Thesis for The Degree of Licentiate of Engineering

Accelerating CNN inference via co-design of convolutional algorithms and long vector processors

Sonia Rani Gupta

Division of Computer and Network Systems
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2025

Accelerating CNN inference via co-design of convolutional algorithms and long vector processors

Sonia Rani Gupta

Department of Computer Science & Engineering
Division of Computer and Network Systems
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2025.

Abstract

Model serving has become crucial for AI applications, with convolutional neural networks (CNNs) driving various applications from object detection to speech recognition. While specialized accelerators and GPUs offer high performance for CNN inference, CPU-based solutions provide better availability and portability for server-side and mobile computing. Vector architectures such as RISC-V Vector extension and ARM Scalable Vector Extension have emerged as a promising solution, offering GPU-like parallel processing capabilities with low latency, high availability, and lower energy consumption.

This thesis investigates co-design opportunities in vector architectures for CNN inference, focusing on the interplay between convolutional algorithmic optimizations and hardware design choices. First, it conducts a co-design study that explores both convolutional algorithm optimizations and hardware parameter tuning such as vector lengths, cache sizes, and vector lanes for CNN inference on ARM-SVE and RISC-VV architectures. Second, it explores the co-design of CNN layers by studying three distinct algorithmic implementations: Direct, im2col+GEMM, and Winograd, in conjunction with hardware parameters for RISC-VV.

While optimizing the im2col+GEMM algorithm, various optimizations have been applied to the GEMM kernel; however, our study shows that not all optimizations benefit different vector architectures equally. Our co-design study using the gem5 simulator demonstrates an $\sim$5$\times$ performance improvement with 16384-bit vector lengths and 256MB of L2 cache, compared to 512-bit vectors and 1MB of L2 cache. Since larger tile sizes cannot be used for the Winograd algorithm due to numerical inaccuracies, this thesis proposes inter-tile parallelism across the input/output channels using 8$\times$8 tiles per channel to utilize longer vector lengths. This approach improves data reuse and achieves an additional performance improvement of 2.4$\times$ (compared to im2col+GEMM) on the A64FX processor. Our co-design study also shows that the Winograd algorithm has lower cache size requirements compared to im2col+GEMM.

The performance of convolutional algorithms depends on layer dimensions (input/output/kernel dimensions, stride, and input/output channels), while computational demands influence SIMD requirements, and cache sharing impacts runtime algorithm selection in model serving. Our study shows that Winograd performs better with smaller vector lengths, whereas the Direct algorithm excels with longer vectors. While im2col+GEMM benefits from larger caches, Direct and Winograd exhibit varying cache sensitivity across VGG16 layers. In contrast, all YOLOv3 layers benefit from the largest simulated L2 cache across all algorithms. To address these complexities, this thesis proposes a random forest classifier that selects the optimal algorithm per layer with 92.8% accuracy and per-layer algorithm selection improves performance by $\sim$2$\times$ compared to using a single algorithm. Finally, our Pareto analysis of area-performance trade-offs for a 7nm RISC-V multicore model shows that algorithm selection leads to increased throughput per area, highlighting the need for co-design in the context of model serving.

Keywords: CNNs, co-design, GEMM, Winograd, Direct, optimizations, long vector architectures, vector length agnostic ISAs, model serving

Acknowledgment

First and foremost, I would like to express my deep and sincere gratitude to my advisor, Professor Miquel Pericàs, for giving me the opportunity to pursue my studies and providing contiguous guidance and support during my studies. His vast knowledge, vision, motivation, and confidence in my work have profoundly encouraged me and taught me the methodology to effectively conduct my research and present my findings with clarity.

I would also like to thank my co-advisor Dr. Nikela Papadopoulou for her insightful feedback and enthusiasm. Her extensive research experience has inspired and taught me to conduct my research work with greater clarity and confidence.

I am grateful to Professor Pedro Petersen Moura Trancoso, who has been my examiner. I would like to say thanks to my, past and present, colleagues and friends at Chalmers, Pirah, Jing, Bhavishya, Hari, Minyu, Nufail, Mahmoud, Fareed, Mateo, Neethu, Per, Monica, Arne, and many others who created a nice and friendly work environment.

Lastly, I would like to express my deepest gratitude to my family for their unwavering support. I am especially thankful to my husband, Amit, and my daughters, Aamia and Aanvi, for always being there for me. I would not have been able to accomplish this without them.

This research was supported by multiple funding sources: The Swedish Research Council (grant no. 2020-04892), and the European High-Performance Computing Joint Undertaking (JU) through several grant agreements: No. 956702 (eProcessor), No. 101036168 (EPI SGA2, under Framework Partnership Agreement No. 800928), and No. 101034126 (The European PILOT). The JU receives support from the European Union's Horizon 2020 research and innovation programme and multiple member states (Spain, Sweden, Greece, Italy, France, and Germany). Additional funding was provided by the Swedish Foundation for Strategic Research through the PRIDE project (CHI19-0048). The simulations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at NSC (grant no. 2018-05973) and the National Academic Infrastructure for Supercomputing in Sweden (NAISS, grant no. 2022-06725), both partially funded by the Swedish Research Council. We thank the Barcelona Supercomputing Center for providing access to an A64FX machine.

List of Publications

Appended publications

This thesis is based on the following publications:

[I] Sonia Rani Gupta, Nikela Papadopoulou, and Miquel Pericàs "Accelerating CNN inference on long vector architectures via co-design"
2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), St. Petersburg, FL, USA, 2023, pp. 145-155, doi: 10.1109/IPDPS54959.2023.00024.

[II] Sonia Rani Gupta, Nikela Papadopoulou, Jing Chen, and Miquel Pericàs "Co-Design of Convolutional Algorithms and Long Vector RISC-V Processors for Efficient CNN Model Serving"
In Proceedings of the 53rd International Conference on Parallel Processing (ICPP '24). Association for Computing Machinery, New York, NY, USA, 73–83. https://doi.org/10.1145/3673038.3673121.

Other publications

The following publications are not included in the thesis.

[a] Sonia Rani Gupta, Nikela Papadopoulou, and Miquel Pericàs "Challenges and Opportunities in the Co-design of Convolutions and RISC-V Vector Processors"
In Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23). Association for Computing Machinery, New York, NY, USA, 1550–1556. https://doi.org/10.1145/3624062.3624232.

Contents

Chapter 1

Introduction

Convolutional Neural Networks (CNNs) play a key role in Artificial Intelligence (AI) inference, powering a wide range of applications such as object detection [1], natural language processing [2], and speech recognition [3]. These models require high throughput and power efficiency due to power constraints (e.g., battery-powered embedded devices or the power caps in data centers). Offloading compute-intensive operations to neural accelerators and GPUs has become a common approach to accelerate CNN inference [4–6]. However, integrating specialized accelerators [7, 8] into general-purpose computing systems remains challenging. Various frameworks [9–11] are optimized for GPUs, but server-side inference requires availability and low latency [12], while mobile and embedded devices benefit from CPU availability and portability [13, 14]. Consequently, optimizing CNNs has become popular on CPUs [14–16], while CPU vendors are increasingly adding deep neural networks (DNN) capabilities to processors [17].

Vector-enabled processors have seen renewed interest as they bring GPU-like parallel processing capabilities to CPUs while maintaining lower energy consumption. In this context, tightly coupled modern long vector processors with low latency, high performance, and power efficiency such as RISC-V Vector Extension (RISC-VV) [18] and ARM Scalable Extension (ARM-SVE) [19] are considered a promising option for efficient CNN inference serving [20, 21]. These vector architectures are based on vector length agnostic (VLA) instruction set architecture (ISA), which ensures code portability across hardware platforms with distinct vector lengths.

Maximizing performance in vector architectures requires efficient algorithmic optimizations and a deeper exploration of the design space, such as leveraging new levels of parallelism introduced by vector units and understanding the relationships between micro-architectural parameters, performance characteristics, and area efficiency for applications of interest. Manual code transformation and optimization are necessary to achieve an efficient algorithmic implementation by exposing the maximum available SIMD parallelism to the vector unit. Additionally, modern architectures are integrating longer vector lengths to support scientific applications and AI workloads. Longer vector lengths increase the high data demand, necessitating larger caches. Furthermore, additional vector lanes become necessary to maintain computational throughput. The relationship between cache capacity, vector lanes, and vector length creates

complex performance trade-offs that must be carefully analyzed. Since these components occupy a significant die area and greatly impact performance, tuning them to the requirements of optimized kernels is essential for designing efficient, high-performing vector architectures for CNN inference.

CNNs consist of consecutive layers, with convolutional layers being the most time-consuming. These convolutional layers can be implemented using various algorithms [22], such as im2col+GEMM, Direct, Winograd, and FFT. The Direct algorithm slides convolutional weights over the input tensor, performing dot products [23]. The im2col+GEMM algorithm transforms the image into a column matrix, turning the convolutional operation into a matrix multiplication by convolving the transformed input matrix with the weight matrix. Winograd and FFT reduce computational complexity, but require transforming the image and weights, performing block-wise multiplications, and applying output transformations. Winograd is effective with small kernel sizes, such as 3×3 or 5×5 [24], while FFT is better suited for larger kernel sizes [25].

Convolutional layers in CNNs vary in their dimensions, including input, output, kernel height and width, input and output channels, and stride. Different convolutional algorithms exhibit varying performance based on each layer's dimensions, due to differences in computational complexity and memory footprints [13]. Additionally, cache size and vector length can impact performance, as some algorithms, like im2col+GEMM, increase the memory footprint. Algorithmic optimizations that exploit longer vector lengths can help achieve higher performance from the vector unit. Moreover, model serving is rapidly becoming the standard approach for deploying AI applications, with cloud providers reporting hundreds of trillions of AI model executions daily [26]. Model serving frameworks [27, 28] enable concurrent model replicas for better resource utilization, although cache contention from co-running inferences affects algorithmic decisions.

Despite extensive research on CNNs and hardware microarchitectural parameters tuning [20, 29–38], the interplay between the optimization space of convolution algorithms and the design space of hardware parameters remains unexplored. The lack of a co-design approach can hamper the design of future vector architectures based on CPUs for CNN inference and model serving.

This thesis fills this gap by providing a co-design study that jointly explores the optimization of convolutional algorithms and the tuning of hardware parameters on long vector architectures, aiming to provide guidance to programmers, hardware designers, and compiler developers. Additionally, it proposes inter-tile parallelism across input/output channels to utilize longer vector lengths for the Winograd algorithm. Furthermore, this thesis conducts a co-design exploration, focusing on the software parameters of convolutional layers and algorithmic implementations alongside hardware parameters for vector architectures, demonstrating that selecting the best algorithm per layer leads to better performance compared to using a single algorithm for all layers. To select the best algorithm at runtime, this thesis proposes a Random Forest classifier that selects the best algorithm for 92.8% of the cases. Furthermore, analyzing the performance-area trade-offs reveals that combining per-layer algorithm selection with model co-location enhances throughput per unit area, emphasizing the importance of co-design in model serving.

1.1 Background

Vector architectures: Vector supercomputers with long vector lengths [39] were first developed in the 1970s to solve scientific problems. A new era of SIMD architectures began in the 1990s with short vectors, initially built for media streaming applications, which later became popular in Digital Signal Processing (DSP) and general-purpose computing [40, 41]. SIMD ISAs with fixed short vector length are now commonly used for general-purpose computing. However, these ISAs provide limited portability, as a new instruction set extension is required if a longer vector length is needed. To overcome this limitation, modern long vector architectures such as RISC-VV [18] and ARM-SVE [19] offer vector length agnostic ISAs, where there is no need to specify a specific vector length. RISC-VV supports MVL up to 16384-bits in powers of 2 whereas ARM-SVE implementations range from 128-bit to 2048-bits with 128-bit increments, as implemented in Fujitsu's A64FX processor with its 512-bit vector length.

Convolutional neural network models The core building block of CNNs is the convolutional layer, which performs the convolution operation between the input data and a set of learnable filters. These layers are followed by other types of layers, including pooling layers, fully connected layers, and normalization layers, depending on the network architecture. YOLOv3, CNN based object detection model, which contains 107 layers of five different types, 75 of these layers are convolutional, making them the most computationally intensive part of the network. Similarly, in VGG16, a model for image classification, 13 of the 25 layers are convolutional, with an additional 3 fully connected layers. the convolutional layers in YOLOv3 and VGG16 consume ∼96% and ∼64% of the total inference time, respectively, when profiled on A64FX using Linux `perf`.

Algorithms for Convolutions: To optimize the performance of CNN-based models, various convolutional algorithms can be employed. In this thesis, I use three convolutional algorithms: im2col+GEMM, Direct, and Winograd to implement the convolutional layers as most CNN-based network models use convolutional layers with small kernel sizes of 1×1, 3×3 or 5×5. I developed an optimized version of the im2col+GEMM algorithm in the Darknet [42] framework, where the convolutional layer is implemented using the im2col+GEMM algorithm. Additionally, I developed an optimized version of the Winograd algorithm from the NNPACK [43] package for implementing these convolutional layers. I then developed and optimized the Direct algorithm for implementing these convolutional layers inside the Darknet framework.

1.2 Related Work

Several works have focused on optimizing convolutions for vector architectures. Specifically, Kelefouras et al. [44] vectorize and optimize the 2D direct convolutions on Intel AVX. Wang et al. [35] optimize the Direct algorithm on ARM NEON. Wang et al. [37] optimize the Winograd algorithm on RISC-V architectures with a custom instruction extension. Louis et al. [29] port and

optimize convolutional and matrix multiplication kernels of CNNs from TensorFlow lite and study the reduced number of instructions on RISC-VV. Alaejos et al. [45]optimize GEMM for deep learning on the ARM-NEON, ARM-SVE, and Intel AVX512 vector extensions. Arm [46] has developed an ARM Compute Library to optimize convolutional kernels for ARM-SVE. Dolz et al. [47] optimize the im2col transformation and Winograd algorithms for ARM-SVE. In another work, Dolz et al. [38] optimize the Winograd algorithm for Intel AVX, ARM NEON, and ARM-SVE architectures. Santana et al. [36] optimize the Direct algorithm for long vector architectures, focusing on the NEC SX Aurora architecture

In the context of microarchitectural parameter tuning of modern long vector architectures, Kodama et al. [33] evaluate the "triad", "gemm" and "nbody" application kernels with multiple vector lengths and number of physical registers using ARM-SVE and evaluated the impact of vector length. Ramirez et al. [32] study the impact of microarchitectural parameters, such as vector lane and vector length, using different vectorized kernels presented in the RISC-V Vectorized Benchmark Suite.

Concerning performance comparisons of different algorithmic implementations of convolutions, Jordà et al. [48] and Xu et al. [49] perform such an analysis on GPUs. Jordà er al. [48] focus on cuDNN and propose that different algorithms should be used depending on the kernel size. Xu et al. [48] also look at cuDNN implementations and propose a scheme for algorithm selection based on the convolution dimensions. Dolz et al. [50] focus on performance-energy tradeoffs of the different algorithms for convolutions on ARM processors. Zlatenski et al. [51] perform a comparative analysis of Winograd and FFT for convolutions using different CNNs on modern CPUs, for full network models.

This thesis focuses on optimizing the im2col+GEMM, Winograd, and Direct convolutional algorithms for long vector architectures using vector length agnostic ISAs. It also presents a co-design study that explores the interaction between microarchitectural parameters and algorithmic optimizations to fully assess the performance potential of vector architectures for CNN inference, providing valuable insights for programmers, hardware designers, and compiler developers. Furthermore, this thesis conducts a co-design study to select the optimal algorithm for each convolutional layer and analyzes its impact on the achievable throughput per area for model serving.

1.3 Problem Statements

Problem 1: Prior work on CNN inference on vector architectures focuses either on applying algorithmic optimizations [20, 29–31, 38] or on studying the hardware micro-architectural parameters design points [32–34]. These studies miss an opportunity to explore the trade-offs in algorithmic and architectural co-design for CNN kernels running on long vector architectures. To address this gap, this thesis investigates the following problem statement.

How to study the design space exploration to co-design an effective vector architecture for high performing CNN inference?

Research questions: To provide a solution to the problem, we need to answer four research questions.

Q1 Are all algorithmic optimizations beneficial for different vector architectures?

Q2 How is the performance of the im2col+GEMM algorithmic implementation affected by very long vector lengths, larger caches, and more vector lanes within a vector unit?

Q3 Can the Winograd algorithm improve the performance of convolutional layers on long vector architectures?

Q4 Can the performance of the Winograd implementation benefit from longer vectors and larger caches?

Problem 2: Previous studies [35–37, 47, 50, 52, 53] have focused on optimizing the performance of specific algorithms on vector architectures, presenting comparative analysis with state-of-the-art libraries for different layers of network models on vector processors, and providing a comparative analysis of different algorithmic implementations of convolutional layers on SIMD ARM-based architectures. Despite the extensive research on convolutional neural networks and various algorithmic implementations, the mutual impact of convolution algorithms and hardware parameters remains unexplored. This limits resource utilization and hampers the task of effectively designing future CPUs for CNN model serving. This thesis performs a co-design study of three convolutional algorithms: Direct, im2col+GEMM, and Winograd and addresses the following problem statement:

How can model serving for CNNs achieve higher performance by co-designing vector architectures?

Research questions: To provide a solution to the problem, four research questions need to be addressed in this context:

Q5 What is the performance of different convolutional algorithms for different layers on vector architectures?

Q6 Is there any single algorithm that benefits all the convolutional layers?

Q7 How to predict the optimal algorithm for each convolutional layer?

Q8 What are the throughput-area tradeoffs to efficiently serve CNNs with vector architectures?

1.4 Contributions

This thesis is based on two papers. *Paper I* addresses the first problem statement and answers the research questions 1, 2, 3 and 4. *Paper I* is the first work that shows the impact of different algorithmic optimizations on different vector architectures (decoupled and integrated) with CNN kernels. *Paper I* also proposes a novel vectorized implementation of Winograd implementation with ARM-SVE that uses inter-tile parallelism across the input/output channels for utilizing the longer vector lengths. The main contributions of this paper are:

- We compare different algorithmic optimizations of the GEMM kernel across two vector architectures: ARM-SVE (tightly integrated vector architecture) and RISC-VV (decoupled vector architecture). Our analysis demonstrates that not all optimizations benefit different vector architectures equally. Further, our co-design study shows that longer vector lengths and bigger caches help to improve the performance by $\sim$5$\times$, however, scalability becomes limited with very long vector lengths (16384-bit).

- We vectorize the Winograd kernels using inter-tile parallelism, which helps to achieve 1.35$\times$ and 1.5$\times$ for YOLOv3 and VGG16 network models, respectively, compared to im2col+GEMM on ARM-SVE. Our co-design study shows that our Winograd algorithm is less sensitive to L2 caches compared to im2col+GEMM.

In *Paper II*, we perform a comparative analysis of Direct, Winograd, and two variants of im2col+GEMM algorithms on each convolutional layer of the YOLOv3 and VGG16 network models. Our co-design analysis shows that the performance of the algorithms for a convolutional layer depends on each layer's dimensions and hardware parameters. To predict the best algorithm for each convolutional layer, in *Paper II*, we train an algorithm selection model using a random forest that takes each layer's dimensions and the hardware's microarchitectural parameters as inputs and outputs algorithm with the lowest execution time for each layer. *Paper II* addresses the second problem statement and addresses the research questions 5, 6, 7, and 8. The main contributions of this paper are:

- Our performance comparison with 512-bit vector length and 1MB L2 cache shows that Winograd is the best choice for layers with 3$\times$3 kernel size, whereas im2col+GEMM works best for layers with skinnier matrices. On the other hand, Direct works best for layers with high input dimensions but low input/output channels.

- Our co-design study shows that Winograd works best with smaller vector lengths while Direct excels with longer vector lengths. im2col+GEMM takes benefit from larger caches except for layers with extremely skinny matrices. Direct takes maximum benefit out of larger caches with longer vector lengths.

