Advancing Direct Convolution using Convolution Slicing Optimization and ISA Extensions

VICTOR FERRARI*, Institute of Computing - UNICAMP, Brazil
RAFAEL SOUSA*, Institute of Computing - UNICAMP, Brazil
MARCIO PEREIRA, Institute of Computing - UNICAMP, Brazil
JOÃO P. L. DE CARVALHO, University of Alberta, Canada
JOSÉ NELSON AMARAL, University of Alberta, Canada
JOSÉ MOREIRA, IBM Research, United States of America
GUIDO ARAUJO, Institute of Computing - UNICAMP, Brazil

Convolution is one of the most computationally intensive operations that must be performed for machine-learning model inference. A traditional approach to computing convolutions is known as the Im2Col + BLAS method. This paper proposes SConv: a direct-convolution algorithm based on an MLIR/LLVM code-generation toolchain that can be integrated into machine-learning compilers. This algorithm introduces: (a) Convolution Slicing Analysis (CSA) — a convolution-specific 3D cache-blocking analysis pass that focuses on tile reuse over the cache hierarchy; (b) Convolution Slicing Optimization (CSO) — a code-generation pass that uses CSA to generate a tiled direct-convolution macro-kernel; and (c) Vector-Based Packing (VBP) — an architecture-specific optimized input-tensor packing solution based on vector-register shift instructions for convolutions with unitary stride. Experiments conducted on 393 convolutions from full ONNX-MLIR machine-learning models indicate that the elimination of the Im2Col transformation and the use of fast packing routines result in a total packing time reduction, on full model inference, of 2.3x – 4.0x on Intel x86 and 3.3x – 5.9x on IBM POWER10. The speed-up over an Im2Col + BLAS method based on current BLAS implementations for end-to-end machine-learning model inference is in the range of 11% – 27% for Intel x86 and 11% – 34% for IBM POWER10 architectures. The total convolution speedup for model inference is 13% – 28% on Intel x86 and 23% – 39% on IBM POWER10. SConv also outperforms BLAS GEMM, when computing pointwise convolutions in more than 82% of the 219 tested instances.

CCS Concepts: • **Software and its engineering** → **Source code generation**; • **Computing methodologies** → *Neural networks*; • **General and reference** → *Performance*.

Additional Key Words and Phrases: Convolution, Packing, Cache Blocking, Compilers

ACM Reference Format:
Victor Ferrari, Rafael Sousa, Marcio Pereira, João P. L. de Carvalho, José Nelson Amaral, José Moreira, and Guido Araujo. 2023. Advancing Direct Convolution using Convolution Slicing Optimization and ISA Extensions. *ACM Trans. Arch. Code Optim.* 1, 1 (September 2023), 26 pages. https://doi.org/XXXXXXX.XXXXXXX

*Both authors contributed equally to this research.

Authors' addresses: Victor Ferrari, v187890@dac.unicamp.br, Institute of Computing - UNICAMP, Campinas, SP, Brazil; Rafael Sousa, rafael.sousa@ic.unicamp.br, Institute of Computing - UNICAMP, Campinas, SP, Brazil; Marcio Pereira, mpereira@ic.unicamp.br, Institute of Computing - UNICAMP, Campinas, SP, Brazil; João P. L. de Carvalho, joao.carvalho@ualberta.ca, University of Alberta, Edmonton, AB, Canada; José Nelson Amaral, jamaral@ualberta.ca, University of Alberta, Edmonton, AB, Canada; José Moreira, jmoreira@us.ibm.com, IBM Research, Yorktown Heights, NY, United States of America; Guido Araujo, guido@unicamp.br, Institute of Computing - UNICAMP, Campinas, SP, Brazil.

1544-3566/2023/9-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Convolution is a mathematical tensor operation commonly used for image processing and in *Convolutional Neural Network (CNN)* models. Convolution is computationally intensive, and it accounts for most of the execution time of a typical CNN model. Thus, many approaches have been proposed to speed up convolution [1, 4, 6, 8, 17, 35]. The most well-known approach to convolution relies on a sequence of two major steps: (a) Im2Col operation to pack the input image and filters into matrices; and (b) GEMM (Generic Matrix Multiplication) to compute the final convolution result. Depending on the convolution layer, the matrices resulting from Im2Col can become very large, potentially leading to poor memory-hierarchy performance if GEMM is not well-optimized. To address the issue, this approach relies on efficient cache-optimized implementations of GEMM available from libraries such as Eigen [10], BLAS [33] and BLIS [29]. These implementations divide the matrices into smaller blocks called tiles that are suitable to the cache hierarchy and then run a machine-dependent GEMM *micro-kernel* [29] on each tiled block.

In previous work, direct convolution outperforms the traditional Im2Col followed by GEMM under certain conditions [2, 35]. This paper presents SConv: a direct-convolution algorithm that uses architectural information to improve convolution's cache utilization and ISA extensions to accelerate data packing and computation, suitable for SIMD architectures. This paper also evaluates SConv in full machine-learning model inference on two architectures (Intel x86 and IBM POWER). The algorithm can leverage Instruction-Set Architecture (ISA) acceleration extensions (*e.g.* IBM POWER10 MMA) to compete with optimization libraries such as BLAS.

Compared with Im2Col followed by GEMM, SConv reduces the data-manipulation overhead and uses a specific cache-tiling technique to reduce the number of cache misses and improve performance. The traditional approach (a) applies Im2Col to the input image and filter set, potentially creating large matrices; (b) applies cache tiling to the matrices to determine the best tile size and their allocation at each cache level (as in [9]); (c) changes the data layout of a tile with a packing routine; and (d) computes GEMM using a micro-kernel. In the proposed algorithm, (a) and (c) are replaced by a single packing step executed after tiling. The central ideas of this paper are encapsulated in two new passes in the software stack. First, a *Convolution Slicing Analysis* (CSA) pass takes the input image, filters, and cache sizes and estimates the best tile sizes, their allocation on the memory hierarchy, and the order in which they should be accessed – CSA is the convolution equivalent to the Goto *et al.* [9] algorithm for matrix multiplication.

After presenting convolution (Section 2), this paper makes the following contributions.

- A compiler-based solution for convolution code generation (Section 3) that does not depend on linking with math optimization libraries such as BLAS;
- Convolution Slicing Analysis (CSA) (Subsection 4.1), a generic compiler analysis pass that can be used to determine a tiling strategy for convolution in different types of CPU architectures, given convolution and hardware information;
- Convolution Slicing Optimization (CSO) (Subsection 4.2), a code-generation pass that results in a direct convolution *macro-kernel* that improves performance when compared to Im2Col+BLAS for two CPU architectures (Intel x86 and IBM POWER10) using specialized micro-kernels;
- Vector-Based Packing (VBP) (Subsection 6.2), an input-tensor packing solution that leverages shift instructions in vector registers for better performance in unitary stride convolutions.

Section 3 gives a quick overview of the compilation and execution flows of the proposed approach. Section 5 describes the design of the micro-kernel using the Intel x86 and the IBM POWER10 MMA architectures. Section 6 explains how packing is performed for the input image and filters. Section 7 presents a comparative experimental evaluation.

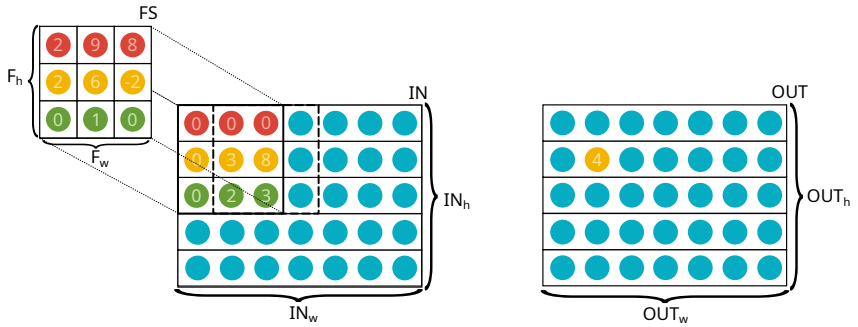


Fig. 1. Convolution step with one 3×3 filter ($OUT_c = 1$), one channel ($IN_c = 1$) and unitary stride. The solid border square indicates the first window, and the dashed border square represents the second window. The convolution between the filter and the first window results in the yellow element in the output.

2 A BACKGROUND ON CONVOLUTION

This section introduces convolution, matrix-multiplication acceleration extensions, and the notation used in this paper.

2.1 Convolution Operation

Convolution in CNNs is formulated as a matrix operation between a set of filters and an input tensor. The filters are also called sets of weights because they are the parameters learned during the training of convolution layers.

A convolution layer of a CNN consists of an input tensor, a set of filters, and one output tensor. In a three-dimensional convolution, the dimensions are the number of channels, height, and width of the tensors. The output tensor OUT has dimensions $OUT_c \times OUT_h \times OUT_w$. The input tensor IN has dimensions $IN_c \times IN_h \times IN_w$, and the set of OUT_c filters (FS) has dimensions $IN_c \times F_h \times F_w$, where IN_c is the number of channels, and F_h and F_w are the height and width of each channel in the filter. Each output channel is the result of a convolution with a different filter. These tensors may be stored in various layouts. For the rest of this paper, consider the intuitive NCHW format for the input and the FCHW format for the filters.

A window is the projection of a filter onto the input tensor, resulting in an $IN_c \times F_h \times F_w$ block of elements. Each output element results from the weighted sum of the input elements in a window, with the weights being the corresponding filter elements. The formation of windows from the input tensor and the computation of a convolution output element are illustrated in Figure 1. Each filter slides over IN with stride s in one dimension between computations.

Tiling a convolution consists of dividing the input tensor IN, filter set FS, and output tensor OUT into tiles that fit in the cache simultaneously. Access to these tiles can be scheduled to take advantage of every cache level in a memory hierarchy.

