


École Normale Supérieure de Lyon

Master 2 Computer Science

Internship report

A study of Convolutions for Efficient Inference of Deep Neural Networks on Embedded Processors

Enrique GALVEZ

Under supervision of
Alix MUNIER
Adrien CASSAGNE

Laboratory: LIP6

ALSOC team

5^{th} of february 2024 – 12^{th} of July 2022

Contents

Introduction

Deep learning algorithms, and more particularly deep neural networks (DNNs) are ubiquitous in modern image processing tasks. For instance, convolutional neural networks (CNNs) have brought a real breakthrough in the fields of algorithmic image classification and object detection [1]. In order to setup a DNN for solving a given task, three main steps have to be considered. Firstly, one should choose or construct the structure of its network. Secondly, the network should be trained to solve a specific task. Lastly, the network makes its predictions and eventually solve the problem asked, this is the inference phase.

The ALSOC team of the LIP6 laboratory, has a strong expertise on Systems-on-Chip (SoCs) architecture. These systems can be characterized by a lower energy consumption compared to high-end servers found in computing centers that are generally used for the training phase of a DNN. As a result, SoCs are a great target for performing deep learning inference. In addition, the ALSOC team is involved in a project aiming to embed a meteor detection software in a satellite [2]. A critical point of this kind of software is to detect moving objects, a task that CNNs can achieve with up to 98.5% detection rate [3]. However, to date, the achieved throughput does not match the real time constraint. The ALSOC team therefore has interests in proposing efficient implementations of CNN inference on SoCs.

This internship is motivated by the perspective of implementing fast and energy efficient CNN inference over several SoCs. The main purpose of this internship is to implement and evaluate several State-of-the-art methods to compute efficient convolutions. Indeed, this study of the convolution layer as an independent operator is the first step to an overall optimization of CNN inference. In the first sections of this report, we will see that the critical operation of the forward pass through a CNN is the convolution layer. The three principal algorithms that we describe in this report are `direct`, `im2row` and `winograd`. We will then characterize this layer and propose several high-level algorithmic designs for efficient convolutions. In a last part of this report, we will evaluate our implementations of the convolution layer in several SoCs.

1 Context about popular CNNs in State-of-the-art

Amongst different types of deep neural networks, convolutional neural networks are certainly the most studied. Inspired by the anatomy of animal visual cortex, CNNs have proven their efficiency in solving image classification and object detection tasks. While the modern structure of a CNN was first described by Yann Le Cun *et al.* in 1989 [4], CNNs became very popular in 2012, after that Krizhevsky *et al.* won ILSVRC image classification contest with their CNN design called AlexNet [5]. AlexNet is one of the most famous CNN architecture, which is similar to LeNet-5 proposed by Yann Le Cun, but with a deeper structure. Right after AlexNet's breakthrough, many CNN designs were created with a similar structure. In this context, ZFNet [6] and VGG networks [7] have been proposed in 2013 and 2014, respectively.

More recently, new mechanisms have been added to CNNs in order to face issues encountered at that time. A good example is the creation of residual networks (ResNet) [8] in 2015, which is the first network to implement residual layers in order to simplify the network and make the learning phase faster. Following the principle of ResNet, DenseNet [9] maximizes the number of connections between separate layers, allowing improvement of accuracy with fewer weights but at the cost of a more complex training. Another notable improvement is the creation of inception layers, featured in GoogLeNet [10] networks, which consists in applying several filters to a same input in order to allow the network to process several filtered version of an image in one layer. Recently, MobileNets [11] have been created to optimize the use of CNNs on embedded systems. In these networks, convolution layers are split in two specific convolution layers: a depthwise convolution and a pointwise convolution.

Despite featuring slight changes with the first network designs, the overall structure of modern CNNs is very similar. The general structure of a CNN always starts by applying learnable features on the input tensor. This first step generally includes dimension reduction which can be achieved with pooling, alternating with convolution layers which apply (and eventually learn) the filters of the model. A second step consists in flattening the output of this feature processing, eventually including numerical transformations. Lastly, the flattened tensor passes through a fully-connected layer which achieves the final prediction.

It is well-known that the convolution layers are critical components of modern CNNs, so a lot of work focuses on optimizing their execution [12]. The goal of this internship is to state guidelines for implementing efficient convolution operators corresponding to the layers of modern CNNs. In this context, the study of network-level optimizations are left as future work.