- We train a Random Forest classifier using 5-fold validation with shuffling, achieving a prediction accuracy of 92.8%. Selecting the optimal algorithm for each layer can boost performance by more than 2$\times$ compared to using a single algorithm. Our performance-to-area trade-offs for both single and multiple model instances demonstrate that carefully selecting the algorithm for each layer enables higher performance in a reduced area.

The rest of the thesis is organized as follows. In Chapter 2, a summary of each paper is presented. Finally, Chapter 3 concludes the thesis, and discusses some possible future research directions.

Chapter 2

Summary of the Papers

2.1 Paper I - Summary

As an alternative to off-chip accelerators, long vector length agnostic architectures such as RISC-VV and ARM-SVE can offer high performance for machine learning workloads and higher energy efficiency, making CPUs suitable for CNN inference. However, manual transformations and optimizations are key to achieving efficient algorithmic implementations and maximizing performance from these long vector architectures. Furthermore, tuning hardware parameters such as vector lengths and L2 cache size to the requirement of algorithmic optimizations can significantly impact performance. Therefore, it is important to have a joint exploration of the design space of vector architectures and optimization space of CNNs to have design points for a high performing vector architecture for CNNs. While existing works focus either on optimizing convolutional algorithms [20, 29–31] or tuning hardware parameters [32–34], they miss the trade-offs between both. In Paper I, we bridge this gap and perform a co-design study between algorithmic optimizations and micro-architectural parameter choices, aiming to give guidance to programmers, compiler developers, and hardware designers.

Convolutional layers are the most time-consuming of CNNs. In Darknet, this layer is implemented using the im2col+GEMM algorithm with GEMM consuming 93.4% of the computation time when compiled with clang on the A64FX system. For experiments, we use the YOLOv3 (object detection) and VGG16(image classification) network models, from the Darknet framework on a 768×576 pixels input image. We use the gem5 simulator [32, 54] to assess the impact of integrating vector units tightly to the core in the case of ARM-SVE and as a decoupled vector architecture attached to the L2 cache in the case of RISC-VV.

To optimize the convolutional layer, we optimize all kernels in the Darknet framework using intrinsic instructions of the respective ISAs, mainly focusing on the GEMM kernel. We optimize the GEMM kernel using two approaches: 3 loops and 6 loops. We apply the following optimizations to the 3-loop implementation: i) vectorization with intrinsic instructions ii) contiguous memory loads/stores to/from vector registers, iii) loop reorder, and iv) loop unrolling. We apply the following BLIS-like optimizations to the 6-loop implementation:

i) loop reorder, ii) matrix packing, iii) block size tuning, iv) loop unrolling, v) prefetching, and vi) vectorization using intrinsic instructions. We simulate the first 4 convolutional layers of the YOLOv3 network on RISC-VV@gem5 with a 512-bit vector length, 1MB of L2 cache and 8 vector lanes, on a single core with different block sizes. We observed that the optimal block size for the 6-loop implementation is 16 × 512 × 128, which differs ∼2% with 3-loop implementation, a difference that is not significant in the simulated environment. This is because VPU is directly attached to the L2 cache, not taking any benefit from bringing packed matrices in the L1 cache beforehand. Additionally, RISC-VV@gem5 does not support software prefetching, which is a desired feature in the 6-loops implementation. On ARM-SVE@gem5, the 6-loop implementation achieves a 15% performance gain over the 3-loop approach, leveraging cache usage despite the lack of software prefetching. On the other hand, we observe a 2× performance improvement with ARM-SVE@A64FX using 6-loop implementation, where the 6-loop implementation is able to take advantage of the caches and prefetching.

Further, we tune the hardware parameters using the optimized 3-loop implementation in the case of RISC-VV@gem5 with the first 20 layers of the YOLOv3 model. Increasing the vector lengths from 512-bit to 16384-bit showcases the performance saturates beyond the 8192-bit vector length. Increasing the L2 cache size from 1MB to 256MB shows a 1.5× - 1.9× performance gain for different vector lengths Our study reveals that larger L2 caches are beneficial with longer vector lengths, but the performance gains of very long vector lengths are limited, as with 256MB L2 cache performance improves by ∼5% from 8192-bit to 16384-bit vector lengths. Additionally, we tune the hardware parameters using 6-loops optimization on ARM-SVE@gem5 which validates the observations made by RISC-VV@gem5 that our optimized kernels can benefit from longer vectors and larger cache sizes.

As an alternative to im2col+GEMM, we have optimized the Winograd implementation of the convolutional layer with 3×3 kernel size from the NNPACK package. For vectorizing the Winograd algorithm to utilize long vector lengths in a VLA way, Paper I proposes an inter-tile parallelism across the input/output channels by using an 8×8 tile from each channel on ARM-SVE. Using 4 input/output channels with one row of 8×8 tiles from each channel as shown in Figure 2.1, we can utilize two 512-bit vector registers. To utilize longer vector lengths, we increase the number of input/output channels accordingly, e.g. 16 channels for 2048-bit vector registers. For vectorizing the tuple multiplications in a VLA way on ARM-SVE, we increase the block size from 3 to 16 with 4 elements in each block i.e., utilizing a maximum of 2048-bit vector lengths. For 512-bit vector lengths, an additional performance of 1.5× is achieved on top of im2col+GEMM.

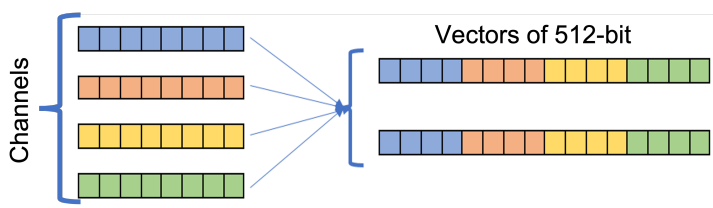

Figure 2.1: Inter-tile parallelism in Winograd

Increasing the vector length from 512 to 2048 bits while keeping the L2 cache at 1MB, we observe a 1.4× performance improvement for both network models. Increasing the L2 cache size from 1MB to 256MB improves performance by 1.75× and 1.4× for all vector lengths in YOLOv3 and VGG16 respectively. All layers in VGG16 use Winograd, which has smaller cache requirements compared to im2col+GEMM, whereas several layers in YOLOv3 invoke im2col+GEMM. This emphasizes that our Winograd implementation does not have high cache requirements.

2.2 Paper II - Summary

CNN models are built upon a series of consecutive layers, where each layer is distinct based on dimensions such as input, output, kernel's height and width, input and output channels, and stride. Multiple algorithms such as im2col+GEMM, Winograd, Direct, and FFT can be used to implement the convolutional layer. These algorithms can demonstrate varying performances for each layer as they have different computational complexities and memory footprints. Moreover, cache memory can impact the performance of these convolutional algorithms, as certain algorithms such as im2col+GEMM increase the memory footprint of a convolutional layer. Additionally, these convolutional algorithms need several optimizations to utilize the SIMD unit efficiently.

Model serving frameworks create replicas of a single model and distribute incoming requests across these replicas by maintaining load balancing. However, concurrent execution competes for cache resources, making the convolutional algorithms dependent on co-running inference tasks. Therefore, it is important to have a mutual impact of convolutional algorithms and hardware parameters to have efficient design points for future vector CPUs for CNN model serving. Previous studies [35–38] have focused on optimizing the performance of specific algorithms on vector architectures and presenting comparative analyses with state-of-the-art libraries for different layers of network models on vector processors. We identify the absence of a joint study of convolutional algorithms and hardware parameters as a missed opportunity. In paper II, we perform a co-design study of three distinct convolutional algorithms, Direct, im2col+GEMM, and Winograd for implementing convolutions on RISC-V based vector architecture. Since large kernel sizes are not common in modern CNNs, we do not consider the FFT algorithm in this work.

We use two variants of the optimized im2col+GEMM algorithm from Paper I. Additionally, we use the optimized Winograd algorithm from our workshop paper [55] on RISC-VV. For the Direct algorithm, we manually vectorize and optimize it with NHWC memory layout (where N refers to batch size, H refers to input height, W refers to input width and C refers to input channels) on RISC-VV using intrinsic instructions of the respective ISA. We evaluate each convolutional layer of the YOLOv3 and VGG16 network model on RISC-VV on a 768×576 pixels input image using the gem5 simulator [56].

We evaluate each convolutional layer of both network models with a 512-bit vector length and 1MB of L2 cache size. Our study showcases that the Direct algorithm is better when input/output dimensions are high, but input/output channels are low. On the other hand, im2col+GEMM with 6 loops prevails

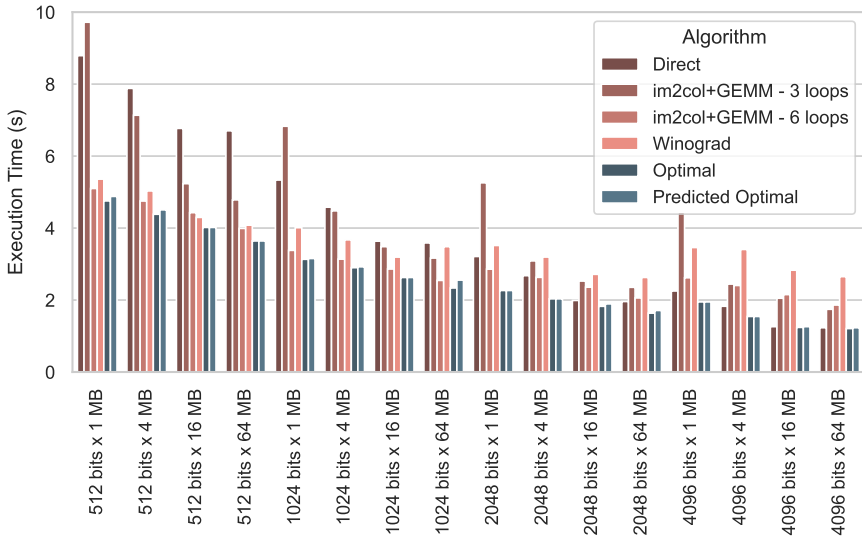

Figure 2.2: Execution time of VGG16 for different vector lengths and L2 cache sizes, when a single algorithm is used for all layers (*Direct*, *im2col+GEMM - 3 loops*, *im2col+GEMM - 6 loops*,*Winograd*), compared against using the *Optimal* algorithm per layer, and using our algorithm selection model to predict the optimal algorithm per layer (*Predicted Optimal*).

and performs better in the case of skinny matrices, i.e., when input/output dimensions are low, but input/output channels are high. Winograd shows better or comparable performance for most layers, except for layers with high input channels. Further, increasing vector lengths from 512-bit to 4096-bit demonstrates that the Direct algorithm shows the maximum scalability of 2.4×–5.8× with longer vector lengths and outperforms the other algorithms. The im2col+GEMM algorithm offers better performance for vector lengths higher than 1024-bit for the skinny matrices. Additionally, increasing the L2 cache from 1MB to 64MB showcases the limited scalability of the Winograd algorithm due to its fixed tile size, which does not utilize the larger caches. Both variants of im2col+GEMM show limited scalability beyond 16MB of L2 cache for extremely skinny matrices. On the other hand, the Direct algorithm benefits the most from larger caches.

Our evaluation shows that the selection of the convolutional algorithms depends upon the layer's dimensions and hardware microarchitectural parameters. Consequently, it is important to select an optimal algorithm for each convolutional layer based upon the co-design study. In Paper II, we propose a random forest classifier that achieves 92.8% prediction accuracy by considering both convolutional layer dimensions and hardware configuration parameters. Selecting the optimal algorithm improves the execution time by up to 1.85× compared to always using the Direct algorithm and up to 1.73× over using the 6-loop implementation of im2col+GEMM in the case of VGG16 as shown in Figure 2.2. For YOLOv3, selecting the optimal algorithm improves the execution time by 1.33× and 2.11× over always using the Direct and 6-loop im-

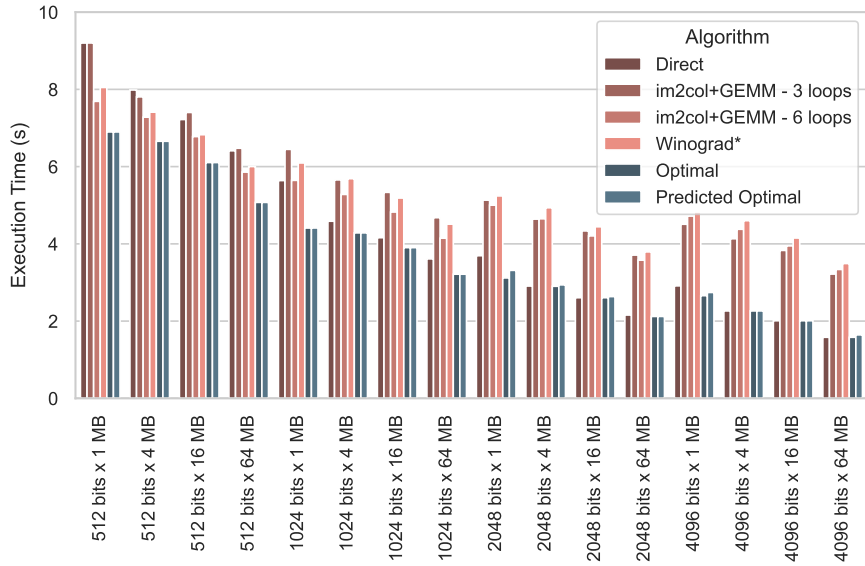

Figure 2.3: Execution time of YOLOv3 (first 15 layers) for different vector lengths and L2 cache sizes, when a single algorithm is used for all layers (*Direct*, *im2col+GEMM - 3 loops*, *im2col+GEMM - 6 loops*, *Winograd**-uses im2col+GEMM for some layers), compared against using the *Optimal* algorithm per layer, and using our algorithm selection model to predict the optimal algorithm per layer (*Predicted Optimal*).

plementation of im2col+GEMM algorithms, respectively as shown in Figure 2.3. Moreover, our performance-area tradeoffs show that algorithm selection allows for better performance in less area, compared to using a single algorithm for each layer.

Chapter 3

Conclusions and Future Work

In this thesis, we present a hardware and software co-design study of CNNs inference with three distinct algorithms, Direct, im2col+GEMM, and Winograd, on modern long vector architectures by considering the architecture's hardware parameters tuning.

This thesis makes the following contributions: Paper 1 concludes that optimizing the algorithms according to the underlying vector architecture is important because not all types of vector architectures benefit from all the optimizations. Manual optimizations on the vector architecture are crucial, as a performance improvement of 3×-6× has been observed compared to auto-vectorization. Paper 1 additionally concludes that tuning the microarchitectural parameters with longer vector lengths, bigger caches, and more vector lanes can improve the performance by ∼5×. However, the performance gain from very long vectors is limited and additional vector lanes only benefit long vector lengths by hiding the startup overhead and pipeline latency. Moreover, our optimized Winograd algorithm with inter-tile parallelism further improves the performance by 1.35× and 1.5× for YOLOv3 and VGG16 network models respectively, while having a lower cache requirement. Paper 2 concludes that the selection of an algorithm per convolutional layer highly depends on the convolutional layer parameters and hardware microarchitectural parameters such as vector lengths and L2 cache sizes. All the algorithms benefit ∼2× with longer vector lengths (2048-bit) compared to 512-bits. Both algorithms im2col+GEMM and Direct benefit about 1.5× from a larger L2 cache (64MB compared to 1MB), but the Direct can have higher speedups if the vector length is large (4096 bits). Moreover, our study shows that there is no single algorithm that fits all the convolutional layers. Therefore, to select the best algorithm, Paper 2 presents our Random Forest classifier, resulting in an average of 92.8% prediction accuracy, with inference time predictions showing at most 10% relative errors. Furthermore, algorithm selection can boost performance more than 2× compared to using a single algorithm and results in more efficient chips for CNN model serving.

So far, we have focused on the optimization space of algorithms for convolutions and the design space of vector architectures. There are various

opportunities to extend this work. One promising direction is toward vision transformers (ViTs), a deep learning architecture that applies to transformer models, designed for natural level processing, object detection, image classifications, etc. These network models are built with self-attention and feedforward layers, where matrix multiplication is the main kernel, contributing a substantial part of total inference time. However, optimizing ViTs on vector architectures presents several challenges. First, many matrices are skinny and irregular, making it challenging to utilize long vector lengths and optimize them effectively. Second, data movement is substantial as each self-attention layer involves two matrix-matrix multiplications along with one softmax kernel. Therefore, mechanisms like data reuse and fusion are proposed [57] to reduce memory accesses and improve performance.

Bibliography

[1] K. Li, W. Ma, U. Sajid, Y. Wu, and G. Wang, "Object detection with convolutional neural networks," *CoRR*, vol. abs/1912.01844, 2019. [Online]. Available: http://arxiv.org/abs/1912.01844

[2] M. M. Lopez and J. Kalita, "Deep learning applied to NLP," *CoRR*, vol. abs/1703.03091, 2017. [Online]. Available: http://arxiv.org/abs/1703.03091

[3] D. Palaz, M. Magimai.-Doss, and R. Collobert, "Convolutional neural networks-based continuous speech recognition using raw speech signal," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 4295–4299.

[4] Y. Hu, Y. Liu, and Z. Liu, "A survey on convolutional neural network accelerators: GPU, FPGA and ASIC," in *2022 14th International Conference on Computer Research and Development (ICCRD)*, 2022, pp. 100–107.

[5] "Whitepaper GPU-based deep learning inference : A performance and power analysis," 2015.

[6] L. Wang, Z. Chen, Y. Liu, Y. Wang, L. Zheng, M. Li, and Y. Wang, "A unified optimization approach for cnn model inference on integrated GPUs," 2019.

[7] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGAs: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[8] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating cnn inference on asics: A survey," *Journal of Systems Architecture*, vol. 113, p. 101887, 2021.

[9] M. Abadi, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2016.

[10] Pytorch. Pytorch GPU. [Online]. Available: https://www.run.ai/guides/gpu-deep-learning/pytorch-gpu/

[11] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *arXiv*, 2018.

[12] J. Park, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," 2018.

[13] S. Mittal, P. Rajput, and S. Subramoney, "A survey of deep learning on cpus: Opportunities and co-optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2021.

[14] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing {CNN} model inference on {CPUs}," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 1025–1040.

[15] E. Georganas and Kalamkar, "Tensor processing primitives: a programming abstraction for efficiency and portability in deep learning workloads," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[16] R. Li and Xu, "Analytical characterization and design space exploration for optimization of cnns," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 928–942.

[17] T. P. Morgan, "IBM BETS BIG ON NATIVE INFERENCE WITH BIG IRON," *The Next Platform*, August 23, 2021. [Online]. Available: "https://www.nextplatform.com/2021/08/23/ibm-bets-big-on-native-inference-with-big-iron/"

[18] (2020) V for vector: software exploration of the vector extension of RISC-V. [Online]. Available: https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/

[19] N. Stephens, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[20] Dan Andrei Iliescu, Francesco Petrogalli. Arm scalable vector extension and application to machine learning. [Online]. Available: https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/arm-scalable-vector-extensions-and-application-to-machine-learning

[21] European Processor Initiative. (2019) V for vector: software exploration of the vector extension of RISC-V. [Online]. Available: https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/

[22] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, aug 2019.

[23] S. Mittal and S. Vaishay, "A survey of techniques for optimizing deep learning on GPUs," *Journal of Systems Architecture*, vol. 99, p. 101635, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762119302656

[24] S. A. Alam, A. Anderson, B. Barabasz, and D. Gregg, "Winograd convolution for deep neural networks: Efficient point selection," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 6, Dec. 2022. [Online]. Available: https://doi.org/10.1145/3524069

[25] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," 2014.

[26] Meta Platforms, Inc. Building meta's genai infrastructure. [Online]. Available: https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/

[27] Nvidia Corporation. Triton inference server: Architecture: Concurrent model execution. [Online]. Available: https://docs.nvidia.com/deeplearning/triton-inference-server/archives/triton_inference_server_1150/user-guide/docs/architecture.html#concurrent-model-execution

[28] BentoML. Bentoml docs: Concurrency. [Online]. Available: https://docs.bentoml.com/en/latest/guides/concurrency.html

[29] M. Sahaya Loui, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. Reddi, and A. Joshi, "Towards deep learning using tensorflow lite on RISC-V," 06 2019.

