` kernel (1D). Each panel reports one dataflow: OS, IS, and WS, from top to bottom. The sequence length is fixed at $L = 256$. Each point is labeled by the channel count, c, with $C_{\text{in}} = C_{\text{out}}$, and the filter size, K. Only the maximum ceiling is reported as stated in Section 5.4. The maximum deviation between the displayed and shape-specific ceilings is 2.8%.

The high operational intensity is a direct consequence of the working set fitting the near-memory subsystem, and all configurations therefore scale close to the theoretical bound (within 1.01× of it). The exception is `c128/K7`, the largest problem size. The resulting off-chip re-reads cost a 1.51× reduction in operational intensity relative to the near-ideal regime.

5.8. QDepthwiseConv

As stated in Section 4.3.4 this operator is memory-bound, dominated by input reads. We characterize it in the regime where it is commonly deployed, the later stages of convolutional networks such as MobileNet [25]. For this reason, the channel count is fixed at 256 (our target VLMAX) to fully leverage the parallelism of the VPU, the image size sweeps over $\{7, 14, 28, 56\}$, and filter sizes are $\{3, 5, 7\}$.

Figure 8 compares the considered depthwise schedules, including the baseline OS implementation and the input-reuse schedule enabled by the sliding ring buffer. IS reaches a peak of 1.99 Ops/Cycle, while WS and OS peak at 1.97 Ops/Cycle and 1.51 Ops/Cycle, respectively. The OS schedule is intentionally left naive, so it re-reads the full input patch at every step and trails the other two. IS adds the ring slide, which advances the patch by loading only the newly exposed column and reusing the rest, cutting input traffic by the K-fold factor. WS is degenerate in the depthwise case: the single $K \times K$ (just for $K = 3$; larger filters do not easily fit in the VRF) tap-set per channel is small enough to stay resident and the output is one accumulator per pixel, so weight- and output-stationarity naturally follow; WS therefore converges to the same optimum as IS. The placement of each point is ruled by whether its working set fits the memory subsystem. At small image sizes the working set stays on-chip and configurations land within 1.02× of the theoretical operational intensity. From `i28` onward the set overflows on-chip capacity and intensity falls at least 1.38× short

of ideal, reaching 3.4× excess off-chip traffic at `i56/K7`; here throughput and intensity decouple, with peak Ops/Cycle still sustained at `i28/K3` and `i56/K3` even though their working sets already overflow and sit 1.4×–1.8× off the ideal intensity. A second constraint emerges along the filter-size axis: at fixed image size, throughput falls steadily with K (`i56/K3` → `i56/K5` → `i56/K7` drops from ∼2 to ∼1 Ops/Cycle and below) because for $K > 3$ the VRF becomes the binding resource: a 7×7 patch needs 49 registers for the filter and as many for the input, exceeding the VRF and forcing the convolution for a single output pixel to be tiled, adding traffic between the VRF, LLC, and main memory.

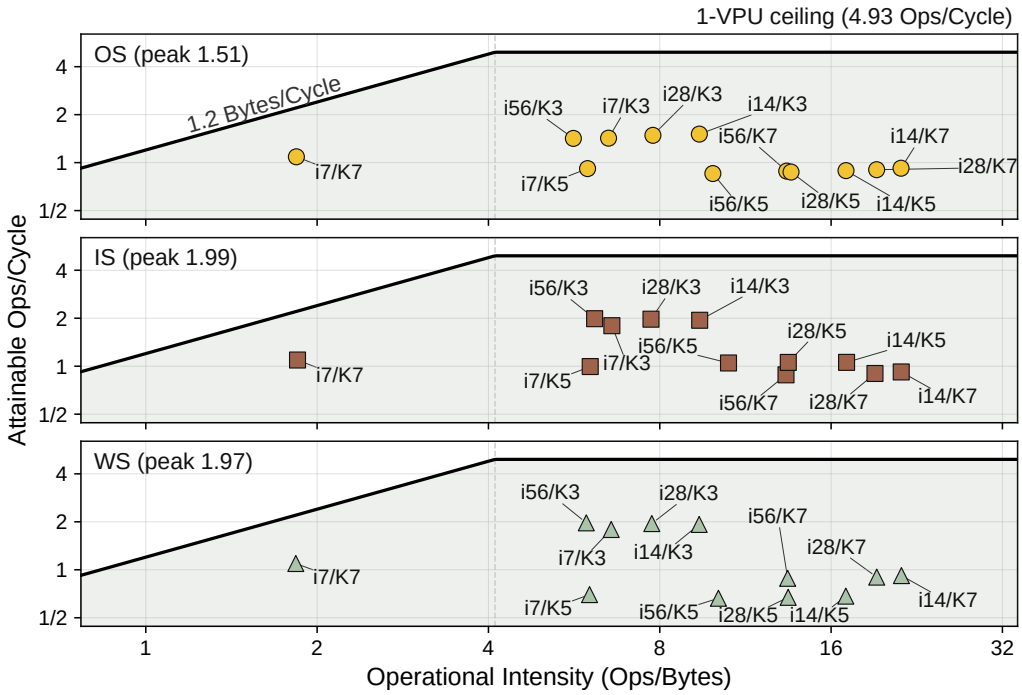

Figure 8. Roofline characterization of the `QDepthwiseConv` kernel. Each panel reports one dataflow: OS, IS, and WS at fixed channel count $C_{in} = C_{out} = 256$. Each point denotes a square image of $i \times i$ and a filter of size $k \times k$. Only the maximum ceiling is reported as stated in Section 5.4. The maximum deviation between the displayed and shape-specific ceilings is 12.6%.