2.2 Im2Col - Image-to-Column

In the Im2Col + BLAS approach [4], an image-to-column transformation expands the input tensor into windows and then multiplies the result by the weight matrix using a highly optimized GEMM routine from a BLAS library. The Im2Col transformation expands the input image by converting each window into a vector of elements and storing this vector as a column of a larger $(IN_c \times F_h \times F_w) \times (OUT_h \times OUT_w)$ tensor. The set of filters forms an $OUT_c \times (IN_c \times F_h \times F_w)$ tensor. A GEMM operation applied to these 2D tensors produces an $OUT_c \times (OUT_h \times OUT_w)$ output tensor. This

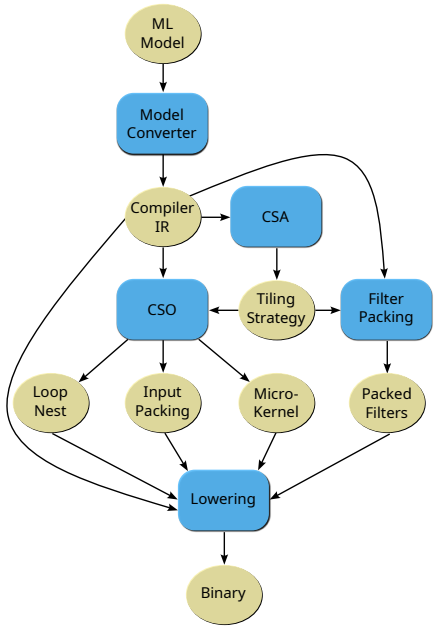


Fig. 2. Compilation flow in a machine-learning compiler using SConv to optimize convolutions.

approach performs two data-movement tasks: (a) the Im2Col data rearrangement; and (b) data packing performed by the GEMM routine to increase spatial locality.

2.3 POWER10 and the MMA Engine

SConv uses an ISA extension for convolution operations in the IBM POWER10 architecture. This processor introduces Matrix-Multiply Assist (MMA) [21], a new built-in engine to accelerate tensor operations accessed through an extension of the POWER ISA v3.1. MMA uses POWER10's 128-bit vector registers (VSRs) as inputs to perform rank-k outer product operations. The result is stored in one of eight new 512-bit accumulator registers representing two-dimensional matrices. The evaluation in this paper indicates that an efficient combination of MMA and VSR operations considerably improves the performance of direct-convolution operations.

3 SCONV COMPILATION FLOW

SConv is a solution to convolution that mixes compile-time data transformation for filter packing and efficient code generation to better utilize memory and architecture resources.

The compilation flow in Figure 2, implemented using ONNX-MLIR and LLVM, can also be used in other compiling frameworks. Blocks are the computation passes, and ellipses are the inputs/outputs. After the conversion of a machine-learning model into an Intermediate Representation (IR), the Convolution Slicing Analysis (CSA) algorithm determines — for each convolution layer: (a) the sizes of the data tiles; (b) the tile scheduling, *i.e.*, the order in which they are loaded at execution time; and (c) the number of tiles that are loaded in each step. Together, (b) and (c) determine the tile distribution in the cache hierarchy, forming a *Tiling Strategy*.

Based on the Tiling Strategy, the Convolution Slicing Optimization (CSO) generates a Loop-Nest macro-kernel for the tiled convolution, an outer-product-based Micro-Kernel to compute a single output tile at peak performance for the architecture, and an Input-tensor Packing routine to rearrange each tile to the required format for the micro-kernel.

For model inference, the filter data is static, and filters can be packed at compile-time based on the tiling strategy. The Lowering pass combines the information generated for each convolution with the rest of the compiler IR to produce an executable binary with tiled convolutions. For training, filter data is not static, and filters are packed at execution time in the macro-kernel.

The four main components of SConv are **CSA** (Subsection 4.1), **CSO** (Subsection 4.2), **Input and Filter Packing** (Section 6) and the **Micro-Kernel** (Section 5). The micro-kernel and packing routines are specific to the target architecture. CSA and CSO use architectural information to create a good tiling strategy and to generate code that implements it. The micro-kernel is designed to leverage ISA acceleration extensions and dictates the packing layout and some tiling dimensions.

4 CONVOLUTION SLICING

An efficient cache-tiling solution that aims to minimize reuse distance is at the core of a direct-convolution algorithm. SConv accomplishes efficient tiling through two compilation passes: (a) Convolution Slicing Analysis (CSA) is responsible for computing tile sizes, distribution, and scheduling; and (b) Convolution Slicing Optimization (CSO) that generates the macro-kernel used to compute the tiled convolution based on the tiling strategy determined by CSA.

4.1 Convolution Slicing Analysis (CSA)

Convolution Slicing Analysis (CSA), a cache-blocking-based algorithm, uses a symbolic simulation heuristic to find a suitable data tiling and scheduling combination that reduces convolution execution time. It aims to maximize input-tensor and filter data reuse on cache memories by: (a) determining the tile size and distribution through the cache hierarchy that maximize cache usage for every level; and (b) selecting the order in which input-tensor/filter tiles are accessed to reduce data movement between the various levels of the cache hierarchy. In this paper IN^T, FS^T, and OUT^T represent, respectively, the tiles of IN, FS, and OUT.

Each convolution output element is related to an input-tensor window, and there is usually an overlap between windows. Thus, each IN^T contains multiple windows to leverage micro-kernel vectorization. CSA determines the number of windows available in each IN^T and the number of filters in each FS^T. The tile sizes are given by Equation 1, where, for example, $|IN^T|$ represents the size of the input tile IN^T. Figure 3 shows that N_{win} is the number of windows in IN^T and N_f is the number of filters used to form FS^T. The values of N_{win} and N_f are a micro-kernel design choice that aims to maximize performance based on the hardware that supports the computation (Section 5).

The CSA algorithm selects tile sizes based on the following constraints:

- the windows in IN^T and the filters in FS^T are only tiled in their channel dimensions;
- full $F_h \times F_w$ filters and windows are used in a tile;
- the maximum number of channels (N_c) is chosen for each tile.

The combination of N_{win} and N_f forms OUT^T, as shown in Equation 1 and Figure 3. DT is the data type size in bytes (*e.g.*, 4 for float32). Within the constraints above, the CSA algorithm must ensure that a IN^T, an FS^T and an OUT^T must fit in the L1 cache.

$$\begin{aligned} |IN^T| &= N_{win} \times N_c \times F_h \times F_w \times DT \\ |FS^T| &= N_f \times N_c \times F_h \times F_w \times DT \\ |OUT^T| &= N_{win} \times N_f \times DT \end{aligned} \tag{1}$$

N_c specifies the number of input channels in a *channel set*. When $N_c = IN_c$, all channels are included in a single IN^T and in a single FS^T, and thus OUT^T can be computed in a single step. When $N_c < IN_c$, each tile combination computes a partial result for an OUT^T, onto which the

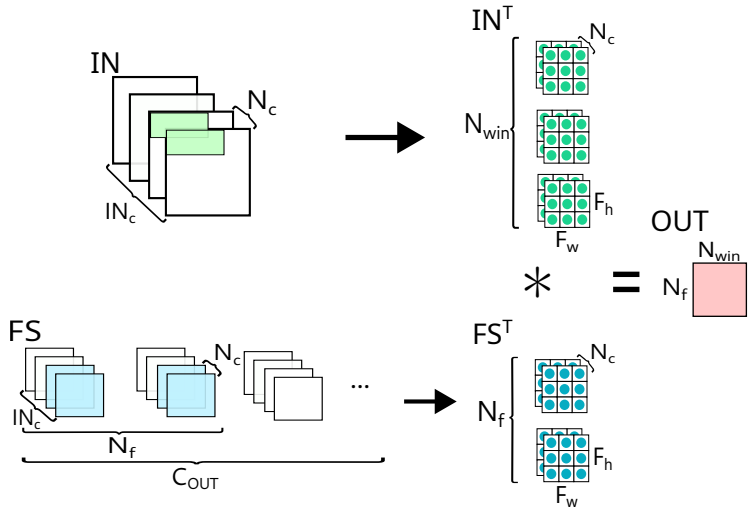


Fig. 3. A CSA-tiled convolution, for $F_h = F_w = 3$, before packing.

results from the rest of the channels need to be accumulated. In this case, the data of OUT^T may be evicted from the L1 cache and then loaded again to continue the accumulation. CSA tries to avoid eviction by maximizing N_c within the constraints in Equation 2, *i.e.*, the total size of IN^T, FS^T, and OUT^T has to be less than the available space in the L1 cache — allowing for the space needed for prefetching and intermediate computation, *e.g.* packing. In Equation 2, the fraction of the L1 cache available for tile storage is represented by α, which depends on the hardware and can be tuned to each convolution. An alternative formulation could subtract the estimated space for prefetching and intermediary values from $|L1|$.

$$|IN^T| + |FS^T| + |OUT^T| \leq |L1| \times \alpha \tag{2}$$

In summary, the convolution operation is divided into tiles along both the window/filter dimension (N_{win}/N_f) and the channel dimension (N_c). The value of N_{win} can also be interpreted as the tiling size for the spatial input tensor, as illustrated in Figure 3. This tiling process ensures that the resulting tiles are small enough to fit into the L1 cache.

In rare cases where Equation 2 cannot be satisfied, even with a single channel in each tile, due to either a small L1 cache or large tiles in each channel, the values of N_{win} and N_f can be further adjusted to decrease the tile size. This slicing approach allows for a more fine-grained partitioning of the convolution and helps mitigate cache size limitations.

CSA calculates the number of IN^T, FS^T, and OUT^T required to compute the convolution. This calculation is done individually for each channel set.

$$\begin{aligned} \#IN^T &= \frac{OUT_h \times OUT_w}{N_{win}} \\ \#FS^T &= \frac{OUT_c}{N_f} \\ \#OUT^T &= \#IN^T \times \#FS^T \end{aligned} \tag{3}$$

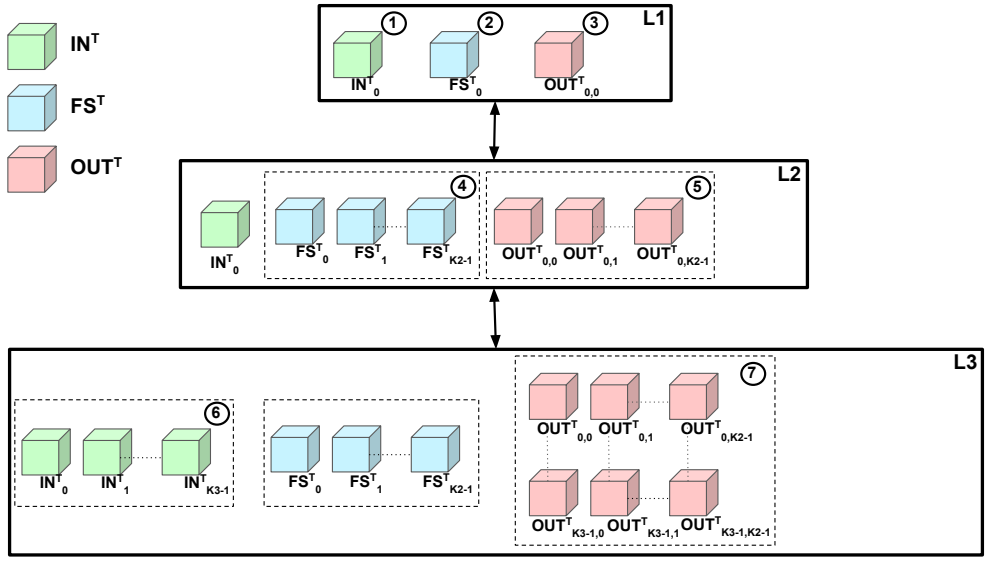


Fig. 4. CSA tile distribution over a three-level cache hierarchy with Input Stationary scheduling.

