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

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

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
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

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

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 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.

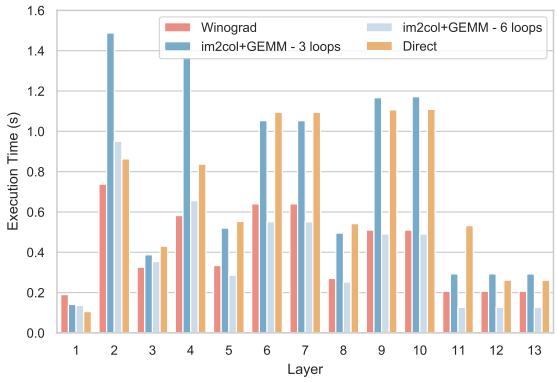

Figure 1: Comparative analysis for different algorithms for VGG-16 convolutional layers on RVV with gem5, with a vector length of 512 bits and 1MB of L2 cache.

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.

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

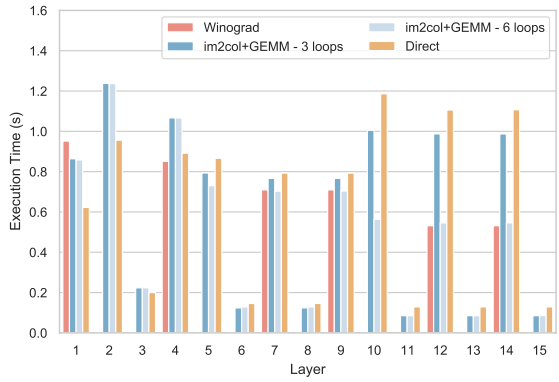

Figure 2: Comparative analysis for different algorithms for the first 15 convolutional layers of YOLOv3 on RVV with gem5, with a vector length of 512 bits and 1MB of L2 cache.

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.

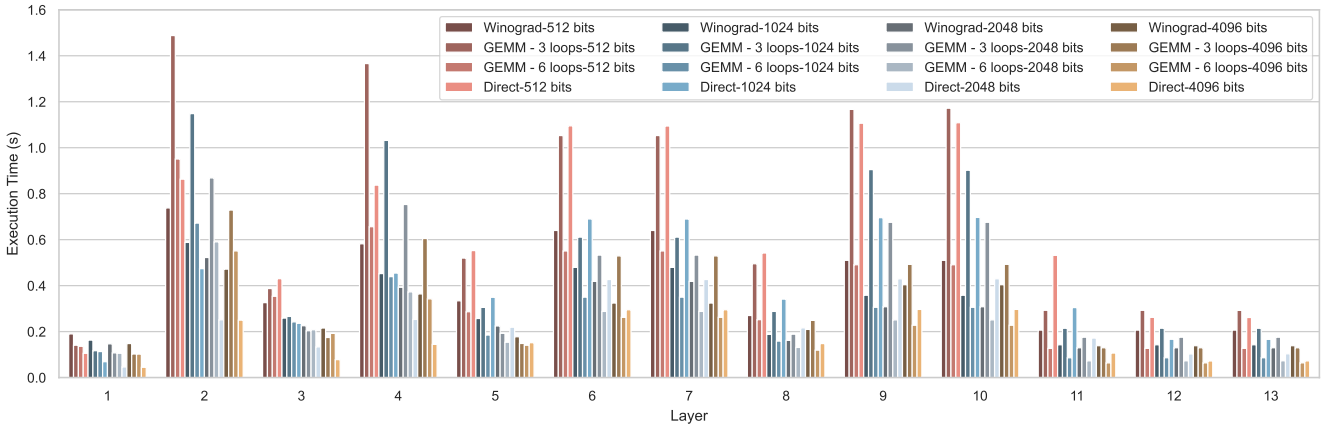

Figure 3: Scalability of different convolutional algorithms with vector lengths from 512 bits to 4096 bits for the convolutional layers of VGG-16, for an L2 cache of 1MB, on RVV with gem5.

