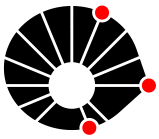
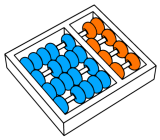




Improving Convolutions with Tensor Hardware Accelerators

V. F. Ferrari *G. C. S. Araujo*

Relatório Técnico - IC-PFG-21-29
Projeto Final de Graduação
2021 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Improving Convolutions with Tensor Hardware Accelerators

Victor Ferrari* Guido Costa Souza de Araújo*

Abstract

Convolutional Neural Network (CNN) models are among the most popular choices for deep learning solutions to problems with huge data sets. Given that CNNs are very computationally expensive, optimizing convolutions is central to enable larger models and speed up inference time. Tensor operations, e.g. matrix multiplication, have increasingly relied on hardware accelerators, such as IBM POWER10's MMA engine.

This work explores how to exploit MMA and the POWER10 architecture to improve convolution performance, and proposes a novel algorithm for the operation, named Convolution Slicing Optimization (CSO), which tiles the instance into multiple subproblems, and schedules the resulting tiles so to minimize DRAM memory accesses. After the convolution is tiled, a micro-kernel is used to increase throughput with the MMA engine.

To evaluate the proposed approach, a set of experiments was performed using a POWER10 CPU, and the results show that CSO is capable of efficiently tile the convolution according to a set of parameters calculated at compile time. Speedups of up to 229% result when comparing the CSO convolution-based slicing technique to a widely used reduction to matrix multiplication.

1 Introduction

The world is becoming increasingly connected, and the volume of data to be processed in a lot of applications is in constant expansion. In this context, machine learning algorithms, more specifically deep learning and neural networks, are rapidly becoming important and powerful tools for building computational models that are able to represent, in a non-analytical way, complex problems involving a large number of variables in a multidimensional hyperspace.

Convolutional Neural Networks (CNN) are particularly useful tools to address problems with huge data sets captured from real-world sensors. The steady increase in the adoption of CNNs is driven mostly by applications in the Computer Vision domain, where it addresses problems like Object Recognition [1, 2, 3], Object Detection [4], and Video Classification [5]. Other areas, like Speech Recognition and Natural Language Processing (NLP) have also benefited from the application of CNN models [6, 7].

Since their origins, the size and complexity of state-of-the-art CNNs have grown significantly. For instance, LeNet-5 [9], a model that recognizes handwritten digits, has less than

*Institute of Computing, University of Campinas (UNICAMP), 13081-970, Campinas-SP, Brazil.

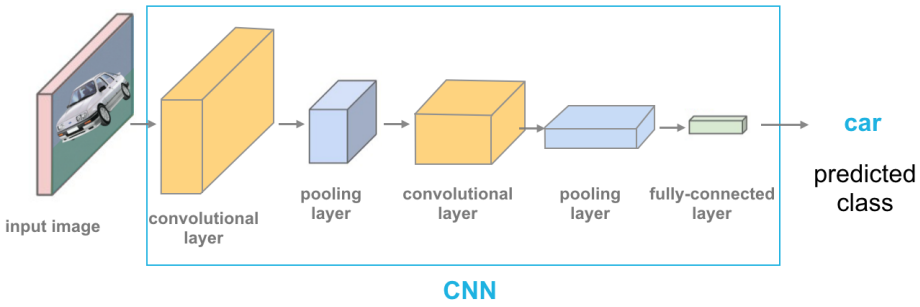


Figure 1: Convolutional Neural Network (CNN) example diagram, for an image classification problem [8].

1 million parameters, while Inceptionv3 [10], which classifies thousands of different object categories, has more than 23 million parameters.

It is well-known that convolution is the most expensive operation of a CNN, accounting for the largest share of a CNN execution [11]. Furthermore, this operation is also important for other applications, such as image processing and other types of signal processing.

Bigger CNN models contain larger convolutions that not only demand more computational power, but also result in a significant increase in data movement between different levels of the memory hierarchy. Consequently, improving convolution performance requires attention to both hardware throughput and memory access.

With the imminent end of Moore's law, advances in CPU architecture have increasingly relied on hardware accelerators via ISA extensions, given their better power/performance metrics for tensor-intensive application domains. Therefore, convolutions could also benefit from this specialized hardware.

1.1 Contributions

This work has the main goal of exploiting the MMA engine [12] and the IBM POWER10 architecture to develop new algorithms that can improve the performance of convolution operations.

Most convolution algorithms used in machine learning frameworks are reductions to other problems, which are then solved by state-of-the-art algorithms implemented in third-party libraries [11, 13]. In this context, the contributions of this work are:

- The construction of a convolution algorithm from the ground up.
- The implementation of a faster convolution process for POWER10 architecture.
- The structure for a possible future compiler intrinsic implementation for POWER10 [14].

This project resulted in an exhibit named "Speeding-up CNN Models using POWER10 MMA Extensions", presented at the CASCON x EVOKE 2021 conference, that took place in November, 2021.

This report is organized as follows. Section 2 discusses the main concepts explored in this work, such as convolutions, CPU architecture and relevant algorithms used for reference and motivation. Other related works are described and put into perspective in Section 3. Section 4 explores how to use IBM's POWER10 architecture to accelerate convolutions with a micro-kernel. Sections 5 and 6 detail proper convolution tiling and packing to achieve good performance and improve memory usage. The results achieved from this work are shown and analyzed in Section 7. Finally, Section 8 summarizes the proposed approach and discusses future works.

2 Background

This section explores some of the main concepts required to understand the project.

2.1 Convolution

A convolution is a mathematical operation with many applications in mathematics, physics, engineering and computer science. Usually, this is an operation between two functions $(f * g)$, but in computer science discrete convolutions are used, and sequences take the place of functions. Usually, these sequences take the form of n-dimensional tensors.

Image filtering is a useful convolution application that can be found in CNNs and image processing. In this case, the convolution is applied between a set of filters (or kernels) and an input image, and the output is a filtered image. In CNNs, the kernels are also called weights (ω). The definition of a 2D convolution in image processing is as follows, where f is the input image. This process is shown in Figure 2.

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy)$$

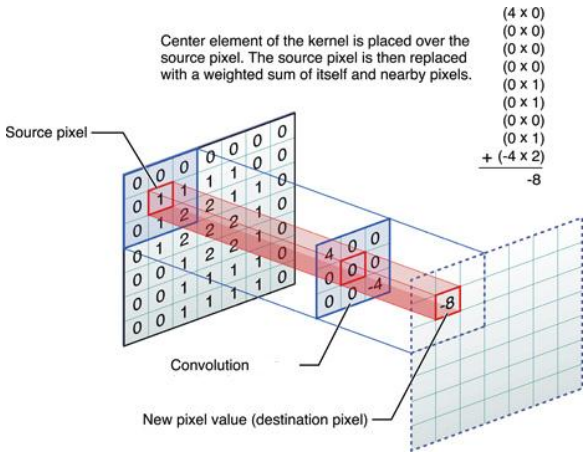


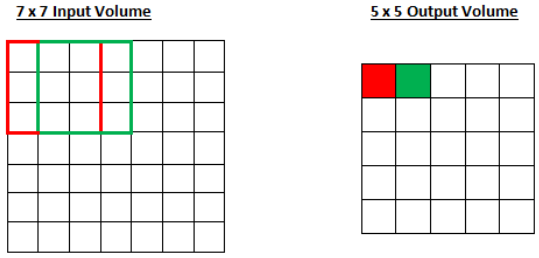
Figure 2: Illustration of an image filtering convolution [15]

Each output element is the result of the weighted sum of the input elements in the same window, which is the abstract structure obtained by superimposing the filter on the input image centered on each element. In 3D convolutions, both kernel and input image have a set of channels, and this process is applied to all channels. The results from each channel are added into the same output elements.

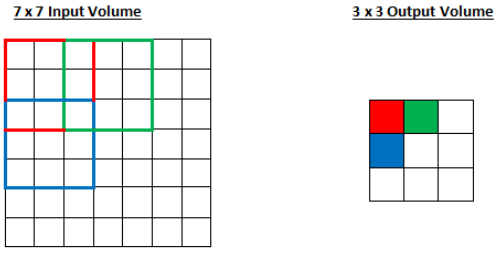
A convolution layer of a CNN consists of two input tensors and one output tensor. The input tensors are an image with dimensions $C \times H \times W$ and a set of M filters with dimensions $C \times k_H \times k_W$. In most cases, $k_W = k_H$, so the windows are square.

The output tensor may have dimensions $M \times H \times W$, but this is not always the case. The output height and width depend on stride and padding. One input image element can only be the center of a window, therefore corresponding to an output value, when there are enough neighboring elements to create a window. This means that the border elements are either ignored, or padding is added to the image to create these neighbors. Without padding, the output tensor has dimensions $M \times (H - kH + 1) \times (W - kW + 1)$.

Another important convolution parameter is the stride. Usually, convolutions use unitary stride, so every input element corresponds to a window, and therefore an output value. With a generic stride s, s elements are skipped between windows in that direction, so the filter is dislocated with a step of s. There can be different strides for both directions. Examples of convolutions with different strides and how this affects the output can be seen in Figure 3. The output tensor from convolutions with padding and stride has dimensions $M \times \frac{H}{s_H} \times \frac{W}{s_W}$.



(a) Stride 1/1



(b) Stride 2/2

Figure 3: Example of convolutions with the same input image, but different strides [16].

2.2 GEMM: General Matrix Multiplication

Matrix multiplication is one of the most important tensor operations, so it has long been the primary operation in linear algebra libraries and the focus of hardware accelerators. This problem has been extensively studied over the years and decades, culminating in very fast implementations that take advantage of architecture.

GEMM (General Matrix Multiplication) is a standard operation that implements regular matrix multiplication between two 2-dimensional arrays with the following signature.

$$C = \alpha \times (A \cdot B) + \beta \times C$$

If the matrix A has dimensions $m \times k$ and B has dimensions $k \times n$, the resulting product has dimensions $m \times n$. k is called the inner dimension of the operation, since it is common between the matrices.

2.2.1 BLAS

The BLAS (Basic Linear Algebra Subprograms) library, originally released in 1979, sought to implement low-level routines for linear algebra operations such as vector and matrix multiplication and addition. Since then, it has become the general specification for a set of standard routines that spawned many implementations, such as OpenBLAS [17], BLIS [18] and others.

BLAS is the most relevant linear algebra library, defining operations such as GEMM. The state-of-the-art implementations of GEMM are based on the work of Goto and de Geijn [19]. With a regular simple matrix multiplication algorithm, but correctly using the memory hierarchy for data reuse and intelligently tiling the matrices into smaller problems, this implementation achieves near-optimal operational intensity, and provides a base for most (if not all) BLAS implementations since. A simple model for the memory hierarchy can be seen in Figure 4.