5.9. Optimization Opportunities

After the scaling analysis for each operator carried out in Sections 5.5–5.8, we identified the best dataflow for each of them. Figure 9 plots how those kernels respond to the two levers the selected near-memory platform exposes: the number of VPUs and the main memory bandwidth using problem sizes chosen so that each cooperating VPU is sufficiently utilized, even at SIMD = 4. As the roofline model [7] prescribes, the most effective optimization depends on whether a kernel is compute- or memory-bound, so we traverse the operators from the compute-bound end of the intensity axis toward the memory-bound one. The platform is instantiated with 1, 2, or 4 cooperating VPUs. While the physical memory ports remain unchanged, a tunable latency module intercepts the off-chip memory traffic and introduces additional response latency to emulate sustained bandwidths of 1.2, 0.8, or 0.5 B/Cycle. These values are used as reference points for three common use-cases: the fastest bandwidth our system can sustain, the measured store-direction bandwidth, and an approximation of a quad-Serial Peripheral Interface (SPI) interface, respectively. All scaling figures below are quoted at 1.2 B/Cycle unless stated otherwise.

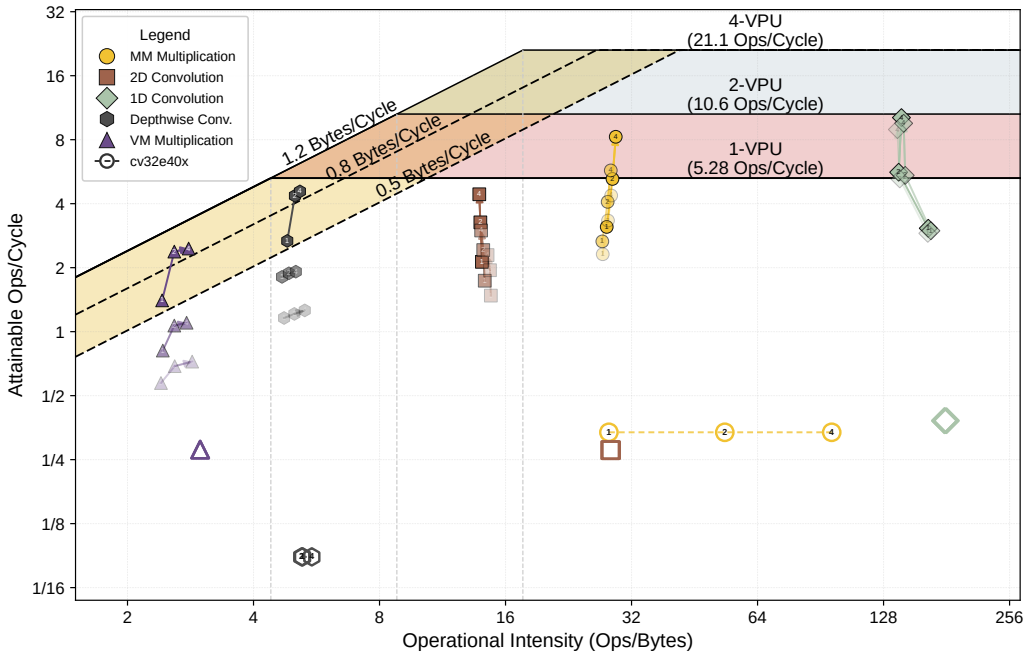

Figure 9. Optimization opportunities and scalability analysis of the various kernel dataflows under SIMD scaling and memory pressure. Each series is one kernel family, drawn with the dataflow that won its per-kernel characterization. Along the markers, labels 1, 2, and 4 denote the number of cooperating VPUs, and the three lines per series correspond to main memory bandwidths of 1.2, 0.8, and 0.5 B/Cycle, fading as the bandwidth decreases. The horizontal roofs mark the highest 1-, 2-, and 4-VPU compute ceilings (5.28 Ops/Cycle, 10.6, and 21.1 Ops/Cycle) across all kernels analyzed here; the diagonal roofs mark the three bandwidths. The scalar-host series report the same operators executed on the `cv32e40x` using the on-chip memory available in each configuration and scalar kernels from μRISCV-NN [9].

At the compute-bound extreme sits the one-dimensional convolution, with $L = 1024$ such that at SIMD = 4 each VPU holds a 256-element slice matching the maximum vector length that yielded the highest throughput in Section 5.7; shorter sequences would leave the units under-occupied and the resulting scaling would reflect vector under-utilization rather than platform behaviour. SIMD partitions the sequence with unit-stride accesses and replication scales throughput almost ideally, by 1.81× at two and 3.05× at four VPUs, the shortfall from 4× reflecting higher LLC pressure. Because the data remain resident in the cache, reducing the bandwidth to 0.5 B/Cycle costs only 1.05× at one VPU and 1.16× at four.