[30] M. Cococcioni, F. Rossi, and E. Ruffaldi, "Fast deep neural networks for image processing using posits and ARM scalable vector extension," *Journal of Real-Time Image Processing*, vol. 17, 06 2020.

[31] ARM. ARM Compute Library. [Online]. Available: https://github.com/ARM-software/ComputeLibrary

[32] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A risc-v simulator and benchmark suite for designing and evaluating vector architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, nov 2020. [Online]. Available: https://doi.org/10.1145/3422667

[33] Y. Kodama, T. Odajima, M. Matsuda, M. Tsuji, J. Lee, and M. Sato, "Preliminary performance evaluation of application kernels using arm sve with multiple vector lengths," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 677–684.

[34] A. Poenaru and S. McIntosh-Smith, "The effects of wide vector operations on processor caches," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 531–539.

[35] P. Wang, W. Yang, J. Fang, D. Dong, C. Huang, P. Zhang, T. Tang, and Z. Wang, "Optimizing direct convolutions on ARM multi-cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023.

[36] A. d. L. Santana, A. Armejach, and M. Casas, "Efficient direct convolution using long simd instructions," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 342–353.

[37] S. Wang, J. Zhu, Q. Wang, C. He, and T. T. Ye, "Customized instruction on risc-v for winograd-based convolution acceleration," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 65–68.

[38] M. F. Dolz, H. Martínez, A. Castelló, P. Alonso-Jordá, and E. S. Quintana-Ortí, "Efficient and portable winograd convolutions for multi-core processors," *The Journal of Supercomputing*, pp. 1–22, 2023.

[39] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: Past, present and future," in *Proceedings of the 12th International Conference on Supercomputing*, ser. ICS '98. NY, USA: ACM, 1998, p. 425–432.

[40] Intel. (2020) Instruction set extensions and future features programming reference. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[41] ARM, "ARM Neon Programmer's Guide," 2013. [Online]. Available: https://documentation-service.arm.com/static/5f731b591b758617cd95559c?token=

[42] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[43] M. Dukhan. (2016) NNPACK. https://github.com/Maratyszcza/NNPACK.

[44] V. Kelefouras and G. Keramidas, "Design and implementation of 2d convolution on x86/x64 processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3800–3815, 2022.

[45] G. Alaejos, A. Castelló, H. Martínez, P. Alonso-Jordá, F. D. Igual, and E. S. Quintana-Ortí, "Micro-kernels for portable and efficient matrix multiplication in deep learning," *The Journal of Supercomputing*, vol. 79, no. 7, pp. 8124–8147, 2023.

[46] D. Li, D. Huang, Z. Chen, and Y. Lu, "Optimizing massively parallel winograd convolution on arm processor," in *Proceedings of the 50th International Conference on Parallel Processing*, ser. ICPP '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3472496

[47] M. F. Dolz, H. Martínez, P. Alonso, and E. S. Quintana-Ortí, "Convolution operators for deep learning inference on the fujitsu a64fx processor," in *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2022, pp. 1–10.

[48] M. Jordà, P. Valero-Lara, and A. J. Peña, "Performance evaluation of cudnn convolution algorithms on nvidia volta gpus," *IEEE Access*, vol. 7, pp. 70 461–70 473, 2019.

[49] R. Xu, S. Ma, and Y. Guo, "Performance analysis of different convolution algorithms in GPU environment," in *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2018, pp. 1–10.

[50] M. F. Dolz, S. Barrachina Mir, H. Martinez, A. Castelló, A. Maciá, G. Fabregat, and A. Tomás, "Performance–energy trade-offs of deep learning convolution algorithms on ARM processors," *The Journal of Supercomputing*, vol. 79, 01 2023.

[51] A. Zlateski, Z. Jia, K. Li, and F. Durand, "FFT convolutions are faster than winograd on modern CPUs, here is why," *ArXiv*, vol. abs/1809.07851, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52339177

[52] M. F. Dolz, H. Martınez, P. Alonso-Jordá, A. Castelló, and E. S. Quintana-Ortı, "Parallel and vectorised winograd convolutions for multi-core processors."

[53] M. F. Dolz, A. Castelló, and E. S. Quintana-Ortí, "Towards portable realizations of winograd-based convolution with vector intrinsics and openmp," in *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2022, pp. 39–46.

[54] Gem5, "Gem5." [Online]. Available: https://gem5.googlesource.com/public/gem5

[55] S. R. Gupta, N. Papadopoulou, and M. Pericàs, "Challenges and opportunities in the co-design of convolutions and RISC-V Vector Processors," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1550–1556.

[56] RISC-V Vector. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases

[57] X. Fu, W. Yang, D. Dong, and X. Su, "Optimizing attention by exploiting data reuse on ARM multi-core CPUs," in *Proceedings of the 38th ACM International Conference on Supercomputing*, ser. ICS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 137–149. [Online]. Available: https://doi.org/10.1145/3650200.3656620

Chapter 4

Paper I

Accelerating CNN inference on long vector architectures via co-design

Sonia Rani Gupta, Nikela Papadopoulou, and Miquel Pericàs

Preprint from

International Symposium on Parallel and Distributed Processing (IPDPS), 2023

Accelerating CNN inference on long vector architectures via co-design

Sonia Rani Gupta
Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
Email: soniar@chalmers.se

Nikela Papadopoulou
Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
Email: nikela@chalmers.se

Miquel Pericàs
Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
Email: miquelp@chalmers.se

***Abstract*—CPU-based inference can be deployed as an alternative to off-chip accelerators. In this context, emerging vector architectures are a promising option, owing to their high efficiency. Yet the large design space of convolutional algorithms and hardware implementations makes the selection of design options challenging. In this paper, we present our ongoing research into co-designing future vector architectures for CPU-based Convolutional Neural Networks (CNN) inference focusing on the im2col+GEMM and Winograd kernels. Using the Gem5 simulator we explore the impact of several hardware microarchitectural features including (i) vector lanes, (ii) vector lengths, (iii) cache sizes, and (iv) options for integrating the vector unit into the CPU pipeline. In the context of im2col+GEMM, we study the impact of several BLIS-like algorithmic optimizations such as (1) utilization of vector registers, (2) loop unrolling, (3) loop reorder, (4) manual vectorization, (5) prefetching, and (6) packing of matrices, on the RISC-V Vector Extension and ARM-SVE ISAs. We use the YOLOv3 and VGG16 network models for our evaluation. Our co-design study shows that BLIS-like optimizations are not beneficial to all types of vector microarchitectures. We additionally demonstrate that longer vector lengths (of at least 8192 bits) and larger caches (of 256MB) can boost performance by 5×, with our optimized CNN kernels, compared to a vector length of 512-bit and 1MB of L2 cache. In the context of Winograd, we present our novel approach of inter-tile parallelization across the input/output channels by using 8×8 tiles per channel to vectorize the algorithm on vector length agnostic (VLA) architectures. Our method exploits longer vector lengths and offers high memory reuse, resulting in performance improvement of up to 2.4× for non-strided convolutional layers with 3×3 kernel size, compared to our optimized im2col+GEMM approach on the Fujitsu A64FX processor. Our co-design study furthermore reveals that Winograd requires smaller cache sizes (up to 64MB) compared to im2col+GEMM.**

***Index Terms*—CNNs, GEMM, Winograd, long vector architectures, vector-length agnostic ISAs, co-design, optimizations**

I. Introduction

Inference via Convolutional Neural Networks (CNNs) is used in many Artificial Intelligence applications such as object detection [1], natural language processing [2] and speech recognition [3]. Most CNN-based object detection network models work with a tight response-time limit and have high and increasing computation costs [4]–[6]. Additionally, these models often operate under tight power constraints, e.g. battery power in embedded systems [7], or power caps in datacenters [8]. Therefore, highly accurate real-time CNNs require highly optimized kernels, running on energy-efficient architectures with large computational capacity.

The popular approach for CNN inference, adopted by many frameworks [6], [9], [10] is to offload the compute-intensive kernels to GPUs [11]–[13]. Specialized neural accelerators also exist [14], [15], but their integration in the general-purpose computing stack is challenging. Nevertheless, many use cases require availability, low-latency, or portability [16]–[18], and therefore benefit from executing deep neural networks (DNNs) on tightly integrated systems. Consequently, many works target software optimization of CNN inference on CPUs [18]–[20], while CPU vendors increasingly add DNN capabilities to processors [21].

In this aspect, vector processors play a leading role [22]. Contemporary vector architectures, such as ARM-SVE [23] and the RISC-V Vector extension (RISC-VV) [24] specify a maximum length of vector registers and allow the usage of different vector lengths. These vector-length agnostic (VLA) Instruction Set Architecture (ISAs) facilitate code portability across iterations of the same machine with different vector lengths.

The effectiveness of vector processors depends on algorithmic optimizations and the hardware design. First, given the limited compiler's ability to perform transformations during auto-vectorization [25], manual transformations and optimizations to expose the available SIMD parallelism to the vector processing units are key to achieving high performance on vector architectures. Second, modern architectures can combine very long vector units, more on-chip vector parallelism and large caches. Tuning the micro-architectural parameters to the requirements of the vectorized and optimized kernels is integral to the design of high-performing, efficient vector architectures.

Existing work on CNN inference on vector architectures focuses either on applying algorithmic optimizations [26]–[29] or on tuning the hardware micro-architectural parameters [30]–[32]. We identify the absence of a combined study as a missed opportunity to uncover algorithmic and architectural trade-offs in the performance of CNN kernels running on vector architectures. In this work, we bridge this gap with a co-design study that performs a joint exploration of the design space of vector architectures and the optimization space of CNNs, aiming to provide guidance to programmers, hardware designers, and compiler developers.

This paper studies the interplay between algorithmic optimizations and micro-architectural parameter choices, demonstrating the trade-offs in co-designing CNNs and vector architectures. For our co-design study, we vectorize all the kernels of the convolutional layer from the Darknet framework [6]

on the RISC-VV and ARM-SVE architectures, using high-level intrinsics of the respective ISAs. We then optimize GEMM, the most time-consuming kernel, using various BLIS-like [33] techniques to reduce the pressure on the memory-subsystem, enforce contiguous memory accesses, and maximize the utilization of vector registers. We additionally optimize the Winograd algorithm from NNPACK [34], with VLA vectorization on ARM-SVE, proposing a novel, inter-tile parallelism scheme. We then use the gem5 [35] simulator to assess the impact of tuning hardware parameters such as vector lengths, vector lanes and L2 cache sizes, on the optimized kernels. We consequently also assess the impact of integrating vector units tightly to the core, in the case of ARM-SVE, or as a decoupled vector architecture, in the case of RISC-VV. Finally, we evaluate the performance of Winograd as an algorithmic replacement for im2col+GEMM.

In summary, we make the following contributions:

1) We demonstrate that not all algorithmic optimizations are beneficial to all different vector architectures, due to traits of their micro-architectural design. To the best of our knowledge, this is the first work that shows the impact of different algorithmic optimizations with CNN kernels on the ARM-SVE and RISC-VV ISAs.
2) We characterize the impact of hardware parameters on convolutional layers with im2col+GEMM, showing that longer vectors can improve the performance by up to 2.5×, and larger caches can further improve performance by up to 1.9× (i.e., a total of almost 5×), when compared to a 512-bit long vector architecture with 1MB of L2 cache.
3) We present a novel, vectorized implementation of Winograd with ARM-SVE in a VLA manner, offering up to 1.35× and 1.5× higher performance with the YOLOv3 and VGG16 network models respectively, compared to im2col+GEMM, on a single core of A64FX. To the best of our knowledge, this is the first implementation of Winograd utilizing long vectors (up to 2048 bits). Moreover, our co-design study on ARM-SVE shows that Winograd is less sensitive to the L2 cache size compared to im2col+GEMM.

The rest of this paper is organized as follows. Section II offers background on vector architectures and CNNs. Section III presents our experimental platforms and setup. Section IV describes the algorithmic transformations and optimizations on the most-time consuming kernels of the convolutional layer, namely im2col+gemm and Winograd. Section V details the hardware parameters we consider in our co-design study. Section VI presents our co-design study on the RISC-VV and ARM-SVE architectures. Section VII evaluates the Winograd kernel and Section IX concludes the paper.

II. Background

A. Vector Architectures

Although long vector lengths were used in supercomputers in the past [36], and short vectors later became popular in general-purpose architectures [37], [38], the high energy efficiency and scalable vector length of vector architectures have led to renewed interest in High-Performance Computing. While SIMD instruction set architectures with a fixed short vector length are available and commonly used for general purpose computing, introducing longer vector lengths requires a new ISA extension, limiting portability. To overcome this limitation, modern architectures such as RISC-VV [39] and ARM-SVE [23] offer vector length agnostic (VLA) ISAs that are portable across different hardware vector lengths.

a) RISC-V Vector Extension (RISC-VV): This is the vector extension of the RISC-V Architecture, with 32 vector registers and a maximum supported vector length (MVL) of 16384 bits. Different vector lengths (*vlen*) in powers of two, not exceeding the MVL (*maximum vector length*), can be used. A vector instruction `vsetvl` determines the granted vector length (*gvl*) at runtime, using the requested vector length (*rvl*) in elements and the element width in bits (*sew*) as input. RISC-VV also supports strided-access, gather-load and scatter-store vector operations.

b) ARM Scalable Vector Extension (ARM-SVE): This is the vector extension of the ARMv8 architecture. The ARM-SVE ISA operates on 32 vector registers and 16 predicate registers. The supported MVL is 2048 bits, allowing to use different vector lengths at runtime, from 128-bit to 2048-bit in increments of 128-bits. Predicate registers are used for per-lane predication, where elements with active lanes get processed and inactive lanes either update the destination or leave the destination unchanged. For the scalar loop tail, ARM-SVE uses loop predication by masking out vector elements and by processing partial vectors. ARM-SVE also provides gather-load and scatter-store vector instructions.

B. Convolutional network models

Convolutional neural networks are implemented in multiple deep learning frameworks. In this work, we focus on Darknet [40], an open-source neural network framework written in C and CUDA. It supports many pre-trained convolutional network models for inference in various applications, such as object detection and image classification. These network models consist of different types of layers, but the computationally dominant layer is the convolutional layer. In Darknet, a convolutional layer is built from the functions `GEMM`, `im2col`, `fill_cpu`, `copy_cpu`, `normalize_cpu`, `add_bias`, `scale_bias` and `activate_array`.

a) CNNs for object detection and image classification: A popular CNN for object detection is YOLOv3, which features 107 layers of five different types, out of which 75 layers are convolutional. A variant for the same task is YOLOv3-tiny, which features 23 layers, out of which 13 are convolutional. VGG16 is an image classification CNN. VGG16 includes 25 layers, out of which 13 are convolutional and 3 are fully-connected layers. The fully connected layers also use compute intensive kernels similar to convolutional layers.

b) Execution time breakdown for CNN inference: We profile the execution time of different kernels in the YOLOv3

TABLE I: Hardware Platforms

	RISC-VV @gem5	ARM-SVE @gem5	A64FX
ISA	RISC-VV v0.8	ARM v8.2+sve	ARM v8.2+sve
Processor	in-order	in-order	out-of-order
Clock Rate	2GHz	2GHz	2GHz
L1 Cache size	64kB, 4-way	64kB,4-way	64kB,4-way
L2 Cache size	1MB, 8-way	1MB, 8way	8-MB, 16-way
Cache line size	64B	64B	256B
Prefetching	No	No	Yes
Vector Length	upto 16384-bit	up to 2048-bit	512-bit
Vector Lanes	upto 8	proportional to vector length	not configurable

network model, compiled with `clang` on the *A64FX* system (see Section III for details) and collect measurements using Linux `perf`. Approximately 92% of the total execution time is spent on computation for inference, while the remaining 8% is used for setting up the network model. We exclude the time for setup, as it occurs only once, and calculate the percentage of time spent on each kernel with respect to the total computation time. The convolutional layer dominates execution, with `GEMM` consuming 93.4% of the computation time.

c) Convolutional layer implementations: Our profiling results show that the convolutional layer is the main building block of CNN network models. In Darknet, this layer is implemented using the im2col+GEMM algorithm, which is also the dominant kernel. We focus on the optimization of the generic im2col+GEMM algorithm, however, a convolutional layer can be implemented with multiple algorithms, as no "one-size-fits-all" strategy exists [41]: *Winograd* [42] works best with convolutional layers with 3×3 or 5×5 kernel sizes [43], *FFT* works best for layers with large kernel sizes, while the *Direct* algorithm is better for 1×1 kernels. We therefore also optimize the Winograd algorithm of the NNPACK [34] package implementation, as in CNN-based network models most of the network models have convolutional layers with kernel sizes of 1×1, 3×3 or 5×5 [44].

III. Methodology

A. Hardware platforms

Our experimental analysis focuses on the RISC-VV and ARM-SVE architectures. For the exploration of hardware parameters, we simulate both architectures with gem5 [35], a cycle-accurate simulator that models the core pipeline, providing accurate timing predictions. For ARM-SVE, we use the Fujitsu A64FX processor that implements the ARMv8-SVE architecture, to evaluate our algorithmic optimizations.

The specifics of the hardware platforms used for our experiments are described in Table I. We note that A64FX has 2 SIMD units, and the vector lengths are not reconfigurable, as this is an actual processor. We use a RISC-V fork of gem5 [30] and the public version of the gem5 simulator [45] with support for modeling vector architectures, for RISC-VV and ARM-SVE, respectively, in system call emulation (SE) mode. We configure gem5 with the in-order "MinorCPU" CPU model, with a frequency of 2GHz for the CPU and vector processor unit (VPU). The memory subsystem is configured with two levels of the data cache. We note that in RISC-VV@gem5, the VPU is connected to the L2 cache. A small VectorCache buffer of 2KB is used, through which the VPU reads and writes data from/to the L2 cache. However, on ARM-SVE@gem5, data for vector registers is accessed through the L1 cache itself.

B. Experimental setup

We evaluate the YOLOv3 network models from the Darknet framework on a 768 × 576 pixels input image. To compile the models, we use the EPI fork of the LLVM `clang` [46] cross-compiler v12.0.0 for RISC-VV, LLVM `armclang` v20.3 [47] for ARM-SVE@A64FX, and *GCC* cross-compiler version 10.2 for ARM-SVE@gem5. For both RISC-VV and ARM-SVE, we use the `-O3` optimization flag. To collect baseline results, we use the `-fno-vectorize` compiler flag in both compilers. Note that the baseline implementation of the network models in Darknet does not include any manual vectorization. The versions of Darknet with our vectorized and optimized kernel implementations for ARM-SVE and RISC-VV are open-source and publicly available [1] [2].

To analyze the impact of the vector lengths, we vary the vector lengths in both simulated architectures from 512 bits up to 2048 bits on ARM-SVE and up to 16384 bits on RISC-VV, in powers of 2. To analyze the impact of on-chip parallelism on RISC-VV, we vary the number of vector lanes from 2 up to 8. To analyze the impact of cache parameters, we increase the L2 cache size on both simulated architectures from 1MB up to 256MB. We calculate the L2 cache latency using the latency of AMD Zen2 L2 [48] (12 cycles @ 7nm tech) and extrapolating it to a cache size of 1MB, using the CACTI tool [49], resulting in a latency of 12 cycles.

To collect time measurements, we perform 100 repetitions for all experiments on A64FX, ensuring that the 95% confidence interval of the mean falls within 5% of the mean.

IV. Algorithmic Optimizations

In this section, we focus on the algorithmic optimizations for im2col+GEMM for the convolutional layer. We additionally describe the optimization of the Winograd implementation of convolutional layers.

A. im2col+GEMM optimizations

To maximize the attainable performance, we begin by vectorizing all kernels of the convolutional layer in Darknet with low-level intrinsic instructions of the respective ISAs on each of our experimental platforms. However, as discussed in Section II, `GEMM` is the most time consuming kernel, and aside from vectorization, manual optimizations are necessary to extract the maximum parallelism out of im2col+GEMM.