2 The convolution layer

2.1 Notations and preliminaries

A convolution is a parameterized operation: kernels with a given size and dilation are applied with a given stride to a tensor with a given size and padding [13]. The notations used in the following of this report to describe convolutions are summarized in Table 1.

For simplicity, this report only considers 2-dimensional convolutions over 4-dimensional tensors. Tensors are multidimensional structures used to represent batches of images. The 4 dimensions of the tensors can be understood as: $batch \times image\_channels \times image\_height \times image\_width$. For the first convolution layer of the CNN, the two inner-most dimensions represents the image while the channels represents the color in RGB format. The batch dimension allows the convolution to operate on several images at once.

MB	batch size	KH, KW	kernel height, width	PH, PW	padding height, width
IC	input channels	OC	output channels	SH, SW	stride height, width
IH, IW	input height, width	OH, OW	output height, width	DH, DW	dilation height, width

Table 1: Notations for the main convolution parameters.

Using the appropriate notations, the convolution operator takes an input tensor of size $MB \times IC \times IH \times IW$ and a weights tensor of size $OC \times IC \times KH \times KW$ to compute an output tensor of size $MB \times OC \times OH \times OW$.

2.2 The convolution operation

As we saw it previously, CNNs can be decomposed in layers, each of them applying a specific treatment to the input tensor. Among those, the convolution layer computes linear combinations between the elements of the input tensor within a sliding kernel and its corresponding input channels.

This operation can be described mathematically as it follows:

$$dst[mb, oc, oh, ow] = bias[oc] + \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} src(mb, ic, ih, iw) \cdot weights[oc, ic, kh, kw], \tag{1}$$

where ih and iw are locally defined as:

$$\begin{cases} ih := oh \cdot SH + kh \cdot (DH + 1) - PH, \\ iw := ow \cdot SW + kw \cdot (DW + 1) - PW. \end{cases} \tag{2}$$

In order to visualize this operation, Figures 1 and 2 shows two kind of forward convolutions, both represented in two dimensions. The first one represents a strided kernel where $SH = SW = 2$ and the second one represents a dilated kernel where $DH = DW = 2$.

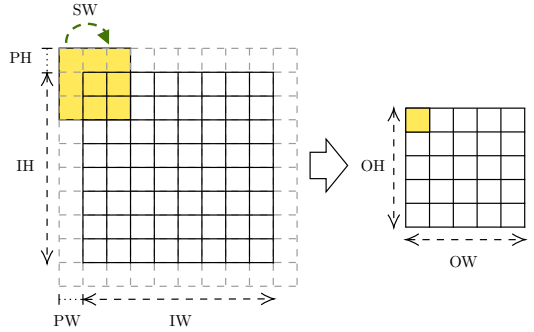

Figure 1: 2-strided 3×3 convolution.

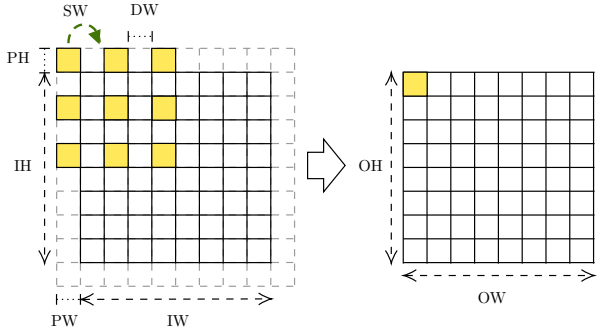

Figure 2: dilated 3×3 convolution: $DW = DH = 1$.

Convolutions in Figures 1 and 2 are represented in two dimensions but in general, they also perform a reduction through the channels, as described by the formula (1). It is worth mentioning that most of the common convolution layers used in CNNs have 3×3 kernels without dilation and are 1-strided. Tensors have a fourth dimension which is the batch: this dimension allows a convolution layer to process several images in only one call to the convolution primitive. In the case of inference, the batch size is considered small: always lower than 8, usually 1 or 2. Other applications, such as training or inferring on a stream of images may require larger batch sizes, but those cases will not be considered in this report.

2.3 Context and motivations for fast convolutions

The essence of any CNN is the succession of layers. A forward pass through a CNN consists in applying several operators to an input tensor. Among those operators, convolution layers generally consume a large fraction of the compute time of an inference.

To illustrate that, we measured the impact of convolution layers compared to other computations during CPU inference using pytorch [14]. Figures 3 and 4 show the impact of convolution layers when running CPU inference of several pytorch models on two modern CPUs. The first CPU is the ARM Cortex-A78AE, embedded in NVIDIA Jetson AGX Orin boards. The second one is a AMD Ryzen 5 5600H CPU, widely used in recent laptops.