Moving left along the operational-intensity axis, we consider the matrix multiplication with $M = K = N = 1024$, so that each of the four VPUs processes 256-element-wide tiles, matching the single-VPU tile width characterized in Section 5.5. It scales sub-ideally, by 1.67× at two and 2.61× at four, because dividing the compute does not shorten the transfer time; the trajectory leans slightly right as VPUs are added, intensity rising from 27.9 to 29.3 Ops/Bytes. Latency is more detrimental here: from 1.2 to 0.5 B/Cycle the four-VPU throughput falls 1.86× to 4.3 Ops/Cycle, 19% below the single-VPU ceiling, and the SIMD speedup collapses from 2.61× to 1.88×, so slow memory cancels the entire benefit of the additional units.

The 2D convolution parallelizes over image height at a fixed 224 × 224 input with $C_{in} = 3$, $C_{out} = 32$ and a 3 × 3 filter, scaling vertically at nearly constant intensity to 2.01× at four VPUs. It is more fragile than the 1D case: the four-VPU throughput falls 1.85× at 0.5 B/Cycle and the SIMD gain itself shrinks to 1.54×.

For memory-bound operators, computation is not the dominant bottleneck; memory latency is therefore addressed first through software prefetching [7]: successive unit-stride DMA transactions are queued so that one VPU computes on its tile while the next is loaded. This double buffering carries throughput from one to two VPUs toward the ceiling, with a smaller further gain from two to four, since two units already suffice to hide the transfer. Depthwise convolution shows both the benefit and its limit: prefetching lifts throughput 1.87× at four VPUs, but at 0.5 B/Cycle it can no longer hide the memory-transfer latency behind computation and the one-, two-, and four-VPU runs collapse onto essentially the same point.

VM multiplication closes the traversal at the memory-bound extreme, motivated by language-model inference. With unit-stride weight loads, throughput improves 1.67× from one to two VPUs but only 1.04× from two to four. Prefetching progressively hides the transfer latency, reducing the stalled share of execution from 58% to 4%. Nevertheless, aggregate throughput remains limited by the bytes the interface delivers, since each weight is read exactly once. At four VPUs it reaches roughly 70% of the 1.2 B/Cycle roof, its closest approach, the residual being loop control. It is correspondingly the most bandwidth-sensitive kernel in the set: raising the bandwidth from 0.5 to 1.2 B/Cycle lifts four-VPU throughput 3.3×, from 0.71 to 2.34 Ops/Cycle.

The scalar-host series bounds what the same operators achieve on the `cv32e40x` with no P- or V-extension, running separately from the QUINN kernels to prevent concurrent memory accesses and using the scalar-fallback kernels of μRISCV-NN [9]. Operands are staged into the near-memory subsystem by the same DMA transfers used by the vector kernels, and since every added VPU contributes to the local storage, each configuration grants the host additional on-chip capacity, so only the execution model differs. An ideal single-issue rate of one operation per cycle places the scalar knee at 0.83 Ops/Bytes, below the intensity computed for each operator, so the host is compute-bound throughout. Its attained throughput is nevertheless limited by the control flow surrounding each multiply–add pair and by the repeated loads and stores traversing the system bus, reaching 0.38 Ops/Cycle on one-dimensional convolution, 0.33 on MM multiplication, 0.28 on VM, 0.27 on two-dimensional convolution, and 0.09 on depthwise.

Table 3 reports both sides of that comparison. For each operator, the throughput gain reported is computed as $G_P = P/P_{\text{scalar}}$, where $P = N_{\text{op}}/C$ and C is the measured execution-cycle count. Since QUINN and the scalar baseline execute the same operator and problem size, N_{op} is identical and therefore $G_P = C_{\text{scalar}}/C$. Similarly, the intensity loss is defined as $L_I = I_{\text{scalar}}/I$, with $I = N_{\text{op}}/B_{\text{off}}$; hence, for the same operation count, $L_I = B_{\text{off}}/B_{\text{off,scalar}}$. Execution cycles and off-chip traffic are measured on the target as described in Section 5.4, with the latter derived from the hardware cache miss and eviction counters. Near-memory vector execution always attains higher throughput than the scalar schedule, at the expense of moderate additional off-chip traffic. The two orderings are unrelated: depthwise achieves the largest throughput gain in the set (51.3×), whereas matrix multiplication buys less throughput for more than three times the cost (24.5× against 3.28). Notably, the scalar μRISCV-NN MM is also the only operator that shows better reuse with additional VPUs, its intensity ratio growing from 1.01 at one VPU to 3.28 at four, since its operation count grows cubically in the problem size while its operand traffic grows only quadratically (Section 5.5). For the other four the ratio is flat under replication. VM multiplication and depthwise convolution perform too few operations per byte loaded for memory capacity to matter. Two-dimensional convolution instead has ample reuse—each input pixel feeds $K_h K_w C_{\text{out}}$ MACs—but the outer-product formulation re-fetches the input band across the C_{out} sweep, which the scalar schedule's output-row tiling avoids and the cache only partly absorbs; the gap does not close under replication

because the cooperating VPUs partition H rather than C_{out}, so each unit still re-reads its own band. Off-chip traffic is therefore governed by the formulation of the execution model, rather than by the arithmetic throughput available, either because the model cannot convert added capacity into reuse (MM) or because its loop ordering trades reuse for additional throughput (2D convolution).

