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

Fig. 2: Optimized 3-loop implementation of GEMM

```
1: $i \leftarrow 0$ , $j \leftarrow 0$, $k \leftarrow 0$
2: long int $gvl$;
3: for $j1 \leftarrow 0$, $j1 < N$, $j1+=blockN$ do
4:     for $k1 \leftarrow 0$, $k1 < K$, $k1+=blockK$ do
5:         Pack MatrixB
6:         for $i1 \leftarrow 0$, $i1 < M$, $i1+=blockM$ do
7:             Pack MatrixA
8:             for $j \leftarrow 0$, $j < blockN$, do
9:                 $gvl \leftarrow vsetvl(blockN - j)$
10:                for $i \leftarrow 0$, $i < blockM$, $i+=U$ 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:                    $VC[i:i+U] \leftarrow C[i:i+U, j:j+gvl]$
15:                    for $k \leftarrow 0$, $k < blockK$, $k++$ do
16:                        Prefetch packed $B$ matrix into L1 cache
17:                        Prefetch packed $A$ matrix into L1 cache
18:                        $VB \leftarrow packedB[k, j:j+gvl]$
19:                        for $it \leftarrow 0$, $it < U$, $it++$ do
20:                            $tmp$ = $alpha \times packedA[it,k]$
21:                            $Vtmp \leftarrow tmp$ //broadcast
22:                            $VC[it] \leftarrow vfmacc$ ($VC[it]$,$Vtmp$,$VB$,$gvl$)
23:                        end for
24:                    end for
25:                    $C[i:i+U, j:j+gvl] \leftarrow VC[i:i+U]$
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.

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

```
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

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

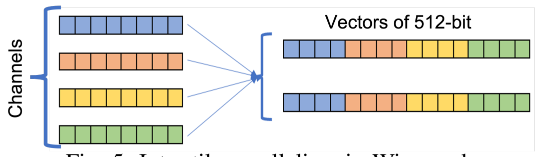

Fig. 5: Inter-tile parallelism in Winograd

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 ~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 $14\times$ 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.

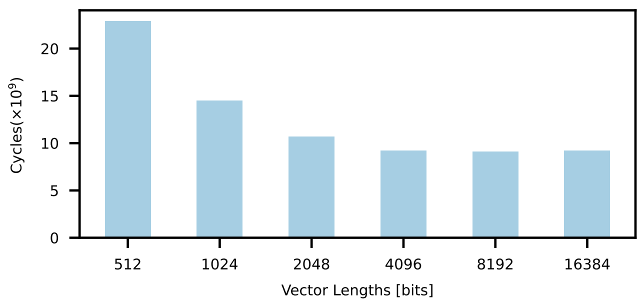

Fig. 6: Impact of vector lengths on RISC-VV@gem5 for YOLOv3 (20 layers), for constant L2 cache 1MB and 8 vector lanes.

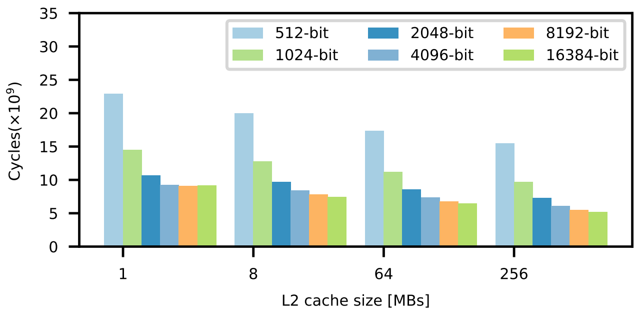

Fig. 7: Impact of the L2 cache size on RISC-VV@gem5 for YOLOv3 (20 layers), for 8 vector lanes.

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.

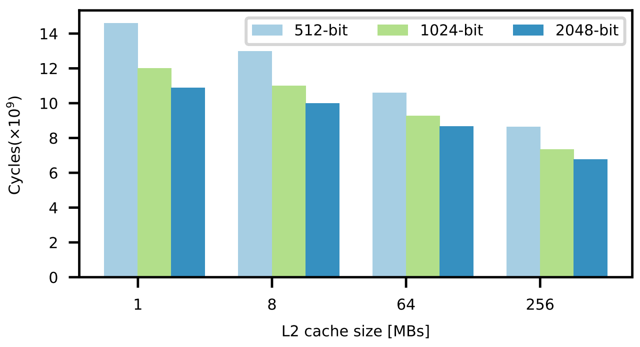

Fig. 8: Impact of vector lengths and L2 cache size on ARM-SVE@gem5 for YOLOv3 (20 layers).

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

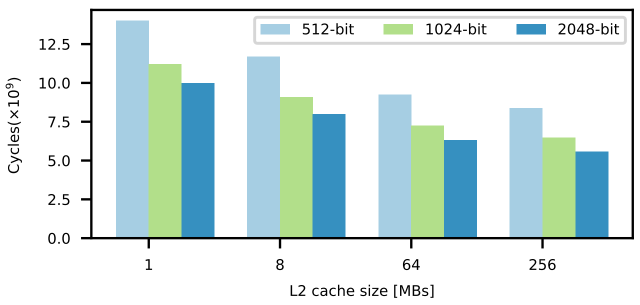

Fig. 9: Impact of vector lengths and L2 cache size with Winograd on ARM-SVE@gem5 for YOLOv3 (20 layers).

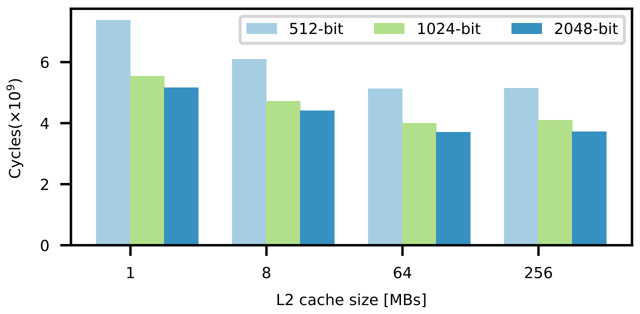

Fig. 10: Impact of vector lengths and L2 cache size with Winograd on ARM-SVE@gem5 for VGG16.

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

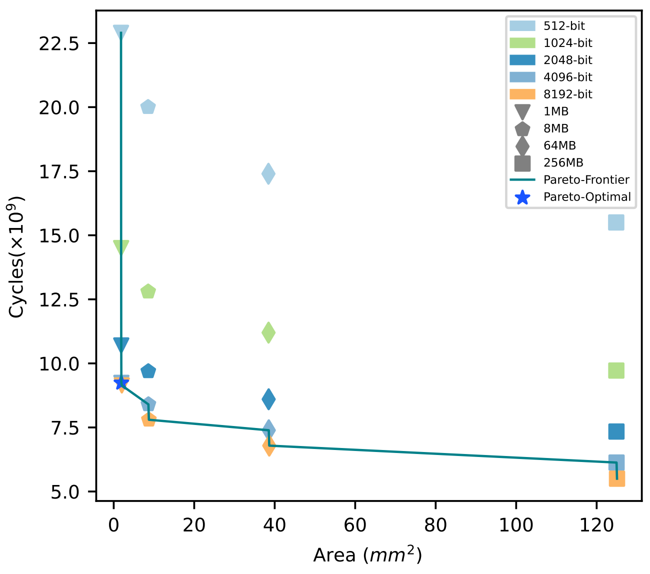

Fig. 11: Pareto frontier for the performance and area of a RISC-VV chip implemented at 7nm, with increasing vector lengths and L2 cache sizes, for YOLOv3 (20 layers).

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.