Edge cases arise when the tiling values are not proper divisors of the convolution information. For instance, when IN_c is not divisible by N_c. These cases are handled by computing smaller tiles when necessary, with smaller N_c, N_{win}, and/or N_f values.

CSA relies on two possible execution orders when scheduling tiles: (a) *Input Stationary (IS)*, keeps one IN^T stationary in L1 cache and reuses it over as many FS^T as possible before proceeding to the next IN^T; (b) *Weight Stationary* (WS), keeps one FS^T stationary in L1, using it to compute multiple OUT^T from many IN^T, before moving to the next FS^T.

Both the IS and the WS scheduling strategies lead to the specification of *K2*, the number of tiles in the L2 cache, and of *K3*, the number of tiles in the L3 cache. These values are set to use the storage available in each cache level while increasing reuse. The L1 cache must always contain one set of IN^T, FS^T, and OUT^T, which are used for the micro-kernel computation.

Figure 4 illustrates the data movements and computation for the IS schedule. Initially, an IN^T is loaded to L1 (1) and kept stationary. Next, a FS^T is loaded to L1 (2) so that the micro-kernel is executed to compute an OUT^T (3), which must also fit into L1. While the IN^T remains in L1, $(K2-1)$ FS^T (4) are in turn brought to L1 to compute other $(K2-1)$ OUT^T (5). The maximum value of *K2* that satisfies the inequality in Equation 4 ensures that the stationary IN^T and the *K2* FS^T and OUT^T fit into the L2 cache. The *K2* FS^T kept in L2 are reused by $(K3-1)$ other IN^T that are kept in L3 (6). The maximum value of *K3* that satisfies the restriction in Equation 5 ensures that a set of *K3* IN^T, *K2* FS^T and the results they generate (7) fit into the L3 cache.

The number of times that each tile is loaded to L1 is given as follows:

- If $K2 = \#FS^T$, then loading each IN^T to L1 one time is enough. Otherwise, other *K2* FS^T are loaded from memory to L2, and *K3* IN^T from L3 are loaded to L1 again;
- If $K3 \neq \#IN^T$, then the same *K2* FS^T are loaded back to L2. In this case, other *K3* IN^T are loaded to L3 and these steps repeat;
- The OUT^T are loaded back to L1 for accumulation $((IN_c/N_c)-1)$ times.

In WS scheduling, the order of IN^T and FS^T are reversed. The share of the caches available to be used in the equations that define $K2$ and $K3$ are represented in Equations 4 – 5 by β and γ. Alternatively, the estimated space needed for packing, prefetching, and other operations, can be subtracted from the cache size.

$$|IN^T| + K2 \times (|FS^T| + |OUT^T|) \leq |L2| \times \beta \quad (4)$$

$$K3 \times |IN^T| + K2 \times |FS^T| + K2 \times K3 \times |OUT^T| \leq |L3| \times \gamma \quad (5)$$

When exploring the tiling space to find the optimal strategy for convolutions, it is important to consider the trade-off between compilation time and model development cycle time. Some approaches, such as TVM [5] or Ansor [36], tolerate long compilation and code generation times (even hours). However, in the context of ML model development, a faster turnaround time (on the order of dozens of minutes) is essential to enable designers to experiment with various options of model architectures.

The CSA approach incorporates specific modeling techniques that restrict the tiling space, but the exploration heuristic can be customized to meet specific requirements. In this work, two greedy heuristics were developed and applied separately to determine values for N_c, $K2$, and $K3$:

- The first heuristic starts with the highest possible value (e.g., IN_c for N_c) and iteratively checks all constraints. If any constraint is not satisfied, the value is halved in each subsequent iteration until all constraints are met. This heuristic prioritizes avoiding incomplete tiles whenever possible because incomplete tiles may result in slower execution.
- The second heuristic employs a binary-search approach to maximize each value based on its corresponding constraint. This heuristic emphasizes cache usage, with a higher priority given to faster cache levels.

For the remainder of this paper, the second heuristic was chosen. It improves performance and is a good balance between exploring the tiling space efficiently and optimizing cache utilization, which aligns with the goal of achieving a faster ML model development cycle.

CSA decides between IS and WS based on a cost model (Section 4.1.1) using the number of L2, L3, and main memory accesses and estimating the number of cycles for those accesses in each strategy.

4.1.1 Cost Model. C_{TOTAL} is the cost of loading tiles from L2 cache, L3 cache, and main memory. It is the sum of the product of the number of loads from each level and the corresponding cost (in cycles) of each load/store operation at that level (Equation 6). The cost of an L1 hit is not included because every load operation goes through the L1 cache. The cost of reloading an output tile when $N_c \neq IN_c$ is also not included in the model because it is the same for both the IS and the WS strategies.

$$C_{TOTAL} = C_{DRAM} \times N_{DRAM} + C_{L3} \times N_{L3} + C_{L2} \times N_{L2} \quad (6)$$

$$N_{DRAM} = N_{DRAM_1} + N_{DRAM_2} \quad (7)$$

The number of DRAM accesses (Equation 7) is the sum of N_{DRAM_1}, the number of *cold misses* required to touch every single tile at some point of the convolution execution; and N_{DRAM_2}, additional touches needed to reload tiles if they are not available in the cache when required.

Let CL be the cache line size of the current architecture. The number of cold misses is given by:

$$N_{DRAM_1} = \frac{IN_c}{N_c} \times \left(\frac{\#IN^T \times |IN^T| + \#FS^T \times |FS^T|}{CL} \right) \quad (8)$$

All other equations in this section depend on the analyzed scheduling strategy. The rest of this section will consider the Input Stationary case. For the Weight Stationary cost equations, replace $\#IN^T$ and $|IN^T|$ with $\#FS^T$ and $|FS^T|$ and vice-versa.

When all the FS^T do not fit simultaneously in the L2 cache, the reloading of tiles during the computation for each set of *K3* IN^T leads to additional DRAM accesses. This data is not available in the L3 cache because it is evicted when loading the other IN^T and FS^T sets that use all the available space. Equations 9 – 11 computes N_{DRAM_2}, the number of additional accesses required.

$$FS^T_{fitmin} = \min\left(\left(\frac{\#FS^T}{K2} - 1\right), 1\right) \tag{9}$$

$$IN^T_{fit} = \frac{\#IN^T}{K3} - 1 \tag{10}$$

$$N_{DRAM_2} = \frac{IN_c}{N_c} \times \frac{FS^T_{fitmin} \times IN^T_{fit} \times \#FS^T \times |FS^T|}{CL} \tag{11}$$

FS^T_{fitmin} represents the condition for additional DRAM accesses, *i.e.* if $\#FS^T$ fits in the L2 cache. If the equation evaluates to zero, no reloads are needed. Otherwise, the number of loads from memory is controlled by IN^T_{fit}, the number of additional IN^T sets.

An IN^T may need to be loaded from the L3 cache more than once. If there is more than one set of *K2* FS^T, the next set will need to be computed with all *K3* IN^T in the L3 cache, so they have to be loaded to L1 again. In the calculation of the number of loads into L3 (Equations 12 – 13), FS^T_{fit} controls the number of reloads from the L3 cache with the number of additional FS^T sets.

$$FS^T_{fit} = \frac{\#FS^T}{K2} - 1 \tag{12}$$

$$N_{L3} = \frac{IN_c}{N_c} \times \frac{FS^T_{fit} \times \#IN^T \times |IN^T|}{CL} \tag{13}$$

For every IN^T, all *K2* FS^T need to be loaded from the L2 cache for computation. Since the tiles initially come from memory directly to the L1 cache, one access should be discharged. N_{L2} is calculated using Equation 14.

$$N_{L2} = \frac{IN_c}{N_c} \times \frac{\left(\#IN^T - 1\right) \times \#FS^T \times |FS^T|}{CL} \tag{14}$$

The cost model represents all data movement between cache levels and main memory, and they depend on the scheduling strategy. A lower cost means better tile reuse and cache usage. Thus, the cost should be calculated for both approaches, and the lowest value should be chosen, for each convolution, for that specific architecture.

4.2 Convolution Slicing Optimization (CSO)

The Convolution Slicing Optimization (CSO) slices the input, filters, and output based on the tiling sizes and scheduling computed by CSA and generates a macro-kernel to compute the tiled convolution. The main idea is to maximize tile reuse to take advantage of the cache hierarchy.

The input-tensor packing routine (Subsection 6.1) expands the input into windows for one tile and packs the tile such that the elements are in the order required by the micro-kernel. The CSO macro-kernel calls the input-tensor packing routine and iterates over all tiles in the order established

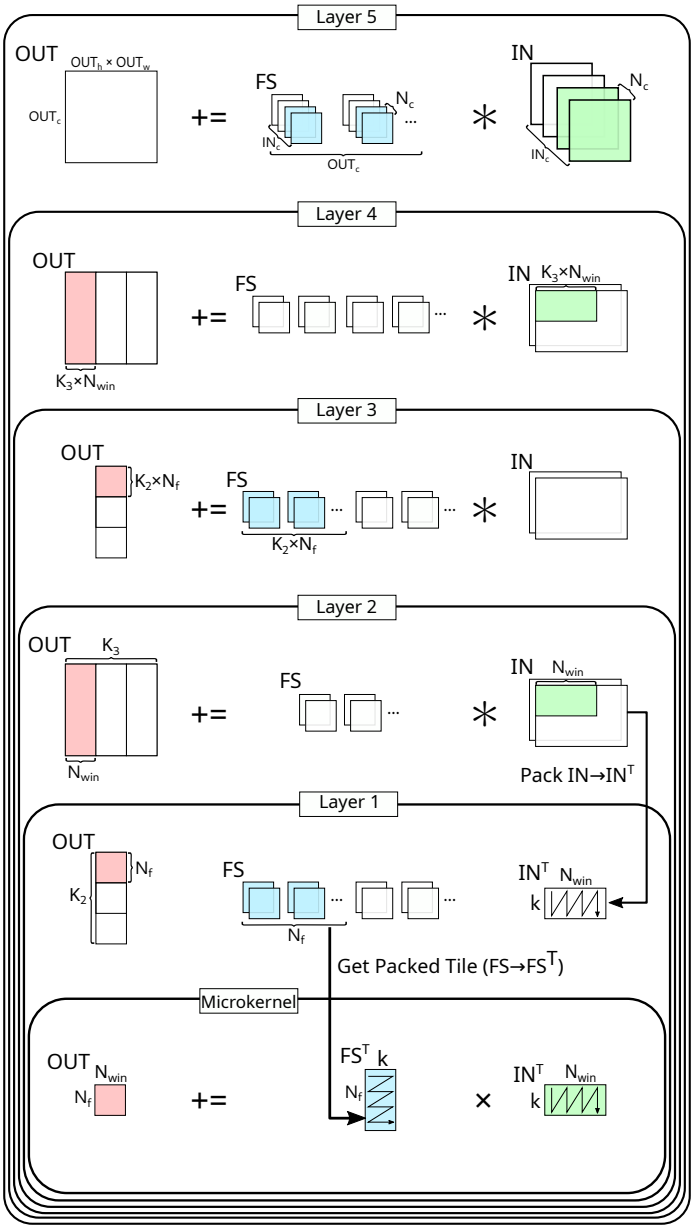


Fig. 5. Convolution loop nest macro-kernel generated by CSO using CSA's tiling parameters. Illustrates the process for Input Stationary scheduling.

by CSA. The loop nest is represented as layers for Input Stationary scheduling in Figure 5. This section details that figure.