Table 3. Throughput gain $G_P = P/P_{\text{scalar}}$ and operational intensity loss $L_I = I_{\text{scalar}}/I$ against the scalar host `rv32imac cv32e40x`, at one and four VPUs and at 1.2 B/cycle. Left block > 1 favours QUINN; right block > 1 favours the scalar schedule. The absolute throughput P [Ops/Cycle] and operational intensity I [Ops/Bytes] achieved by QUINN and the scalar host, taken from Figure 9 at 1 and 4 VPUs, are reported alongside each ratio. In the G_P column, bold marks the highest throughput gain, while in the L_I column it marks the lowest operational intensity loss.

	Throughput P [Ops/Cycle]						**Operational Intensity I [Ops/Bytes]**					
	QUINN		**Host**		G_P		**QUINN**		**Host**		L_I	
Kernel	**1 V**	**4 V**	**1 V**	**4 V**	**1 V**	**4 V**	**1 V**	**4 V**	**1 V**	**4 V**	**1 V**	**4 V**
1D convolution	2.97	10.16	0.380	0.382	7.8	**26.6**	136	141	179	179	1.32	**1.27**
MM	3.11	8.26	0.339	0.337	9.2	**24.5**	27.9	29.3	28.3	96.2	**1.01**	3.28
2D convolution	2.13	4.43	0.277	0.281	7.7	**15.8**	14.1	13.9	28.5	28.6	**2.02**	2.06
Depthwise	2.67	4.57	0.085	0.089	31.4	**51.3**	4.82	5.16	5.24	5.51	1.09	**1.07**
VM	1.40	2.46	0.281	0.280	5.0	**8.8**	2.42	2.80	2.98	2.98	1.23	**1.06**

That same distinction between execution model and formulation lets us separate how much of the throughput gain in Table 3 comes from the vectorized execution substrate itself and how much from the QUINN kernel design. We isolate the two by running the general `QLinearMatMul`-OS kernel constrained to $M = 1$ against the dedicated VM kernel of Section 4.3.1 at the `VM256` configuration ($K = N = 256$) of Section 5.5: both execute the same vector instructions on the same hardware, so any residual gap is attributable to the kernel formulation rather than to vectorization. We choose the VM case because it is memory-bound, so compute throughput contributes least and the comparison is most sensitive to the formulation. The dedicated kernel achieves a 1.5× speedup corresponding to 33% fewer cycles and a saving of 570k cycles: unlike the general kernel, it overlaps the DMA load of the next weight tile with computation on the current one, hiding transfer latency that the general formulation cannot recover once its loop structure collapses to a single output row. The generic kernel on the vector substrate therefore already yields most of the gain over the scalar host, while the operator-specific dataflow adds a further, non-negligible factor on top of it. In autoregressive language-model inference, where the VM kernel is invoked once per decoding step, this per-call saving accumulates across invocations, widening the advantage well beyond the single-call figure.

5.10. Energy and Architectural Generalization

To assess whether the dataflow ranking and optimization guidelines of Section 5.9 generalize beyond the single NM-Carus [6] configuration characterized so far, and to place them on a common footing with energy, we sweep cache size, VRF/VLEN size, and lane count for the `QLinearMatMul` kernel at $M = K = N = 256$, holding SIMD replication and memory latency fixed at the values already characterized in Section 5.9. Figure 10 reports the resulting roofline for the three dataflows, and Table 4 the corresponding energy per operation and FPGA resource utilization. Cache and VRF/VLEN size are increased jointly at each labeled point, from 64 to 128 to 256 cache [kiB]/VLEN [elements]. For OS and IS this raises operational intensity with diminishing returns (3.09–3.32× then 1.98–2.02× for OS; 3.05–3.24× then 1.64–1.67× for IS): as on-chip capacity approaches the working set, a growing fraction of the operand traffic is already resident, so each further doubling converts a smaller residual of off-chip traffic. WS departs from this trend, nearly flat from

64 to 128 kiB (1.19–1.20×) then jumping 4.09–4.20× at 256 kiB: below 256 kiB the `int32` partial sums it must spill on every non-final K-tile do not fit on-chip, so the extra cache is absorbed by other structures and leaves the spill traffic unaffected; only at 256 kiB the capacity is sufficient to retain them, collapsing the spill traffic. Lane count acts orthogonally: at fixed capacity, doubling lanes from two to four and four to eight raises throughput by 1.51–1.91× per step, a sub-2× scaling consistent with the control-flow and DMA-descriptor overhead identified in Section 5.9, while operational intensity is essentially unchanged (e.g., OS spans only 26.2–27.5 Ops/Bytes across two, four, and eight lanes at 64 kiB). The three knobs thus recover the qualitative structure of the fixed NM-Carus [6] configuration: OS remains the most data-efficient dataflow at every point, and the OS, IS, WS ordering is preserved across the full range tested.