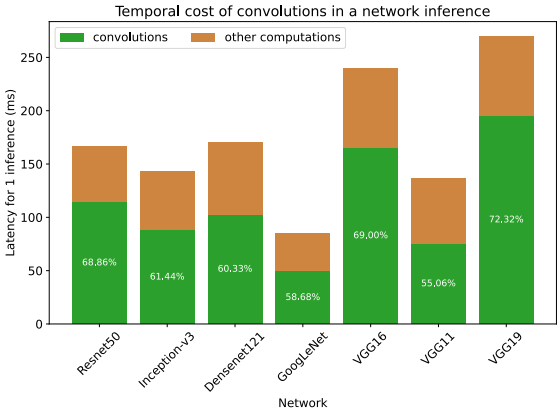

Figure 3: Cost of convolutions on ARM Cortex-A78AE.

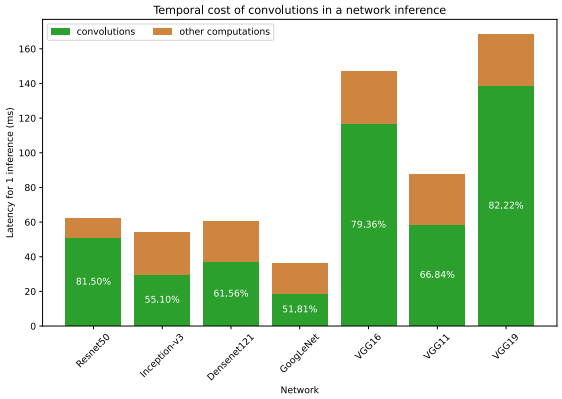

Figure 4: Cost of convolutions on AMD Ryzen 5 5600H.

We can see in the previous graphs that convolutions represent more than half of the entire computational time for a CNN. This important cost of the convolution layer motivates an important amount of work aiming at developing efficient convolutions. However, due to the complexity of the operation, the literature cannot state a proper and unique way to implement it to support any scenario.

During this internship, I implemented forward convolutions in several ways defined by State-of-the-art, with high-level optimizations that I found appropriate. The aim of this work is to provide a fair comparison between three well-known State-of-the-art high performance implementations of forward convolutions: direct method, GEMM-based methods with im2row/im2col transformation and Winograd's method.

3 Optimization framework and programming model

3.1 Accessing memory in RAM or caches

Before explaining the optimization of a complex operator such as convolution layers, we must define some concepts of high performance computing. Indeed, the convolution operator requires a lot of accesses to computer's memory, which has a limited throughput called memory bandwidth. Modern CPUs have access to several levels of memory with increasing size but decreasing bandwidth.

We consider an architecture model in which data is loaded in cache lines each time the program loads data from RAM. When the program tries to access data already loaded in cache, it can use this data directly instead of loading it from RAM, taking profit of the higher bandwidth offered by the cache.

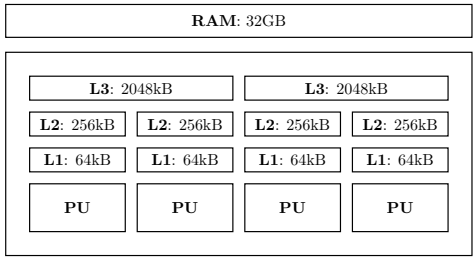

Figure 5: Different levels of memory in modern CPUs

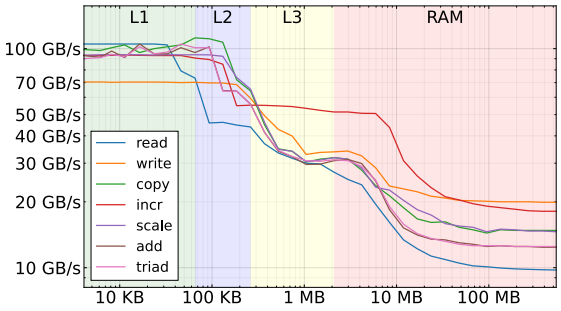

Figure 6: Bandwidth among memory levels

Figure 5 illustrates an example of architecture for a multicore processor in which each process unit (PU) has access to 3 cache levels. The L1 and L2 caches are specific to one PU while the L3 cache is common to 2 PUs. The RAM is usually the higher level of memory with the lowest bandwidth and is common to all the PUs. This kind of architecture is common and very similar to the architecture of the SoCs studied in the performance evaluation section.