For efficiency, the micro-kernel in the innermost loop (Section 5) computes with tiles already loaded into the L1 or L2 caches. Thus, the algorithm implements *Packing on Demand*: layer two of the loop nest packs input-tensor tiles right before they are used. The filter-packing routine, used

in layer 1, returns the already packed tile since the filters have already been packed during the model's compilation.

The outermost loop (layer five) iterates over channel sets with no reuse between them. Once a channel set is selected, the tiles within this channel set need to be further divided so that a few stay at each cache level. This is done via the *K2* and *K3* parameters.

Layers 2 and 4 are responsible for managing all input-tensor tiles (IN^T). Layer 2 iterates over a single set, and layer 4 iterates over different sets of *K3* input-tensor tiles, packing one tile at a time. Similarly, Layers 1 and 3 iterate over all filter-set tiles (FS^T). Layer 1 calls the micro-kernel. This setup ensures that every combination of input-tensor and filter-set tiles is computed.

The input-tensor tile selected by layer 2 is kept stationary in the L1 cache because it is used with *K2* filter-set tiles in layer 1. When the next input tile is selected, these filter-set tiles are in the L2 cache to be reused. Likewise, *K3* input-tensor tiles are kept in the L3 cache afterward for reuse with the next set of filter-set tiles.

In the Weight Stationary scheduling, the process is the same, switching IN^T and FS^T in Figure 5. The input-tensor tiles need to be packed for the L2 cache in this case. Thus, the packing process happens in layer 3 for all *K2* tiles at once, and each tile is fetched when needed. If the filters cannot be packed at compile time, the tiles are also packed on demand whenever they are needed, using the same process used for the input tensor.

5 AN OUTER-PRODUCT MICRO-KERNEL

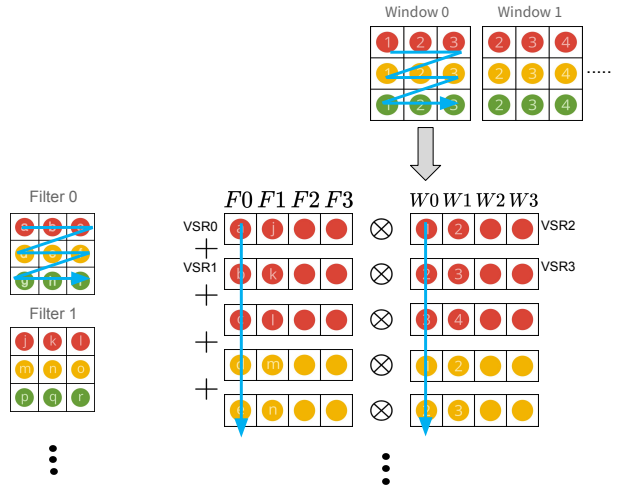
The micro-kernel, located in the most internal layer of Figure 5, computes the arithmetic operations that form the convolution. For most efficiency, the micro-kernel should be treated as an extension of the hardware instead of a high-level routine. A micro-kernel implementation is tailored to the architecture and uses any available specific instructions to increase throughput. This work focuses only on a single-precision (32-bit) floating-point micro-kernel, as different datatypes can drastically change the entire process, especially in more complex architectures.

SConv uses an outer-product-based micro-kernel. The outer product is generally useful due to its versatility for computing higher-rank operations and high throughput, computing n^2 output elements from $2n$ input elements. For those reasons, it is widely used in high-performance linear algebra libraries, and it is being incorporated in CPU ISA extensions such as IBM POWER10 MMA [21]. The micro-kernel computes a sum of outer products between vectors of floating-point elements, loaded from memory with unitary stride. A highly optimized implementation of such a micro-kernel is found in BLAS GEMM, which can be used as the inner layer of the algorithm, repurposed by the packing layout and macro-kernel. This brings an additional benefit: those architectures with a BLAS implementation can easily integrate their micro-kernel into SConv. Moreover, a new micro-kernel designed for some future architecture could seamlessly be used by both a BLAS library and SConv.

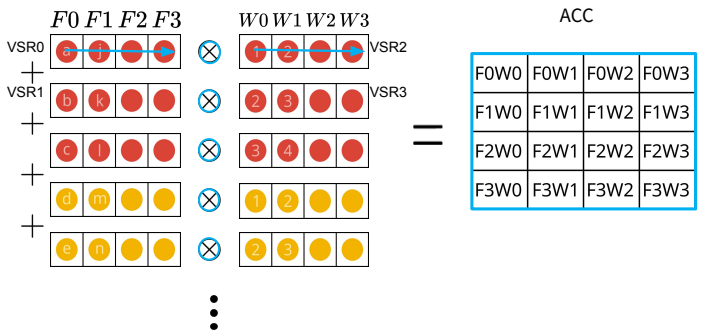
5.1 Micro-Kernel for IBM's MMA

For the POWER10 architecture, the MMA engine can compute multiple output elements at once. Each input element is used multiple times to compute different output elements. Thus, the input should be split into windows when packing.

The outer-product-based design of MMA enables vectorization of the convolution, as shown in Figure 6a. For 32-bit data types, each VSR contains four elements. One VSR contains elements from four different filters and the other from four different windows. An outer product between two VSRs computes sixteen partial output elements, as shown in Figure 6b. In the two-dimensional matrix produced, each row represents a convolution with one filter, and each column represents convolutions with one window. The partial products are accumulated in place in this accumulator. The accumulated values are stored into memory after the complete set of $F_h \cdot F_w \cdot N_c$ outer product



(a) Window and filter data distribution over a sequence of outer products.



(b) Sequence of outer products to generate one accumulator matrix with part of the micro-kernel's convolution output.

Fig. 6. Execution of a POWER10 convolution micro-kernel for 32-bit floating point. Four filters and four windows are used in each VSR of the outer product.

operations have been executed. The resulting data layout is correct for the convolution output, and thus there is no need for unpacking or reordering.

Combining all available VSRs, a larger accumulator can be emulated with a layout-dependent size. Its bigger dimension should be used for computing more windows in each micro-kernel call because there are usually more windows than filters in a convolution. This "super accumulator" has 8×16 elements in total [3], so the micro-kernel consumes 8 filters and 16 windows per call.

For POWER10 CSA uses $N_f = 8$ and $N_{win} = 16$ as described in Subsection 4.1.

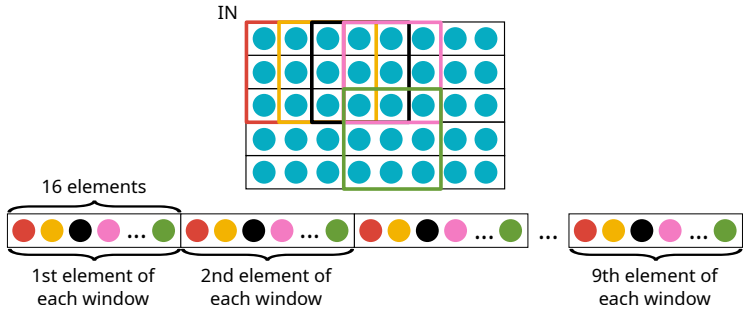


Fig. 7. SConv's input-tensor packing layout for a 16-window micro-kernel, applied to a convolution with $F_h = F_w = 3$, $IN_c = 1$ and unitary stride. For each of the 16 windows, the elements from the same position are stored sequentially.

5.2 Micro-Kernel for Intel's AVX-512

Even though the Intel Skylake x86 architecture does not feature a matrix engine, the data layout for the input and output structures is the same as for the POWER10 micro-kernel. In this architecture, vector instructions in AVX-512 vector registers emulate the outer-product computation using the following algorithm:

- load two AVX-512 registers with 4 elements (128 bits) each;
- broadcast the elements to fill the rest of each register with copies;
- permute the results so that copies of the same element are consecutive;
- perform a Fused Multiply-Add (FMA) operation between the registers.

This micro-kernel computes 16 windows (N_{win}) and 24 filters (N_f) per call. A similar process can be done to emulate outer products in other architectures with SIMD computing.

6 PACKING ON DEMAND FOR THE MICRO-KERNEL

Since the micro-kernel is responsible for computing the convolution and is the innermost step in the loop nest, it should be the focus for maximizing performance. This entails minimizing CPU stalls and data manipulation overhead. Therefore, the required data should be stored sequentially in memory, in the correct order, and ideally available in the cache. Therefore, SConv includes packing steps for both the input tensor and filters, as indicated in layers 2 and 1 from Figure 5, respectively.

The objective of the packing steps is to reshape and, in the input-tensor case, to expand a tile of data into the format required by the micro-kernel. Thus, this format can differ between architectures. The packing routines are called for a single tile at a time, and the data is loaded into the L1 cache for the micro-kernel to use thereafter (Packing on Demand).

6.1 Input-Tensor Packing

The input must be expanded into windows, which are then reshaped so that the micro-kernel can be used to compute the convolution. Instead of doing this process in two separate steps as done by Im2Col+BLAS, the SConv input-tensor packing routine combines both into a single pass. The convolution stride is also handled in this routine, ensuring that the correct elements are contiguous.

Figure 6a illustrates the micro-kernel's access pattern in the POWER10 architecture (for 32-bit floating point data). A window's elements are distributed between multiple outer products, so a vector register contains elements from the same position in different windows. In Figure 6, VSR2 contains the first element of each window, VSR3 has the second element of each window, and so on. Therefore, the elements in memory should follow this pattern. Other outer-product-based micro-kernels, such as the one for Intel Skylake x86, follow this same access pattern.