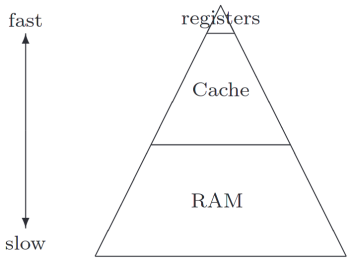


Figure 4: Simple model for the hierarchical memories, viewed as a pyramid [19].

Goto's GEMM uses a structure with an outer kernel (also called macro-kernel) and an inner kernel (also called micro-kernel). The latter provides a high-performance low-level routine for the actual computation, using what the architecture offers without much memory manipulation, and the former provides a high-level memory-focused way of using the micro-kernel as best as possible.

2.3 Im2Col: Image To Column

The most common and widely used convolution algorithm in machine learning frameworks and beyond is a reduction to GEMM, named Im2Col [11]. In this method, the input image is transformed, and the result is multiplied with the weight matrix using BLAS's GEMM routine. Since BLAS GEMM is in theory very optimized, this allows for a fast solution with little extra effort and maintenance, with better memory access than a naive direct convolution implementation.

Each output element of the convolution is computed from a series of accumulated multiplications between pairs of elements from the window and kernel, as explained in Section 2.1 and Figure 2. Therefore, it can be modeled as the inner product between two arrays.

Expanding this idea to the entire convolution, one filter can be applied to the input image with inner products between its flattened array and the flattened array from each input window. If a larger matrix is made from the expansion of each window (or patch) into a column, this can be solved with BLAS's GEPM method [19]. Adding multiple flattened filters to the kernel tensor, one per row, the solution is the multiplication of these two matrices, which can be achieved with GEMM.

The process of expanding every convolution window into one column of a larger matrix is named Im2Col, or Image to Column. The result is named input patch matrix, with dimensions $(C \times k_H \times k_W) \times (H \times W)$, each element converted into a column with the contents of its window, including every channel. Since the window size is the same as the kernel size, this is the inner dimension, so the result has dimensions $M \times HW$.

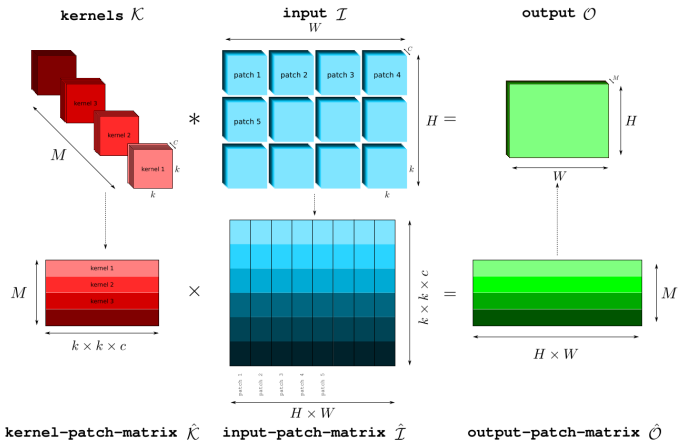


Figure 5: Im2Col + GEMM method with square filters [13].

Depending on the layout and read order (row-major or column-major), reordering of the kernel elements may not be necessary. The output is already in the correct layout. Sometimes the Im2Col process is referred to as packing, not to be confused with GEMM packing, which happens inside BLAS.

2.3.1 Problems

Since it is easy to implement, Im2Col and similar methods are used to reduce convolutions to GEMM in many machine learning frameworks, such as PyTorch [20] and Caffe [21]. However, using this process is not as efficient as it seems.

One problem of Im2Col is memory utilization. The input image is expanded by a factor of $O(k_H \cdot k_W)$ during the transformation, which can be very expensive for larger instances.

A second problem is in the GEMM routine. Goto's algorithm is optimized for common high performance computing and scientific matrix shapes, in which the inner dimension is small when compared to the outer dimensions. The resulting patch matrix from Im2Col, however, may not fall into that category. The inner dimension ($k_H \times k_W \times C$) is in most cases larger than M, and sometimes even larger than $H \times W$ [22]. Therefore, using BLAS GEMM can result in suboptimal performance.

Since there is already a packing step inside of the GEMM routine, with the addition of Im2Col there are two data movement steps, which affect cache behavior and result in expensive memory operations.

2.4 POWER10 and MMA

In 2020, IBM published the specification of Power ISA v3.1 [23], with the set of instructions for the upcoming POWER10 architecture. This introduced the Matrix-Multiply Assist (MMA) [12] instructions as an extension of the Vector-Scalar Extension (VSX) facility.

These instructions serve as hardware accelerators for matrix multiplications, performing rank-k operations with 128-bit vector-scalar registers (VSRs) as inputs. The result is stored in new 512-bit accumulator registers that represent matrices. There are eight accumulators, and each one is associated with four VSRs, which become unusable while the corresponding accumulator is assembled.

Rank-k operations compute the product between a $m \times k$ matrix and a $k \times n$ matrix, resulting in a $m \times n$ matrix. In MMA, the k factor is related to the size of the data type, which can be up to 8 (4-bit integers) or down to 1 (32-bit and 64-bit floating point numbers). The result is either a 4×4 single-precision 32-bit floating point (`fp32`) matrix or a 4×2 double-precision 64-bit floating point (`fp64`) matrix, stored in an accumulator.

The basic rank-1 operations are outer products, which can be used to compute higher rank operations by accumulating the results. Furthermore, outer products have high computational density, since mn elements are computed from $m + n$ input values. In most processors, this operation has to be emulated, but MMA provides direct hardware support. The outer product between a u vector of size m and a v vector of size n is given by the following equation.

$$u \otimes v = \begin{bmatrix} u_1v_1 & u_1v_2 & \cdots & u_1v_n \\ u_2v_1 & u_2v_2 & \dots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \dots & u_mv_n \end{bmatrix}$$

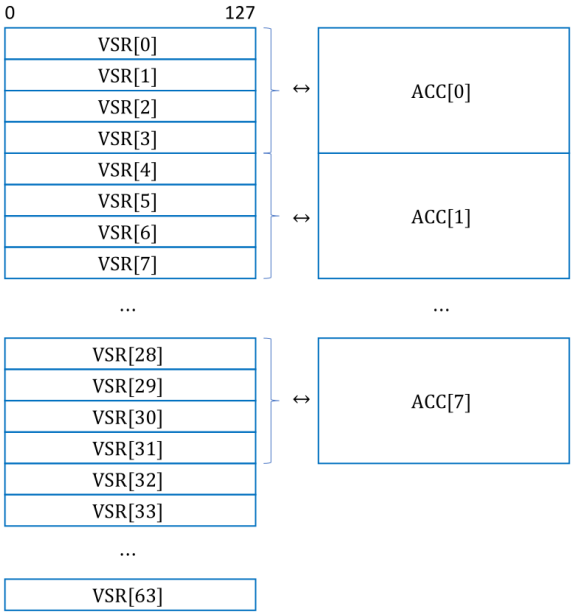


Figure 6: MMA accumulators and corresponding VSRs [12].

A larger accumulator can be emulated by combining all eight accumulators and multiple instructions. An example can be seen in Figure 7. The MMA engine can easily be used to speed up GEMM, but other tensor-related operations may also benefit from these new operations.

The MMA instructions can be accessed via compiler built-ins, which act as wrapper functions for the low-level instructions with compiler vector types [24, 25]. This also allows for emulation in other architectures.

2.4.1 Dot Product With Outer Products

Rank-k update operations can be computed by rank-1 outer products. Therefore, matrix multiplication can be done as a series of accumulating outer products. Instead of completing a direct output value per instruction, one outer product operation computes one term from each output element.

To do this efficiently, the memory layout should be modified, since in $A \cdot B$ one column of A and one row of B are the inputs to the outer product. A should be in column-major format, and B should be in a row-major layout. An example of decomposing dot product (or GEMM) into outer products can be seen in Figure 8. This reordering happens inside the packing routines in BLAS GEMM.

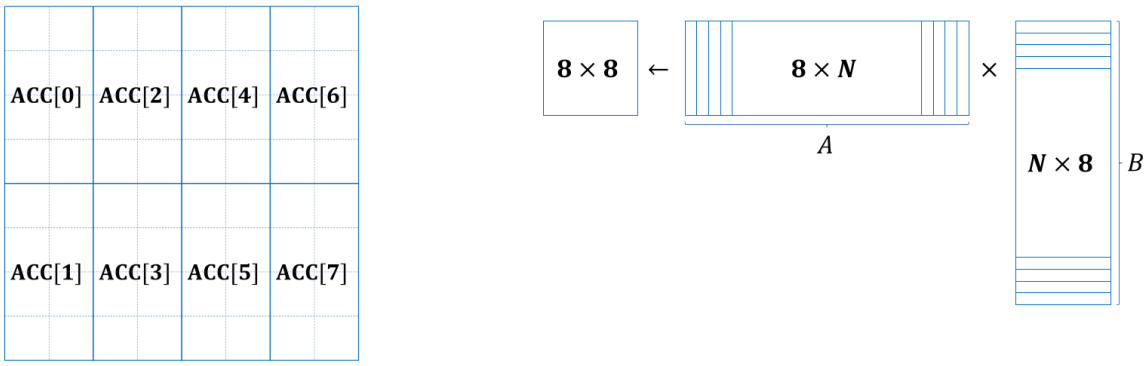


Figure 7: Example of GEMM with 64-bit floating point numbers using all eight accumulators. By accumulating N outer products, an 8×8 output can be computed [12].

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & b_{12} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} \\ a_{21}b_{11} & a_{21}b_{12} \end{bmatrix}$$

$$\begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix} \otimes \begin{bmatrix} b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{12}b_{21} & a_{12}b_{22} \\ a_{22}b_{21} & a_{22}b_{22} \end{bmatrix}$$

Figure 8: Decomposition of a 2×2 matrix multiplication into outer products.

3 Related Works

With the rise of CNNs in machine learning, convolutions became important operations to many applications, and research on the topic increased. Many approaches were studied, such as improvements to the Im2Col procedure described in Section 2.3, other reductions to GEMM, and directly computing the convolution with better data locality. This section explores some of the works that directly or indirectly influenced this project.

3.1 Low-Memory GEMM Reductions

To solve one of the main problems of Im2Col, its memory footprint, Anderson et al. [13] developed a series of convolution reductions to GEMM with increasingly less extra memory required.