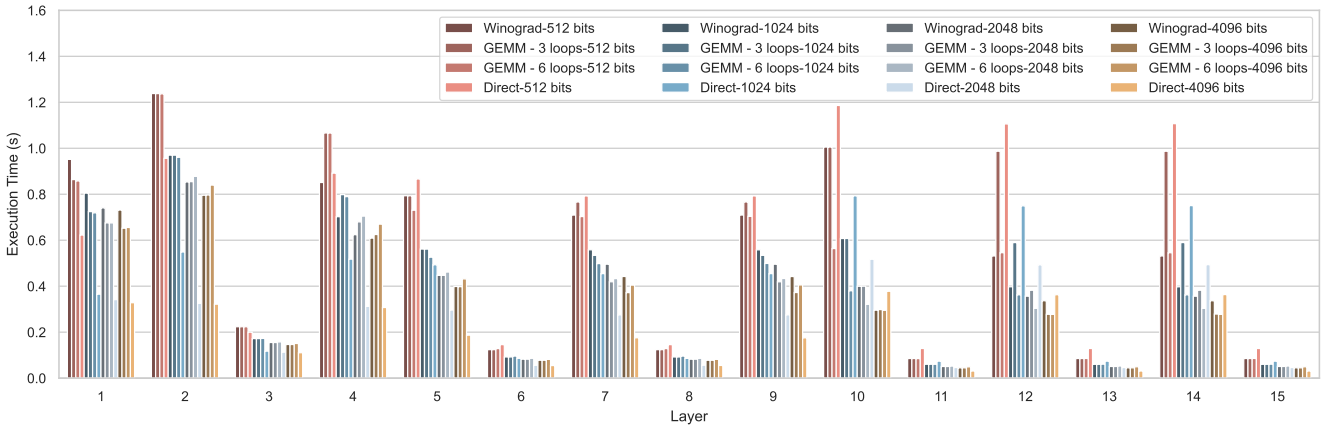

Figure 4: Scalability of different convolutional algorithms with vector lengths from 512 bits to 4096 bits for the first 15 convolutional layers of YOLOv3, for an L2 cache of 1MB, on RVV with gem5.

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, 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

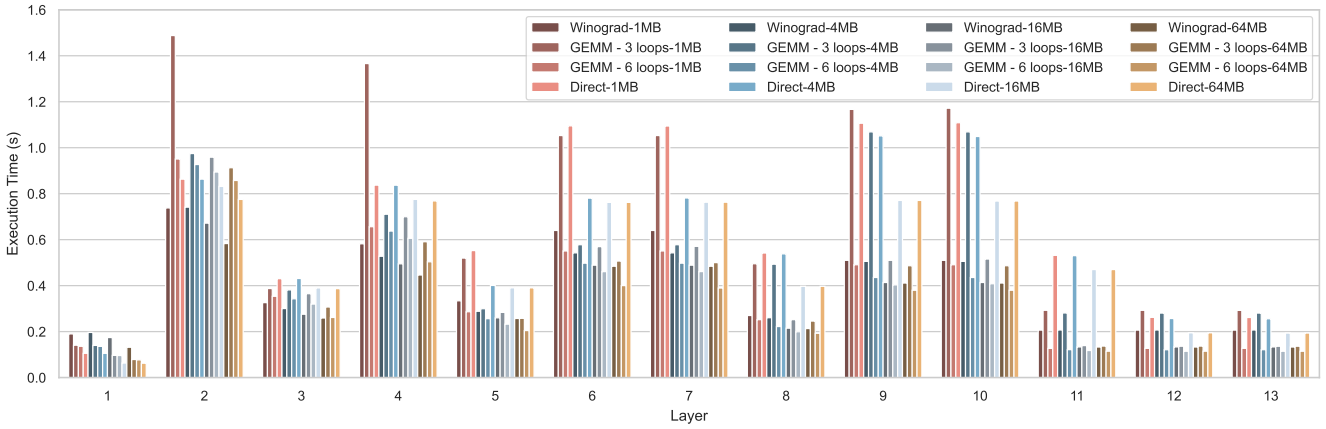

Figure 5: Scalability of different convolutional algorithms with L2 cache sizes from 1MB to 64MB for the convolutional layers of VGG-16, for a vector length of 512 bits, on RVV with gem5.