Figure 7 shows the packing layout derived from the micro-kernel's access pattern. Elements are loaded multiple times. Thus, the data is packed so that the elements from the same position in every window are stored sequentially in memory. This pattern is repeated for all N_c channels in the tile so that the packing routine takes advantage of both spatial and temporal data locality per row, especially in unitary stride convolutions.

6.2 Vector-Based Input Packing

Packing the input tensor in SConv requires data replication because one element can be a part of many windows. In fact, most elements in an input tensor participate in many windows.

The most straightforward way to optimize for multiple loads of the same element is to load from the L1 cache whenever possible. However, when the convolution has a unitary stride in NCHW format, the first element from the next window is the second element from the current window, excluding border cases (as evidenced by Figure 7). Therefore, a shift-left operation with a serial-in element is enough to get the next element from each of the windows in a packed tile.

This observation enables the use of Vector-Based Packing (VBP), where a vector shift operation reduces loads: the next element of each window in a row is obtained by shifting the current set of elements to the left, which can be done by loading all elements into vector registers. This operation is available in most architectures, so the technique is general. Both POWER10 and Intel Skylake x86 provide instructions to shift a pair of vector registers together to the left with the second vector register appended to the right of the first. The least significant element of the first vector register is fed by the most significant element of the second vector register.

Padding the input tensor at the start of the convolution process addresses edge cases. Convolutions with 1×1 filters can also benefit from vector register use, even if no shifts are needed.

6.3 Filter Packing

Filters are loaded in the same way as windows in the micro-kernel. Thus, the packing strategy is similar: a position from each tile filter is packed before advancing to the next, and this process repeats for every channel. However, in contrast with the input tensor, there is no data replication while packing the filters, only the rearrangement of their elements.

Filter packing differs between the two tested architectures because their micro-kernels compute a different number of filters per call.

7 COMPARING SCONV WITH IM2COL + BLAS

This experimental evaluation indicates that the strengths of SConv come from the use of faster micro-kernels that are tailored for each architecture and from the careful packing that reduces cache misses.

7.1 Full-Model Performance Evaluation Using ONNX-MLIR

The SConv prototype evaluated in the two CPU architectures specified in Table 1 is integrated into the ONNX-MLIR framework [19] to allow for full-model performance evaluation under realistic conditions. The loop nest, input-tensor packing routine, and micro-kernel are implemented as a library instead of being completely generated by CSO at compile-time. All the 393 convolutions found in seven machine-learning models from the ONNX Model Zoo [22] are executed with SConv. Each model takes one input image; thus, the batch is 1. Unless otherwise specified, the filters are packed at compile time.

The baseline, called Base, is an industry-standard convolution algorithm where the input is packed into a patch-matrix using the Im2Col routine from the Caffe framework [15], followed by a call to an optimized GEMM routine from the OpenBLAS library [33]. This library is also

Name	IBM Power S1024	Intel Xeon Silver 4208
CPU Architecture	IBM POWER10	Intel Cascade Lake (x86)
Matrix/SIMD Engines	MMA/VSX	None/AVX-512
L1 Cache Size	32 kB	32 kB
L2 Cache Size	1 MB	1 MB
L3 Cache Size	4 MB	4 MB
CPU Frequency	3.9 GHz	2.1 GHz
Matrix/SIMD Units	2/4	0/1
Max Throughput	249.6 GFLOPS/s	67.2 GFLOPS/s

Table 1. Details of CPU architectures on which tests were conducted.

Convolution	OUT_c	IN_c	IN_h/IN_w	F_h/F_w	x86				P10			
					N_c	$K2$	$K3$	Schd	N_c	$K2$	$K3$	Schd
VGG16-2	64	64	224	3	17	72	3	WS	29	8	196	IS
GoogleNet-50	128	32	7	5	6	4	6	WS	10	16	4	IS
SqueezeNet-6	64	16	55	1	16	190	3	WS	16	8	190	IS
ResNet152-39	1024	256	14	1	154	13	43	WS	256	94	13	IS
VGG16-1	64	3	224	3	3	256	3	WS	3	8	1935	IS

Table 2. CSA results for various convolutions from the tested models

optimized for each architecture, using vector and tensor units when available. For a fair comparison, the micro-kernels used in SConv are also from this library because they are designed by vendor developers and therefore have very efficient usage of each hardware.

Equation 2 and Equations 4 – 5 introduce the α, β, and γ parameters to account for extra space needed for non-tile data in each level of the memory hierarchy. These parameters can be tuned for each use case. For this performance evaluation, the values of all three parameters were set to 0.8 for all convolutions and both machines.

Both SConv and Base use single-precision floating-point datatype, are executed in single-threaded mode and are compiled with Clang 14. Cache performance measurements for x86 use the Perf Events API from the Linux kernel. All results are averaged over 100 runs, with less than 5% observed standard deviation for both architectures.

Both SConv and Base are evaluated using complete end-to-end model inference measurements, preceded by a few warm-up runs to ensure consistent results. Performance measurements of individual convolutions and other relevant values are also captured within this context. The other operators in the model are not modified and retain their original ONNX-MLIR implementations. The unit tests provided with the framework verify the correctness of the model inference results and ensure consistency between methods. The CSA algorithm uses a binary-search heuristic for space exploration, as described in Subsection 4.1.

7.2 CSA and Cache Usage

Table 2 displays the CSA values for a selection of convolutions from the tested models, encompassing various sizes and shapes. The calculated parameters indicate efficient cache utilization, with 84% of convolutions on POWER10 and 90% on x86 using more than 95% of available L1 cache space. L2 and L3 cache usage is determined by the scheduling strategy. For instance, a lower L2 cache usage can be advantageous if it allows all tiles in the filter set to fit simultaneously.

The difference between heuristics becomes evident when examining the N_c values of VGG16-2 and GoogleNet-50, as determined by the binary search heuristic. In the case of VGG16-2, the first convolution operates on 64 channels, which are tiled into three sets of 17 channels and one set of 13 channels on x86. This creates an edge case where the binary search heuristic selects a configuration that uses slightly less L1 cache space compared to a simpler heuristic, which would divide the tiles into four sets of 16 channels and avoid the edge case.

Model	Convolution Speedup		Model Speedup		Convolution Time Share	
	x86	P10	x86	P10	x86	P10
GoogleNet [27]	1.20	1.26	1.11	1.11	0.48	0.45
InceptionV2 [14]	1.23	1.28	1.18	1.19	0.77	0.64
ResNet-18 [11]	1.28	1.39	1.27	1.34	0.94	0.85
ResNet-50 [11]	1.16	1.28	1.16	1.22	0.90	0.75
ResNet-152 [11]	1.18	1.24	1.17	1.19	0.93	0.78
SqueezeNet [12]	1.13	1.23	1.11	1.16	0.75	0.60
VGG-16 [25]	1.28	1.28	1.17	1.13	0.70	0.51
Geo Mean	1.21	1.28	1.17	1.19	0.76	0.64

Table 3. SConv performance speedup over Base in machine-learning models.

The second convolution benefits from the binary search heuristic on both architectures. On POWER10, this convolution is tiled into three sets of 10 channels and one set of 2 channels. On x86, it is divided into five sets of 6 channels and one set of 2 channels. In contrast, the basic heuristic would divide the tiles into four sets of 8 channels for POWER10 and eight sets of 4 channels for x86. This alternative approach would result in additional reloads of the output and packing, reducing efficiency.

Whenever feasible, all channels are included in the convolution tiles ($N_c = IN_c$). However, this may not be sufficient to fill the available space in the L1 cache, as shown in the convolutions of SqueezeNet-6, VGG16-1, and, to a lesser extent, ResNet152-39 on POWER10. The L1 cache may be underutilized in such cases because the spatial dimensions of the tiles are not considered during the space exploration process. Convolutions with $F_h = F_w = 1$, as discussed in Subsection 7.6, result in tiles with smaller channel sizes and need a higher number of channels to fill the L1 cache. As a result, these convolutions are more susceptible to this limitation.

The convolution scheduling, determined by the cost model (Subsection 4.1.1), is sensitive to the micro-kernel shape ($N_f \times N_{win}$). The POWER10 micro-kernel computes 8×16 elements, while the x86 micro-kernel computes 24×16 elements. The change in the N_f value leads to a larger filter-set tile, which may favor its reuse in the L1 cache. This difference leads to the selection of weight-stationary scheduling for x86 and input-stationary scheduling for POWER10.

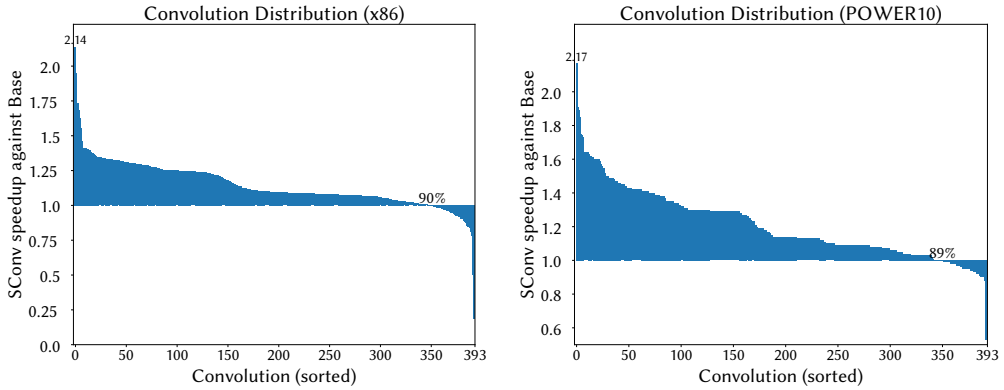
7.3 SConv Outperforms Base in Every Model

Table 3 shows the convolution-only speedup and the whole-model inference speedup, including every operation from each model. SConv outperforms Base in every model on both architectures with convolution speedups ranging from 13% to 39%. As expected, higher speedups are observed in models with large convolutions, such as VGG-16 and ResNet-18. The fraction of the time spent in convolution in ONNX-MLIR determines the effect of using SConv over Base in the whole-model performance. On average, whole models are between 17% and 19% faster.

A more in-depth performance analysis needs to examine the performance changes in individual convolutions. Figure 8 shows the results for all individual convolutions sorted by speedup. SConv outperforms Base on both architectures in at least 89% of the convolutions. The maximum speedup for x86 is 2.14×, and for POWER10 is 2.17×. SConv reaches 78% of the theoretical peak throughput in POWER10 and 90% in x86.