Assuming a convolutional layer with a $k \times k$ kernel size, on an input image of dimensions $h \times w \times c$, where h, w, c are the height, width, and number of channels respectively, for n number of filters, GEMM takes as input a weight matrix $M \times K$, and an input matrix $K \times N$, where $M = n$, $K = k \times k \times c$, and $N = h \times w$.

[1] https://github.com/chalmers-hart/Darknet-ARM-SVE.git
[2] https://github.com/chalmers-hart/Darknet-RISCVV.git

```
1: i ← 0 , j ← 0, k ← 0
2: for i ← 0, i < M, i++ do
3:    for k ← 0, k < K, k++ do
4:       tmp = alpha * A[i,k]
5:       for j ← 0, j < N, j+= 1 do
6:          C[i,j] += tmp * B[k,j]
7:       end for
8:    end for
9: end for
```

Fig. 1: Naive implementation of GEMM

```
1: i ← 0 , j ← 0, k ← 0
2: long int gvl;
3: for j ← 0, j < N do
4:    gvl ← vsetvl(N − j) //compute 'granted vector length'
5:    for i ← 0, i < M, i+= U do //U is unrollfactor
6:       VC[i : i + U] ← C[i : i + U, j : j + gvl]
7:       for k ← 0, k < K, k++ do
8:          VB ← B[k, j : j + gvl]
9:          for it ← 0, it < U, it++ do
10:            tmp = alpha × A[it, k]
11:            Vtmp ← tmp //broadcast
12:            VC[it] ← vfmacc (VC[it],Vtmp, VB, gvl)
13:         end for
14:      end for
15:      C[i : i + U, j : j + gvl] ← VC[i : i + U]
16:   end for
17:   j+= gvl
18: end for
```

Fig. 2: Optimized 3-loop implementation of GEMM

```
1: i ← 0 , j ← 0, k ← 0
2: long int gvl;
3: for j1 ← 0, j1 < N, j1+= blockN do
4:    for k1 ← 0, k1 < K, k1+= blockK do
5:       Pack MatrixB
6:       for i1 ← 0, i1 < M, i1+= blockM do
7:          Pack MatrixA
8:          for j ← 0, j < blockN, do
9:             gvl ← vsetvl(blockN − j)
10:            for i ← 0, i < blockM, i+= U do
11:               Prefetch block of C matrix into L1 cache
12:               Prefetch packedA matrix into L2 cache
13:               Prefetch packedB matrix into L2 cache
14:               VC[i : i + U] ← C[i : i + U, j : j + gvl]
15:               for k ← 0, k < blockK, k++ do
16:                  Prefetch packed B matrix into L1 cache
17:                  Prefetch packed A matrix into L1 cache
18:                  VB ← packedB[k, j : j + gvl]
19:                  for it ← 0, it < U, it++ do
20:                     tmp = alpha × packedA[it, k]
21:                     Vtmp ← tmp //broadcast
22:                     VC[it] ← vfmacc (VC[it],Vtmp,VB,gvl)
23:                  end for
24:               end for
25:               C[i : i + U, j : j + gvl] ← VC[i : i + U]
26:            end for
27:            j+= gvl
28:         end for
29:      end for
30:   end for
31: end for
```

Fig. 3: Optimized 6-loop implementation of GEMM

Fig. 1 shows the pseudocode for the naive implementation of GEMM ($C = alpha \cdot A \cdot B + beta \cdot C$), as implemented in Darknet. In the pseudo code, A ($M \times K$) represents the weight matrix, B ($K \times N$) represents the input matrix and C ($M \times N$) represents the output matrix.

To optimize GEMM, we follow two approaches. The first approach optimizes the *3-loop implementation*, depicted in Fig. 2. The second approach tiles the matrices, resulting in a *6-loop implementation* depicted in Fig. 3, where we apply optimizations.

We apply the following optimization to the *3-loop implementation*: i) vectorization with intrinsic instructions ii) contiguous memory loads/stores to/from vector registers, iii) loop reorder, and iv) loop unrolling. Loop reordering reduces the pressure on the memory subsystem by maximizing the reuse of the vector registers. Loop unrolling hides the pipeline latency by maximizing the vector register utilization and increasing the parallelism in the algorithm.

Figure 2 shows the pseudocode for the optimized 3-loop implementation of the GEMM kernel. In this algorithm, we use the jik loop order, and we unroll the intermediate loop j to reuse the vector data of matrix B by performing U $(unrollfactor)$ times dot products with different A matrix elements. The loop in line 3 is incremented by the vector length gvl to take advantage of VLA and the loop in line 5 is incremented by U to take advantage of loop unrolling. Loops are reordered to reuse the loaded vector data as much as possible. Low level intrinsics are used to manually vectorize the algorithm. For RISC-VV, the vector length is calculated using the `vsetvl` intrinsic instruction. Once matrices are loaded to the vector registers ($VB, VC, Vtmp$), we use a fused multiply-add vector intrinsic `vfmacc` to calculate the multiplication and addition for the intermediate resultant matrix VC. $Vtmp$, a scalar value broadcasted to the vector register, is passed as the second parameter to the `vfmacc` intrinsic. The compiler internally uses vector-scalar multiply-add intrinsics and avoids the use of the broadcast intrinsic instruction. The resulting multiple multiply-add operations hide the pipeline latency.

Furthermore, we optimize the *6-loop implementation*, where the original matrices in GEMM are tiled in blocks of dimensions $blockM$, $blockN$, $blockK$. We apply the following BLIS-like [33] optimizations: i) loop reorder, ii) matrix packing, iii) block size tuning, iv) loop unrolling, v) prefetching, and vi) vectorization using intrinsic instructions. We perform loop reorder and unrolling for the same reasons as in the *3-loop implementation*. We pack matrices to facilitate contiguous memory accesses. We tune the block sizes to the size of the caches, in order to minimize memory accesses and maximize reuse. Finally, prefetching assists in hiding load latencies.

Fig. 3 shows the pseudocode for the optimized 6-loop implementation of the GEMM kernel. The 6-loop implementations allow us to pack the blocks of matrices A and B so that the innermost loop performs contiguous accesses. The order in the innermost 3 loops is jik, where matrix A is accessed in continuous order from the packed A matrix to perform the

dot product with the packed B matrix. The first three loops in lines 3, 4, and 6 are incremented by block sizes $blockM$, $blockN$, and $blockK$, tuned to the architecture. Matrices are packed in lines 5 and 7, to facilitate contiguous cache access in the inner-most loop and facilitate prefetching. Matrix packing operations are also vectorized using intrinsic instructions.

The inner loops (lines 8 and 10) are incremented by gvl (granted vector length) and U ($unrollfactor$), as in the 3-loop implementation, to make use of VLA and facilitate loop unrolling. Here, $gvl \times U$ is also called the "macro-block" size. As in the 3-loop implementation, we perform loop reorder. Additionally, in this implementation, the blocks of matrix C are prefetched into the cache before storing them in the vector registers. We also prefetch the A and B packed matrix data into the L1 cache. The remainder of the inner-most loop is vectorized in the same way as in the 3-loop implementation.

We note that prefetching capabilities vary among the platforms. The toolchain used for RISC-VV does not yet support software prefetching (Zicbop extension), therefore any relevant intrinsic instructions are ignored by the compiler. In the case of ARM-SVE, the compiler generates the assembly instructions for prefetching, which take effect on the A64FX processor, but are treated as no-ops on our gem5 platform, which currently does not support software prefetching.

1: $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$, $tileitr \leftarrow 0$
2: $elements = 4$
3: VL = svcntw() // get vector length
4: $interchannels = VL/elements$
5: **if** $channels >= 4$ **then**
6: $tiles = interchannels$
7: **for** $tileitr \leftarrow 0$, $tileitr < channels$, $tileitr+ = tiles$ **do**
8: //Buffer preparation for longer vectors
9: **for** $k \leftarrow 0$, $k < tiles$, $k+ = 1$ **do**
10: **for** $i \leftarrow 0$, $i < 8$, $i+ = 1$ **do**
11: **for** $j \leftarrow 0$, $j < 4$, $j+ = 1$ **do**
12: $buff1[(i \times VL) + (j + (k \times 4)))]$ = pack row-wise 0-3 elements of 8x8 tile
13: $buff2[(i \times VL) + (j + (k \times 4)))]$ = pack row-wise 4-7 elements of the same 8x8 tile
14: **end for**
15: **end for**
16: **end for**
17: nnp_iwt8x8_3x3_with_offset_sve_vectorized()
18: Store the transposed data in their respective tiles across channels.
19: **end for**
20: **else**
21: // single tile
22: nnp_iwt8x8_3x3_with_offset_sve_vectorized()
23: **end if**

Fig. 4: Input transformations code snippet from winograd showcasing the inter-tile parallelism across channels

B. Winograd optimizations

As an alternative to im2col+GEMM, for convolutional layers with small filter sizes, we target the Winograd algorithm from the NNPACK [34] package. NNPACK, Arm Compute Library and other implementations [28], [50], [51] vectorize Winograd with ARM-NEON. We vectorize Winograd by building on the NEON implementation in NNPACK in a VLA way, to utilize the longer vector lengths up to 2048-bit on ARM-SVE. The Winograd implementation requires an input, weight, and output transformation and a tuple multiplication, and operates on a default tile size of 8×8. Vectorizing the transformations with longer vector lengths would require a larger tile size, however, in this case, the numerical accuracy would drop. Therefore, we employ a scheme of inter-tile parallelism across the input/output channels by using an 8×8 tile from each channel, which allows us to vectorize the transformation kernels using long vector lengths. Using 4 input/output channels with one row of 8×8 tiles from each channel as shown in Fig. 5, we can utilize two 512-bit vector registers. To utilize longer vector lengths, we increase the number of input/output channels accordingly, e.g. 16 channels for 2048-bit vector registers.

The pseudocode in Fig. 4 shows our inter-tile parallelization across the channels for the input transformation in Winograd. Lines 2 to 4 select the vector length, and determine the number of channels at runtime, in a VLA manner. For example, for a 512-bit vector length with 16 single precision elements, the number of channels will be 4. If the number of channels is more than 4, inter-tile parallelism is enabled. Lines 6-16 create the buffers `buff1`, `buff2` to utilize the specified vector length. In Line 17, these buffers are used as a input for the SVE-vectorized input transformation kernel. We optimize the kernels for the weight and output transformation for longer vector lengths in a similar way, using the same inter-tile parallelization scheme, and applying the corresponding vectorized transformation (replacing the function in line 17).

We additionally vectorize the tuple multiplication in a VLA way, which can use up to 2048-bits of vector length. To utilize the longer vector lengths for tuple multiplication, we increase the number of blocks for the GEMM kernel, using 16 blocks with 4 elements in each block. Therefore, there will be a total of 64 elements to utilize the maximum 2048-bit vector length.

V. Hardware Tuning

In this section, we detail our methodology to study the impact of tuning hardware parameters with the optimized kernels for CNN inference. We focus on three parameters: vector lengths, L2 cache sizes, and the number of vector lanes. To support scientific applications and AI workloads, the latest chips are integrating longer vector lengths for fast processing. As the vector length increases, so does the pressure on the memory subsystem. Adding larger caches to alleviate the pressure can increase the access time. Even if we assume constant access time, we still need to determine the exact cache size that

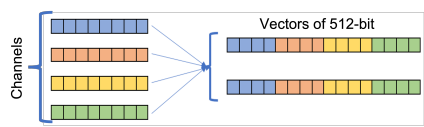

Fig. 5: Inter-tile parallelism in Winograd

is beneficial. Along with the cache sizes, there is a need to have more on-chip parallelism. However, bigger caches and more on-chip parallelism can influence the performance differently for different vector lengths. Therefore, it is important to study the trade-off between these micro-architectural parameters, as these hardware components occupy significant die area, while having an important influence on performance.

Vector length agnostic ISAs can work with different vector lengths without any modifications to the ISA, hence making vector length a hardware parameter in designing vector architectures. With recent ISAs supporting very long vector lengths, this raises the question of *how long the vector lengths should be*. Tuning the vector lengths to the demands of optimized CNN kernels can guide hardware designers in the selection of the appropriate vector lengths on future architectures.

Longer vector lengths require more on-chip storage, which consequently may require larger cache to effectively handle locality. Larger cache sizes can reduce the cache miss rate, therefore this raises the question of *how large should the L2 cache be for different vector lengths*. This question also relates to the length of vector registers, since longer vector registers can lead to increased pressure on the memory subsystem.

The number of vector elements to be processed per cycle is determined by the available on-chip parallelism. To achieve this, additional pipelines can be added to a vector architecture. However, this raises the question of *how many vector lanes are required for different vector lengths*, as adding more pipelines increases the start-up overhead, which can potentially degrade the performance with short vector lengths.

To respond to the aforementioned questions, our analysis shows the trade-offs between these three micro-architectural parameters. We highlight that other micro-architectural parameters, such as in-order vs. out-of-order cores, the core frequency, or the number of registers, can also be important, but are beyond the scope of this paper.

VI. Evaluation of im2col+GEMM

In this section, we present the results of our co-design study of CNN inference on RISC-VV and ARM-SVE. We first showcase the impact of algorithmic optimizations and hardware parameter tuning on RISC-VV, using a single core. For ARM-SVE, we evaluate our algorithmic optimizations in detail on the A64FX processor and perform the hardware parameter tuning on ARM-SVE@gem5. In all experiments with gem5, we report performance in terms of execution cycles. We exclude cycles spent on the initialization phase, such as network setup, as this is a constant overhead not incurred when continuously running inference over a stream of images.

A. Algorithmic optimizations with RISC-VV

We first analyze the performance of the algorithmic optimizations for im2col+GEMM on RISC-VV@gem5. To vectorize the inner-most kernels of the optimized 3-loop and 6-loop implementations, we use the EPI builtins [52]. In the 3-loop implementation we have tuned the loop unrolling factor by utilizing up to 32 vector registers. Our study shows no significant improvement beyond utilizing 16 registers for RISC-VV. In fact, by utilizing the 32 register, we experienced a performance drop by $\sim$15% due to register spilling. Therefore, we set the $unrollfactor$ as 16 for the 3-loop and 6-loop implementations. Moreover, for the optimized 6-loop implementation, we tune the block sizes of the matrices, determined by the $blockM$, $blockN$, $blockK$ parameters, to fit the packed matrices into the L2 cache, since, in the RISC-VV@gem5 model, the VPU is connected to the L2 cache.

TABLE II: Relative execution time of YOLOv3 (4 layers) with the optimized 6-loop implementation, compared to the optimized 3-loop implementation of im2col+GEMM on RISC-VV@gem5

Block sizes	Normalized Performance
128×1024×256	0.9
16×1024×128	0.95
16×512×128	0.98
16×512×256	0.96
32×512×128	0.97
64×1024×128	0.95

We simulate the first 4 convolutional layers of the YOLOv3 network on RISC-VV@gem5, with 1MB of L2 cache and 8 vector lanes, on a single core, using the optimized 3-loop implementation and the optimized 6-loop implementation, with different block sizes. The relative execution time of the 6-loop implementation over the 3-loop implementation is presented in Table II, for block sizes of different dimensions. We observe that the optimal block size for the 6-loop implementation is $16 \times 512 \times 128$, where the two implementations only differ by 2%, a difference that is not significant in the simulated environment.

Overall, the results indicate that the optimized 6-loop implementation does not offer any performance benefit over the optimized 3-loop implementation on RISC-VV, despite the BLIS-like optimizations. We attribute this to the following two reasons. First, the 6-loop implementation packs the matrices to facilitate contiguous cache accesses during the inner-most loop and prefetches the packed B and A matrices in the L2 and L1 caches. The rationale behind tuning the block size in BLIS-like optimizations is to fit matrix B in the last-level cache (L2) and matrix A in the L1 cache. However, in RISC-VV modeled with gem5, the VPU is connected to the L2 cache. Therefore, data in the L1 cache is not directly accessed by the VPU, and practically, the implementation benefits only from caching in L2. Additionally, as also explained in Section IV, RISC-VV does not support prefetching, which is a desired feature in the 6-loop implementation, in order to hide the latencies associated with matrix packing.

We conclude that **BLIS-like optimizations do not boost the performance of convolutional layers on RISC-VV.** We highlight that, after vectorizing all the kernels of the convolutional layer and by optimizing the im2col+GEMM kernel with the 3-loop implementation, we observe 14 × higher performance compared to the naive baseline for the YOLOv3-Tiny network model.

TABLE III: Average vector length and L2 cache miss rate

Vector length	YOLOv3	L2 cache miss rate(%)
512-bit	512	32
1024-bit	1022.9	36
2048-bit	2041.9	39
4096-bit	4063.7	42
8192-bit	8111.9	61
16384-bit	15902.2	79

B. Hardware parameters tuning with RISC-VV

Using the optimized 3-loop implementation, which demonstrates the best performance on RISC-VV, we proceed our experimentation with tuning hardware parameters of the architecture. We experiment with the first 20 layers of the YOLOv3 model, out of which 15 are the convolutional layers.

a) Scalability with different vector lengths: Fig. 6 demonstrates the impact of different vector lengths on the performance of the convolutional layers on RISC-VV. For this experiment, we consider a fixed L2 cache size of 1MB and a fixed number of vector lanes, equal to 8 on gem5, varying only the vector length. We note that longer vector lengths hide the pipeline latency of vector lanes, thus any overheads associated to the start-up time become minimal. Moving from 512-bit to 16384-bit vector lengths, the performance increases by 2.5×, but effectively, the performance saturates beyond the 8192-bit vector length. To analyze this effect, we present the consumed average vector lengths and L2 cache miss rate, collected in gem5, in Table III. Although the 16384-bit vector length is almost fully utilized, the L2 cache miss rate increases significantly both for the 8192-bit and 16384-bit vector lengths. We therefore attribute this performance saturation to the increase in the L2 cache miss rate. Although longer vector lengths help in hiding latency, which should boost the performance without increasing the cache size, they also require more data to be processed per cycle, therefore to be transferred from the memory to the cache and then to the VPU, hence the increased L2 cache miss rate.

b) Scalability with different L2 cache sizes: We continue our hardware parameter tuning with the L2 cache sizes. We examine the impact of L2 caches for different vector lengths, since we have observed an increase in L2 cache miss rate for the 1MB cache as the vector length increases. For this experiment, we consider a fixed number of vector lanes equal to 8, on gem5. We expect a larger L2 cache to reduce the miss rates, however, one should note that larger caches come with increased access latencies and require more chip area.

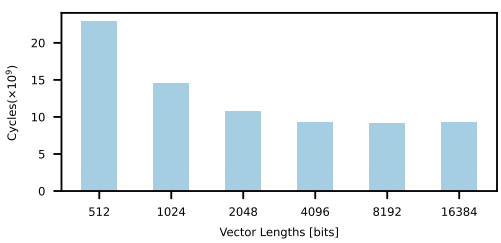

Fig. 6: Impact of vector lengths on RISC-VV@gem5 for YOLOv3 (20 layers), for constant L2 cache 1MB and 8 vector lanes.

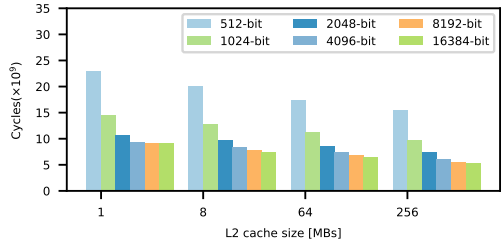

Fig. 7: Impact of the L2 cache size on RISC-VV@gem5 for YOLOv3 (20 layers), for 8 vector lanes.