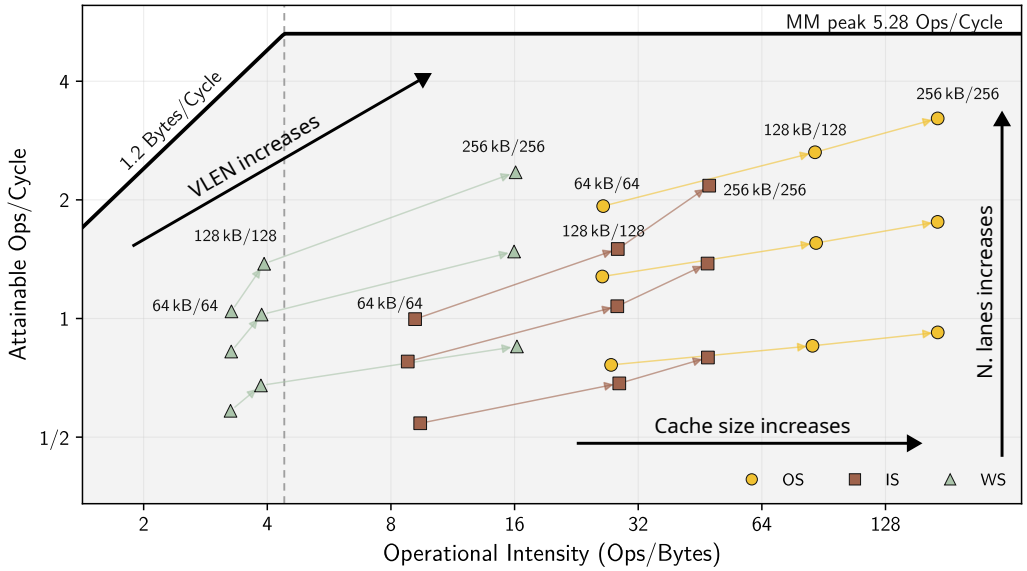

Figure 10. Sensitivity of the `QLinearMatMul` roofline to cache size, VRF/vector length (VLEN), and VPU lane count, at fixed problem size $M = K = N = 256$ with a single VPU (SIMD = 1) and zero added memory latency. Each marker series corresponds to one dataflow (OS, IS, WS); within a series, cache size and VLEN are swept jointly over {64, 128, 256} kiB/elements (labeled pairs), while the three parallel lines per dataflow report lane counts of 2, 4, and 8. Horizontal displacement along a line reflects the operational-intensity gain from enlarging on-chip capacity; vertical displacement between lines at matching labels reflects the throughput gain from added lane parallelism, which leaves operational intensity essentially unchanged.

Table 4. `QLinearMatMul` with M = K = N = 256: energy per op, energy saving vs. the scalar host, and FPGA resource utilization across lanes × cache. Power is extracted from post-implementation dynamic power analysis using Vivado 2020.2 with target FPGA XCZU7EV-2FFVC1156. Bold values indicate the lowest energy per operation among the three proposed dataflows.

Lanes	Cache [kiB]	Energy per op [pJ/op]			Energy Saving (OS)	FPGA Resources			
		OS	IS	WS	vs. CPU	LUT	FF	BRAM	DSP
2	64	**327.4**	386.7	372.8	36.7×	4253	855	24	2
2	128	**380.2**	442.2	428.2	31.6×	4545	867	40	2
2	256	**488.0**	565.2	529.8	24.6×	5129	891	72	2
4	64	**295.3**	388.3	380.4	40.7×	7185	1321	24	4
4	128	**320.9**	412.5	406.3	37.5×	7477	1333	40	4
4	256	**376.7**	480.5	447.9	31.9×	8061	1357	72	4
8	64	**414.0**	641.4	639.8	29.0×	13,628	2255	24	8
8	128	**421.8**	631.4	638.8	28.5×	13,920	2267	40	8
8	256	**450.4**	667.2	617.0	26.7×	14,504	2291	72	8

Energy per operation is obtained by multiplying the measured kernel execution cycles by the dynamic power drawn by the near-memory VPU, extracted from post-implementation power analysis with Vivado 2020.2 for the target `xczu7ev-2ffvc1156`, and dividing by the operation count of Equation (1); the ratio to the same quantity measured for the scalar host (`cv32e40x`) executing the corresponding μRISCV-NN [9] kernel gives the energy-saving metric in Table 4. The VLEN axis is omitted from the energy/resource results in Table 4, since a larger VRF mainly raises the dynamic power drawn by the register file without materially changing the energy-per-operation trend, adding noise to the comparison. The lowest energy per operation, 295.3 pJ/op, occurs at four lanes and 64 kiB, the same point as the highest saving (40.7×). The two-lane, 64-kiB configuration draws less power and occupies far fewer resources (4253 LUTs and 855 FFs against 7185 and 1321 at four lanes), yet its energy per operation, 327.4 pJ/op, is higher: with the datapath under-occupied, more control and DMA-descriptor switching activity is spent per useful operation, and the longer runtime outweighs the lower average power. At eight lanes energy rises again to 414.0–450.4 pJ/op, as the added toggling logic draws more dynamic power than the sub-2× throughput gain over four lanes can amortize. Cache size behaves monotonically: enlarging it from 64 to 256 kiB raises energy per operation by 1.29–1.37× across all dataflows and lane counts, the additional Block RAM (BRAM) contributing switching power without a compensating throughput gain once the working set already fits the smaller cache. Consistent with Section 5.9, OS remains the lowest-energy formulation at every configuration, its shorter runtime following from the absence of partial-output spills. These three views do not select the same configuration, and none is sufficient alone. Throughput peaks with lane count (eight lanes), energy per operation with a balanced datapath (four lanes, 64 kiB), and operational intensity with on-chip capacity (256 kiB); the optima are disjoint. Operational intensity must moreover be read alongside energy per operation precisely because the latter excludes energy drawn from off-chip memory accesses: a configuration that minimizes on-chip energy per operation but thrashes the cache pays that omitted cost in main-memory fetches, which the intensity metric exposes as a leftward drift on the roofline. Choosing well therefore means reading Figure 10 and Table 4 together against the target's constraints such as the latency budget, the energy envelope, or the off-chip bandwidth ceiling. In providing this joint characterization, QUINN is useful beyond the kernels themselves: it lets a designer size lanes, cache, and VRF for a near-memory system, and a programmer select the dataflow based on the measured trade-offs rather than by per-platform trial, before committing to a given configuration.