This extra memory is replaced with more GEMM calls, with the intuition that a convolution with kernels of size 1×1 is equivalent to GEMM, scaling each pixel by one value in each kernel and summing up the different channels, and a $k \times k$ convolution can be decomposed into k^2 1×1 convolutions. With this approach and clever border case treatment,

similar performance to the Im2Col + GEMM method was achieved, with even better results in embedded and other low-memory devices. This approach solved one of the problems mentioned in Section 2.3.1, but to do that, added the overhead of calling GEMM multiple times.

3.2 ConvGEMM and Packing On-The-Fly

Instead of using BLAS GEMM as a black box, Juan et al. [26] reduced the extra memory requirement for convolution by altering the BLIS library [18] GEMM implementation. The result is a convolution method that uses the tiling structure from Goto's method, with new packing routines to apply the Im2Col transform on-the-fly.

This method removes the need for $O(k_H \cdot k_W)$ expansion in memory, with only small extra buffers required inside the packing routines, eliminates the double packing problem from Im2Col, and takes advantage of the multi-threaded parallelization that BLIS provides. The performance is similar to the standalone GEMM call with the full input patch matrix.

3.3 High Performance Direct Convolutions

In the midst of the development of many GEMM-based convolution methods, Zhang et al. [22] provided an argument for direct convolutions instead. This work documents the problems described in Section 2.3.1 and includes a breakdown on how to properly implement direct convolutions for a model architecture with a predetermined data layout.

The main problem with usual direct convolutions is bad memory locality, so a favorable data layout, combined with tiling the input and kernels for register and cache sizes, result in near-optimal performance. This method combines correct architecture usage with better parallelism and a problem-specific bottom-up approach.

A convolution can be modeled as a series of FMA (fused multiply-add) operations, and most modern processors have hardware support for this instruction. With clever blocking and a highly specific data layout, no memory replication is needed. The achieved throughput is similar to BLAS GEMM for HPC matrices, close to the theoretical peak for some architectures, and up to 4 times better than the Im2Col + GEMM approach.

The downside to this approach is that the starting data layout is affected by architecture information, so at some point there needs to be a data reordering step. There is also no direct support for higher throughput operations such as outer products. Nonetheless, it shows the importance of data locality and convolution-specific tiling to achieve the best performance.

4 MMA Convolution: The Micro-Kernel

This section explores how to exploit the MMA instructions and accumulators from POWER10 described in Section 2.4 to compute convolutions. This means analyzing convolution more closely, and establishing various methods for different data types.

4.1 Convolution as FMA Operations

Section 2.1 provides mathematical definitions for convolutions and describes the process through which it is calculated. To compute it, only two basic floating point operations (FLOPS) are required: scalar multiplication and addition. For each output element, every element of its window multiplies one corresponding value from the kernel, and the results are accumulated. Luckily, most modern processors include FMA (fused multiply-add) instructions, which combine those two operations [22]. A naive implementation is written as pseudo-code in Algorithm 1.

Algorithm 1 Naive Convolution Algorithm [22]

Input: Input I, Kernel Weights F
Output: Output O
for $i \leftarrow 1$ to C **do**
  for $j \leftarrow 1$ to M **do**
    for $k \leftarrow 1$ to W **do**
      for $l \leftarrow 1$ to H **do**
        for $m \leftarrow 1$ to k_W **do**
          for $n \leftarrow 1$ to k_H **do**
            $O_{j,k,l} += I_{i,k+m,l+n} \times F_{i,j,m,n}$

This means that a convolution can be perceived as a set of FMA instructions between corresponding elements of kernels and windows of the input image. Let A be the set of FMA operations between every triple of elements from the kernels, input image and output. Convolution can be defined as a subset of A restricted by the process described earlier, given in Algorithm 1 by the i, j, k, l, m, n indices and their usage.

Reducing this apparently complex operation into its most basic form allows for separation between the actual computation of the FMA operations, done by the micro-kernel, and data manipulation, done by the macro-kernel. The rest of this section will focus on the micro-kernel.

4.2 MMA Instructions

Algorithm 1's micro-kernel has only one FMA instruction per call. A simple way to improve performance is to vectorize it, applying vector FMA instructions, and thus increasing the throughput by computing multiple values per cycle. POWER10's MMA engine allows for even further vectorization, with tensor operations described in Section 2.4. The downside is that the data needs to be in a certain layout, adding a packing layer.

However, considering that the data is already available to the micro-kernel at minimal cost, the approach depends on the data size. For half-precision floating point values (`fp16`) and brain floating point format (`bfloat16`), the available instruction is a rank-2 update, with a dot product between two 4×2 matrices. For single-precision (`fp32`) and double-precision (`fp64`) floating point values, the instruction is a rank-1 update, an outer product

between flat vectors. There are special instructions for integers, but those won't be discussed in this work.

In any case, the concept of a complete input image is no longer useful. Since each output element is connected to a window, and windows may overlap, vectorizing the computation can mean replicating data from the input. So from now on, the input will be referred to as a set of windows.

4.3 Dot Product: fp16 and bfloat16

One output element of the convolution can be computed as the inner product between flattened vectors of one filter and one window, as can be seen in Figure 9. There is no inner product instruction in POWER10, but the rank-2 update instruction can serve a similar purpose, with more vectorization.

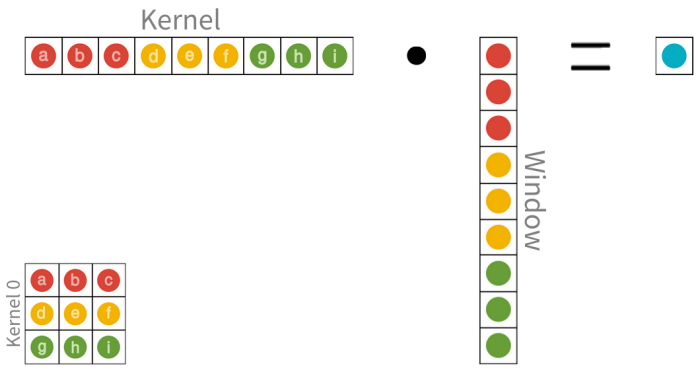
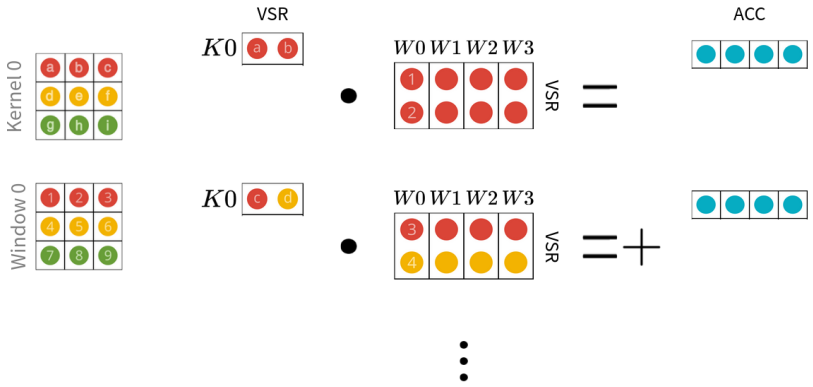


Figure 9: Convolution output element computed from the inner product of one kernel and one window of size 3×3.

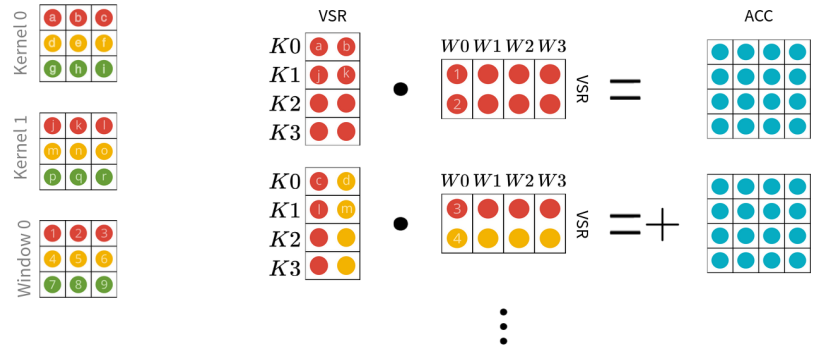
The `fp16` and `bfloat16` instructions consume 4×2 matrices, generating a 4×4 accumulator. Taking only 2 elements from each VSR, the result is the inner product of these vectors. If the VSRs contain kernel and window elements, a partial output value is obtained. Adding 3 more windows, 4 partial output elements are computed, as seen in Figure 10a.

To fully utilize the accumulator, more output elements need to be computed with the same loaded window values. This is only possible if there are multiple kernels in the convolutions, which is generally the case for CNNs. Every filter needs to be applied to every window, so the first VSR can be filled with elements from 3 other kernels. Repeating this process for every element from every channel, and accumulating the results, the final output values are obtained. This process is exemplified in Figure 10b.

This approach of calculating multiple partial inner products per instruction using the dot product rank-2 update MMA operation is optimal for floating point values in the POWER10 architecture, but it is not universal, since these instructions are only available for small data types. Although many applications use `fp16` and `bfloat16`, it is important to also handle single and double precision, which requires a different approach.



(a) Single Kernel



(b) Multiple Kernels

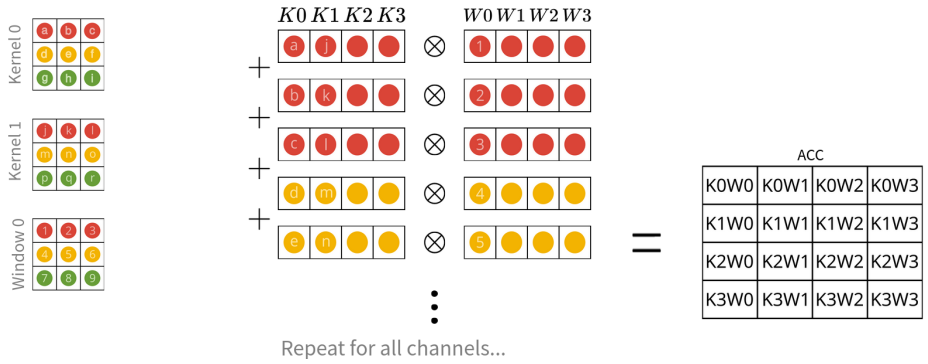
Figure 10: Dot product approach to convolution using MMA, with kernels and windows of size 3×3.