Fig. 7 shows the performance of YOLOv3 from 1MB to 256MB with different vector lengths. We observe that for vector lengths up to 4096 bits, the performance increases by 1.5× as we increase the L2 cache size. For the longer 8192-bit and 16384-bit vector length, the equivalent performance improvement is 1.7×-1.9×. We additionally observe that, with a 256MB L2 cache, performance improves by ∼5% from 8192-bit vector length to 16384-bit vector lengths and L2 cache miss rates are 2.4% and 2.6% respectively. Therefore, we conclude that larger caches are beneficial, given that their latency remains low. Moreover, **it is important to use larger L2 caches for longer vector lengths, but the performance gains of very long vector lengths are limited.** Note that we have performed the same experiment on YOLOv3 using the optimized 6-loop implementation of im2col+GEMM, with block sizes tuned for an 8MB L2 cache, validating our conclusions regarding the L2 cache size tuning.

c) Scalability with different numbers of vector lanes: We finalize our analysis of hardware parameters by tuning the number of vector lanes, i.e. the parallel SIMD units in the vector architecture that determine the on-chip parallelism. We examine the impact of this hardware parameter for different vector lengths, as increasing the number of vector lanes also increases the startup time; execution starts only after filling all the vector lanes. We note, however, that this analysis is limited by gem5 capabilities, which only allows to simulate up to 8 vector lanes. For this experiment, we consider a fixed 1MB L2 cache. Increasing the vector lanes from 2 to 8 with different vector lengths, we observe a performance improvement of ∼1.25× for the 8192-bit vector length. In the case of 512-bit, performance scales from 2 to 4 lanes, but becomes saturated beyond 4 lanes. We therefore conclude that **additional vector lanes are more beneficial to longer vector lengths.**

C. Algorithmic optimizations with ARM-SVE

Similarly to RISC-VV, for ARM-SVE, we analyze the performance of the algorithmic optimizations for im2col+GEMM. For this, we use A64FX. We let the compiler to auto-vectorize all the kernels and manually vectorize kernels that the compiler

TABLE IV: Arithmetic Intensity and Sustained performance of YOLOv3 convolutional layers on A64FX

Layers	M	N	K	AI	% of Peak
L1	32	369664	27	7.32	46
L2	64	92416	288	26	72
L3	32	92416	64	11	50
L5	128	23104	576	52	77
L6	64	23104	128	21	70
L10	256	5776	1152	101	81
L11	128	5776	256	42	75
L38	256	1444	512	76	82
L44	1024	361	4608	126	83
L45	512	361	1024	88	78
L59	255	361	1024	65	75
L61	256	1444	768	85	91
L62	512	1444	2304	162	83
L75	255	5776	256	63	75

fails to vectorize, such as normalization and activation. We manually vectorize the inner-most kernels of the optimized 3-loop and 6-loop implementations on SVE.

Comparing the 6-loop implementation to the 3-loop implementation with ARM-SVE on A64FX, we observe a 2× performance improvement using the 6-loop, BLIS-like optimized GEMM kernel on the YOLOv3 network model. Unlike the case of RISC-VV modeled with gem5, which poses the limitation of the VPU being attached to the L2 cache, on A64FX, the 6-loop implementation is able to take advantage of the caches and improve the performance of GEMM. Moreover, since prefetching is a hardware feature of A64FX, the prefetching instructions boost the performance of the 6-loop implementation. We note, however, that the 6-loop implementation outperforms the 3-loop implementation by 15% on ARM-SVE@gem5 which does not support prefetching, with a 512-bit vector length. Comparing the optimized 6-loop implementation to the naive GEMM in Darknet, we observe a ∼32× performance improvement for YOLOv3 on A64FX.

a) Per-layer sustained performance: We assess the sustained performance of the convolutional layers in YOLOv3, with respect to their arithmetic intensity, as per the roofline model, on A64FX, using our optimized kernels. YOLOv3 has 75 convolutional layers, but some layers work on the same input sizes. We therefore consider the 14 discrete convolutional layers which work with discrete matrix sizes, and compute the arithmetic intensity (AI) per layer as follows:

$$AI = \frac{ArithmeticOperations}{Bytes} = \frac{2\times M\times N\times K}{4\times(M\times N+K\times N+M\times K)}$$

where M, N, K correspond to the sizes of the weight, input and output matrices. We showcase the results in Table IV. We note that the peak performance of a single A64FX core is 62.5 GFLOPs. The results indicate that some layers have low AI and sustained performance, especially the layers with small M and K values, i.e., small weight matrix size. There is additional room for performance improvement for these layers, which is, however, beyond the scope of this paper, where we optimize kernels focusing primarily on portability across ISAs with VLA vector extensions.

b) Performance Analysis of Manual vs Auto-vectorization: To understand the effectiveness of auto-vectorization, we performed a comparative analysis for manual optimization and auto-vectorization. Both clang and gcc compilers were able to vectorize most of the CNN kernels, however with performance limitations. As a reference, auto-vectorization achieved ∼6.3x speedup compared to the baseline for YOLOv3-tiny. Forcing the compiler to unroll loops while auto-vectorizing, with different unroll degrees, we achieved ∼9x speedup. With manual vectorization and optimizations, we were able to achieve ∼21× speedup compared to the baseline, on ARM-SVE on A64FX.

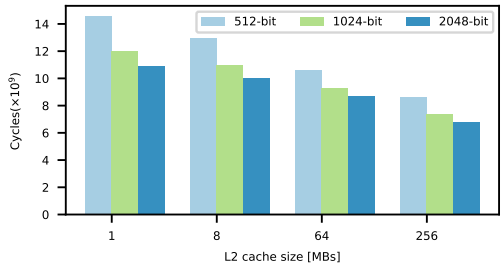

Fig. 8: Impact of vector lengths and L2 cache size on ARM-SVE@gem5 for YOLOv3 (20 layers).

D. Hardware parameters tuning with ARM-SVE

Similarly to RISC-VV, we study the impact of micro-architectural parameters with ARM-SVE using gem5. As the ARM-SVE model in gem5 sets the number of vector lanes proportional to the vector length, we focus only on tuning the vector length and the L2 cache size. We do not further tune the block sizes of the 6-loop implementation, as they fit in the smallest simulated cache. Fig. 8 shows the impact of different vector lengths and L2 cache sizes on the performance of the first 20 layers of YOLOv3. We observe that, for a cache of 1MB, moving from 512-bit to 2048-bit vector lengths, the performance improves by 1.34×. Additionally, similarly to RISC-VV, performance benefits from larger caches, with a performance improvement of 1.6× as we increase the L2 cache size from 1MB to 256MB for 2048-bit vector length. Our findings for ARM-SVE agree with our observations for RISC-VV: our optimized kernels can benefit from longer vectors and larger cache sizes, which can significantly boost the performance of CNN inference on vector architectures.

VII. Evaluation of Winograd

As explained in Section IV, we vectorize the transformation and tuple multiplication kernels of Winograd in a VLA way, using intrinsic instructions on ARM-SVE. Our kernels adapt the different vector lengths and can be executed with 512-bit, 1024-bit and 2048-bit vector lengths. We use these kernels in Darknet, to implement convolutional layers with kernel sizes of 3×3 and stride 1 and 2. For convolutional layers of different kernel sizes, we fall back to our optimized im2col+GEMM.

For our Winograd implementation on ARM-SVE, we use intrinsics to create tuples of four vectors and then transpose

these vectors. On RISC-VV, currently, no specific intrinsics are available to perform these operations. We therefore implemented a solution that uses temporary buffers and additional store and gather-load intrinsics. This however limits the performance improvement and the potential insights of running Winograd on the RISC-VV with very long vectors. Because of this reason, we do not include RISC-V results in the Winograd analysis.

A. Algorithmic optimizations with ARM-SVE

We evaluate the performance of the optimized Winograd implementation in Darknet on the A64FX processor. As a baseline for comparison, we use our optimized im2col+GEMM. We note that a naive implementation of Winograd is slower than using the naive implementation of im2col+GEMM, therefore we use our optimized im2col+GEMM as the baseline for comparison. A primary analysis revealed that the weight transformation is a major bottleneck, but it can be performed offline for inference. After excluding the weight transformation time, we achieve a speedup of 1.5× compared to im2col+GEMM for VGG16, where all convolutional layers use 3×3 kernel-sized filters. For YOLOv3, where 38 out of the 75 use 3×3 kernel-sized filters, the equivalent speedup is 1.35×. Out of these 38 layers, the 32 with stride 1 perform 2.4× better with Winograd compared to im2col+GEMM, while for the 6 layers with stride 2, Winograd is 1.4× slower than im2col+GEMM. The remaining layers use 1×1 kernel-size filters and default to im2col+GEMM. We therefore conclude that **our optimized Winograd algorithm offers significant performance improvement for layers with stride 1, however, different algorithmic optimizations are required to achieve high performance for layers with stride 2.** Still, convolutional layers require careful algorithmic selection related to the kernel sizes and strides.

B. Hardware parameter tuning with ARM-SVE

Similarly to our approach for im2col+GEMM, we study the impact of hardware parameters on the performance of our optimized Winograd algorithm for ARM-SVE, using Gem5. As indicated by our evaluation on A64FX, we use Winograd for all convolutional layers with 3×3 kernel sizes and stride 1, and default to our optimized im2col+GEMM implementation for all other cases. In particular, we study the impact of the L2 cache size, ranging from 1MB up to 256MB, and the impact of different vector lengths, i.e. 512-bit, 1024-bit and 2048-bit. The number of vector lanes is propotional to vector lengths.

We showcase the results of our analysis for the first 20 layers of YOLOv3 in Fig. 9, and for VGG16 in Fig. 10. For both network models, for an L2 cache of 1MB, we observe a performance improvement of 1.4× as we increase the vector lengths from 512 to 2048 bits, due to increased throughput and decreased pressure on the memory subsystem.

Evaluating the impact of L2 cache sizes, we observe that, for the first 20 layers of YOLOv3, performance improves by 1.75× for all vector lengths, when increasing the caches from 1MB to 256MB. For VGG16, the performance improves by

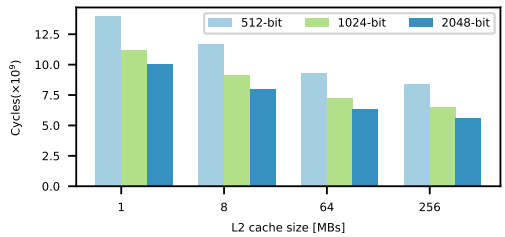

Fig. 9: Impact of vector lengths and L2 cache size with Winograd on ARM-SVE@gem5 for YOLOv3 (20 layers).

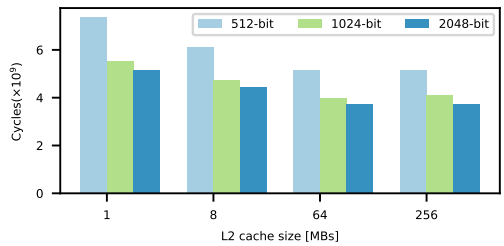

Fig. 10: Impact of vector lengths and L2 cache size with Winograd on ARM-SVE@gem5 for VGG16.

1.4× from 1MB to 64MB, but the network does not benefit from a larger cache. We note that all layers in VGG16 use Winograd, which has smaller cache requirements compared to im2col+GEMM, whereas several YOLOv3 layers invoke im2col+GEMM. As a conclusion, **longer vectors are highly beneficial to the performance of Winograd-enabled convolutional layers and networks. With respect to the L2 cache size, our optimized Winograd algorithm does not have high cache requirements, and therefore is able to perform well with moderately large L2 cache sizes.**

We finally compare the performance of VGG16 using Winograd, compared to im2col+GEMM, with different vector lengths of 512, 1024 and 2048 bits, with 1MB of L2 cache. The performance improves by 1.4×, 1.5×, and 1.3× respectively, compared to im2col+GEMM, for the different vector lengths, showing that Winograd is a good alternative to im2col+GEMM for any vector length.

VIII. Performance-Area analysis

Our analysis so far has shown that the performance of CNN inference can benefit from longer vector lengths and larger caches. This, however, will require a larger chip area. To evaluate this performance-area tradeoff, as well as the attainable performance in a fixed area envelope, we examine the scenario of a RISC-VV core with a decoupled VPU of 8 lanes, like the one simulated in Section III, implemented in 7nm FinFET technology. Given the results in [53], we estimate the area of the core, VPU and vector register file (VRF) in 22nm, based on the assumption that only the VPU VRF area will increase proportionally to the vector length, while the core and VPU FPU area will remain constant. Our analysis

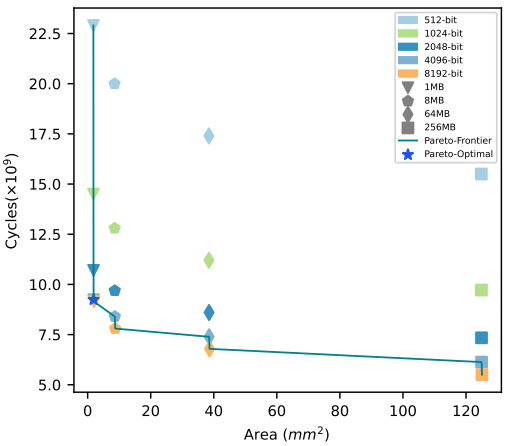

Fig. 11: Pareto frontier for the performance and area of a RISC-VV chip implemented at 7nm, with increasing vector lengths and L2 cache sizes, for YOLOv3 (20 layers).

estimates that the chip area dedicated to the VRF consumes 3%, 6.9%, 12.68%, 22.5%, and 36.9% of the total chip area, as we increase the vector length from 512 bits to 8192 bits. We then scale the total area to a 7nm FinFET technology, which translates to a conservative estimate of a 6.2× increase in transistor density [54], [55]. PCacti [56] is used to estimate the area of L2 caches in 7nm.

We present our analysis for the performance of the first 20 layers of the YOLOv3 network against the expected chip area in Figure 11. It is evident that the impact of longer vector lengths on area is minimal, but it is significant for performance. Most of the points on the Pareto frontier correspond to longer vector lengths. On the other hand, the cache size has a more significant impact on the total area, driving the chip area up to 125.1mm^2 for the largest configuration, with less significant impact on performance. We find the Pareto-optimal configuration for both performance and chip area to use the smallest examined L2 cache size, i.e., 1MB, with one of the larger vector lengths, i.e., 4096 bits. Although we expect that technology scaling will further decrease the required area, making hardware designs with larger caches feasible, we highlight that the caches still consume most of the area and power of the chip [57] and algorithmic implementations which are less sensitive to the cache size, e.g., Winograd instead of im2col+GEMM, need to be considered for effective co-design of future vector architectures.

IX. Conclusion

In this paper, we presented a hardware and software co-design study of CNN inference on modern vector architectures with variable vector lengths. Focusing on the most time-consuming kernels in convolutional layers, we have developed efficient, VLA-vectorized, optimized implementations of im2col+GEMM and the Winograd algorithm.

Experimenting with two different ISAs, RISC-VV and ARM-SVE, we conclude that certain optimizations are not portable across vector architectures, and highlight the following portable optimizations: i) maximize utilization/reuse of vector registers, ii) use unstrided load/store instructions, for contiguous memory accesses, iii) use multiple multiply-add instructions to hide the pipeline latency. We additionally conclude that longer vector lengths improve performance even with smaller caches, however larger caches with low latencies can help minimize any adverse effects from increased cache misses. Finally, more vector lanes can hide the pipeline and startup latency for longer vector lengths.

Our algorithmic optimizations using VLA ISAs for im2col+GEMM improve the performance of CNN inference by 14× for YOLOv3-Tiny on RISC-VV and by 32× for YOLOv3 on ARM-SVE, compared to the naive implementation of im2col+GEMM in Darknet. Our vectorized Winograd algorithm offers additional performance improvement of 1.35× and 1.5× to YOLOv3 and VGG16 respectively, while having lower cache requirements.

We believe that our work is useful to programmers, hardware designers and compiler developers. In the future, we aim to extend our algorithmic optimizations for vector architectures to more kernels in DNN inference and examine additional, influential architectural and micro-architectural features.

Acknowledgement

This work has been supported by the Swedish Research Council via registration number 2020-04892. The simulations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at NSC partially funded by the Swedish Research Council through grant agreement no. 2018-05973. We thank the Barcelona Supercomputing Center for providing access to an A64FX machine.

References

[1] K. Li, W. Ma, U. Sajid, Y. Wu, and G. Wang, "Object detection with convolutional neural networks," *CoRR*, vol. abs/1912.01844, 2019. [Online]. Available: http://arxiv.org/abs/1912.01844

[2] M. M. Lopez and J. Kalita, "Deep learning applied to NLP," *CoRR*, vol. abs/1703.03091, 2017. [Online]. Available: http://arxiv.org/abs/1703.03091

[3] D. Palaz, M. Magimai.-Doss, and R. Collobert, "Convolutional neural networks-based continuous speech recognition using raw speech signal," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 4295–4299.

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[5] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=HJzgZ3JCW

[6] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[7] S. Wang, A. Pathania, and T. Mitra, "Neural network inference on mobile socs," *IEEE Design & Test*, vol. 37, no. 5, p. 50–57, Oct 2020.

[8] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No "power" struggles: Coordinated multi-level power management for the data center," vol. 42, 03 2008, pp. 48–59.

[9] M. Abadi, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016.

[10] Pytorch. Pytorch gpu. [Online]. Available: https://www.run.ai/guides/gpu-deep-learning/pytorch-gpu/

[11] Y. Hu, Y. Liu, and Z. Liu, "A survey on convolutional neural network accelerators: Gpu, fpga and asic," in *2022 14th International Conference on Computer Research and Development (ICCRD)*, 2022, pp. 100–107.

[12] "Whitepaper gpu-based deep learning inference : A performance and power analysis," 2015.

[13] L. Wang, Z. Chen, Y. Liu, Y. Wang, L. Zheng, M. Li, and Y. Wang, "A unified optimization approach for cnn model inference on integrated gpus," 2019.

[14] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[15] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating cnn inference on asics: A survey," *Journal of Systems Architecture*, vol. 113, p. 101887, 2021.

[16] J. Park, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," 2018.

[17] S. Mittal, P. Rajput, and S. Subramoney, "A survey of deep learning on cpus: Opportunities and co-optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2021.

[18] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing {CNN} model inference on {CPUs}," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 1025–1040.

[19] E. Georganas and Kalamkar, "Tensor processing primitives: a programming abstraction for efficiency and portability in deep learning workloads," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[20] R. Li and Xu, "Analytical characterization and design space exploration for optimization of cnns," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 928–942.

[21] T. P. Morgan, "Ibm bets big on native inference with big iron," *The Next Platform*, August 23, 2021. [Online]. Available: "https://www.nextplatform.com/2021/08/23/ibm-bets-big-on-native-inference-with-big-iron/"

[22] C. Lemuet, J. Sampson, J. Francois, and N. Jouppi, "The potential energy efficiency of vector acceleration," 12 2006, pp. 1 – 1.

[23] N. Stephens, "The arm scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[24] E. P. Initiative, "V for vector: software exploration of the vector extension of risc-v," https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/, 2019.

[25] N. Adit and A. Sampson, "Performance left on the table: An evaluation of compiler auto-vectorization for risc-v," *IEEE Micro*, pp. 1–9, 2022.

[26] M. Sahaya Loui, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. Reddi, and A. Joshi, "Towards deep learning using tensorflow lite on risc-v," 06 2019.

[27] Cococcioni, "Fast deep neural networks for image processing using posits and arm scalable vector extension," *J. Real-Time Image Process.*, vol. 17, no. 3, 2020.

[28] ARM. Arm compute library. [Online]. Available: https://github.com/ARM-software/ComputeLibrary

[29] F. P. Dan Andrei Iliescu. Arm scalable vector extension and application to machine learning. [Online]. Available: https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/arm-scalable-vector-extensions-and-application-to-machine-learning

[30] C. Ramírez, "A risc-v simulator and benchmark suite for designing and evaluating vector architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3422667

[31] Y. Kodama, T. Odajima, M. Matsuda, M. Tsuji, J. Lee, and M. Sato, "Preliminary performance evaluation of application kernels using arm sve with multiple vector lengths," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 677–684.

[32] A. Poenaru and S. McIntosh-Smith, "The effects of wide vector operations on processor caches," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 531–539.