Figure 6 illustrates that memory bandwidth is higher on lower levels of memory and decreases on the higher levels. In order to build an efficient and memory aware algorithm, we thus need to maximize the reuse of cached data.

3.2 Working with tensors in memory

According to its definition, the convolution operation works on multidimensional memory objects called tensors. In particular, these objects represents the inputs and outputs of the convolution operator. An interesting point to consider when working with tensors is how they are stored in memory. Indeed, memory is an unidimensional structure so tensors need to be unfolded in order to be stored in memory.

A memory format describes how tensors are unfolded in memory. We refer to a memory format using a multiplication between its dimensions, in the same order as the dimensions are stored in memory.

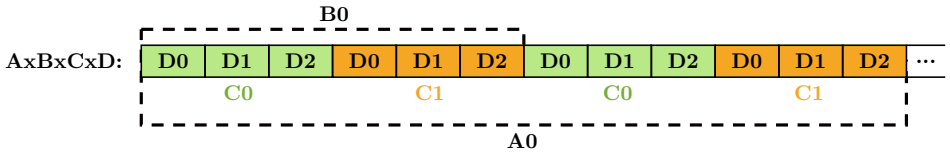

Figure 7: Tensor with format $A \times B \times C \times D$ in memory

Figure 7 illustrates how tensors are stored in memory according to $A \times B \times C \times D$ format. An important thing to note is that with this format, for fixed A, B and C, the elements among the D dimension are contiguous in memory. This remark is important because an appropriate data layout can help the program reusing data stored in the same cache line, benefiting from a higher memory bandwidth. Moreover, if data is loaded from contiguous locations in memory, modern compilers are able to automatically generate code using single instruction multiple data (SIMD) parallelism [15].

4 Direct convolutions

4.1 Naive implementation

The forward convolution operator can be naively implemented using nested for loops. We call `direct` the implementation which computes convolutions as defined in the formula (1).

Algorithm 1: Naive direct convolution

1 **Input tensor** src: $(MB \times IC \times IH \times IW)$
2 **Weights tensor** wei: $(OC \times IC \times KH \times KW)$
3 **for** mb=0 *to* MB **do**
4 **for** oc=0 *to* OC **do**
5 **for** oh=0 *to* OH **do**
6 **for** ow=0 *to* OW **do**
7 $d \leftarrow 0$
8 **for** ic=0 *to* IC **do**
9 **for** kh=0 *to* KH **do**
10 **for** kw=0 *to* KW **do**
11 $ih \leftarrow oh \cdot SH + kh \cdot (DH + 1) - PH$
12 $iw \leftarrow ow \cdot SW + kw \cdot (DW + 1) - PW$
13 $d \leftarrow d + src[mb, ic, ih, iw] * wei[oc, ic, kh, kw]$
14 $dst[mb, oc, oh, ow] \leftarrow d$
15 **Return** dst: $(MB \times OC \times OH \times OW)$

The naive way of implementing direct convolutions is to loop over the output tensor and accumulate the value of each output pixel independently through the convolution kernel and input channels. This naive implementation is described in Algorithm 1: loops from lines 3 to 6 iterates through the elements of output tensor and loops from lines 8 to 10 iterates through the kernel and the corresponding channels.

4.2 State-of-art optimizations

Despite being relatively straight-forward, the previous version suffers from poor performance. We can explain this poor performance by a non optimal reuse of cached data. Different approaches to optimize direct convolutions were described in State-of-the-art and their common point is to optimize memory accesses.

To the best of our knowledge, the first article to describe a high performance implementation of direct convolution was written by Zhang *et al.* in 2018 [16]. This article states a proper way of defining direct convolutions in order to maximize the reuse of cached memory.

A first change that can be made to Algorithm 1 is to reorder its loops. Zhang *et al.* propose to nest the loops in the following order: $MB, OH, KH, KW, IC, OW, OC$. This order allow the program to reuse $src[mb, ic, ih, iw]$ among the last loop and partially computes output starting by the OW dimension. This results in a better reuse of memory already loaded in caches.

Another optimization described by Zhang *et al.* is *cache blocking*. By dividing the loops IC, OW, and OC into blocks, we allow our implementation to keep reusing cached data between several iterations of previous loops even when IC, OW, or OC are big. These cache blocking techniques are very popular and usually provide significant performance improvements when blocking parameters are designed to fit the target architecture.