4.4 Outer Product: fp32 and fp64

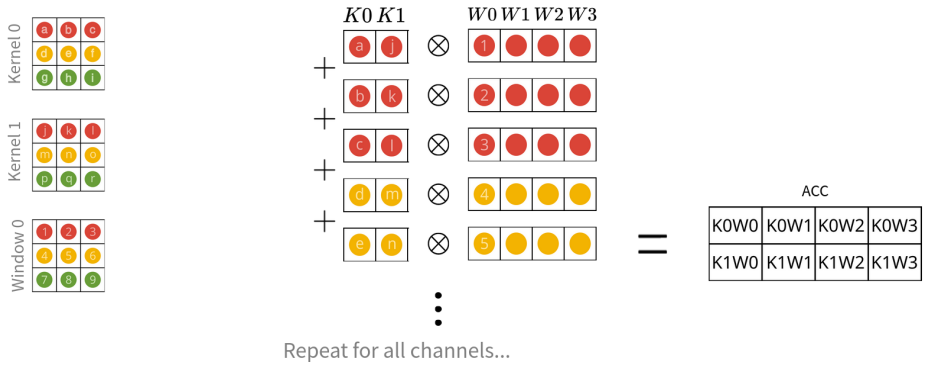
The `fp32` and `fp64` MMA instructions are rank-1 update, outer product operations. As discussed in Section 2.4.1, it is possible to convert a dot product into a series of outer products. Applying this logic to Figure 10b, the outer products should have the first column of the kernel VSR, and the first row of the window VSR. This means that the first VSR has one element from each kernel, and the second has one element from each window. This process is then repeated for every element in the filter and window, and every channel.

The `fp32` instruction consumes two vectors of size 4, with a 4×4 accumulator as a result. This can be seen in Figure 11a. This process is similar for `fp64`, with a size restriction, since the instruction uses two VSRs as one input of size 4, and one VSR of size 2 as the other, resulting in a 4×2 accumulator, as illustrated in Figure 11b.

This solution is universal, since smaller data types can be converted to `fp32` with simple cast instructions, but is not optimal for `fp16` and `bfloat16`.



(a) Single Precision Floating Point



(b) Double Precision Floating Point

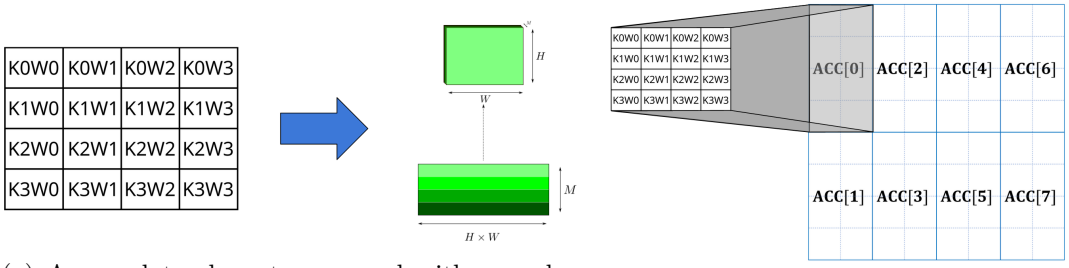
Figure 11: Outer product approach to convolution using MMA, with kernels and windows of size 3×3.

4.5 Multiple Accumulators

The output of both solutions is one accumulator matrix, in which each row represents a convolution with one kernel, and each column represents convolutions with one window. This output data layout is correct, so there is no need for unpacking or reordering, as can be verified in Figure 12a.

Each MMA instruction consumes VSRs and produces one accumulator, but POWER10 can have up to eight accumulators at once. Combining all available registers, a larger accumulator can be emulated, with a layout-dependent size. The best accumulator layout is the one which maximizes perimeter and area, using as many windows and kernels as possible and in every combination. This means a rectangle-like shape, such as the one in Figure 12b.

Since there are usually more windows then filters in a convolution, the larger dimension should be used for computing more windows. Therefore, the large accumulator created has a 2×4 shape, resulting in 8×16 elements per micro-kernel call for `fp32` and smaller, and 4×16 elements per call for `fp64`.



(a) Accumulator layout compared with convolution output [13].

(b) Using all accumulators for convolution [12].

Figure 12: Accumulator layout and combination.

4.6 Similarities to GEMM

Since the micro-kernel does not handle data reordering and packing, only the FLOPS are important to develop it, and both direct convolution and Im2Col + GEMM have the exact same FLOPS, what changes is the data layout. Taking the Im2Col structures from Figure 5 and converting the GEMM into outer products, the result is the method described in Section 4.4.

This means that, for the best layout for MMA convolution, a GEMM micro-kernel is enough, there is no need for a specific implementation. GEMM can be converted into convolution by changing the macro-kernel and keeping the underlying MMA structure [26].

5 Convolution Slicing Analysis

There's more to taking advantage of architecture than using specific components and low-level instructions. Section 4 describes how a micro-kernel can be implemented to use MMA instructions to increase throughput, but much of the performance can be lost by not correctly using the cache hierarchy and memory bursts. The program can be limited not by how much it can compute per cycle, but by how well it can get the data to the computation unit, and this is often the case as more powerful instructions are developed with hardware support.

A common solution to this potential issue when dealing with tensor operations is to divide the operation into smaller tiles [19], as briefly discussed in Section 2.2.1. In the regular Im2Col + GEMM method, tiling is handled by BLAS GEMM, but this can be sub-optimal for the matrices generated by Im2Col (Section 2.3.1), so it would be better to develop a convolution-specific tiling process.

The micro-kernel is used in this section as a black box, and assumed to be optimal. This being the case, the correct amount of data should be prepared so that this micro-kernel can run with as little stalls as possible. In other words, the data should be in the L_1 cache, and should be tiled so that each iteration completely uses this cache. These resulting tiles should then be arranged so to minimize loads from main memory by reusing as much data as possible from the L_2 and L_3 caches. This process is called Convolution Slicing

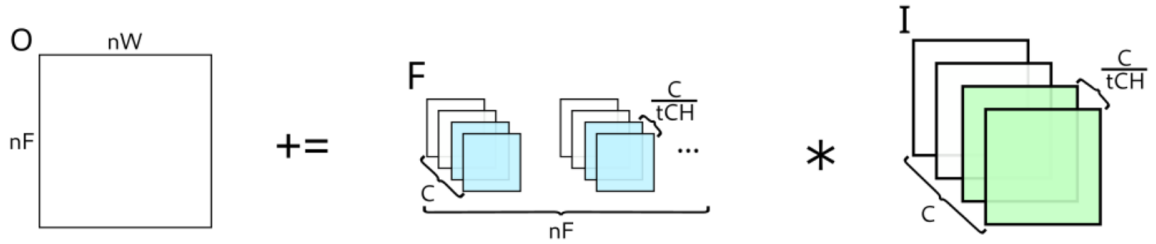


Figure 13: CSA channel tiling, where n_F and n_W are the amount of windows and filters in the convolution, C is the amount of channels in the input and t_{CH} is the amount of channel tiles to divide the convolution into.

Analysis (CSA), originally designed for NMP architecture [27] and adapted for general-purpose architectures.

5.1 Channel Tiling

Each micro-kernel execution computes output elements out of sets of windows and filters. These structures have the same dimensions for height (k_H) and width (k_W) and the same amount of channels (C), so these are the candidates for tiling. By reducing one of these dimensions, the output will no longer be complete until the rest is computed. k_H and k_W are usually very small compared to C, and different channels are usually very distant in memory, so tiling on the amount of channels is the clear choice.

The amount of channel tiles (t_{CH}) directly impacts the size of the sets of windows and filters. Knowing the size of the L_1 cache, the parameter is calculated by maximizing the following inequation with constraints, where I_{SIZE}, F_{SIZE}, O_{SIZE} are the sizes of the micro-kernel input, filter and output in bytes, respectively, SET_W and SET_F are the amounts of windows and filters computed by the micro-kernel, and D_{SIZE} is the data type length:

$$I_{SIZE} + F_{SIZE} + O_{SIZE} \leq |L_1|$$
$$I_{SIZE} = SET_W \times k_H \times k_W \times \frac{C}{t_{CH}} \times D_{SIZE}$$
$$F_{SIZE} = SET_F \times k_H \times k_W \times \frac{C}{t_{CH}} \times D_{SIZE}$$
$$O_{SIZE} = SET_W \times SET_F \times D_{SIZE}$$

One solution to this optimization problem is obtained with an iterative heuristic. By computing the convolution for only a fraction of the channels at a time, the L_1 cache is correctly used and data locality is improved. A global view is presented in Figure 13.

5.2 Scheduling

With the convolution properly tiled to fill the L_1 cache, the next step is to schedule the computation of each tile for the best temporal locality, so that DRAM accesses can be resolved in cache levels near the CPU. The convolution is now tiled by the amount of windows

and filters computed by the micro-kernel, and in channel sets. Since every combination of filter and window sets are computed, they can be arranged so that data is reused from the L_2 and L_3 caches.

There are two main ways for arranging the tiles, either keeping one input tile stationary in the L_1 cache and filling the L_2 cache with weight tiles, or keeping one weight tile stationary in the L_1 cache and filling the L_2 cache with input tiles. The L_3 cache is then used to keep more of the stationary tile type. The first approach is called Input Stationary Scheduling and the second is called Weight Stationary Scheduling. This section will explore each of these further.

5.2.1 Input Stationary

In the input stationary scheduling, an input tile is kept in the L_1 cache, and is reused for many weight tiles stored in the L_2 cache before moving to the next input set. More input sets are kept in the L_3 cache, which are then used with all of the previous weight tiles. This process is shown in Figure 14.

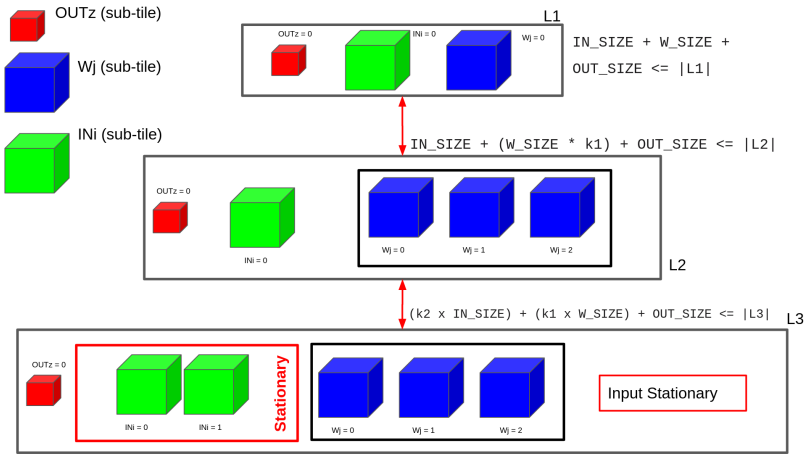


Figure 14: CSA Input Stationary Scheduling diagram.