[33] V. Zee, "The blis framework: Experiments in portability," *ACM Trans. Math. Softw.*, vol. 42, no. 2, jun 2016.

[34] M. Dukhan. (2016) Nnpack. https://github.com/Maratyszcza/NNPACK.

[35] N. Binkert, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011.

[36] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: Past, present and future," in *Proceedings of the 12th International Conference on Supercomputing*, ser. ICS '98. NY, USA: ACM, 1998, p. 425–432.

[37] Intel. (2020) Instruction set extensions and future features programming reference. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[38] ARM, ""ARM Neon Programmer's Guide"," 2013. [Online]. Available: https://documentation-service.arm.com/static/5f731b591b758617cd95559c?token=

[39] (2020) V for vector: software exploration of the vector extension of risc-v. [Online]. Available: https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/

[40] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[41] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, aug 2019.

[42] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251–280, 1990, computational algebraic complexity editorial. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747717108800132

[43] S. A. Alam, A. Anderson, B. Barabasz, and D. Gregg, "Winograd convolution for deep neural networks: Efficient point selection," *ACM Trans. Embed. Comput. Syst.*, mar 2022, just Accepted.

[44] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on fpgas," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 101–108.

[45] Gem5, "Gem5." [Online]. Available: https://gem5.googlesource.com/public/gem5

[46] BSC. Llvm epi compiler. [Online]. Available: https://ssh.hca.bsc.es/epi/ftp/

[47] ARM. Arm c/c++ compiler. [Online]. Available: https://documentation-service.arm.com/static/5f187b3e20b7cf4bc524c620?token=

[48] WikiChip. Zen 2 - microarchitectures - amd. [Online]. Available: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2

[49] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 3–14.

[50] P. P. Maji, "Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus," *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pp. 1–5, 2019.

[51] D. Li, D. Huang, Z. Chen, and Y. Lu, "Optimizing massively parallel winograd convolution on arm processor," in *50th International Conference on Parallel Processing*, ser. ICPP 2021. NY, USA: ACM, 2021.

[52] R. Ferrer. epi-builtins-ref. [Online]. Available: https://repo.hca.bsc.es/gitlab/rferrer/epi-builtins-ref

[53] C. Lazo, E. Reggiani, C. Rojas, R. Bagué, L. Vargas, M. Salinas, M. Cortés, O. Unsal, and A. Cristal, "Adaptable register file organization for vector processors," 11 2021.

[54] M. Bohr, "22ffl technology," *URL: https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/03/Mark-Bohr-22FFL-2017. pdf*, 2017.

[55] P. Oldiges, R. A. Vega, H. K. Utomo, N. A. Lanzillo, T. Wassick, J. Li, J. Wang, and G. G. Shahidi, "Chip power-frequency scaling in 10/7nm node," *IEEE Access*, vol. 8, pp. 154 329–154 337, 2020.

[56] Pcacti, "Sport lab." [Online]. Available: https://sportlab.usc.edu/downloads/packages/

[57] E. Arima, Y. Kodama, T. Odajima, M. Tsuji, and M. Sato, "Power/performance/area evaluations for next-generation hpc processors using the a64fx chip," in *2021 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*. IEEE, 2021, pp. 1–6.

Chapter 5

Paper II

Co-Design of Convolutional Algorithms and Long Vector RISC-V Processors for Efficient CNN Model Serving

Sonia Rani Gupta, Nikela Papadopoulou, Jing Chen, and Miquel Pericàs

Preprint from International Conference on Parallel Processing (ICPP), 2024

Co-Design of Convolutional Algorithms and Long Vector RISC-V Processors for Efficient CNN Model Serving

Sonia Rani Gupta
Chalmers University of Technology
Gothenburg, Sweden
soniar@chalmers.se

Nikela Papadopoulou
University of Glasgow
Glasgow, UK
nikela.papadopoulou@glasgow.ac.uk

Jing Chen
Chalmers University of Technology
Gothenburg, Sweden
chjing@chalmers.se

Miquel Pericàs
Chalmers University of Technology
Gothenburg, Sweden
miquelp@chalmers.se

ABSTRACT

The performance of convolutional algorithm depends on the size, stride, and input/output channels of the convolutional kernel. Moreover, the varying computational demands of convolutional layers influence the requirement for SIMD support on multicore processors. Finally, sharing cache resources in scenarios such as inference serving also impacts the runtime choice of the best algorithm. To identify the best settings, we perform a co-design exploration, focusing on the software parameters of the convolutional layers of convolutional neural networks (CNNs), and three distinct algorithmic implementations: Direct, im2col+GEMM, and Winograd, jointly with hardware parameters for vector architectures. Our simulation-based study identifies that Winograd is suitable for convolutional layers with a 3×3 kernel size and stride 1, specifically for shorter vector lengths and L2 cache sizes. For layers with more input/output channels, im2col+GEMM performs better. Looking at VGG-16, our study shows that not all the layers benefit from our biggest simulated cache memory when using the Direct and Winograd implementations, while the im2col+GEMM implementation scales to an L2 cache memory of 64MB with all layers. In contrast, all the simulated layers of YOLOv3 benefit from an L2 cache memory of 64MB, for all convolutional algorithms. To select the best implementation at runtime, we develop a random forest predictor that selects the best algorithm in over 90% of the cases, with limited degradation when a sub-optimal configuration is selected. We conclude with a Pareto analysis of the area-performance trade-off in an inference serving scenario, on a 7nm RISC-V multicore model with a vector unit supporting vectors of 512 up to 4096 bits.

ACM Reference Format:
Sonia Rani Gupta, Nikela Papadopoulou, Jing Chen, and Miquel Pericàs. 2024. Co-Design of Convolutional Algorithms and Long Vector RISC-V Processors for Efficient CNN Model Serving. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673121

1 INTRODUCTION

Model-serving is a major source of computing cycles, with some cloud providers reporting over hundreds of trillion of AI model executions per day [31]. Within AI model serving, Convolutional neural networks (CNNs) are commonly used in image and vision tasks. CNNs are computationally intensive and require high computing power to accelerate their performance. GPUs (Graphics Processing Units) have been widely used to accelerate CNNs due to their parallel processing capabilities [4, 22, 42]. As an alternative, CNNs have also become popular on CPUs [18, 27, 28] where they benefit from higher availability, and the increasing parallel processing capabilities offered by larger core counts and SIMD units. In particular, emerging long vector architectures are a promising direction for efficient inference serving [11, 23].

CNN models are built upon a series of consecutive layers, with convolutional layers being the most time-consuming. Various algorithms can be employed to implement these convolutional layers, including Direct, im2col+GEMM, Winograd, and FFT. The Direct convolutional operation involves sliding convolutional weights over the input tensor and calculating the dot product between the weight and input [33]. On the other hand, the im2col+GEMM algorithm transforms the image into a column matrix, turning the convolutional operation into a matrix multiplication by convolving the transformed input matrix with the weight matrix. This matrix multiplication operation can significantly enhance the performance of the convolutional layers because of the well-established optimizations for GEMM on most computing platforms [19]. While im2col+GEMM shares the computational complexity of Direct, it does increase the memory footprint [35] because of the image-to-column transformation (im2col). Winograd and FFT necessitate the initial transformation of both the image and weights, followed by block-by-block multiplications on the transformed input and weight matrices, concluding with output transformation. These methods enhance the convolutional implementation's performance by reducing computational complexity. Winograd is effective with small kernel sizes, such as 3×3 or 5×5 [6], while FFT is better suited for larger kernel sizes [29]. Since large kernel sizes are not common in modern CNNs, we do not further consider the FFT algorithm in this work.

Common convolutional network models consist of multiple convolutional layers with distinct dimensions, dictated by the input, kernel, and output's width and height, the input and output channels, and the stride of the convolution [37]. Notably, the different algorithms (im2col+GEMM, Direct, Winograd) that can be used to implement convolutional layers demonstrate varying performance depending on the convolution dimensions, as a consequence of their varying algorithmic complexity and memory footprint [32].

Moreover, the underlying computer architecture affects the performance of each algorithm. On the one hand, cache sizes, and memory bandwidth influence the performance, as certain algorithmic implementations, such as im2col+GEMM, increase the memory footprint of a convolutional layer. On the other hand, vector units can offer high performance to algorithms, albeit they require several algorithmic optimizations to exploit increasing vector lengths.

Model-serving frameworks [7, 10] aim to optimize inference performance through techniques like concurrent model execution. This approach creates replicas of a single model, enabling parallel processing on a single hardware unit (GPU or CPU). Load balancing distributes incoming requests across these replicas, maximizing resource utilization. While beneficial for cost-effective deployment, particularly for smaller models, concurrent execution introduces competition for caching resources. Consequently, the selection of the optimal algorithm becomes dependent on the characteristics of co-running inference tasks.

Previous studies [14, 20, 21, 40, 43, 44] have focused on optimizing the performance of specific algorithms on vector architectures, presenting comparative analyses with state-of-the-art libraries for different layers of network models on vector processors, and exploring the interplay of algorithmic optimizations with hardware parameters of long vector architectures. Several works [12, 13, 16] provide a comparative analysis of different algorithmic implementations of convolutional layers on SIMD ARM-based architectures. Despite the extensive research on convolutional neural networks and various algorithmic implementations, the mutual impact of convolution algorithms and hardware parameters remains unexplored, reducing utilization and hampering the task of effectively designing future CPUs for CNN model serving.

In this paper, we conduct a performance investigation and co-design study on three distinct algorithms, Direct, im2col+GEMM, and Winograd, for the implementation of convolutions on RISC-V based architectures implementing the "V" vector extension v1.0 [2] (RVV). RVV enables a style of programming called vector length agnostic programming (VLA), which allows the same program to run unmodified on processors implementing different vector lengths. We simultaneously explore the characteristics of convolutional neural network models and the hardware parameters of long vector architectures, focusing on the vector length and the size of the L2 cache. Building upon our previous work, where we have developed optimized versions of im2col+GEMM and Winograd for vector architectures [20, 21], we use two variants of im2col+GEMM [21], a 3-loop implementation and a 6-loop implementation, and the Winograd algorithm [20] implemented and optimized for RVV within the Darknet framework [38]. We additionally implement a vectorized implementation of the Direct algorithm on RVV, following the implementation proposed in [40] for the oneDNN framework. Our simulations on an RVV-enabled fork of gem5 [1] show that blocking of the input channels is not a beneficial optimization for the Direct algorithm over naive vectorization. Instead, loop reordering provides a greater improvement in performance (3×). Subsequently, we present a comparative analysis of the three algorithms, considering the tradeoffs between algorithmic optimizations, shared last-level cache sizes, and vector lengths. Our simulation results show that there is no optimal algorithmic choice for all convolutional layers. Hence, to support efficient model serving, we train several classification algorithms, finding that random forests exhibit high accuracy in selecting the optimal algorithm for each case. We conclude the paper with insights into the trade-off between the performance, throughput, and resource considerations for the long vector architectures.

We hereby highlight the main contributions of our paper:

- We vectorize the Direct algorithm and perform a comparative analysis involving our previously developed, vectorized implementations of im2col+GEMM [21] and Winograd [20], on an RVV model with a vector length of 512 bits and an L2 cache of 1MB. The analysis shows Winograd to be the best choice for layers with 3×3 kernel size, whereas, the 6-loop implementation of im2col+GEMM is the best choice for layers with large numbers of input and output channels and skinnier matrices. The Direct algorithm performs best when the input and output dimensions are high, but the number of channels is relatively small.
- Our co-design analysis demonstrates that the Winograd algorithm exhibits adequate performance with small vector lengths for 3×3 kernel sizes, while the Direct algorithm excels with longer vector lengths. The im2col+GEMM variants require larger L2 caches owing to their larger memory footprints, however, the 6-loop im2col+GEMM variant scales well and exhibits high performance for layers with large input and output dimensions. The Direct algorithm benefits from large cache sizes, especially as the vector length increases, and performs well with large input and output dimensions when the matrices are not skinny.
- We evaluate several classification algorithms and train an algorithm selection model using random forests, which deliver the best prediction accuracy. The trained model selects the optimal algorithm in 92.8% of the cases, on average. The overall slowdown introduced by mispredicted layers is negligible in most cases, and never above 10%.
- We employ Pareto frontiers to analyze the trade-off between execution time, model serving throughput, and area when using the different algorithms with the VGG16 and YOLOv3 network models. We show that, for the case of a single network, algorithm selection allows for better performance in less area, compared to using a single algorithm for each layer. Our analysis for co-located model instances shows that co-location and algorithm selection offer throughput that scales linearly with area, making a compelling case for co-design for model serving.

2 RELATED WORK

Several works have focused on optimizing convolutions for vector architectures. Specifically, Alaejos et al. [5] optimize GEMM for deep learning on the ARM-NEON, ARM-SVE and Intel AVX512 vector extensions. Wang et al. [44] optimize the Winograd algorithm on RISC-V architectures with a custom instruction extension. Dolz et al. [16] optimize the im2col transformation and Winograd algorithms for ARM-SVE. In another work, Dolz et al. [15] optimize the Winograd algorithm for Intel AVX, ARM NEON, and ARM-SVE architectures. Santana et al. [40] optimize the Direct algorithm for long vector architectures, focusing on ARM-SVE. Kelefouras et al. [25] vectorize and optimize the 2D direct convolutions on Intel AVX. Wang et al. [43] optimize the Direct algorithm on ARM NEON. In our previous work [20, 21], we optimize the im2col+GEMM and

Winograd algorithms on ARM-SVE and RISC-V Vector extensions, also performing a co-design study concerning the vector length, vector lanes, and L2 cache size.

Concerning performance comparisons of different algorithmic implementations of convolutions, Jordà et al. [24] and Xu et al. [45] perform such an analysis on GPUs. Jordà et al. focus on cuDNN and propose that different algorithms should be used depending on the kernel size. Xu et al. also look at cuDNN implementations and propose a scheme for algorithm selection based on the convolution dimensions. Dolz et al. [12] focus on performance-energy tradeoffs of the different algorithms for convolutions on ARM processors. Zlatenski et al. [46] perform a comparative analysis of Winograd and FFT for convolutions using different CNNs on modern CPUs, for full network models.

In this paper, we utilize optimized algorithmic implementations for im2col+GEMM, Winograd, and Direct-based convolutions. We focus on the emerging, long vector architectures with the vector-length-agnostic RVV ISA, and perform not only a comparative analysis of algorithms at a per-layer basis, but also a co-design study, and a performance-area analysis, seeking to optimize future vector architectures for convolutions. In contrast to our previous work [20, 21], where we seek to explore the performance potential of vector architectures for CNNs via co-design, we focus on the aspect of algorithm selection and its impact on the attainable throughput per area in the scenario of model serving.

3 METHODOLOGY

3.1 Experimental Platform

In this work, we focus on the RISC-V Vector Extension [2] (RVV) within the RISC-V Architecture. Including 32 vector registers, the RISC-V Vector architecture supports a maximum vector length (MVL) of 16384 bits. RVV allows the utilization of various vector lengths (`vlen`), expressed as powers of two, provided they do not exceed the MVL. The architecture employs the concept of vector length to specify the number of elements to be processed within a vector. The vector instruction `vsetvl` instruction is used to dynamically determine the vector length at runtime. This instruction takes the requested vector length (`rvl`) in elements and the element width in bits (`sew`) as inputs. The output of this instruction is the granted vector length (`gvl`) in elements. In this way, Vector Length Agnostic (VLA) code generation with different `vlen` is handled at runtime.

We perform all our experiments on a fork of the gem5 simulator [1], a cycle-accurate simulator configured with the RISC-V in-order *RiscvMinorCPU* CPU model, with a core frequency of 2GHz. The simulator implements a tightly integrated vector unit targeting the RVV v1.0. In our experiments, we vary the maximum vector length of the vector units from 512 bits up to 4096 bits. The memory subsystem is configured with DDR3 1600 memory technology with 12.8GiB/s bandwidth per core, which is not far from the measured per-core bandwidth of a recent Intel Xeon Max 9480 with HBM (~19GB/s) [30]. Additionally, the simulated CPU integrates two levels of data cache. We fix the L1 cache size to 64KB and vary the L2 cache size from 1MB up to 64MB in our experiments. We note that this fork of gem5 models a constant latency for all vector instructions. In practice, the latency of the instructions will vary with the implementation of RVV. Also, we note that the simulator supports vector lengths only up to 4096 bits. To validate the results of convolutional layers, we additionally use Spike [3], a RISC-V ISA simulator that supports vector lengths up to 4096 bits and supports the RVV v1.0 extension.

3.2 Algorithms for Convolutions

In this paper, we focus on three different algorithms commonly used to implement convolutional layers, namely Winograd, im2col+GEMM, and Direct. We employ two variants of the optimized im2col+GEMM algorithm for the RVV architecture, as described in [21], a 3-loop implementation and a 6-loop implementation, hereby denoted as *im2col+GEMM - 3 loops* and *im2col+GEMM - 6 loops*. Although this previous work shows that the *GEMM - 3 loops* implementation performs better on RVV, in this work, we simulate a tightly integrated RISC-V vector unit, which resembles the architecture of the Fujitsu A64FX processor, where the *GEMM - 6 loops* implementation has been shown to perform more efficiently. We tune the block size to fit in the L2 cache of our architecture, at a size of 16×512×128. We utilize the vectorized and optimized Winograd implementation outlined in [20]. The aforementioned implementations are open-source and publicly available. For the Direct algorithm, we leverage the algorithms described by Santana et al. [40], which target long vector architectures and have been evaluated on the NEC Vector Engine.

Implementing the Direct algorithm for RVV. We implement the Direct algorithm in the Darknet framework [38]. Following the rationale in [40], the Direct algorithm can be best optimized with the NHWC memory layout of the input (where N refers to the number of images in the batch, H refers to the input height, W refers to the input width, and C refers to the input channels). Therefore, we transform the input and weights from the NCHW format to the NHWC format, before starting the computations. Subsequently, we "naively" vectorize the Direct algorithm across the input channels IC. Following this, we implemented blocking of the input channels, as proposed in [40], however, we did not observe any performance improvement on top of the naive vectorization, as the memory footprints of the subtensors produced by blocking are smaller than the L2 cache size of 1MB we simulate. This is partly because the proposed blocking scheme in [40] aims to optimize the algorithm on an L2 cache size of 256KB, with a long cache line of 128 bytes. To further optimize the vectorized algorithm, we instead followed a loop reordering strategy, accessing the input channels after the output channels and dimensions, improving performance by 3× over the naive vectorized version. Furthermore, we utilize and reuse the maximum possible vector registers by unrolling the loops around the output width (OW) and output height (OH). We choose the unrolling factor in such as way that the algorithm utilizes the maximum possible vector registers by avoiding landing on the tail loop if possible, to avoid any potential bottleneck in the performance due to the tail loop. If the tail loop is unavoidable, we also vectorize it using RVV intrinsic instructions.

3.3 Experimental Setup

In this paper, we evaluate two popular CNN models. The first one is YOLOv3 [39], an object detection network, which features 107 layers of five different types, out of which 75 layers are convolutional. We profile the execution time of the convolutional layers

Table 1: Convolutional layers of the VGG-16 (top) and YOLOv3/20 layers (bottom) network models. IC = Input Channels, OC = Output Channels, IH = Input Height, IW = Input Width, OH = Output Height, OW = Output Width, KH= Kernel Height, KW = Kernel Width

Layers	IC	OC	IH,IW	OH,OW	KH,KW	stride
1	3	64	224	224	3	1
2	64	64	224	224	3	1
3	64	128	112	112	3	1
4	128	128	112	112	3	1
5	128	256	56	56	3	1
6,7	256	256	56	56	3	1
8	256	512	28	28	3	1
9,10	512	512	28	28	3	1
11	512	512	14	14	3	1
12,13	512	512	14	14	3	1
Layers	**IC**	**OC**	**IH,IW**	**OH,OW**	**KH,KW**	**stride**
1	3	32	608	608	3	1
2	32	64	608	304	3	2
3	64	32	304	304	1	1
4	64	64	304	304	3	1
5	64	128	304	152	3	2
6,8	128	64	152	152	1	1
7,9	64	128	152	152	3	1
10	128	256	152	76	3	2
11	256	128	76	76	1	1
12,14	128	256	76	76	3	1
13,15	256	128	76	76	1	1

of YOLOv3, as implemented in the Darknet framework, finding that the convolutional layers contribute ~96% of the total execution time. The second model is VGG-16 [41], an image classification model, which includes 25 layers, out of which 13 are convolutional and 3 are fully connected. Profiling VGG-16 within the Darknet framework, we find the convolutional layers contributing ~ 64% of the total execution time.