5.11. MobileNetV2: Application-Level Projection

To move beyond isolated operators, we estimate the cost of MobileNetV2 [25] on QUINN. We use MobileNetV2-1.0 [25], a convolutional network built from inverted-residual blocks, trained for 1000-class image classification on ImageNet [29], that pairs a depthwise `QLinearConv` for spatial mixing with pointwise convolutions for channel mixing, and whose int8 quantized form carries roughly 3.7 MB of weights and performs about 300 M MACs per 224 × 224 inference. Its depthwise–pointwise structure exercises exactly the two kernels most sensitive to layout and dataflow in the preceding characterization. Because coupling the library to a compiler would require additional work outside the present scope, we take each layer of the pre-quantized ONNX model from the ONNX Model Zoo [30] and execute the matching QUINN kernel at that layer's shape, back-to-back in ONNX order, on the FPGA setup of Section 5.1: depthwise layers run on the depthwise kernel of Section 4.3.4 and pointwise layers on the matrix-multiplication kernel of Section 4.3.1, all in the HWC layout these kernels prefer; additionally, a final fully-connected classification layer runs on the matrix-multiplication kernel of Section 4.3.1, at the 8-lane, 128 kiB configuration.

Cycles are summed from the `mcycle` CSR and off-chip traffic from the cache performance counters. Each kernel is individually bit-exact against ONNX Runtime by the validation of Section 5.3, so the summed figure is a faithful first-order estimate of a complete evaluation cost; realizing it as a running deployment, however, additionally requires operand lowering, weight staging, and inter-layer layout management, which a full Bring Your Own Code (BYOC) integration in TVM [31] would provide and which we leave to future work. Table 5 reports the estimate at SIMD = 1 and SIMD = 4. Replication yields only a 1.07× latency reduction, far below the 2–3× per-kernel gains of Section 5.9: the network is dominated by its early, high-resolution but narrow-channel layers, where few channels leave the additional VPUs under-occupied, whereas the deeper channel-heavy layers that replication favours contribute a smaller share of the total. Off-chip traffic is essentially unchanged under replication (186.2 against 187.0 MiB), consistent with replication being a throughput-side lever that does not alter the bytes moved. The network's absolute traffic is instead set by its layer-to-layer dataflow: activations are largely produced and consumed once in a producer–consumer fashion between successive layers, offering little inter-layer reuse for the cache to exploit, so the working set is dominated by the per-layer activation footprint rather than by reused operands. The energy figures make the trade-off explicit: because SIMD = 4 draws four times the dynamic power for a 1.07× speedup, it spends 4.73 J per image against 1.27 J at SIMD = 1—a 3.7× energy penalty for a marginal latency gain. As in Section 5.10, replication pays only where it shortens runtime proportionally; on a network whose bottleneck layers cannot use the extra units, a single VPU is the energy-optimal choice, and the added units are justified only when latency is the binding constraint. This estimate shows that the characterized kernels can be applied across the QUINN-supported layers of MobileNetV2 [25] without per-layer kernel redesign.

Table 5. MobileNetV2-1.0 estimate on QUINN, all layers executed in ONNX order at the 8-lane, 128 kiB configuration under HWC layout, averaged over five runs at 80 MHz. The operation count N_{op} follows Equation (1); energy uses the estimated dynamic VPU power of 0.108 W per unit. The best value in each row is shown in bold.

Metric	SIMD = 1	SIMD = 4
N_{op}	638.5×10^6	
Total cycles [$\times 10^6$]	941.3	**875.7**
Latency per image @80 MHz [ms]	11,766.2	**10,946.3**
Off-chip traffic [MiB]	**186.2**	187.0
Throughput [Ops/Cycle]	0.68	**0.73**
Energy per image @80 MHz [J]	**1.27**	4.73

6. Conclusions

This paper identifies a minimal RISC-V Vector Extension (RVV) subset for near-memory VPUs executing integer-precision kernels that are bit-exact with the quantized ONNX operator semantics consumed by deployment tools. Using NM-Carus [6], we provide an operator-level roofline characterization combining attained throughput and measured off-chip traffic across realistic bandwidths of $\{1.2, 0.8, 0.5\}$ B/Cycle and configurations instantiating $\{1, 2, 4\}$ VPUs. The optimal dataflow is kernel-dependent: OS wins matrix multiplication (1.49× at the largest tested size) and two-dimensional convolution (1.13×), and IS wins depthwise (1.01×) and causal one-dimensional convolution (1.06×). Compute-bound kernels favour output-resident accumulation and VPU parallelism; memory-bound ones favour input reuse and prefetching. Replication pays only while the memory interface keeps up: at 0.5 B/Cycle the four-VPU matrix multiplication falls to 4.3 Ops/Cycle, below the single-VPU compute ceiling, cancelling the benefit of the