To decide how many tiles should be stored in each cache level, k_1 for L_2 cache and k_2 for L_3, the following inequations should be maximized:

$$I_{SIZE} + (k_1 \times F_{SIZE}) + O_{SIZE} \leq |L_2|$$

$$(k_2 \times I_{SIZE}) + (k_1 \times F_{SIZE}) + O_{SIZE} \leq |L_3|$$

Iterative heuristics are also used to obtain solutions these optimization problems.

5.2.2 Weight Stationary

In the weight stationary scheduling, a weight tile is kept in the L_1 cache, and is reused for many input tiles stored in the L_2 cache before moving to the next weight set. More weight

sets are kept in the L_3 cache, which are then used with all of the previous input tiles. This process is shown in Figure 15.

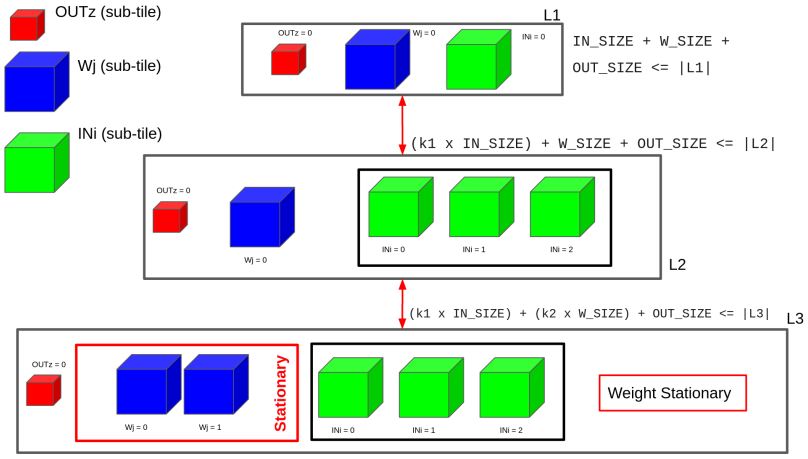


Figure 15: CSA Weight Stationary Scheduling diagram.

To decide how many tiles should be stored in each cache level, k_1 for L_2 cache and k_2 for L_3, the following inequations should be maximized:

$$(k_1 \times I_{SIZE}) + F_{SIZE} + O_{SIZE} \leq |L_2|$$
$$(k_1 \times I_{SIZE}) + (k_2 \times F_{SIZE}) + O_{SIZE} \leq |L_3|$$

Iterative heuristics are also used to obtain solutions for these optimization problems.

5.2.3 Deciding the Best Approach

The main objective of scheduling is to minimize the amount of loads from memory (DRAM) required to compute the convolution, reusing as much as possible from the L_2 and L_3 caches while keeping the stationary tile in the L_1 cache. That being the case, the indicator of how appropriate an approach is for the current instance and architecture is its load cost, the weighted sum of loads from L_1, L_2 and L_3 caches, as well as from memory. The sum weights are also dependent on architecture, the latency in cycles of reading from each cache level and memory.

The first terms of the cost correspond to the amount of load operations that need to go to the main memory to get the corresponding data. Every single tile needs to be loaded from DRAM at some point. Considering I_{TOTAL} and W_{TOTAL} as the total amount of tiles for the input and weights, respectively, MEM as the latency of reading from memory, and n_{CL} the number of cache lines, this cost is defined by the following equation:

$$cost = MEM \times ((I_{TOTAL} \times I_{SIZE}) + (W_{TOTAL} \times W_{SIZE}))/n_{CL}$$

The other cost terms are dependent on the scheduling and amount of channel tiles (t_{CH}). Therefore, consider ST an alias for the stationary structure, and R an alias for the other

structure, to be reused. So in input stationary, ST is the input and R is the group of weights, and vice-versa for weight stationary.

Additional accesses to DRAM may be required to load R tiles, if they don't all fit into the L_2 cache. In this case, the R tiles will need to be read again if the L_3 cache cannot fit all ST tiles. Considering $ST_{t_{CH}}$ and $R_{t_{CH}}$ the amount of tiles per channel set for the respective structures, these accesses can be calculated by the following equations:

$$R_{fit} = \min((\frac{R_{t_{CH}}}{k_1} - 1), 1)$$

$$ST_{fit} = \frac{ST_{t_{CH}}}{k_2} - 1$$

$$cost+ = MEM \times t_{CH} \times (R_{fit} \times ST_{fit} \times R_{t_{CH}} \times R_{SIZE})/n_{CL}$$

If R does not fit for each channel set in the L_2 cache, the amount of loads from memory is controlled by ST_{fit}.

It may be required to load the ST tiles from L_3 more than once. If the R tiles do not fit L_2, when loading the other k_1 R tiles the same ST tiles have to be loaded to L_1. If L_3 is the latency of reading from the L_3 cache:

$$R_{fit} = \frac{R_{t_{CH}}}{k_1} - 1$$

$$cost+ = L_3 \times t_{CH} \times (R_{fit} \times ST_{t_{CH}} \times ST_{SIZE})/n_{CL}$$

Finally, the cost of accessing the R tiles from the L_2 cache should be added as well, for every ST tile. Since the tiles initially come from memory, one access should be disconsidered.

$$cost+ = L_2 \times t_{CH} \times (ST_{t_{CH}} - 1 \times R_{t_{CH}} \times R_{SIZE})/n_{CL}$$

This cost should be calculated for both scheduling approaches, and the one with the lowest value should be chosen for the instance and architecture.

6 Convolution Slicing Optimization: The Macro-Kernel

With the development of the micro-kernel, the computation of 8×16 output elements is optimized by using the MMA instructions. However, as discussed in Section 5, this does not guarantee good performance for the entire convolution. Arguably more important than the highly optimized micro-kernel is how to get the data in place for it.

When powerful SIMD (Single Instruction, Multiple Data) instructions are used, the bottleneck then becomes the latency to read the data from memory. For that reason, the macro-kernel is very important for achieving the best performance for the convolution as a whole. The macro-kernel is responsible for using the cache hierarchy and minimizing memory impact when calling the micro-kernel, as well as using the micro-kernel to compute a convolution (as described in Section 4.1).

This work is called Convolution Slicing Optimization (CSO), and consists of input pre-processing, convolution tiling using CSA's (Section 5) calculated parameters and data packing for the micro-kernel.

6.1 Tiling

For a complete convolution, every window needs to compute output elements with every filter. For large convolutions, though, it would be helpful to solve multiple sub-problems which are then merged at the end of the process. A micro-kernel such as the ones described in Section 4 already requires tiling with a specific layout to be used. Aside from micro-kernel tiling, the problem can be further divided for cache usage.

Section 5 details how to tile a convolution so that the cache hierarchy is used to help avoid stalls in the micro-kernel and unnecessary loads from main memory. Based on the parameters calculated by CSA, the macro-kernel can tile the convolution into multiple channel sets, and then schedule the internal micro-kernel tiles. This process consists of five loops, and is shown for input stationary scheduling in Figure 16.

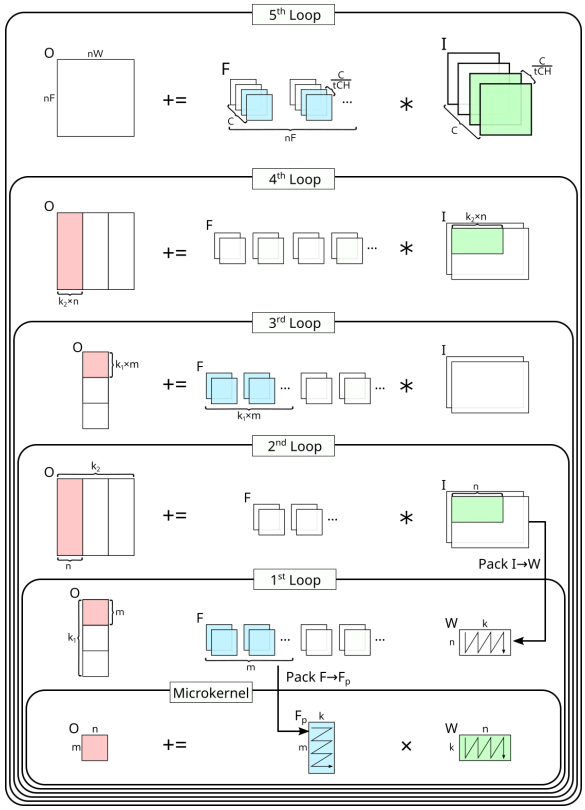


Figure 16: CSO tiling process for input stationary scheduling.

The outermost loop selects the current channel tile, with $\frac{C}{t_{CH}}$ channels. Then, k_2 input (or weight, in weight stationary scheduling) tiles are selected, enough to fill the L_3 cache, and k_1 weight (or input, in WS) tiles are chosen, enough to fill the L_2 cache. The two innermost loops take one tile of each structure and pack them in the correct layout for the micro-kernel.

6.2 Packing

When using a micro-kernel, the input structures usually cannot be used directly. In the case of a GEMM micro-kernel, the input needs to be divided into windows in a specific layout, and the filters need to be reordered to column-major layout. This process is called packing, and can happen at any level. Im2Col (Section 2.3) can be seen as a form of input prepacking, but BLAS has its own packing routines before its micro-kernel, so the data is packed twice.

CSO takes a BLAS-like approach, packing the window and filter tiles as they are needed in the tiling process, named Packing on Demand (PoD). This method differs from BLAS as it does not pack on the size of the L_2 cache, but directly on the L_1 cache size, right before the micro-kernel is called. Since tiles are reused, instead of packing them again a large buffer is used to store k_1 tiles of the non-stationary structure. This way, the tiles are first loaded from memory to the L_1 cache, then kept at the L_2 cache until they are used again.

As discussed in Section 4, for `fp32` one MMA operation takes one element out of each window and filter, and the different elements are accumulated on subsequent instructions. That being the case, the best layout to organize the data for the micro-kernel for maximum data locality uses this information. The elements with same index from every window and filter in the tile are stored before the next. This layout is exemplified in Figure 17. The process is similar for other data types.