We evaluate the layers of YOLOv3 and VGG-16 network models from the Darknet [38] framework on a 768×576 pixels input image, using a batch size of 1, which is a common case for inference. As described above, we use a fork of gem5 for our experiments. To acquire feasible simulation times, we limit our evaluation to the first 20 layers of the YOLOv3 network, out of which 15 are convolutional layers. We provide details on the dimensions of the convolutional layers of VGG-16 and YOLOv3 in Table 1. We note that we use single-precision floating point numbers for the weights, and thus for all computations. We use the EPI-Builtins [17] to vectorize the Direct convolutional algorithm on RVV in a VLA way. We use the EPI fork of the LLVM [9] Clang cross-compiler version 17.0.0 for RVV, with -O3 optimization flag to compile all the three convolutional algorithms in our setup.

For our co-design study, we vary the maximum vector length from 512 to 4096 bits on RVV, incrementing in powers of 2. To analyze the impact of cache parameters, we increase the L2 cache size from 1MB up to 64 MB. We consider a constant latency of 20 cycles for all the L2 cache sizes.

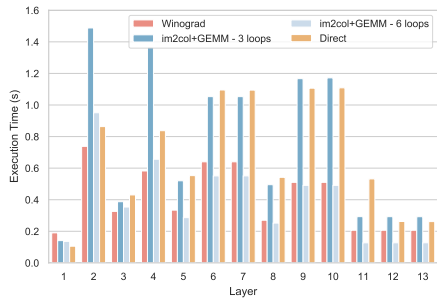

Figure 1: Comparative analysis for different algorithms for VGG-16 convolutional layers on RVV with gem5, with a vector length of 512 bits and 1MB of L2 cache.

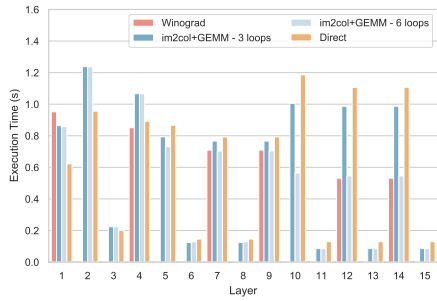

Figure 2: Comparative analysis for different algorithms for the first 15 convolutional layers of YOLOv3 on RVV with gem5, with a vector length of 512 bits and 1MB of L2 cache.

4 EVALUATION

In this section, we first showcase our findings for the best suitable algorithm for each layer in the CNN-based VGG-16 and YOLOv3 network models on RVV, on a single core. Subsequently, we present the results of our per-layer co-design study for both network models. In all experiments with gem5, we report the per-layer performance in terms of execution time in seconds. We then implement a predictor for algorithm selection, and showcase the performance-area tradeoffs for model serving on a 7nm RVV chip.

4.1 Performance Comparison of Convolution Implementations

We start our evaluation with a performance comparison of convolutions with the 3 different algorithms, i.e. Winograd, im2col+GEMM (with the two variants of GEMM), and Direct. We evaluate each convolutional layer of YOLO-v3 and VGG-16 on RVV using the gem5 simulator, for a fixed vector length of 512 bits, and an L2 cache size of 1MB. Figure 1 shows the per-layer performance for the VGG-16 network model. We observe that, for layers #1 and #2, where the input and output width/height are high (IH, IW, OH, OW), the Direct algorithm performs well, although the winner

algorithm for layer 2 is Winograd. For layers #3 to #13, Winograd, as well as the 6-loop implementation of im2col+GEMM perform better than all other algorithms. For layers #5 to #13, where the input and output matrices become skinny but the number of input and output channels (IC, OC) increase, the 6-loop im2col+GEMM variant prevails. Although Winograd reduces the computational complexity of the convolutional layer by reducing the number of multiplications, the increased numbers of input and output channels add transformation overheads to the Winograd algorithm, leading to its inferior performance, compared to im2col+GEMM. In the case of layer #1, Winograd underperforms compared to all other algorithms, as the number of input channels is too low for the algorithm to use the inter-tile parallelism approach described in [21].

Figure 2 shows the per-layer performance of the different algorithms for the first 15 convolutional layers of the YOLOv3 network model. The YOLOv3 network model has convolutional layers with kernel sizes 3×3 with stride 1 or 2 and 1×1 kernel sizes. We note that the Winograd algorithm is only appropriate for layers with kernel sizes of 3×3 and strides of 1, due to issues of numerical stability. We therefore only present results with Winograd for layers with these properties. Similar to the case of VGG-16, Direct offers superior performance for layer #1, where the input and output dimensions are high but the number of input channels is low. Additionally, the Winograd algorithm demonstrates high performance for all the layers where it is applicable and comparable performance to the 6-loop im2col+GEMM implementation, for many of these layers. The Direct algorithm offers high performance to layers #1-#3, where the input and output dimensions are high, but as the matrices become skinnier, for layers #5-#15, the 6-loop im2col+GEMM implementation prevails. It is noteworthy that the performance of the 3-loop and 6-loop implementation of im2col+GEMM is comparable for the first layers, but the 6-loop transformation proves beneficial to skinny matrices.

4.2 Co-designing Convolutions on Long Vector Architectures

In this section, we jointly explore the effect of the hardware parameters of vector architectures and the algorithm selection for the implementation of convolutional layers. As discussed, we focus on the vector length and the L2 cache size.

4.2.1 The effect of the vector length. We experiment with vector lengths ranging from 512 bits to 4096 bits, while keeping the L2 cache size fixed to 1MB, and observe the scalability of the different algorithms on the convolutional layers of VGG-16 and YOLOv3.

In Figure 3, we show the scalability of the different algorithms on the layers of VGG-16. The Winograd algorithm scales from ~1.3× to ~1.7× as we increase the vector length from 512 to 2048 bits. However, moving from 2048 bits to 4096 bits, we observe limited scaling, especially for skinny matrices. We attribute this behavior to the need for larger sub-block sizes to leverage longer vector lengths for the tuple multiplication in the Winograd algorithm. As a consequence, the block sizes for the input and output channels are reduced, requiring increased loop iterations for transformations and tuple multiplications, resulting in increased overhead despite the use of longer vector lengths.

On the other hand, the 3-loop im2col+GEMM variant scales from ~1.4× to ~3.5× when transitioning from vector lengths of 512 to 4096 bits. However, layers #6 and #7 exhibit no scalability beyond 2048 bits, and layer #8 experiences performance degradation beyond 2048 bits. We attribute this to increased pressure to the L2 cache, and, examining the L2 cache miss rate for these layers, we observe a very high cache miss rate of ~ 98% for vector lengths of 4096 bits. The 6-loop im2col+GEMM variant algorithm demonstrates scalability of ~1.4× up to ~2.1× for all layers as we increase the vector length from 512 bits to 4096 bits. The Direct algorithm demonstrates the best scalability for all layers, with performance improvements of ~2.4× - 5.8× transitioning from 512 bits to 4096 bits of a vector length. We do note, however, that the 6-loop im2col+GEMM variant can offer better performance than the Direct algorithm with vector lengths of 2048 bits for layers #6 to #13.

We conduct a similar analysis for YOLOv3 in Figure 4. The Winograd algorithm is applicable on a total of 6 convolutional layers, and these layers exhibit scaling between ~1.3× and ~1.6×, as we increase the vector length from 512 bits to 4096 bits, however, we observe no noticeable scalability as from 2048 bits to 4096 bits. The 3-loop im2col+GEMM variant scales between ~1.3× and ~3.5× for the YOLOv3 layers. On the other hand, the im2col+GEMM 6 loops kernel demonstrates scaling between ~1.3× and ~2.0× for the YOLOv3 layers. Similarly to the case of VGG-16, the Direct algorithm exhibits better and robust scalability, scaling between ~1.9× and ~4.6× for all layers. Moreover, the Direct algorithm outperforms the other algorithms for most layers, except for those involving skinnier matrices (i.e. layers #10, #12, and #14), where the im2col+GEMM variants offer better performance for vector lengths higher than 1024 bits.

4.2.2 The effect of the L2 cache size. We further experiment with the L2 cache size, as it can significantly reduce the pressure on the main memory. We consider L2 cache sizes of 1MB to 64MB, fixing the vector length at 512 and 4096 bits.

We showcase the scalability of the different algorithms for the L2 cache size, for the layers of VGG-16, in Figures 5 and 6, for vector lengths of 512 bits and 4096 bits respectively. For the case of Winograd, the algorithm scales ~1.3× - 1.5× for layers #1 to #4 when increasing the L2 cache from 1MB to 64MB, for the vector length of 512 bits, and ~1.3×-1.6×, for the vector length of 4096 bits. The number of input and output channels in this case is small and allows the algorithm to scale. For layers #5 to #13, we observe a scalability of ~1.2× - 1.5× as we increase the L2 cache size from 1MB to 16MB, but the algorithm does not benefit from further increasing the cache to 64MB, for any vector length.

The 3-loop im2col+GEMM algorithm scales well, with performance improvements of ~1.8× - 2.4× for the smaller vector length of 512 bits, as we scale the L2 cache from 1MB to 64MB, and achieving remarkably high performance for the 64MB cache. For the 4096-bit vector length, the algorithm benefits intensively from the 64MB cache for layers #4 to #10, which exhibit high L2 cache miss rates with longer vectors, scaling up to ~3.6×. However, for the more compute-intensive layers #11 to #13, the algorithm does not benefit from L2 caches larger than 16MB, and the performance does not scale further. On the other hand, the 6-loop im2col+GEMM cache-friendly variant benefits less from the larger L2 cache sizes,

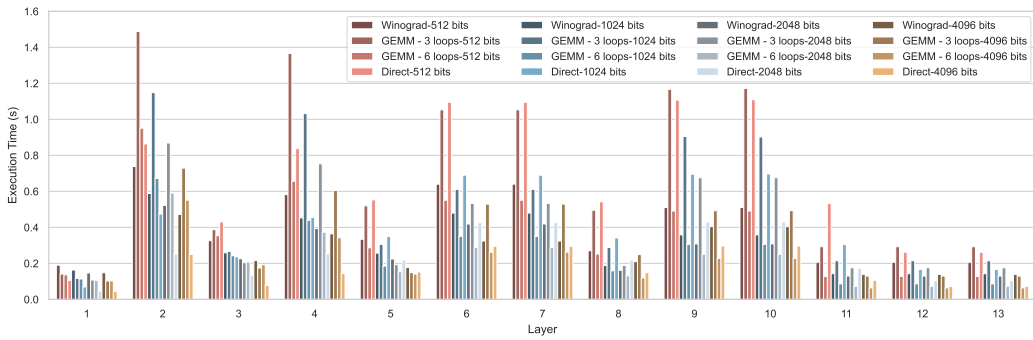

Figure 3: Scalability of different convolutional algorithms with vector lengths from 512 bits to 4096 bits for the convolutional layers of VGG-16, for an L2 cache of 1MB, on RVV with gem5.

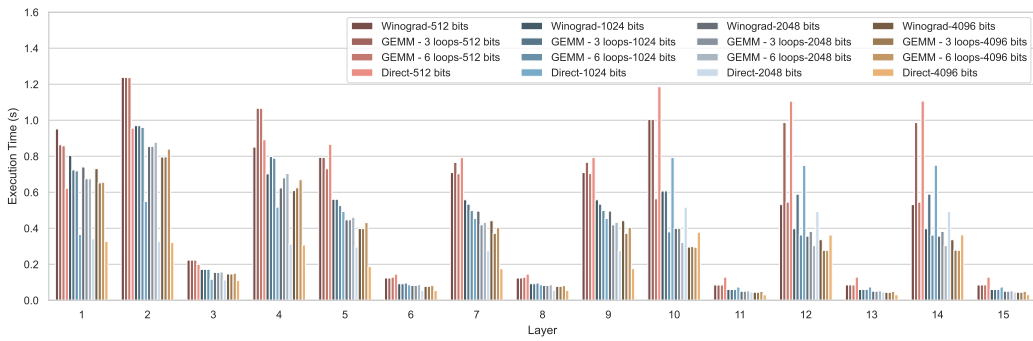

Figure 4: Scalability of different convolutional algorithms with vector lengths from 512 bits to 4096 bits for the first 15 convolutional layers of YOLOv3, for an L2 cache of 1MB, on RVV with gem5.

improving ~1.1×-~1.76× as we move from 1MB to 64MB, for both the smaller and larger vector length. Similarly to the 3-loop variant, it does not show any further improvement by increasing the cache size further than 16MB in the case of the more compute-intensive layers #11 to #13.

The Direct algorithm scales moderately as we increase the L2 cache size from 1MB to 16MB, with improvements of ~1.1× - 1.4× for the vector length of 512 bits, and ~1.2× for layers #2 to #4 and ~2.2× for layers #5 to #13 for the vector length of 4096 bits. The only exception is layer #2 for the case of 512 bits, which has high input and output dimensions, where the Direct algorithm benefits from the 64MB of cache.

We similarly examine the scalability of the layers of the YOLOv3 model, in Figures 7 and 8, for 512 bits and 4096 bits of vector lengths respectively. For the layers where the Winograd algorithm is applicable, we observe a scalability of ~1.2× - 1.3× and ~1.3× - 1.4× when increasing the cache size from 1MB to 64MB, for the case of 512 bits and 4096 bits of vector lengths respectively. Notably, the layers with higher input and output dimensions benefit more when increasing the cache size from 16MB to 64MB. For the case of the 3-loop im2col+GEMM variant, we observe scaling of ~1.1× - 2×, for both vector length sizes, but the last layers #10-#15 only lightly benefit from increasing the cache size from 16MB to 64MB. The 6-loop variant of im2col+GEMM shows similar scalability of ~1.1× - 2× as we increase the L2 cache size. The Direct algorithm, on the other hand, sees a significant performance boost from scaling the L2 cache size, especially for the case of the longer vector length of 4096 bits, with scalability of ~1.3×-~2.8×. We point out that layers #2 and #4 are those that benefit the most from increasing the cache size from 16MB to 64MB, however, the performance of the last layers #10 to #15 experience no further scalability from 16MB to 64MB of L2 cache. In both the cases of im2col+GEMM variants and the Direct algorithm, the layers #3, #6, and #8, with the smaller

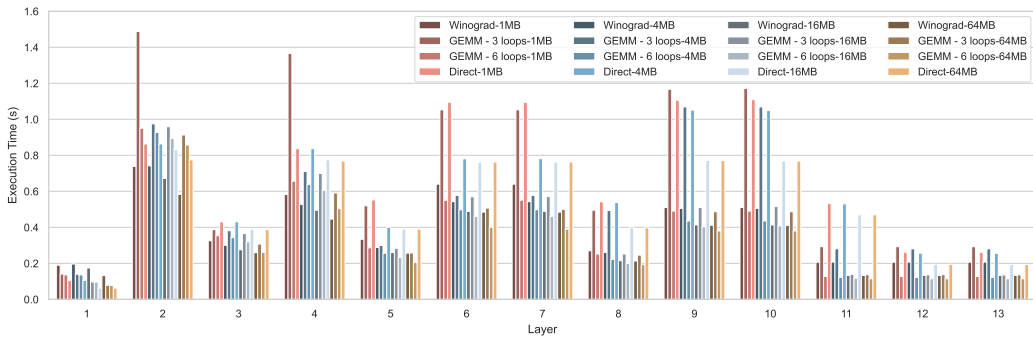

Figure 5: Scalability of different convolutional algorithms with L2 cache sizes from 1MB to 64MB for the convolutional layers of VGG-16, for a vector length of 512 bits, on RVV with gem5.

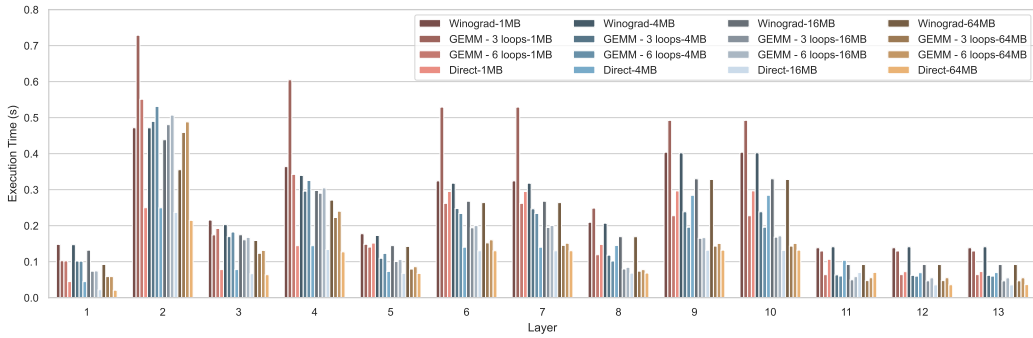

Figure 6: Scalability of different convolutional algorithms with L2 cache sizes from 1MB to 64MB for the convolutional layers of VGG-16, for a vector length of 4096 bits, on RVV with gem5.

kernel size of 1×1 and higher input/output dimensions benefit more from the increase of the L2 cache size.

In summary, we attribute the limited scalability of the Winograd algorithm to the fixed tile size, which does not fully utilize larger cache sizes, for both shorter and longer vector lengths. Both variants of im2col+GEMM benefit up to more than 2× from larger cache sizes, for any vector length, especially for layers with moderate numbers of input and output channels but high input and output dimensions. Conversely, the Direct algorithm benefits the most from the larger L2 cache when the vector length is long and the input and output dimensions are high, as is the case of the first convolutional layers of YOLOv3.

4.3 Algorithm Selection

Our results show that there is no single algorithm that minimizes the execution time across all layers. Therefore, to minimize the execution time of a full network model, a machine learning framework should be able to select the appropriate algorithm per layer at compile time or at runtime (autotuning). To this end, we construct a fast and accurate predictor that performs algorithm selection, depending on the layer dimensions, and the hardware configuration.

To predict the optimal algorithm among Winograd, the two variants of im2col+GEMM and Direct, we experiment with several classification models, available in the Python `scikit-learn` 1.3.2 package, including a Support Vector Machine, K-Nearest Neighbors, Naive Bayes, a Multilevel Perceptron, a Decision Tree, Gradient Boosting, and Random Forests, to predict the best performing algorithm per layer. We use as input parameters the vector length, the L2 cache size, the input channels, height, width, stride, and padding, the output channels, height, and width, and the kernel height and width, totaling 12 parameters, out of which 2 are relevant to the architecture and 10 are drawn from the convolution dimensions. The model outputs the algorithm predicted to perform the best. We

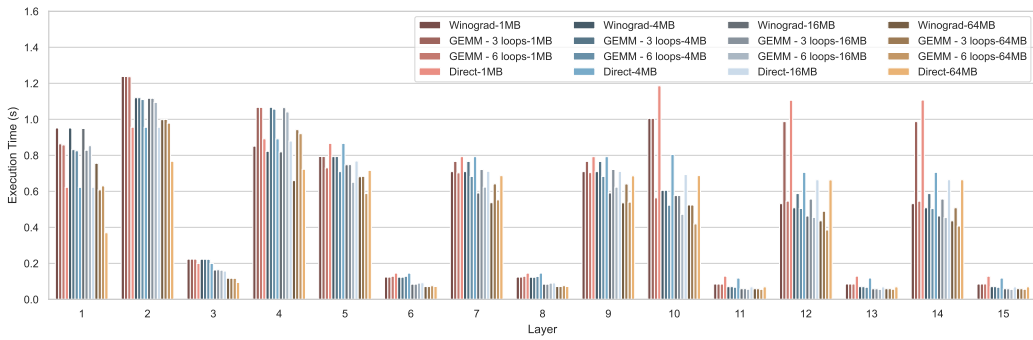

Figure 7: Scalability of different convolutional algorithms with L2 cache sizes from 1MB to 64MB for the first 15 convolutional layers of YOLOv3, for a vector length of 512 bits, on RVV with gem5.