additional units. Against a scalar host executing the same operators on the same target, the vector kernels sustain 5.0–51.3× the throughput, but the scalar schedule retains higher operational intensity on all five operators, so that throughput is bought with additional off-chip traffic in every case, though the gain exceeds the cost by at least 3.8×. That cost has two distinct sources: a formulation that cannot convert added capacity into reuse (matrix multiplication, whose intensity gap grows from 1.01 to 3.28 as VPUs are added) and a loop order that forgoes reuse the operator offers at any capacity (two-dimensional convolution, flat at $\sim 2\times$). Off-chip traffic is therefore governed by the formulation the execution model admits rather than by the arithmetic throughput available, which is what makes the per-operator dataflow selection consequential. The characterization is performed per operator and under cold-cache conditions; whether the per-operator dataflow winners remain optimal once operators are composed, with the cache state carried between layers, is left to future work together with end-to-end deployment of quantized networks and autoregressive models.

Author Contributions: Conceptualization, V.P. and F.G.; methodology, V.P. and F.G.; software, V.P. and F.G.; validation, V.P. and F.G.; writing, V.P. and F.G.; review and supervision, G.M. and M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The QUINN kernels are implemented against the interface exposed by the near-memory VPUs of [6], which is not publicly distributable; the kernel sources are therefore available from the authors on request. The raw data supporting the conclusions of this article are likewise available from the authors on request.

Acknowledgments: The authors gratefully acknowledge the Embedded Systems Laboratory (ESL) led by David Atienza at the École Polytechnique Fédérale de Lausanne (EPFL) for providing access to the NM-Carus IP during the visiting period in which this work was carried out.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
CSR	Control and Status Register
CV-X-IF	Core-V eXtension Interface
DMA	Direct Memory Access
DSP	Digital Signal Processing
FF	Flip-Flop
FLOPS	Floating Point Operations per Second
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
IS	Input Stationary
ISA	Instruction Set Architecture
LUT	Look-Up Table
MAC	Multiply-accumulate
ML	Machine learning
MM	Matrix-Matrix
HWC	Height, Width, Channels
NMC	Near-Memory Computing
NN	Neural Network
ONNX	Open Neural Network Exchange
OS	Output Stationary

PIM	Processing-in-Memory
RVV	RISC-V Vector Extension
SIMD	Single Instruction, Multiple Data
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TCN	Temporal Convolutional Network
LLC	Last-Level Cache
VM	Vector-Matrix
VPU	Vector Processing Unit
VRF	Vector Register File
WS	Weight Stationary
LSB	Least Significant Bit
BRAM	Block RAM
BYOC	Bring Your Own Code
DNN	Deep Neural Network

References

1. Zhou, Z.; Chen, X.; Li, E.; Zeng, L.; Luo, K.; Zhang, J. Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing. *Proc. IEEE* **2019**, *107*, 1738–1762. [CrossRef]
2. Saha, S.; Xu, L. Vision transformers on the edge: A comprehensive survey of model compression and acceleration strategies. *Neurocomputing* **2025**, *643*, 130417. [CrossRef]
3. Abadade, Y.; Temouden, A.; Bamoumen, H.; Benamar, N.; Chtouki, Y.; Hafid, A.S. A Comprehensive Survey on TinyML. *IEEE Access* **2023**, *11*, 96892–96922. [CrossRef]
4. Oquab, M.; Darcet, T.; Moutakanni, T.; Vo, H.; Szafraniec, M.; Khalidov, V.; Fernandez, P.; Haziza, D.; Massa, F.; El-Nouby, A.; et al. DINOv2: Learning Robust Visual Features without Supervision. *arXiv* **2023**, arXiv:cs.CV/2304.07193. [CrossRef]
5. Gholami, A.; Yao, Z.; Kim, S.; Hooper, C.; Mahoney, M.W.; Keutzer, K. AI and Memory Wall. *IEEE Micro* **2024**, *44*, 33–39. [CrossRef]
6. Caon, M.; Choné, C.; Schiavone, P.D.; Levisse, A.; Masera, G.; Martina, M.; Atienza, D. Scalable and RISC-V Programmable Near-Memory Computing Architectures for Edge Nodes. *IEEE Trans. Emerg. Top. Comput.* **2025**, *13*, 1003–1018. [CrossRef]
7. Williams, S.; Waterman, A.; Patterson, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* **2009**, *52*, 65–76. [CrossRef]
8. Garofalo, A.; Rusci, M.; Conti, F.; Rossi, D.; Benini, L. PULP-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors. *Philos. Trans. R. Soc. A* **2019**, *378*, 20190155. [CrossRef] [PubMed]
9. van Kempen, P.; Jones, J.P.; Mueller-Gritschneder, D.; Schlichtmann, U. muRISCV-NN: Challenging Zve32x Autovectorization with TinyML Inference Library for RISC-V Vector Extension. In Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions, New York, NY, USA, 7–9 May 2024; pp. 75–78. [CrossRef]
10. Yu, M.S.; Yuan, C.Y.; Chen, T.L.; Lee, J.K. Case Study: Optimization Methods with TVM Hybrid-OP on RISC-V Packed SIMD. *IEEE Access* **2024**, *12*, 64193–64211. [CrossRef]
11. Parashar, A.; Raina, P.; Shao, Y.S.; Chen, Y.H.; Ying, V.A.; Mukkara, A.; Venkatesan, R.; Khailany, B.; Keckler, S.W.; Emer, J. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Madison, WI, USA, 24–26 March 2019; pp. 304–315.
12. Mei, L.; Houshmand, P.; Jain, V.; Giraldo, S.; Verhelst, M. ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators. *IEEE Trans. Comput.* **2021**, *70*, 1160–1174. [CrossRef]
13. Gilbert, M.; Wu, Y.N.; Emer, J.S.; Sze, V. LoopTree: Exploring the Fused-Layer Dataflow Accelerator Design Space. *IEEE Trans. Circuits Syst. Artif. Intell.* **2024**, *1*, 97–111. [CrossRef]
14. Scherer, M.; Macan, L.; Jung, V.J.B.; Wiese, P.; Bompani, L.; Burrello, A.; Conti, F.; Benini, L. Deeploy: Enabling Energy-Efficient Deployment of Small Language Models on Heterogeneous Microcontrollers. *Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2024**, *43*, 4009–4020. [CrossRef]
15. Palacios, P.; Medina, R.; Ansaloni, G.; Atienza, D. HEEPstor: An Open-Hardware Co-design Framework for Quantized Machine Learning at the Edge. In Proceedings of the 22nd ACM International Conference on Computing Frontiers: Workshops and Special Sessions, New York, NY, USA, 28–30 May 2025; pp. 22–25. [CrossRef]
16. Chiu, K.L.; Eichler, G.; Lin, C.T.; Di Guglielmo, G.; Carloni, L.P. WOLT: Transparent Deployment of ML Workloads on Lightweight Many-Accelerator Architectures. In Proceedings of the 2024 IEEE 42nd International Conference on Computer Design (ICCD), Milan, Italy, 18–20 November 2024; pp. 637–644. [CrossRef]

17. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 2704–2713. [CrossRef]
18. Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *arXiv* **2017**, arXiv:cs.CV/1703.09039. [CrossRef]
19. Yang, C.; Wang, Y.; Farrell, S.; Kurth, T.; Williams, S. Hierarchical Roofline Performance Analysis for Deep Learning Applications. *arXiv* **2020**, arXiv:cs.DC/2009.05257. [CrossRef]
20. Verhelst, M.; Benini, L.; Verma, N. How to Keep Pushing ML Accelerator Performance? Know Your Rooflines! *IEEE J. Solid-State Circuits* **2025**, *60*, 1888–1905. [CrossRef]
21. Kooli, M.; Heraud, A.; Charles, H.P.; Giraud, B.; Gauchi, R.; Ezzadeen, M.; Mambu, K.; Egloff, V.; Noel, J.P. Towards a Truly Integrated Vector Processing Unit for Memory-bound Applications Based on a Cost-competitive Computational SRAM Design Solution. *J. Emerg. Technol. Comput. Syst.* **2022**, *18*, 1–26. [CrossRef]
22. Yang, Y.; Yuan, Y.; Wang, X.; Li, X.; Wu, H.; Liu, Q.; Tang, W.; Fu, X.; Zhang, F. A Multicore Programmable Variable-Precision Near-Memory Accelerator for CNN and Transformer Models. *IEEE J. Solid-State Circuits* **2026**, *61*, 2725–2737. [CrossRef]
23. Eggermann, G.A.; Ansaloni, G.; Atienza, D. Keep All in Memory with Maxwell: A Near-SRAM Computing Architecture for Edge AI Applications. In Proceedings of the 2025 26th International Symposium on Quality Electronic Design (ISQED), San Francisco, CA, USA, 23–25 April 2025; pp. 1–7. [CrossRef]
24. Prieto, A.; Nouripayam, M.; Westring, K.; Mohedano, S.C.; Allfjord, A.; Svensson, L.; Andersson, P.; Rodrigues, J. NMCu-CNN: A Scalable Near-Memory Computing Co-Processor on a RISC-V MCU in 22 nm FDSOI. In Proceedings of the 2025 IEEE Workshop on Signal Processing Systems (SiPS), Hong Kong, China, 1–4 November 2025; pp. 1–5. [CrossRef]
25. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 4510–4520. [CrossRef]
26. Chen, Y.H.; Krishna, T.; Emer, J.S.; Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid-State Circuits* **2017**, *52*, 127–138. [CrossRef]
27. Giuffrida, L.; Schiavone, P.D.; Caon, M.; Masera, G.; Martina, M.; Atienza, D. Scalability analysis of multi-bank near-memory computing in low-power SoCs. In Proceedings of the 2025 IFIP/IEEE 33rd International Conference on Very Large Scale Integration (VLSI-SoC), Puerto Varas, Chile, 12–15 October 2025; pp. 1–4. [CrossRef]
28. ONNX Runtime Developers. ONNX Runtime. Version 1.21.0. 2021. Available online: https://onnxruntime.ai/ (accessed on 25 July 2026).
29. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Miami, FL, USA, 20–25 June 2009; pp. 248–255. [CrossRef]
30. ONNX Model Zoo Contributors. MobileNetV2-12-int8: Quantized MobileNetV2 ONNX Model. 2022. Available online: https://github.com/onnx/models/blob/main/validated/vision/classification/mobilenet/model/mobilenetv2-12-int8.onnx (accessed on 25 July 2026).
31. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Cowan, M.; Shen, H.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 578–594.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.