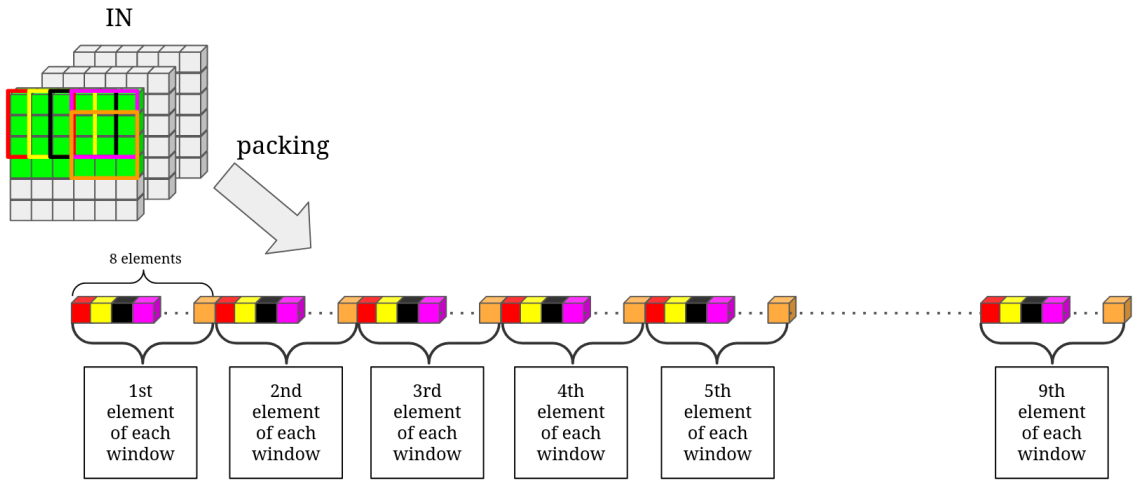


Figure 17: Packing layout for `fp32` and a 3×3 window, considering 8 windows in a tile, and a single channel.

Just as the micro-kernel, this step is highly dependent on implementation. The routine needs to be carefully implemented, so to use as much data locality as possible. In this case, there are two ways of doing this: reading in order (and storing out of order) or storing in order (and reading out of order). Since the convolution requires the input image to be divided into windows, storing in order was chosen for this structure. There is some overlap

between windows, but the elements will already be in the cache when they are loaded again. For filters, reading in order was chosen, since packing is a simple rearrangement of the elements.

6.2.1 VSR Packing

Packing for a convolution when using a MMA micro-kernel requires data replication, because one element can be part of many windows. In fact, most elements in an input image participate in many windows.

The most straightforward way to optimize this need for multiple loads of the same element is to take advantage of data locality and load from the L_1 cache when possible. However, when the convolution has unitary stride a property can be used: the first element from the second window is the second element from the first window, aside from border cases (Figure 18). This means that 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.

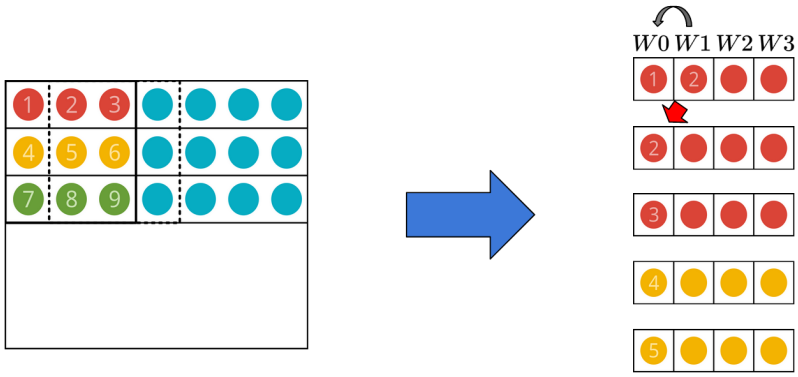


Figure 18: Useful observation for stride 1 convolutions: the second element from a window is the first element of the next one, and so on. The data is of type `fp32`.

This indicates an opportunity to use what the architecture has to offer. In POWER10, there are vector registers (VSRs) which can be loaded and stored with their own instructions, and have a separate computation unit which can be used in parallel with MMA or scalar operations.

Furthermore, POWER10's vector and VSX instruction sets have left shift operations. One of them is called Vector Shift Left Double by Octet Immediate (`vsldoi`), and it concatenates two VSRs, shifting the subsequent temporary large vector to the left by c bytes and storing the first 16 bytes of the result in another register, as seen in Figure 19. The VSX instruction extension introduced the VSX Vector Shift Left Double by Word Immediate (`xxsldwi`) instruction, which is the same but shifting by words instead of bytes.

Using `vsldoi` or `xxsldwi`, one VSR can serve as a serial in producer to another register. This can be used to accelerate packing, so that the elements are only loaded once. At the beginning of the packing process, every element to be packed from the same row is loaded into N VSRs. Then, the registers with the first elements from the windows are stored, and

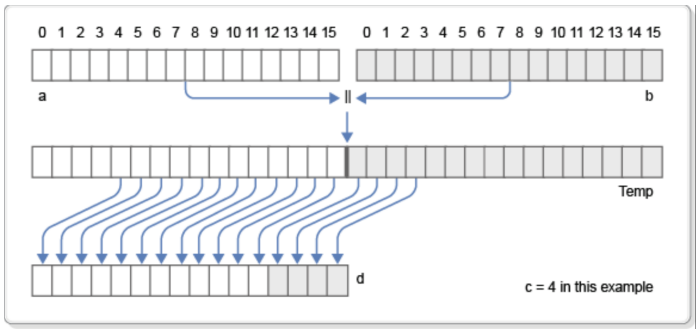


Figure 19: Diagram for the Vector Shift Left Double by Octet Immediate vector instruction for POWER architecture [28].

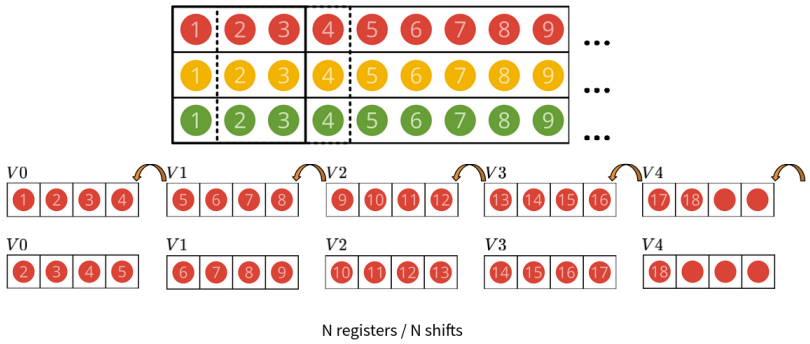


Figure 20: VSR Packing for packing 16 windows of size 3×3. Only one shift sequence is shown, and this process is done a total of $kW - 1$ times for each row. The data type is `fp32`.

by using a sequence of N pairwise shifts the next elements are put in place, as exemplified in Figure 20. This process is done $kW - 1$ times for each of the kH window rows.

For this method to work, the micro-kernel should consume an amount of windows which is a multiple of the size of the VSRs, so that the number of elements to store per iteration fits exactly in the vector registers. The serial in registers need to be shifted as well, to prepare the next element to be inserted into the last VSR to be stored.

At the end of each row there are border cases for VSR packing, since there are elements that are part of fewer windows, and elements should be skipped. For that reason, this process is only used when there are enough windows in the same row. This routine is specific to POWER architecture, but most modern architectures have similar registers and instructions. Unitary stride is also required, so that the initial observation holds true.

6.2.2 Filter Packing

In most instances, input packing is responsible for most of the packing time. However, some convolutions from the end of CNN models have larger filters and a smaller input image, and in these cases the filter packing time can be dominant not only when compared to input

packing, but with the entire process.

One alternative is to use as much data locality as possible and reorder the filters before tiling. However, in most machine learning models this can be implemented as a separate pass, since the filter is known at compile time.

7 Results

This work was implemented as described in Sections 5 and 6, with a GEMM micro-kernel from the OpenBLAS [17] library. Since in machine learning frameworks the dimensions of the convolutions are known at compile time, CSA was implemented as a separate application which outputs a header file with the calculated parameters. CSO is then compiled with this configuration, which decides scheduling and channel tiling.

Aside from the CSA parameters, other configuration options are present at compile time, mainly related to packing. A version with input prepacking was implemented to help measure the time split between packing and micro-kernel. The filters can be packed on demand, prepacked, or reordered before starting the process, emulating a separate pass to pack these kernels.

For compatibility between different padding parameters in the low-level routines, padding is explicitly constructed in a larger input image when necessary. Since the micro-kernel tile size is the same for every call, extra padding may also be required in the packed sets and output. This results in the need for an output unpacking step before returning the final solution.

The baseline for this work is an implementation of the Im2Col + BLAS GEMM method, using the Caffe [21] framework's Im2Col routine and OpenBLAS GEMM.

7.1 Testing Conditions

Both the baseline and CSO were tested for correctness and performance on a POWER10 machine with single-threaded execution. The cache sizes are 32 KB, 1 MB and 4 MB for L_1, L_2 and L_3, respectively.

On the software side, the experiments employed the Ubuntu Linux distribution 20.04.2, a modified version of the LLVM compiler Clang 14, and OpenBLAS 0.3.17 for both GEMM and the micro-kernel. The BLAS library was compiled using the regular Clang 13 compiler, with MMA support.

These tests were conducted by computing each convolution 1000 times, executing the binary every time so that cache and memory allocations are reset, and the current run does not influence the next one. The wall time is measured internally, and both the total time and elapsed time for each step are calculated. These results are then averaged.

7.2 Small Convolutions

The first experiments to be conducted were for small convolutions that fit entirely in the L_1 cache. In this case, CSA tiling is not needed, and the results can be seen in Table 1. The speedup achieved for these convolutions indicates that the overhead of calling BLAS

GEMM is very large compared to CSO, which is confirmed by looking at the time split for the baseline.

Instance		Small-1	Small-2
Input Shape	CHW	$3 \times 28 \times 28$	$16 \times 8 \times 8$
Filter Shape	MCHW	$8 \times 3 \times 3 \times 3$	$32 \times 16 \times 3 \times 3$
Baseline	Total (μs)	121	112
	Im2Col (μs)	11	6
	GEMM (μs)	100	97
CSA + CSO	Total (μs)	24	12
Speedup		**5.04**	**9.33**

Table 1: Results for CSA + CSO against the baseline for small convolutions that fit in the L_1 cache.

7.3 CNN Convolutions

After testing our implementation against the baseline with small convolutions, larger instances from real-world CNN models were selected with different shapes and parameters such as stride and padding. Measuring CSO with these instances provides a more realistic outlook on the project's performance. A breakdown of the convolutions can be found in Table 2.