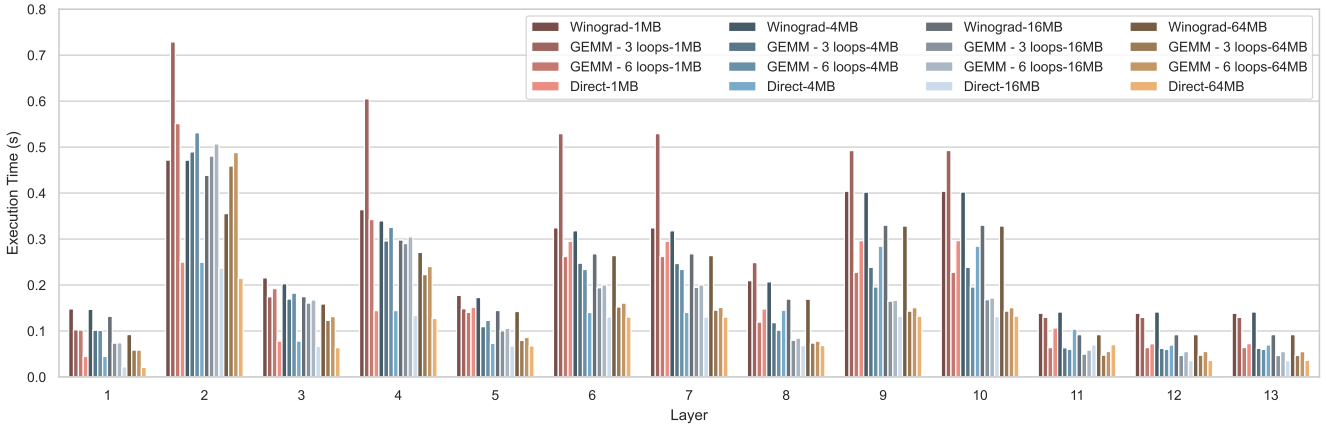

Figure 6: Scalability of different convolutional algorithms with L2 cache sizes from 1MB to 64MB for the convolutional layers of VGG-16, for a vector length of 4096 bits, on RVV with gem5.

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 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

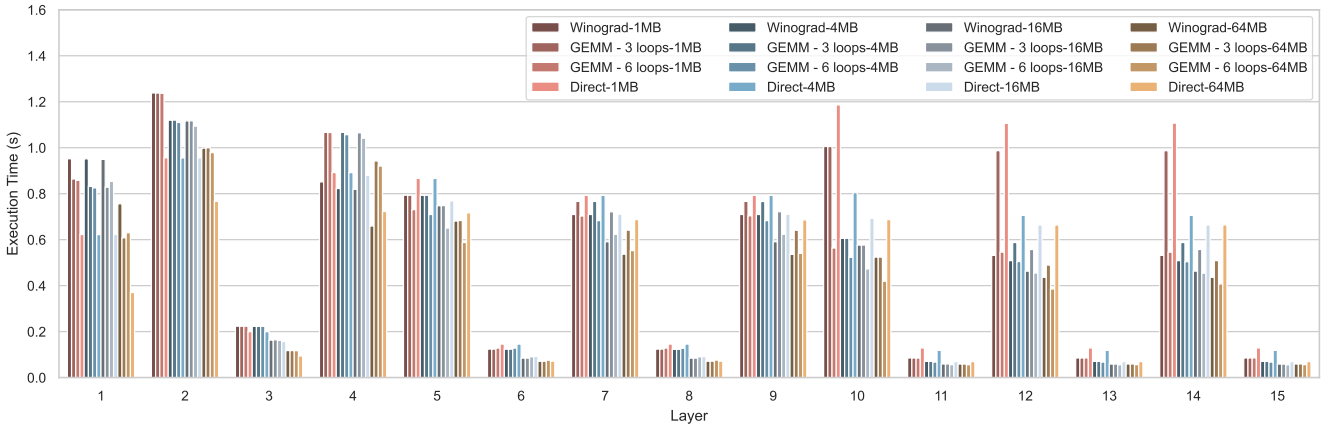

Figure 7: Scalability of different convolutional algorithms with L2 cache sizes from 1MB to 64MB for the first 15 convolutional layers of YOLOv3, for a vector length of 512 bits, on RVV with gem5.

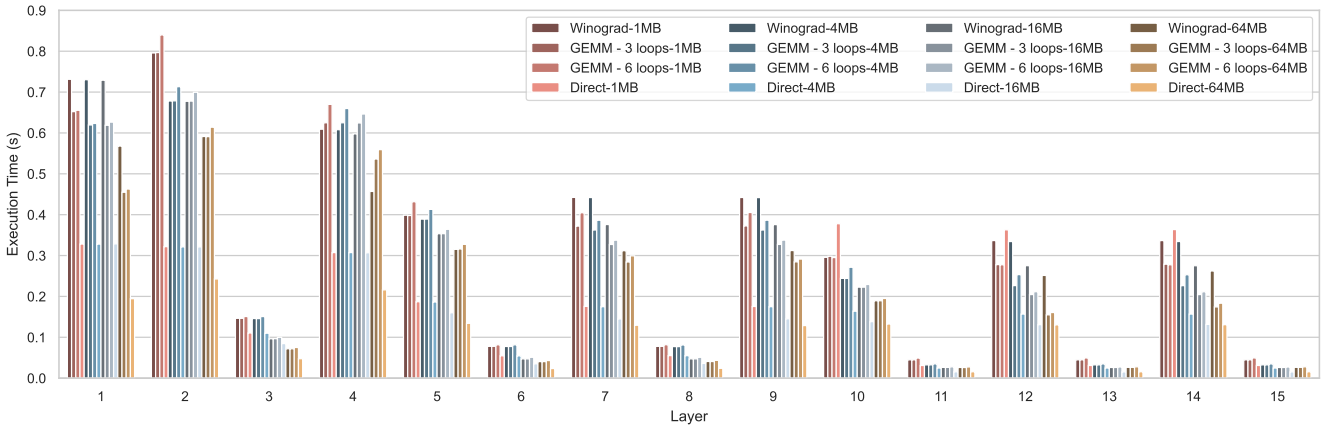

Figure 8: Scalability of different convolutional algorithms with L2 cache sizes from 1MB to 64MB for the first 15 convolutional layers of YOLOv3, for a vector length of 4096 bits, on RVV with gem5.