Lastly, memory layouts of the several tensors considered in this algorithm can help to optimize the reuse of cached data. Indeed, in a program containing several nested loops, ensuring that data is loaded close enough in memory can allow the compiler to perform automatic vectorization which can lead to significant performance improvement.

4.3 An optimized implementation

Following the principles stated in the previous section, we implemented an optimized version of direct convolutions. This implementation, described in Algorithm 2, implements cache blocking with blocks of size $OC_b \times OW_b \times IC_b$ and loop reordering to reuse the local variable s among the OC_b loop. According to previous section, our implementation is inspired from the algorithm proposed by Zhang *et al.* [16].

Algorithm 2: Optimized direct convolution

```
1  Input tensor src: MB × IH × IW × IC
2  Weights tensor wei: ⌈OC/OC_b⌉ × ⌈IC/IC_b⌉ × KH × KW × IC_b × OC_b
3  for mb=0 to MB do
4  |  for oc_b = 0 to ⌈OC/OC_b⌉ do
5  |  |  for ow_b=0 to ⌈OW/OW_b⌉ do
6  |  |  |  for oh=0 to OH do
7  |  |  |  |  for ic_b=0 to ⌈IC/IC_b⌉ do
8  |  |  |  |  |  for kh=0 to KH do
9  |  |  |  |  |  |  for kw=0 to KW do
10 |  |  |  |  |  |  |  ih ← oh · SH + kh · (DH + 1) − PH
11 |  |  |  |  |  |  |  for ic=ic_b × IC_b to (ic_b + 1) × IC_b do
12 |  |  |  |  |  |  |  |  for ow=ow_b × OW_b to (ow_b + 1) × OW_b do
13 |  |  |  |  |  |  |  |  |  iw ← ow · SW + kw · (DW + 1) − PW
14 |  |  |  |  |  |  |  |  |  s ← src[mb, ih, iw, ic]
15 |  |  |  |  |  |  |  |  |  for oc=oc_b × OC_b to (oc_b + 1) × OC_b do
16 |  |  |  |  |  |  |  |  |  |  w ← wei[oc, ic, kh, kw]
17 |  |  |  |  |  |  |  |  |  |  d ← s ∗ w
18 |  |  |  |  |  |  |  |  |  |  dst[mb, oc, oh, ow] ← dst[mb, oc, oh, ow] + d
19 Return dst: MB × OH × OW × OC
```

In Algorithm 2, the 3 inner-most loops correspond to an iteration within a block of size $OC_b \times OW_b \times IC_b$. These cache blocking parameter are critical for the performance of `direct`. In order to exploit the full potential of this algorithm, we should use appropriate cache blocking parameters, fitting the architecture we target. These parameters will be statically defined for each target of the performance evaluation.

The implementation considered in the performance evaluation section implements OpenMP parallelism across the loops from lines 3 to 6. Keeping the loop of line 7 inside a parallel task avoid concurrent writes into *dst* tensor. Moreover, at least one of the dimensions MB, OC, OH and OW is in general big enough to allow consistent parallelism across a loop of size $MB \times \lceil OC/OC_b \rceil \times \lceil OW/OW_b \rceil \times OH$.

5 im2row/im2col methods

5.1 The im2row and im2col transformations

A well-known alternative to direct convolution algorithm uses a specific transformation on the input tensor in order to compute the convolution as a multi-dimensional matrix-matrix multiplication (GEMM) [17]. These GEMM-based methods are widely used in State-of-the-art because the GEMM operation is already very well optimized in many high performance mathematics libraries [18]. Moreover, GEMM operations are very well supported by hardwares such as CPUs and GPUs [19].

The transformation operations applied to a tensor to turn it into a matrix to compute a convolution as a GEMM are called `im2col` or `im2row` depending on the memory format of the tensor.