The convolutions where SConv underperforms Base indicate some of the algorithm limitations. As discussed in Subsection 7.2, when a convolution has a small number of channels, the L1 cache might be underutilized because of restrictions in CSA spatial tiling. For small convolutions, edge cases such as partial tiles can also be responsible for a significant performance impact. Pointwise convolutions also have other sensibilities that are further explored in Subsection 7.6. Nearly all cases in which SConv has worse performance than Base include a combination of these factors.

As discussed in Subsection 7.2, there are tradeoffs for different CSA space-exploration heuristics. The simpler heuristic underperforms binary search on the convolutions of each ML model by 2%-10% on POWER10 (geomean: 4%) and 1%-2% on x86 (geomean: 2%). Besides that, it is notable the increase in L1 cache usage, which goes from 66% to 76% (total available space is 80%).



(a) x86 (b) POWER10

Fig. 8. SConv performance speedup against Base for each of the 393 individual convolutions from the evaluated machine learning models. The x-axis is sorted by speedup (y-axis). Base's performance is normalized to 1. The graphs also indicate the percentage of convolutions in which SConv outperforms Base.

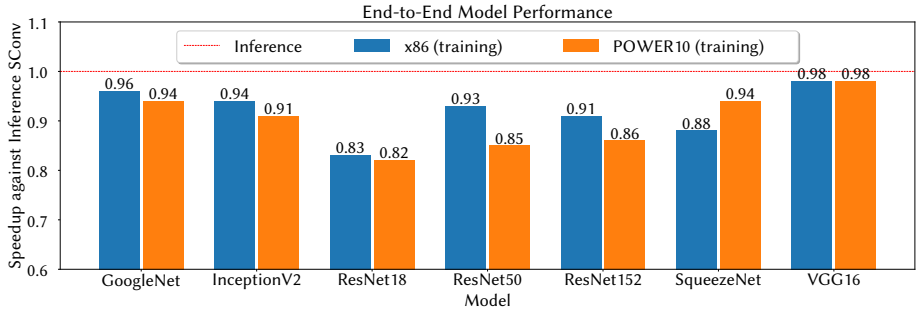
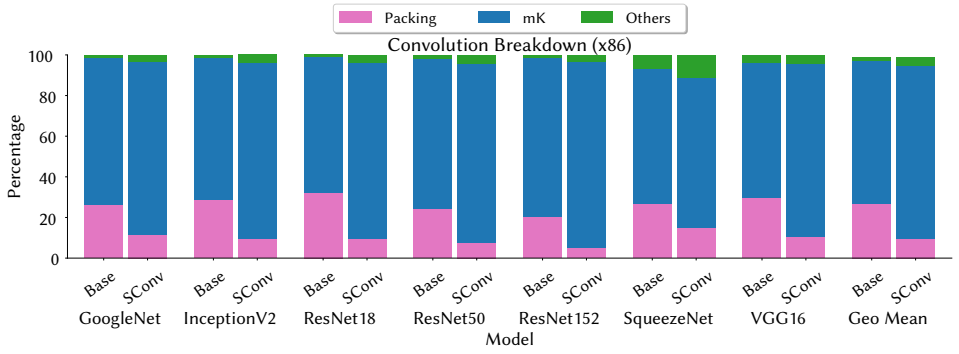


Fig. 9. Performance differences between the training and inference versions of SConv, normalized to the inference version.

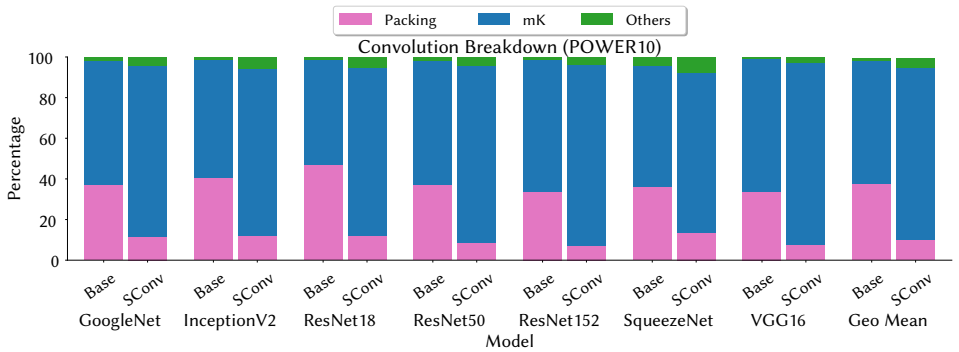
7.4 Filter Packing at Compile Time Improves Performance

As mentioned in Section 3, during the compilation process for inference, the filter set of each convolution in the model can be static and known at compile time. This information allows for packing optimization by rearranging the elements during that stage. However, this scenario may not always be applicable, and there is a need for a version of SConv that can pack filter set tiles on demand.

Figure 9 shows a relative comparison between these two versions of SConv. As expected, the training version is slower than the optimized inference SConv. Packing the filter set on demand not only introduces another routine to the loop nest and more data loads but also interferes with



(a) x86



(b) POWER10

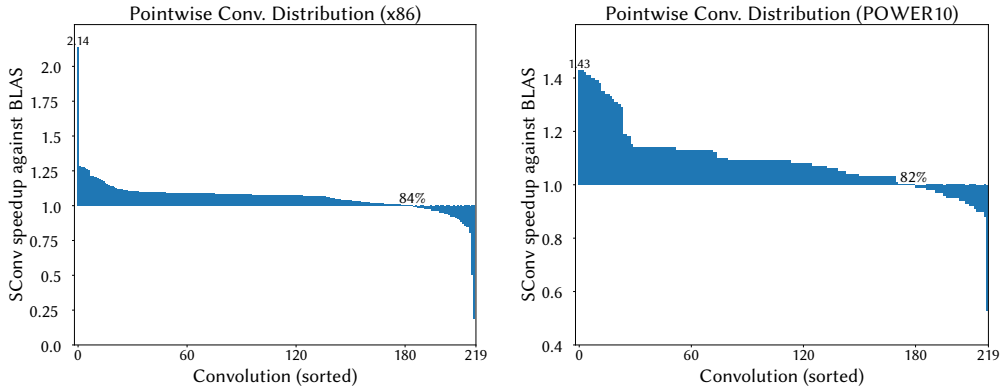
Fig. 10. Breakdown of each model's total convolution time distributed between packing, micro-kernel (mK) and other steps (Others), for Base and SConv.

the cache while packing. However, even with a significant impact on performance, SConv is still, on average, faster than Base.

The ResNet models, along with the SqueezeNet model on x86, experience the most significant performance decline when introducing filter packing at execution time on both architectures because a considerable number of convolutions in these models have small input tensors and a large number of filters. As a result, the filter-packing step represents a larger share of the overall execution time. The VGG16 model is the least affected by this change because it has large convolutions with relatively smaller filters, which mitigates the impact of the filter packing step on the overall execution time.

7.5 Faster Micro-Kernels Make Packing Time More Relevant

Figure 10 shows the convolution execution time divided into (a) "Packing", including Im2Col and GEMM packing; (b) "mK", the micro-kernel execution, which is the same for both approaches; (c) "Others", which represents the remaining boilerplate code (*e.g.*, bias, control code, padding, etc.) that is different for Base and SConv. In SConv, packing has a smaller contribution to the execution time in comparison with Base. SConv provides three major improvements in packing: elimination



(a) x86 (b) POWER10

Fig. 11. SConv performance speedup against BLAS for each of the 219 pointwise convolutions from the evaluated machine learning models. The x-axis is sorted by speedup (y-axis). BLAS' performance is normalized to 1. The graphs also indicate the percentage of pointwise convolutions in which SConv outperforms BLAS.

of the Im2Col step, reduction of unnecessary loads by Vector-Based Packing, and optimization of filter-packing at compile-time.

The x86 results show that Base has, on average, 26% of its execution time dedicated to packing (20% — 32%), while SConv's packing share averages 10% (5% — 15%). The POWER10 results indicate a larger improvement, with Base's packing share averaging 38% (34% — 47%) and SConv's averaging 10% (7% — 13%), which is a packing time improvement of 2.9x on x86 and 4.7x on POWER10. The improvement range is 2.3x — 4.0x on x86 and 3.3x — 5.9x on POWER10. Faster hardware-supported micro-kernel makes the packing time more significant in the overall execution. This effect is evident in the better POWER10 results, which translates to overall speedup in Table 3 and Figure 8.

Packing data is most significant for large convolutions, such as those in ResNet18 and VGG16, that contribute most to the model time. Large convolutions are the most sensitive to some of the improvements in SConv, such as VBP and the elimination of Im2Col. Thus, packing optimizations are the main contributors to the results shown in Subsection 7.3.

7.6 Im2Col is Not Needed for Pointwise Convolutions

A pointwise convolution comprises 1×1 windows with unitary stride. This type of convolution can be solved by matrix multiplication since the Im2Col transformation produces a simple copy of the input matrix. Thus, for pointwise convolutions, Base is only the BLAS library's GEMM routine, and SConv does not have the advantage of fewer packing steps or Vector-Based Packing. Such convolutions are frequently found in ML models — 219 out of the 393 (55%) convolutions in our experimental evaluation. Even though they are numerous, they are usually smaller and faster to compute, so often do not contribute as much to the model execution time as the larger convolutions.

SConv's results for these convolutions, shown in Figure 11, have a pattern that matches, to some degree, the one in Figure 8, indicating good performance even when there is no Im2Col overhead in the baseline. SConv outperforms BLAS GEMM in 84% of all pointwise convolutions on x86, and 82% on POWER10, with larger average improvements in the positive cases than slowdowns in the negative ones. These results evidence the efficiency of CSA's tiling strategy for different convolution shapes and the advantage of packing filters at compile-time. Compile-time packing is only important in convolutions with many filters and channels and small IN_h and IN_w, and this is reflected in which convolutions achieve higher speedups.

Most convolutions for which Base outperforms SConv are pointwise: 93% on POWER10 and 83% on x86. For instance, the largest pointwise convolutions found in the tested models have $IN_h = IN_w = 56$, and are consistently faster in Base. These cases fit the best-performance scenario of the BLAS GEMM routine because the inner dimension (IN_c) is small compared to the dimensions of the output matrix [35]. Furthermore, filter packing has a smaller impact on the overall execution time when the input tensor is large.