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 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

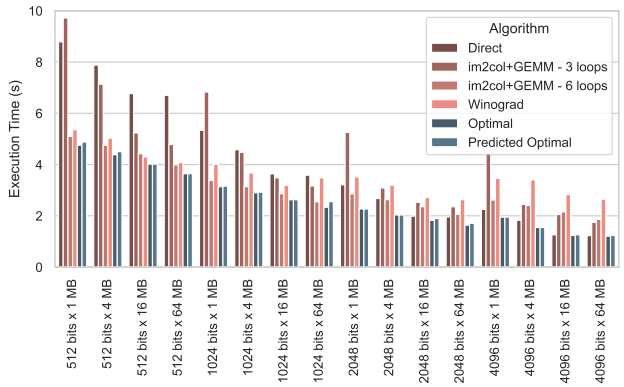

Figure 9: Execution time of VGG-16, for different vector lengths and L2 cache sizes, when a single algorithm is used for all layers (*Direct, im2col+GEMM - 3 loops, im2col+GEMM - 6 loops, Winograd*), compared against using the *Optimal* algorithm per layer, and using our algorithm selection model to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

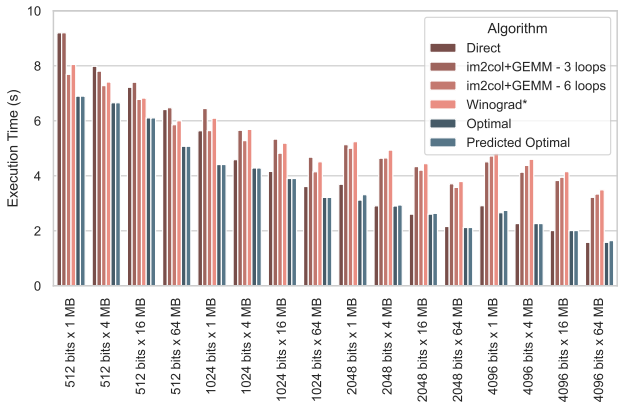

Figure 10: Execution time of YOLOv3 (first 15 layers), for different vector lengths and L2 cache sizes, when a single algorithm is used for all layers (*Direct, im2col+GEMM - 3 loops, im2col+GEMM - 6 loops, Winograd-uses im2col+GEMM for some layers), compared against using the *Optimal* algorithm per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

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 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].

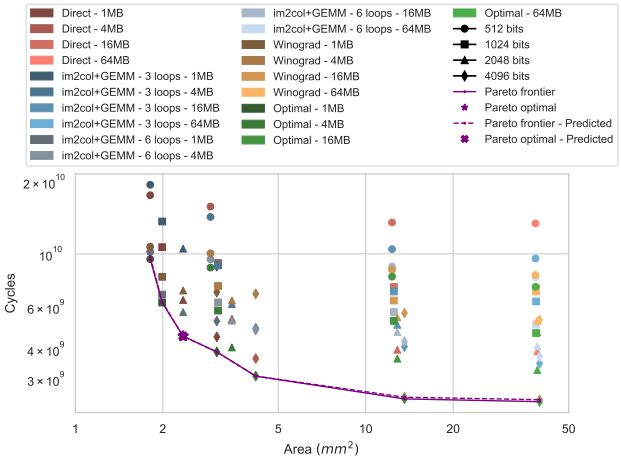

Figure 11: Performance-area tradeoff and Pareto frontier for a single instance of VGG-16 on an RVV chip at 7nm.

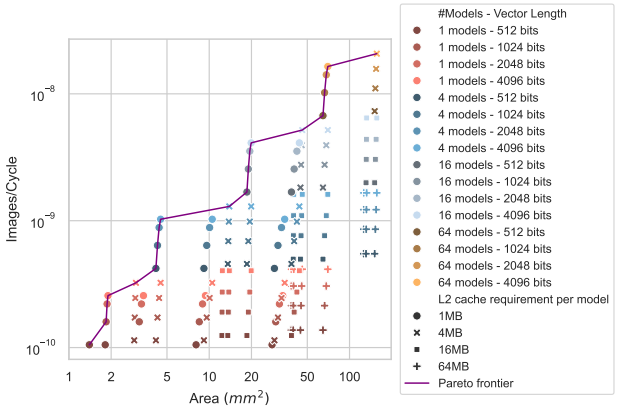

Figure 12: Throughput-area tradeoff and Pareto frontier for multiple instances of VGG-16 on an RVV chip at 7nm, using the optimal algorithm per layer.

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 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