Instance	Input Shape CHW	Filter Shape MCHW	Stride	Padding
Inceptionv3 1	$80 \times 73 \times 73$	$192 \times 80 \times 3 \times 3$		0/0
Inceptionv3 2	$32 \times 149 \times 149$	$32 \times 32 \times 3 \times 3$	1	0/0
Inceptionv3 3	$448 \times 8 \times 8$	$384 \times 448 \times 3 \times 3$		1/1
Inceptionv2 1	$3 \times 299 \times 299$	$32 \times 3 \times 3 \times 3$	2	
Inceptionv2 2	$64 \times 73 \times 73$	$80 \times 64 \times 1 \times 1$	1	0/0
Inceptionv2 3	$2048 \times 8 \times 8$	$80 \times 2048 \times 1 \times 1$	1	
ResNet 1	$3 \times 224 \times 224$	$64 \times 3 \times 7 \times 7$	2	3/3
ResNet 2	$64 \times 56 \times 56$	$64 \times 64 \times 3 \times 3$	1	1/1
ResNet 3	$64 \times 56 \times 56$	$64 \times 64 \times 1 \times 1$	1	0/0
MobileNet 1	$3 \times 224 \times 224$	$32 \times 3 \times 3 \times 3$	2	1/1
MobileNet 2	$256 \times 14 \times 14$	$256 \times 256 \times 3 \times 3$	1	
VGG16 1	$64 \times 224 \times 224$	$64 \times 64 \times 3 \times 3$	1	1/1
VGG16 2	$256 \times 56 \times 56$	$256 \times 256 \times 3 \times 3$		

Table 2: CNN convolution instances used to test CSO against the baseline.

The largest instances are from the VGG16 [1] model, followed by some from Inceptionv3 [10]. The other instances are from the Inceptionv2 [29], ResNet [2] and MobileNet [30]

models. In some cases, the filter tensor is larger than the input, as in `Inceptionv3 3` and `Inceptionv2 3`.

7.3.1 CSA

CSA decides, for a specific instance and architecture, which is the best scheduling, and calculates the t_{CH}, k_1 and k_2 parameters, as described in Section 5. This process was executed on all instances for two different architectures: the POWER10 machine used for testing (Section 7.1), and a generic x86 architecture, considering the same micro-kernel, cache line size and latency for each cache level and main memory. The x86 architecture has cache sizes of 32 KB, 256 KB and 1 MB for L_1, L_2 and L_3, respectively. The results can be seen in Table 3. Since the L_1 size is the same in both architectures, t_{CH} is also the same.

Instance	Output Shape MHW	t_{CH}	POWER10			x86		
			Scheduling	k_1	k_2	Scheduling	k_1	k_2
Inceptionv3 1	$192 \times 71 \times 71$	4	IS	24	316	WS	39	24
Inceptionv3 2	$32 \times 147 \times 147$	1		4	168	WS	21	4
Inceptionv3 3	$384 \times 8 \times 8$	16		48	4	IS	24	4
Inceptionv2 1	$32 \times 149 \times 149$	1	IS	4	1388	IS	4	347
Inceptionv2 2	$80 \times 73 \times 73$	1		10	334		10	167
Inceptionv2 3	$192 \times 8 \times 8$	8		24	4		24	4
ResNet 1	$64 \times 112 \times 112$	1	IS	8	392	IS	8	98
ResNet 2	$64 \times 56 \times 56$	2		8	196	WS	24	8
ResNet 3	$64 \times 56 \times 56$	1		8	196	IS	8	196
MobileNet 1	$32 \times 112 \times 112$	1	IS	4	784	IS	4	392
MobileNet 2	$256 \times 14 \times 14$	8		32	13		16	13
VGG16 1	$64 \times 224 \times 224$	2	IS	8	196	WS	24	8
VGG16 2	$256 \times 56 \times 56$	8		32	196		24	32

Table 3: CSA results for the POWER10 testing machine and a x86 generic architecture.

Since the POWER10 machine's L_2 and L_3 caches are four times larger than the x86's, input stationary scheduling is selected in every instance, with a large k_2 parameter to store many input tiles in the L_3 cache. When the caches are smaller, such as in the x86 architecture, weight stationary scheduling is more often selected, especially on the larger instances, to avoid repeated loads from DRAM.

7.3.2 CSO

The parameters from Table 3 were used to tile each convolution inside CSO, and the performance was measured for various configurations of the method. The different versions refer to changes in packing for both the input image and filters. Both structures can be prepacked or packed on demand.

In the prepacking configuration, the respective structure is completely packed into a single large buffer before any of the computation is done. This allows for measuring how

much time is spent on packing alone, but does not utilize the memory hierarchy to its full potential. Tile scheduling is still used, but the data may not be in the L_1 cache when the micro-kernel is called, since it was prepacked. A mixed configuration was also tested, prepacking only the filters, and packing the input image on demand.

The performance results for CSO and the baseline, as well as the speedup achieved by the PoD version for each instance, are shown in Table 4.

Method	Baseline (ms)	CSA + CSO (ms)			Speedup
Input	Im2Col	Prepacked	PoD	PoD	
Filters	-	Prepacked	Prepacked	PoD	
Inceptionv3 1	10.727	10.735	8.959	8.956	**1.20**
Inceptionv3 2	8.110	5.941	3.743	3.798	**2.14**
Inceptionv3 3	1.990	2.541	2.413	2.034	0.98
Inceptionv2 1	1.270	1.562	1.337	1.338	0.95
Inceptionv2 2	0.929	0.923	0.814	0.821	**1.13**
Inceptionv2 3	0.695	0.648	0.564	0.488	**1.42**
ResNet 1	3.441	3.499	2.719	2.769	**1.24**
ResNet 2	3.027	2.630	1.980	1.966	**1.54**
ResNet 3	0.529	0.450	0.377	0.379	**1.40**
MobileNet 1	0.767	0.922	0.797	0.794	0.97
MobileNet 2	2.043	2.390	2.188	2.050	1.00
VGG16 1	45.805	40.926	28.924	28.590	**1.60**
VGG16 2	24.877	27.221	21.683	21.502	**1.16**

Table 4: CSA + CSO and Baseline results for every instance detailed in Table 2. Instances in which CSO achieves better performance than the baseline have their speedup in bold.

The results show that, for most instances, CSO is up to 2.14 times faster than the Im2Col + BLAS GEMM baseline. Even in the cases where it is not faster, the performance is similar, with at most 5% slowdown.

As expected, prepacking the input image resulted in a considerable performance drop, being up to 36% slower than its counterparts. This observation shows that packing on demand achieves its purpose of using improved memory accesses to speed up the convolution process.

Analyzing the convolutions for which CSO was slower than the baseline, two patterns can be observed in Table 2. Two out of the three convolutions with larger stride values achieved slowdowns when compared to the baseline. The other two slower instances have very large filter tensors when compared to their respective input images, which indicates that filter packing is responsible for a big part of the runtime in these cases.

Section 6.2.2 discussed different options for kernel packing, one of them being implementing it as a separate pass inside of a machine learning framework. In these cases, the filter would only need to be packed once for many executions, with the downside of impacting cache usage in CSO. For this reason, another tested CSO version uses filters which are already in the correct order before the execution begins. This approach is called “Ready”,

and its performance results are shown in Table 5.

Method Input Filters	Baseline (ms) Im2Col -	CSO (ms) PoD PoD	Speedup	CSO (ms) PoD Ready	Speedup
Inceptionv3 1	10.727	8.956	**1.20**	8.822	**1.22**
Inceptionv3 2	8.110	3.798	**2.14**	3.716	**2.18**
Inceptionv3 3	1.990	2.034	0.98	1.305	**1.52**
Inceptionv2 1	1.270	1.338	0.95	1.327	0.96
Inceptionv2 2	0.929	0.821	**1.13**	0.815	**1.14**
Inceptionv2 3	0.695	0.488	**1.42**	0.303	**2.29**
ResNet 1	3.441	2.769	**1.24**	2.712	**1.27**
ResNet 2	3.027	1.966	**1.54**	1.899	**1.59**
ResNet 3	0.529	0.379	**1.40**	0.376	**1.41**
MobileNet 1	0.767	0.794	0.97	0.790	0.97
MobileNet 2	2.043	2.050	1.00	1.745	**1.17**
VGG16 1	45.805	28.590	**1.60**	27.983	**1.64**
VGG16 2	24.877	21.502	**1.16**	21.208	**1.17**

Table 5: Comparison between the baseline results and both of the best versions of CSO, for every instance detailed in Table 2. "Ready" filters mean that the filters are already packed in the correct order at the start of execution. Instances in which CSO achieves better performance than the baseline have their speedup in bold. Instances in which kernel packing has a large performance hit have their speedups in red.

Even though the cache usage can be worse than its PoD counterpart, using "ready" kernels improves the performance of CSO for every tested instance, since the kernel packing time is eliminated from the runtime. For most instances, this improvement is small, of up to 3%, but some cases greatly benefit from this approach, such as the ones with large filter tensors.

When the filter tensor is larger than the input image, using "ready" filters can improve the speedup when compared to the baseline in up to 87%, and when compared to the PoD version in up to 48%. With this approach, only two of the stride 2 convolutions remain slower than the baseline.

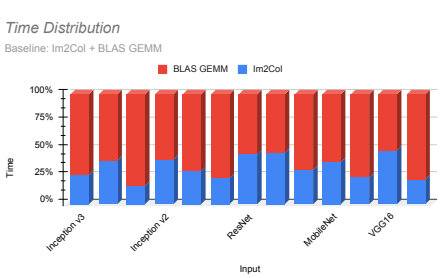
The baseline does not benefit from "ready" kernels, since the operation expects the tensor in the original format, and kernel packing is a step taken inside the BLAS library.

7.3.3 Time Split

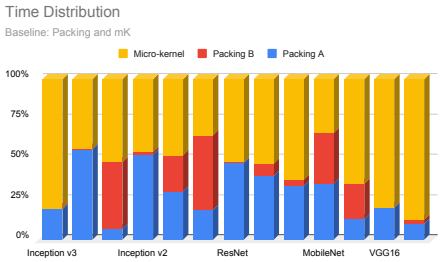
Since noticing that the kernel packing step takes a large part of the runtime for some instances helped to develop solutions for the performance issue, it is relevant to analyze the time split of each of these instances for every step.

The baseline is divided in two main steps: Im2Col packing, and BLAS GEMM. The GEMM routine can be subdivided into three other steps: input packing, kernel packing,

and the micro-kernel. The time split for the baseline and inside GEMM can be seen in Figure 21.



(a) Baseline execution time split percentages between Im2Col and BLAS GEMM.

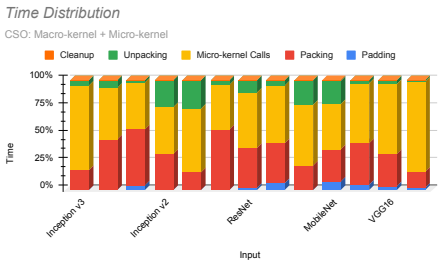


(b) GEMM execution time split percentages between input packing (Packing A), filter packing (Packing B) and micro-kernel.