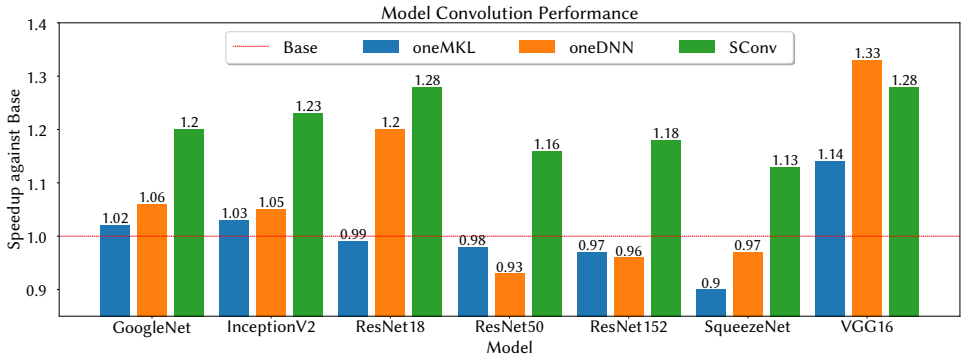


Fig. 12. Performance speedup of SConv and oneAPI routines over Base in machine learning models.

7.7 CSA Scheduling Impacts Performance

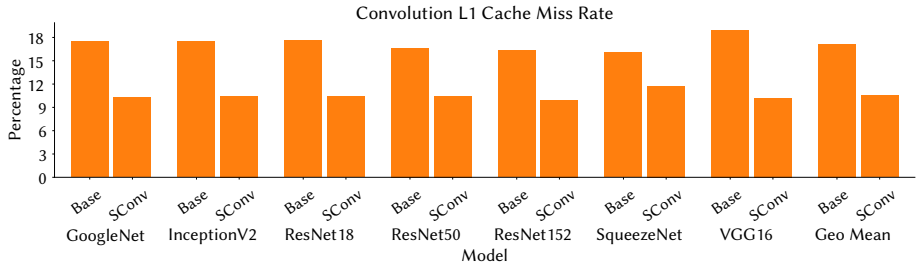
The CSA algorithm aims to choose between the IS and WS scheduling strategies based on a cost model. To evaluate the effectiveness of this selection, we conducted experiments on the x86 architecture where the chosen scheduling policy is different than the one selected by the cost model. Given that the cost model always prefers WS for the tested convolutions on x86, this alternate version always chooses IS scheduling. An analysis of the results reveals that WS outperforms IS in 96% of the convolutions (379 out of 393), with an average convolution model speedup of 7.5%. The speedup ranges from 6.4% to 16.1% for SqueezeNet and GoogleNet, respectively. The ResNet models run between 7.4% and 7.8% faster, while InceptionV2 exhibits a speedup improvement of 10.4%. However, an exception occurs with the VGG16 model, which experiences a slowdown of 2.5%. WS outperforms IS in 9 out of 13 convolutions in VGG16. This slowdown primarily stems from the second convolution in VGG16 (VGG16-2), which contributes significantly to the overall execution time. Although the micro-kernel in this convolution is faster in WS, less time is required for packing input tiles in IS, outweighing the advantages of WS. The CSA algorithm may occasionally suggest a suboptimal solution because it does not consider the packing time in its cost model.

7.8 SConv is Faster Than oneMKL and oneDNN

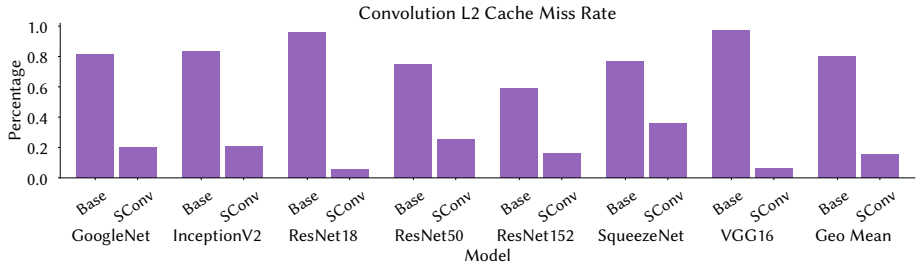
This section compares, on the x86 architecture, SConv against the following routines from oneAPI [13]: (a) the Caffe Im2Col transformation followed by oneMKL GEMM, and (b) the forward convolution routine from oneDNN. The performance speedup of SConv and the oneAPI routines compared to Base is depicted in Figure 12. Across all models, SConv surpasses oneMKL, achieving a speedup difference ranging from 14% to 29% for VGG16 and ResNet18, respectively. Similarly, when comparing with oneDNN, SConv exhibits a speed advantage between 8% and 23% for ResNet18 and ResNet152, respectively, except for VGG16, where it experiences a slowdown of 5%. The slowdown in VGG16 is primarily due to the costly packing time associated with this convolution in SConv for VGG16-2. SConv outperforms oneDNN in 8 out of the 13 convolutions within this model. Overall,

out of the 393 tested convolutions, SConv has superior performance in 78% (305) and 84% (329) of them compared to oneMKL and oneDNN, respectively.

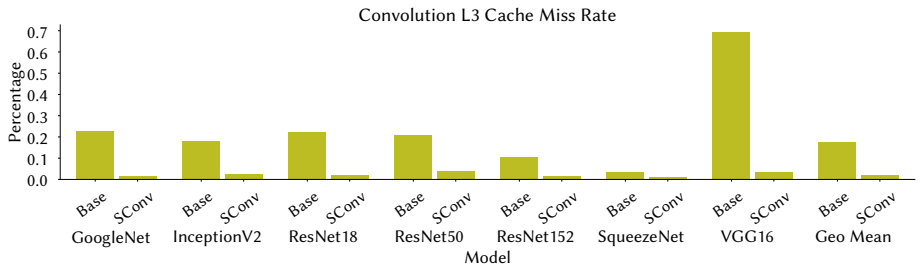
7.9 SConv Reduces Cache Misses in All Levels of Cache



(a) L1



(b) L2



(c) L3

Fig. 13. Breakdown of each model's cache miss performance in each level, for Base and SConv on x86. Shows the percentage of cache misses in each level from the total loads.

The main goal of the slicing strategy in SConv is to improve cache performance. The Im2Col step is bad for cache performance since most tiles, after expansion, need to be reloaded from a more distant cache level or DRAM for GEMM packing and computation. Removing Im2Col allows for Packing On Demand, which improves cache locality by keeping the tiles in the cache after packing for the micro-kernel. Furthermore, the tiling strategy determined by CSA optimizes tile reuse by scheduling in order to minimize the data flow between memory levels. However, the addition of explicit padding in SConv adds extra loads, which may lead to more cache misses.

SConv has fewer loads than Base in 63% of all tested convolutions (245). Even when SConv has more loads than Base (148, 37%), the impact on cache misses is not significant: SConv has more L1 misses on 0.5% of the convolutions (2), more L2 misses on 4% (17) and more L3 misses on 5% (21). In these cases, the difference is small.

The miss rates shown in Figure 13 are for all loads in the model. For SConv, around 90% of all loads resolve in L1 cache, compared with 83% for Base — a 1.9× improvement. The difference between the methods increases at lower cache levels. On average, SConv has 5.8x fewer L2 misses and 9.9x fewer L3 misses, though both represent a small percentage of the total number of loads. In the case of VGG-16, SConv's improvements on larger convolutions resulted in a big drop in L3 misses because, with the input not fitting into the cache, data has to be reloaded from DRAM in Base, which does not happen in SConv. Not only SConv is better at minimizing expensive memory operations, but it also often requires fewer load operations to compute the convolution.

8 RELATED WORK

Chellapilla *et al.* proposed the computation of convolution using the Im2Col transformation to reduce the problem to GEMM and solve it using a BLAS library [4]. The BLAS's GEMM routine already contains packing steps. Thus two separate steps require data manipulation: expansion (Im2Col) and packing. This approach has a large memory footprint because of the large matrix resulting from Im2Col. Also, the GEMM tiling can be inefficient when the inner dimension is large. Other authors address the memory footprint issue while still using BLAS GEMM as a foundation to solve the convolution problem [1, 7, 31]. However, they still use GEMM's tiling with multiple data-manipulation steps.

Zhang *et al.* [35] discuss the limitations and inefficiencies of the Im2Col + BLAS convolution algorithm and present an argument for using direct convolution instead. They use a model architecture with fused multiply-add instructions to explore cache blocking and data reuse, given a friendly memory layout. Unlike the work proposed by Zhang *et al.* , SConv explores more scheduling strategies and tile distribution in the cache hierarchy, provides a template that can be used in different architectures, and does not impose a different memory layout. Thus SConv can be deployed in other AI/ML frameworks without changing any other operator implementation.

Goto and de Gejin [9] introduce the idea of exploiting an optimized micro-kernel (called "inner-kernel") by tiling the problem in an external macro-kernel. These tiles are then packed to a friendly layout for the micro-kernel. The CSA algorithm resembles the algorithm proposed by Goto and de Gejin in its functionalities, but it applies the strategy to convolution instead of applying it to GEMM.

Li *et al.* [20] proposes a tiling algorithm that initially assesses how different permutations (loop orders) of the nested tiling loops may affect data movement. This analysis results in eight different permutations that are evaluated through a cost model that uses a non-linear optimization problem (min-max) to produce tile sizes and tries to reduce the data movement between each cache level. CSA goes in the same direction, but it works by first determining the tile sizes, the number of tiles produced from the convolution data, and how these tiles are distributed over the caches.

Patabandi *et al.* [24] describes an analytical model that aims to identify from a search space a solution that minimizes data movement overheads for convolutions. Their solution is composed of a cost model that quantifies the number of accesses to the convolution data, starting from the innermost loop and going all the way to the outermost loop. CSA also works by quantifying the memory accesses to the convolution data at each cache level to decide on a solution through a cost model. But instead of analyzing each loop individually, CSA estimates costs from a tile perspective and does not assume that the multiple accesses to a tile must be served from the same cache level.

Tollenaere *et al.* [28] design a solution that selects up to two micro-kernels which together are divisible by the output height dimension, as a way to avoid partial tiling. Similar to CSA, they also prioritize the input-channel dimension to reuse the partial accumulations but differ in how the tile sizes are selected. CSA selects the tile sizes aiming to maximize data reuse at each cache level in a deterministic fashion: either the input or the filter tiles are kept stationary. In contrast, Tollenaere *et al.* randomly select a subset of solutions to validate on hardware, sometimes taking several minutes to achieve that. Contrary to CSA, they randomly select which convolution data is tiled at each memory hierarchy level.