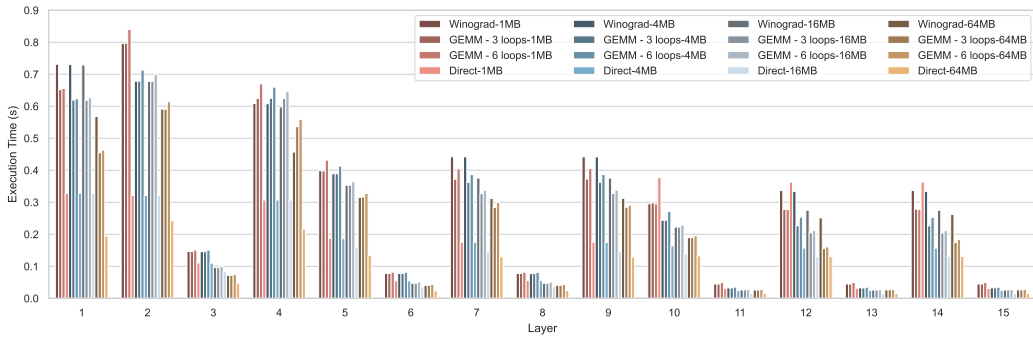

Figure 8: Scalability of different convolutional algorithms with L2 cache sizes from 1MB to 64MB for the first 15 convolutional layers of YOLOv3, for a vector length of 4096 bits, on RVV with gem5.

select random forests as the classifier with the best performance. We partition the data (of 448 data points) into 80% for training and 20% of testing, and use 5-fold cross-validation and shuffling, therefore all points in a testing set are not included in the corresponding training set, i.e. are previously unseen by the model. We tune the hyperparameters of the Random Forest classifier, resulting in a maximum tree depth of 10, and the usage of bootstrapping.

Our evaluation shows that the algorithm selection model achieves an average prediction accuracy of 92.8% (ranging from 91% to 96%) across the 5 cross-validation sets, indicating the model's proficiency in correctly selecting the best-performing algorithms under various contexts. Notably, within the 7.1% of misclassified layers/configurations, if the predicted algorithm is employed instead of the optimal one, the mean absolute percentage error in the performance of layers is only 20.4%.

To demonstrate the importance of our algorithm selection model, and to further evaluate its accuracy, we assess the inference time of VGG-16 and YOLOv3 in Figures 9 and 10 respectively. Specifically, we compare the execution time of each network model when always using the same algorithm for all layers, against using the *Optimal* algorithm per layer, or the *Predicted Optimal* algorithm per layer, namely the output of our algorithm selection model. For VGG-16, we observe that selecting the optimal algorithm per layer results in reduced execution compared to using any single algorithm, for all examined hardware configurations. Indicatively, selecting the optimal algorithm can improve the execution time by up to 1.85× over always using the Direct algorithm and up to 1.73× over using the 6-loop implementation of im2col+GEMM. Concerning the predictive ability of our algorithm selection model, the average error compared to the optimal configuration is 1.67% and the maximum error is 8.4%. For YOLOv3, selecting the optimal algorithm can improve the execution time by up to 1.33× and 2.11× over always using the Direct and 6-loop implementation of im2col+GEMM algorithms, respectively. The average error from the predicted

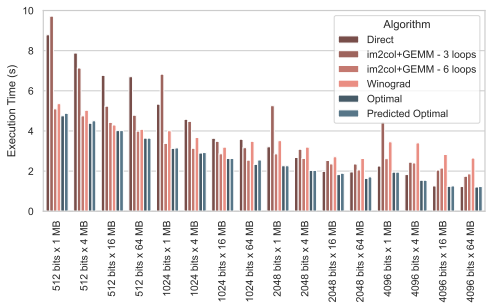

Figure 9: Execution time of VGG-16, for different vector lengths and L2 cache sizes, when a single algorithm is used for all layers (*Direct, im2col+GEMM - 3 loops, im2col+GEMM - 6 loops, Winograd*), compared against using the *Optimal* algorithm per layer, and using our algorithm selection model to predict the optimal algorithm per layer (*Predicted Optimal*).

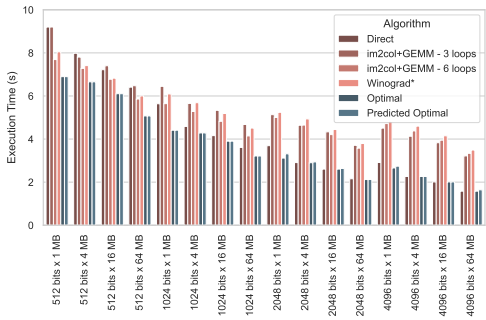

Figure 10: Execution time of YOLOv3 (first 15 layers), for different vector lengths and L2 cache sizes, when a single algorithm is used for all layers (*Direct, im2col+GEMM - 3 loops, im2col+GEMM - 6 loops, Winograd-uses im2col+GEMM for some layers), compared against using the *Optimal* algorithm per layer, and using our algorithm selection model to predict the optimal algorithm per layer (*Predicted Optimal*).**

optimal configuration against the optimal configuration is 0.95%, and the maximum error is 5.9%. We also point out that, even in configurations where our algorithm selection model introduces some error, it still manages to provide configurations that are better than always using a single algorithm to compute all layers.

4.4 Performance-Area Tradeoffs

Our analysis so far has shown that convolutional layers scale with longer vector lengths and larger cache capacity, for all the different algorithms. However, the longer vector lengths and larger L2 cache sizes require a larger chip area. To evaluate this performance-area tradeoff, as well as the attainable performance in a fixed area envelope, we first examine the scenario of a single model instance executing on an RVV core with an integrated VPU, like the one simulated in Section 3, implemented in 7nm FinFET technology. Building on the results in [26], we estimate the area of the core, VPU, and vector register file (VRF) in 22nm, based on the assumption that both the VPU and VRF area will increase proportionally to the vector length. In contrast, the core area will remain constant. Our analysis estimates that the chip area dedicated to the VPU and VRF consume ~28%, ~43%, ~60% and ~75% of total chip area, as we increase vector lengths from 512 bits to 4096 bits. We then scale the total area to a 7nm FinFET technology, which translates to a conservative estimate of a 6.2× increase in transistor density [8, 34]. We use PCacti [36] to estimate the area of L2 caches in 7nm.

We showcase the performance (in cycles) - area (in mm^2) tradeoff, accompanied by the Pareto curve, for VGG-16 in Figure 11. Due to space limitations, we omit the relevant figure for YOLOv3. It is evident that the impact of longer vector lengths on the area is minimal, but it is significant for performance, while the cache size has a more significant impact on the total area. The Pareto frontier consists of 7 data points, including all possible configurations with the smallest possible cache, i.e. 1 MB of cache, as well as all configurations with a vector length of 4096 bits. All the Pareto frontier points correspond to selecting the optimal algorithm per layer. The Pareto-optimal point for both VGG-16 and YOLOv3 is given by the configuration with 2048 bits and 1MB of L2 cache, with a total area of $2.35mm^2$. Using the optimal algorithm per layer results in 1.18× - 1.6× better performance compared to using a single algorithm for YOLOv3, and in 1.26×-2.32× better performance for VGG-16. Inversely, the Direct algorithm would require an area of $3.07mm^2$, i.e. 30% more area, to achieve the same level of performance both for YOLOv3 and VGG16, while im2col+GEMM can only achieve the same performance for YOLOv3 at $13.6mm^2$.

We then consider the case of a multi-core RVV chip. We consider configurations with 1, 4, 16, and 64 cores, of 512 up to 4096 bits of vector lengths, with a shared L2 cache of 1, 4, 16, 64, and 256 MB, at 7nm, as a realistic server in a model-serving context, resulting in 200 different hardware configurations. To simplify our analysis, we consider the existence of some static cache partitioning mechanism, e.g. similar to Intel CAT, which grants isolated cache ways to each hosted application. We also assume that the memory bandwidth does not become a bottleneck in this system, which is known to be the case for some systems with high-bandwidth memory [30]. On each of these configurations, we co-locate from 1 up to 64 identical instances of each network model, with the assumption that the cores and L2 cache do not become oversubscribed. We then analyze the tradeoff between the achieved throughput, in terms of Images/Cycle, and the required area, for VGG-16, in Figure 12. We observe that by co-locating multiple instances of VGG-16, we achieve an increase in throughput that is equivalent to the increase in area, as we add a larger cache size, a longer vector length, or

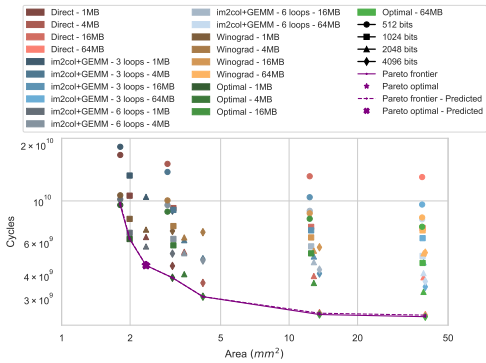

Figure 11: Performance-area tradeoff and Pareto frontier for a single instance of VGG-16 on an RVV chip at 7nm.

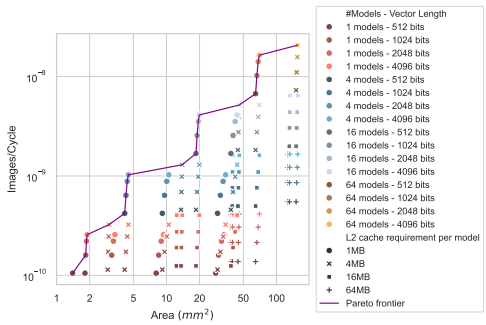

Figure 12: Throughput-area tradeoff and Pareto frontier for multiple instances of VGG-16 on an RVV chip at 7nm, using the optimal algorithm per layer.

additional cores. We highlight that all points of the Pareto frontier correspond to co-locating as many models as possible with the lowest possible L2 cache per model (1MB or 4MB at most). Such a configuration is enabled by the presence of high external memory bandwidth. Notably, though, such configurations will increase the energy consumption on external memory accesses. Finally, our analysis reveals that for a hardware configuration of 64 cores, 256 MB of cache, and vector units of 4096 bits, co-locating model instances with the optimal algorithm per layer leads to improved overall throughput by 1.16×, compared to using the best-performing algorithm (Direct), across all layers.

5 CONCLUSION

In this paper, we explore CNN co-design involving three distinct algorithms: direct, im2col+GEMM (two variants), and Winograd, on the convolutional layers of two CNN models i.e., YOLOv3 and VGG-16, with hardware parameter tuning for the RVV architecture, targeting model serving of CNNs. Our co-design exploration focused on tuning the vector length from 512 bits to 4096 bits, and the L2 cache size from 1MB to 64MB. Our study shows that the choice of the best algorithm depends on several parameters, including the kernel size, the dimensions of the activations, the vector length, and the L2 cache size. To select the best algorithm for each layer we build a Random Forest classifier, resulting in an average of 92.8% prediction accuracy, with inference time predictions showing at most 10% of relative errors. Finally, we analyze performance/area tradeoffs in the case of a single, as well as multiple model instances, showing that carefully selecting the algorithm per layer allows for higher performance in a reduced area. Coupled with model co-location, algorithm selection leads to increased throughput per area, highlighting the need for co-design in the context of model serving.

In the future, we aim to parameterize algorithmic optimizations, and include more hardware features in our exploration, enhancing our search space. We will also consider alternative neural network architectures and additional computational kernels, such as point-wise and depth-wise convolutions and attention mechanisms.

ACKNOWLEDGMENTS

This work has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 956702 (eProcessor), under Framework Partnership Agreement No. 800928 and Specific Grant Agreement No. 101036168 (EPI SGA2), and under grant agreement No. 101034126 (The European PILOT). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Sweden, Greece, Italy, France, Germany. Additionally, this work has received funding from the project PRIDE from the Swedish Foundation for Strategic Research with reference number CHI19-0048, and the project P4PIM from the Swedish Research Council (VR) with project ID 2020-04892. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725.

REFERENCES

[1] [n.d.]. plct-gem5. https://github.com/plctlab/plct-gem5
[2] [n.d.]. RISC-V Vector. https://github.com/riscv/riscv-v-spec/releases
[3] [n.d.]. Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim.
[4] 2015. Whitepaper GPU-Based Deep Learning Inference : A Performance and Power Analysis.
[5] Guillermo Alaejos, Adrián Castelló, Héctor Martínez, Pedro Alonso-Jordá, Francisco D Igual, and Enrique S Quintana-Ortí. 2023. Micro-kernels for portable and efficient matrix multiplication in deep learning. *The Journal of Supercomputing* 79, 7 (2023), 8124–8147.
[6] Syed Asad Alam, Andrew Anderson, Barbara Barabasz, and David Gregg. 2022. Winograd Convolution for Deep Neural Networks: Efficient Point Selection. *ACM Trans. Embed. Comput. Syst.* (mar 2022). Just Accepted.
[7] BentoML. [n.d.]. BentoML Docs: Concurrency. https://docs.bentoml.com/en/latest/guides/concurrency.html
[8] Mark Bohr. 2017. 22FFL technology. (2017). https://en.wikichip.org/w/images/e/e1/22FFL-2017.pdf
[9] BSC. 2023. LLVM EPI Compiler. https://ssh.hca.bsc.es/epi/ftp/
[10] Nvidia Corporation. [n.d.]. Triton Inference Server: Architecture: Concurrent Model Execution. https://docs.nvidia.com/deeplearning/triton-inference-server/archives/triton_inference_server_1150/user-guide/docs/architecture.html#concurrent-model-execution
[11] Francesco Petrogalli Dan Andrei Iliescu. [n.d.]. Arm Scalable Vector Extension and application to Machine Learning. https://developer.arm.com/solutions/hpc/

resources/hpc-white-papers/arm-scalable-vector-extensions-and-application-to-machine-learning
[12] Manuel F. Dolz, Sergio Barrachina Mir, Hector Martinez, Adrián Castelló, Antonio Maciá, Germán Fabregat, and Andrés Tomás. 2023. Performance–energy trade-offs of deep learning convolution algorithms on ARM processors. *The Journal of Supercomputing* 79 (01 2023).
[13] Manuel F. Dolz, Adrián Castelló, and Enrique S. Quintana-Ortí. 2022. Towards Portable Realizations of Winograd-based Convolution with Vector Intrinsics and OpenMP. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 39–46.
[14] Manuel F Dolz, Héctor Martınez, Pedro Alonso-Jordá, Adrián Castelló, and Enrique S Quintana-Ortı. [n.d.]. Parallel and Vectorised Winograd Convolutions for Multi-core Processors. ([n. d.]).
[15] Manuel F Dolz, Héctor Martínez, Adrián Castelló, Pedro Alonso-Jordá, and Enrique S Quintana-Ortí. 2023. Efficient and portable Winograd convolutions for multi-core processors. *The Journal of Supercomputing* (2023), 1–22.
[16] Manuel F. Dolz, Héctor Martínez, Pedro Alonso, and Enrique S. Quintana-Ortí. 2022. Convolution Operators for Deep Learning Inference on the Fujitsu A64FX Processor. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 1–10.
[17] Roger Ferrer. 2022. epi-builtins-ref. https://repo.hca.bsc.es/gitlab/rferrer/epi-builtins-ref
[18] Evangelos Georganas and Kalamkar. 2021. Tensor processing primitives: a programming abstraction for efficiency and portability in deep learning workloads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[19] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (may 2008).
[20] Sonia Rani Gupta, Nikela Papadopoulou, and Miquel Pericàs. 2023. Challenges and Opportunities in the Co-Design of Convolutions and RISC-V Vector Processors. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 1550–1556.
[21] Sonia Rani Gupta, Nikela Papadopoulou, and Miquel Pericàs. 2023. Accelerating CNN inference on long vector architectures via co-design. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 145–155.
[22] Yunxiang Hu, Yuhao Liu, and Zhuovuan Liu. 2022. A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC. In *2022 14th International Conference on Computer Research and Development (ICCRD)*. 100–107.
[23] European Processor Initiative. 2019. V for vector: software exploration of the vector extension of RISC-V. https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/
[24] Marc Jordà, Pedro Valero-Lara, and Antonio J. Peña. 2019. Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs. *IEEE Access* 7 (2019), 70461–70473.
[25] Vasilios Kelefouras and Georgios Keramidas. 2022. Design and Implementation of 2D Convolution on x86/x64 Processors. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3800–3815.
[26] Cristóbal Ramírez Lazo, Enrico Reggiani, Carlos R. Morales, Roger Figueras Bagu'e, Luis Alfonso Villa Vargas, Marco Antonio Ramírez Salinas, Mateo Valero Cortés, Osman Sabri Unsal, and Adrián Cristal. 2021. Adaptable Register File Organization for Vector Processors. *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2021), 786–799.
[27] Rui Li and Xu. 2021. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 928–942.
[28] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing {CNN} Model Inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1025–1040.
[29] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2014. Fast Training of Convolutional Networks through FFTs. arXiv:cs.CV/1312.5851
[30] John D. McCalpin. 2023. Bandwidth Limits in the Intel Xeon Max (Sapphire Rapids with HBM) Processors. In *High Performance Computing*, Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse (Eds.). Springer Nature Switzerland, Cham, 403–413.
[31] Inc. Meta Platforms. [n.d.]. Building Meta's GenAI Infrastructure. https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/
[32] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. 2021. A survey of deep learning on CPUs: opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems* 33, 10 (2021), 5095–5115.
[33] Sparsh Mittal and Shraiysh Vaishay. 2019. A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture* 99 (2019), 101635.
[34] Phil Oldiges, Reinaldo A Vega, Henry K Utomo, Nick A Lanzillo, Thomas Wassick, Juntao Li, Junli Wang, and Ghavam G Shahidi. 2020. Chip power-frequency scaling in 10/7nm node. *IEEE Access* 8 (2020), 154329–154337.
[35] Jongseok Park, Kyungmin Bin, and Kyunghan Lee. 2022. MGEMM: Low-Latency Convolution with Minimal Memory Overhead Optimized for Mobile Devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22)*. Association for Computing Machinery, New York, NY, USA, 222–234.
[36] Pcacti. [n.d.]. SPORT lab. https://sportlab.usc.edu/downloads/packages/
[37] Felizia Quetscher. [n.d.]. A comprehensible explanation of the dimensions in CNNs. https://towardsdatascience.com/a-comprehensible-explanation-of-the-dimensions-in-cnns-841dba49df5e
[38] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.
[39] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *arXiv* (2018).
[40] Alexandre de Limas Santana, Adrià Armejach, and Marc Casas. 2023. Efficient Direct Convolution Using Long SIMD Instructions. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 342–353.
[41] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:cs.CV/1409.1556
[42] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. 2019. A Unified Optimization Approach for CNN Model Inference on Integrated GPUs. arXiv:cs.DC/1907.02154
[43] Pengyu Wang, Weiling Yang, Jianbin Fang, Dezun Dong, Chun Huang, Peng Zhang, Tao Tang, and Zheng Wang. 2023. Optimizing Direct Convolutions on ARM Multi-Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 70, 13 pages.
[44] Shihang Wang, Jianghan Zhu, Qi Wang, Can He, and Terry Tao Ye. 2021. Customized Instruction on RISC-V for Winograd-Based Convolution Acceleration. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 65–68.
[45] Rui Xu, Sheng Ma, and Yang Guo. 2018. Performance Analysis of Different Convolution Algorithms in GPU Environment. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 1–10.
[46] Aleksandar Zlateski, Zhen Jia, Kai Li, and Frédo Durand. 2018. FFT Convolutions are Faster than Winograd on Modern CPUs, Here is Why. *ArXiv* abs/1809.07851 (2018). https://api.semanticscholar.org/CorpusID:52339177