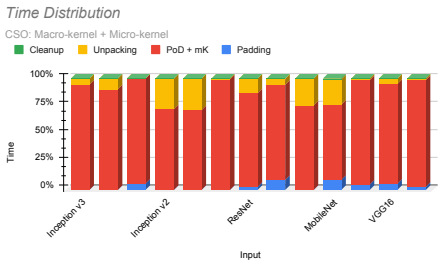
Figure 21: Time split graphs for the baseline and inside GEMM.

Figure 21a contains the time split between Im2Col and BLAS GEMM. In the baseline, the Im2Col process is responsible for 16% to 48% of the runtime, even though the data is packed again inside BLAS. Indeed, as seen in Figure 21b, packing can take up to 66% of GEMM's runtime. This means that, in total, packing data can be responsible for up to 78% of the runtime, as is the case for the `MobileNet 1` instance. However, other instances can be packed very quickly, and most of the time is spent on computation inside the micro-kernel, as is the case for the `VGG16 2` instance, with 67% of the runtime dedicated to the micro-kernel.

Figure 21b also confirms that in some instances the kernel packing step takes a large part of the runtime, even in the baseline. In `Inceptionv2 3` and `Inceptionv3 3`, almost half of the GEMM time is dedicated to packing the filters.



(a) Prepacking



(b) PoD with Ready Kernels

Figure 22: Time split graphs for CSO.

Figure 22 shows the time split for the prepacked and PoD with "ready" filters versions of CSO. The version with input and kernel prepacking shown in Figure 22a allows for an approximate analysis of packing time in CSO. Packing the data only once, without the

need for two data manipulation steps, improves the time share that the packing step takes. However, it can still dominate the runtime, taking up to 55% of the total execution time.

Still, for most instances the dominating step of the process is the actual computation, inside the micro-kernel, instead of data manipulation and reordering, even when considering filter packing. If the micro-kernel is optimal, the largest its time share becomes, the closer to optimal performance the process gets. The micro-kernel is responsible for up to 81% of the runtime.

Aside from packing and the micro-kernel, another step with a relevant time share is output unpacking. In the best version of CSO, packed on demand with "ready" filters, this step can take up to 28% of the runtime, as seen in Figure 22b. The two instances which are slower than the baseline for this best configuration also have a large time share dedicated to output unpacking, 28% for `Inceptionv2 1` and 23% for `MobileNet 1`.

8 Conclusion

This work aimed to improve convolution performance by developing and presenting a novel algorithm based on BLAS GEMM by Goto and de Geijn [19] to tile, pack and compute the convolution without relying on third-party libraries, while exploiting the current architecture, namely IBM POWER10. The main focus was on improving CNN convolutions, so some properties of machine learning models were used.

Most widely used methods reduce the problem to GEMM, and use a BLAS implementation to solve it with good memory hierarchy usage. This is a simple process, but the Im2Col transformation results in high memory utilization, the data is packed again inside GEMM, and the BLAS GEMM tiling is not optimal for the resulting shapes [22].

The implementation for this project was split between a micro-kernel, responsible for exploiting the MMA engine to increase the throughput of the operation, and a macro-kernel, responsible for optimizing data access by tiling the input image and filters to reduce the amount of DRAM loads, and by exploring data locality and the architecture when packing the data for the micro-kernel.

The novel strategy taken in this work was to use efficient convolution-specific slicing instead of the regular BLAS GEMM tiling. CSO was adapted from a work on NMP architecture [27], and slices the convolution structures based on micro-kernel and cache size, before packing. Furthermore, it schedules these tiles so to maximize reuse from cache.

The packing routines were implemented so to explore data locality and exploit the architecture to eliminate multiple loads of the same element, under certain conditions, with vector registers and shift operations. The data is packed only when needed, before calling the micro-kernel, in a process called packing on demand.

Early experiments with very small convolutions showed that the overhead of calling the BLAS GEMM routine greatly surpasses CSO's runtime, the latter being more than five times faster than the former in both instances. For larger convolutions from CNN models, which include tiling in both approaches, the CSA + CSO method achieved speedups of up to 214% when comparing to the baseline, and was faster in most tested instances.

Looking at the breakdown of how much time is spent in each step of each instance, it was

observed that filter packing can be responsible for a large part of the runtime. In machine learning frameworks, a separate pass can be implemented to pack the kernels ahead of time, since they are known in the inference pass. Removing filter packing from CSO, an increase in performance was seen on every instance, in some cases achieving speedups of up to 87% when comparing to the baseline, and 48% when comparing to the previous version.

8.1 Future Work

This project achieved promising results comparing single convolutions between the common Im2Col + GEMM baseline and CSA + CSO. With kernels packed ahead of time, only two instances had worse performance in CSO when comparing to the baseline. Both of these instances have over 20% of their respective runtimes dedicated to output unpacking, a step at the end of the process in which the output is copied from an intermediate possibly larger buffer to the final one.

The unpacking step is necessary since the micro-kernel receives a constant amount of windows and filters in each call, and the total amount of windows or filters may not be multiples of the tile sizes. In this case, extra padding is added to both the packed sets and the output, which needs to be removed before returning the result. If the micro-kernel can be used with a variable tile size, unpacking is not needed, so this change is a promising next step.

Another future path is to integrate this work with an existing machine learning framework. This would allow for streamlining of the testing process, and measuring performance for entire models instead of single convolutions, in a more realistic context.

This implementation of CSO does not use parallelism in any way, being a single-threaded application, but the process is parallelizable. Evidences show that, for the matrix shapes generated by Im2Col, BLAS GEMM does not scale well with multiple threads [22], so exploring this side is a relevant research path.

8.1.1 Compiler Intrinsic Function

In CNNs, all kernel and input image dimensions are known at compile time, as well as other parameters such as stride and padding. This means that CSA can be executed at compile time as well, generating the parameters for tiling the convolution, and the testing methodology reflects that.

Still, the dimensions of the convolution are still read at runtime in CSO. Since every parameter is known beforehand, optimized code can be generated for each instance without the need for a generic implementation. By moving CSA and CSO inside of the compiler, every loop can be unrolled and optimized at IR level, creating an implementation with as few branches as possible.

This can be done with LLVM in multiple ways, the first of which being with the MLIR (Multi-Level Intermediate Representation) [31] infrastructure. The ONNX [32] project includes an open-source compiler project using MLIR, called ONNX-MLIR [33], which would be a good fit for this approach, given that CSO can be compared with its current convolution implementation. Another path would be to integrate directly in LLVM with an

intrinsic function.

LLVM intrinsic functions represent an extension mechanism for LLVM IR, enabling specialization to specific architectures and preventing duplication of code within the compiler infrastructure. An intrinsic can be viewed as a function call at the IR level that is lowered into either target-agnostic or target-specific inlined generated code.

There is no intrinsic function for convolution in LLVM as of this report's writing, but proposals for tensor extensions with intrinsics have been submitted for comments which include convolution [14].

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2015.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[3] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/3fe94a002317b5f9259f82690aeea4cd-Paper.pdf

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587.

[5] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[6] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[7] Y. Kim, "Convolutional neural networks for sentence classification," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 08 2014.

[8] C. Camacho, "Convolutional neural networks," 2018, [Accessed November 30, 2021]. [Online]. Available: https://cezannec.github.io/Convolutional_Neural_Networks/

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[10] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[11] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006. [Online]. Available: https://hal.inria.fr/inria-00112631

[12] J. E. Moreira, K. Barton, S. Battle, P. Bergner, R. Bertran, P. Bhat, P. Caldeira, D. Edelsohn, G. Fossum, B. Frey, N. Ivanovic, C. Kerchner, V. Lim, S. Kapoor, T. M. Filho, S. M. Mueller, B. Olsson, S. Sadasivam, B. Saleil, B. Schmidt, R. Srinivasaraghavan, S. Srivatsan, B. W. Thompto, A. Wagner, and N. Wu, "A matrix math facility for power ISA(TM) processors," *CoRR*, vol. abs/2104.03142, 2021. [Online]. Available: https://arxiv.org/abs/2104.03142

[13] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "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)*, 2020, pp. 99–106.

[14] A. Kothari, A. R. Noor, D. Khaldi, V. Adve, Y. Luo, S. Sengupta, M. Girkar, and C. Mendis, "Proposal for TLX: Tensor LLVM eXtensions," UIUC, Intel and Amazon AWS, Tech. Rep., 2021. [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2021-November/153725.html

[15] M. Basavarajaiah, "6 basic things to know about convolution," 2019, [Accessed November 30, 2021]. [Online]. Available: https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411

[16] A. Deshpande, "A beginner's guide to understanding convolutional neural networks part 2," 2016, [Accessed December 14, 2021]. [Online]. Available: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

[17] Z. Xianyi, M. Kroeker, W. Saar, W. Qian, Z. Chothia, C. Shaohu, and L. Wen, "Openblas: An optimized blas library." [Online]. Available: https://www.openblas.net/

[18] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 14:1–14:33, June 2015. [Online]. Available: https://doi.acm.org/10.1145/2764454

[19] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, may 2008. [Online]. Available: https://doi.org/10.1145/1356052.1356053

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[21] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[22] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 5776–5785. [Online]. Available: http://proceedings.mlr.press/v80/zhang18d.html

[23] *Power ISA Version 3.1*, IBM Corporation, May 2020. [Online]. Available: https://ibm.ent.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0

[24] P. Bhat, J. Moreira, and S. K. Sadasivam, *Matrix-Multiply Assist (MMA) Best Practices Guide*. IBM Corporation, 2021.

[25] *Power Vector Intrinsic Programming Reference*, OpenPOWER Foundation, October 2020.

[26] P. S. Juan, A. Castello, M. F. Dolz, P. Alonso-Jorda, and E. S. Quintana-Orti, "High performance and portable convolution operators for multicore processors," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2020, pp. 91–98. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SBAC-PAD49847.2020.00023

[27] R. Sousa, J. Byungmin, J. Kwak, M. Frank, and G. Araujo, "Efficient tensor slicing for multicore npus using memory burst modeling," in *2021 33th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2021.

[28] *z/OS 2.5 XL C/C++ Programming Guide*, IBM Corporation, 2021.

[29] S. Ioffe and C. Szegedy, "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*, ser. ICML'15. JMLR.org, 2015, p. 448–456.

[30] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *ArXiv*, vol. abs/1704.04861, 2017.

[31] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[32] J. Bai, F. Lu, K. Zhang *et al.*, "ONNX: Open Neural Network Exchange," https://github.com/onnx/onnx, 2019.

[33] T. D. Le, G.-T. Bercea, T. Chen, A. E. Eichenberger, H. Imai, T. Jin, K. Kawachiya, Y. Negishi, and K. O'Brien, "Compiling onnx neural network models using mlir," *ArXiv*, vol. abs/2008.08272, 2020.