Park *et al.* [23] propose mGEMM, a zero-memory-overhead solution for mobile CPUs that requires a specific memory layout for better memory access. The tiling in this work prioritizes the accesses through the input-channel dimension to reuse the accumulators in registers, and the filters are tiled in L2 while the input tiles are kept in L1. Wang *et al.* [32] also propose a zero-memory-overhead solution for mobile CPUs. Their tiling focuses on keeping the filter elements in registers until they are no longer needed and continuously fetches elements from the input and output tiles from L1. To reduce the memory footprint, they also divide the input channels into groups of tiles. In CSA, these tiling solutions are aligned with different scheduling strategies, which enable better memory reuse, depending on the micro-kernel and on the architecture.

Juan *et al.* [17] modify the BLIS library's [29] GEMM routine to apply Im2Col in its packing step. The result is a convolution and GEMM hybrid named ConvGEMM, which takes advantage of Goto and de Gejin's GEMM tiling and structure with the Im2Col transformation applied on the fly, eliminating the double-packing problem. This process is similar to Packing on Demand, but the tiling is still GEMM-based, that is, 2D slicing without the concept of scheduling, and therefore has different cache usage, tile reuse, and packing patterns. Dukhan proposes an indirect-convolution algorithm that modifies GEMM all the way to the micro-kernel to avoid materializing the Im2Col matrix in memory [8].

Korostelev *et al.* [18] combine the ideas of modifying the GEMM routine and decomposing convolution into multiple GEMM operations [1] to create a new method that avoids packing redundancy while keeping the overall GEMM structure (tiling, packing, micro-kernel). Contrary to the approach in SConv, [18] works only on unitary stride convolutions and improves only non-pointwise convolutions. Moreover, their experiments focus on isolated convolutions and do not provide results for complete models.

Following the work from Zhang *et al.* [35], Barrachina *et al.* [2] propose two new direct-convolution algorithms for the NHWC layout (batch N, height H, width W, and channels C) on ARM processors. Like SConv, they tile in the channel dimension and use a BLAS micro-kernel. In SConv, however, besides considering the partitioning of the input tensor and of the filter set, tiling also considers the two scheduling alternatives — input or weight stationary — and the tile distribution over the cache hierarchy. SConv shows that direct convolution can benefit from: (a) novel ISA extensions, such as the POWER10 MMA, for faster tensor-algebra computation; (b) vector-register extensions, such as Intel x86 AVX and POWER VSX, to implement Vector-Based Packing. The experimental evaluation demonstrates the flexibility of SConv by using two different CPU architectures and the common NCHW layout.

An adaptation of the GEMM routine to a convolution algorithm by lazy Im2Col packing was made for GPU hardware by Chetlur *et al.* [6]. Yan *et al.* [34] also focused on GPUs with an optimized Winograd convolution method. Jordá *et al.* [16] eliminates the data transformation requirement for coalesced accesses with the CuConv algorithm. Adapting SConv to GPUs would require a different tiling analysis focusing on the SIMT properties of the hardware and memory coalescing instead of CPU cache hierarchies. This variation would require fundamental changes to the algorithm described in this work but may preserve the compilation flow described in Section 3.

This project follows the work done by Sousa *et al.* [26] in tiling 3D convolutions targeting NPUs, by generalizing its tiling analysis for CPUs in the CSA pass. In that work, scheduling strategies are used in a different context, and the modeling focuses on taking advantage of memory bursts while loading data to the scratchpad memories. SConv, however, creates a tiling solution suitable for CPUs with a traditional cache hierarchy. Different accelerator architectures may have distinct sensibilities, such as a larger focus on massive parallelism or a specific memory structure with programmable scratchpad memories. A completely separate tiling analysis might be required to account for these variations.

9 CONCLUSION AND FUTURE WORK

As stated in the introduction, the objective of the algorithm presented in this paper is to provide a good implementation of direct convolution for different CPU architectures. SConv achieves this goal, outperforming Im2Col + BLAS in experiments performed in realistic conditions with full machine learning model inference (Subsection 7.3) on both tested architectures by reducing total data manipulation time (Subsection 7.5). SConv also outperforms regular BLAS GEMM when computing pointwise convolutions in over 83% of the 219 instances tested (Subsection 7.6).

Currently, the CSA tiling analysis pass creates tiles based solely on micro-kernel information and the IN_c value. More robust space exploration can be done by relaxing the spatial constraints of the tiles to better utilize the L1 cache in convolutions with few channels. Another alternative would be to modify CSA/CSO to consider multiple tiles in the L1 cache. Moreover, since the approach is integrated with machine-learning compilers, a fully code-generation-based implementation can leverage compile-time information to simplify operations such as padding and remove boilerplate code overhead, if a common abstraction for architectures with different features is provided [30]. Finally, a hybrid solution with Im2Col + BLAS might perform best for each convolution.

REFERENCES

[1] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2020. High-Performance Low-Memory Lowering: GEMM-based Algorithms for DNN Convolution. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 99–106. https://doi.org/10.1109/SBAC-PAD49847.2020.00024

[2] Sergio Barrachina, Adrián Castelló, Manuel F. Dolz, Tze Meng Low, Héctor Martínez, Enrique S. Quintana-Ortí, Upasana Sridhar, and Andrés E. Tomás. 2023. Reformulating the direct convolution for high-performance deep learning inference on ARM processors. *Journal of Systems Architecture* 135 (2023), 102806. https://doi.org/10.1016/j.sysarc.2022.102806

[3] João P. L. de Carvalho, José E. Moreira, and José Nelson Amaral. 2022. Compiling for the IBM Matrix Engine for Enterprise Workloads. *IEEE Micro* (2022), 1–8. https://doi.org/10.1109/MM.2022.3176529

[4] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, Suvisoft, La Baule (France). https://hal.inria.fr/inria-00112631

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan M. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *ArXiv* abs/1410.0759 (2014).

[7] Minsik Cho and Daniel Brand. 2017. MEC: Memory-Efficient Convolution for Deep Neural Network. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 815–824.

[8] Marat Dukhan. 2019. The Indirect Convolution Algorithm. *ArXiv* abs/1907.02129 (2019).

[9] 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), 25 pages. https://doi.org/10.1145/1356052.1356053

[10] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[12] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *ArXiv* abs/1602.07360 (2016).

[13] Intel Corporation 2022. *oneAPI Specification*. Intel Corporation. Retrieved May 19th, 2023 from https://spec.oneapi.io/versions/latest/index.html

[14] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (Lille, France) *(ICML'15)*. JMLR.org, 448–456.

[15] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[16] Marc Jordà, Pedro Valero-Lara, and Antonio J. Peña. 2022. CuConv: CUDA Implementation of Convolution for CNN Inference. *Cluster Computing* 25, 2 (apr 2022), 1459–1473. https://doi.org/10.1007/s10586-021-03494-y

[17] P. San Juan, A. Castello, M. F. Dolz, P. Alonso-Jorda, and E. S. Quintana-Orti. 2020. High Performance and Portable Convolution Operators for Multicore Processors. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE Computer Society, Los Alamitos, CA, USA, 91–98. https://doi.org/10.1109/SBAC-PAD49847.2020.00023

[18] Ivan Korostelev, João P. L. De Carvalho, José Moreira, and José Nelson Amaral. 2023. YaConv: Convolution with Low Cache Footprint. *ACM Trans. Archit. Code Optim.* 20, 1, Article 18 (feb 2023), 18 pages. https://doi.org/10.1145/3570305

[19] Tung D. Le, Gheorghe-Teodor Bercea, Tong Chen, Alexandre E. Eichenberger, Haruki Imai, Tian Jin, Kiyokuni Kawachiya, Yasushi Negishi, and Kevin O'Brien. 2020. Compiling ONNX Neural Network Models Using MLIR. *ArXiv* abs/2008.08272 (2020).

[20] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. 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.

[21] José E. Moreira, Kit Barton, Steven Battle, Peter Bergner, Ramon Bertran, Puneeth Bhat, Pedro Caldeira, David Edelsohn, Gordon Fossum, Brad Frey, Nemanja Ivanovic, Chip Kerchner, Vincent Lim, Shakti Kapoor, Tulio Machado Filho, Silvia Melitta Mueller, Brett Olsson, Satish Sadasivam, Baptiste Saleil, Bill Schmidt, Rajalakshmi Srinivasaraghavan, Shricharan Srivatsan, Brian W. Thompto, Andreas Wagner, and Nelson Wu. 2021. A matrix math facility for Power ISA(TM) processors. *CoRR* abs/2104.03142 (2021). arXiv:2104.03142 https://arxiv.org/abs/2104.03142

[22] ONNX 2019. *ONNX Model Zoo*. Retrieved June 26th, 2022 from https://github.com/onnx/models

[23] 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*. 222–234.

[24] Tharindu R Patabandi, Anand Venkat, Rajkishore Barik, and Mary Hall. 2021. SWIRL++: Evaluating Performance Models to Guide Code Transformation in Convolutional Neural Networks. In *Languages and Compilers for Parallel Computing: 32nd International Workshop, LCPC 2019, Atlanta, GA, USA, October 22–24, 2019, Revised Selected Papers 32*. Springer, 108–126.

[25] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2015).

[26] Rafael Sousa, Byungmin Jung, Jaehwa Kwak, Michael Frank, and Guido Araujo. 2021. Efficient Tensor Slicing for Multicore NPUs using Memory Burst Modeling. In *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE Computer Society, Los Alamitos, CA, USA, 84–93. https://doi.org/10.1109/SBAC-PAD53543.2021.00020

[27] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[28] Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P Sadayappan, and Fabrice Rastello. 2023. Autotuning Convolutions is Easier Than You Think. *ACM Transactions on Architecture and Code Optimization* 20, 2 (2023), 1–24.

[29] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33. https://doi.acm.org/10.1145/2764454

[30] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. arXiv:2202.03293 [cs.PL]

[31] Aravind Vasudevan, Andrew Anderson, and David Gregg. 2017. Parallel Multi Channel convolution using General Matrix Multiplication. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 19–24. https://doi.org/10.1109/ASAP.2017.7995254

[32] Qinglin Wang, Dongsheng Li, Songzhu Mei, Siqi Shen, and Xiandong Huang. 2020. Optimizing one by one direct convolution on armv8 multi-core cpus. In *2020 IEEE International Conference on Joint Cloud Computing*. IEEE, 43–47.

[33] Zhang Xianyi, Martin Kroeker, Werner Saar, Wang Qian, Zaheer Chothia, Chen Shaohu, and Luo Wen. [n. d.]. OpenBLAS: An optimized BLAS library. https://www.openblas.net/

[34] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing Batched Winograd Convolution on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 32–44. https://doi.org/10.1145/3332466.3374520

[35] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. 2018. High Performance Zero-Memory Overhead Direct Convolutions. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 5776–5785. http://proceedings.mlr.press/v80/zhang18d.html

[36] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.