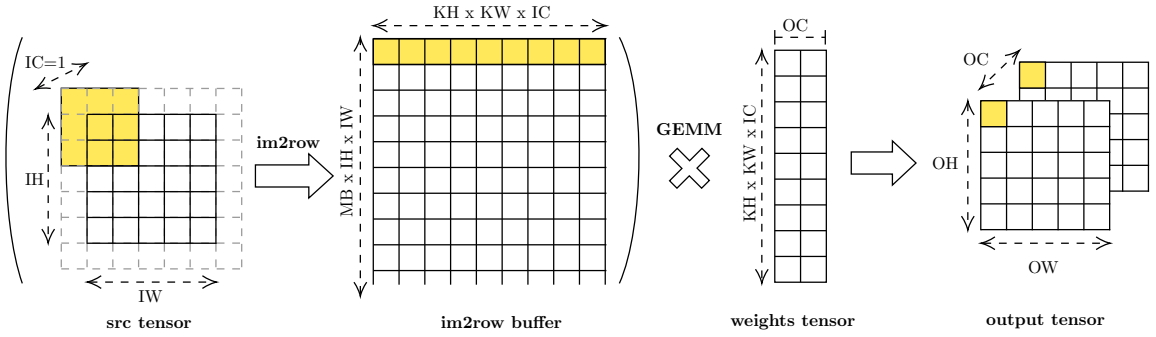

Figure 8: Computing a convolution using `im2row`.

Figure 8 represents how `im2row` moves the data from the source tensor to a buffer in order to allow the convolution to be computed as a GEMM. `im2row` transformation gathers the elements of each convolution kernel and store them in a row of the im2row buffer. As a result, im2row buffer has size $MB \times OH \times OW \times IC \times KH \times KW$.

Algorithm 3: `im2row` transformation.

Input tensor src: $(MB \times IH \times IW \times IC)$
for mb=0 *to* MB **do**
  for oh=0 *to* OH **do**
    for ow=0 *to* OW **do**
      for ic=0 *to* IC **do**
        for kh=0 *to* KH **do**
          for kw=0 *to* KW **do**
            $ih \leftarrow oh \cdot SH + kh \cdot (DH + 1) - PH$
            $iw \leftarrow ow \cdot SW + kw \cdot (DW + 1) - PW$
            $buf[mb, oh, ow, ic, kh, kw] \leftarrow src[mb, ih, iw, ic]$
Return buf: $(MB \times OH \times OW \times IC \times KH \times KW)$

The implementation of `im2row` can be derived from the implementation of direct convolutions. Indeed, Algorithm 3 shows that im2row transformation only iterates through the dimensions of `im2row` buffer and fill its rows with values of source tensor by iterating through the convolution kernels. The only difference with direct convolution implementation is that `im2row` does not compute any product between source and weights tensor but it only accesses elements of the source tensor to store them in a buffer. As a result, `im2row` efficiency is only bounded by memory while `direct` efficiency is also bounded by computational power.

The main advantage of `im2row` and `im2col` based methods is that all the floating point computations are performed in a GEMM operation with a high performance library such as BLAS. Moreover, the `im2row` transformation can be parallelized by collapsing the loops MB, OH, OW, IC and the parallelism of the GEMM is managed by the high performance framework used.

5.2 Implementation details

By using the `im2row` transformation introduced in previous section, the whole convolution implementation can be described by Algorithm 4.

Algorithm 4: im2row convolution.

Input tensor src: $MB \times IH \times IW \times IC$
Weights tensor wei: $(IC \times KH \times KW) \times OC$
$im\_buf \leftarrow$ im2row(input) ; // im_buf : $MB \times OH \times OW \times (IC \times KH \times KW)$
$dst \leftarrow$ blas_gemm(im_buf, wei)
Return dst ; // dst: $MB \times OH \times OW \times OC$

In this implementation of im2row convolutions, we can notice that src and dst tensors are stored with channels as inner-most dimension in memory. This format avoids having to reorder the output tensor after the gemm. In addition, the format with channel inner-most is widely used for other layers of CNNs and has shown state-of-art performance improvement [20].

If we suppose that the weights of the networks are stored in $IC \times KH \times KW \times OC$ format, then the GEMM can interprate these tensors as two matrices of format $(MB \times OH \times OW) \times (IC \times KH \times KW)$ and $(IC \times KH \times KW) \times OC$. Finally, the result of the GEMM of these two matrices with common dimension $(IC \times KH \times KW)$ has appropriate format $MB \times OH \times OW \times OC$.

However, this interpretation does not stand if tensors are stored with format $MB \times IC \times IH \times IW$. If we want to use such a memory format, we should consider an `im2col` transformation which transforms src tensor to a buffer with format $(IC \times KH \times KW) \times OH \times OW$. A proper im2col convolution implementation is introduced in Algorithm 5.

Algorithm 5: im2col convolution.

Input tensor src: $MB \times IC \times IH \times IW$
Weights tensor wei: $OC \times (IC \times KH \times KW)$
for $i \leftarrow 0$ *to* MB **do**
  $im\_buf \leftarrow$ im2col(input) ; // im_buf: $(IC \times KH \times KW) \times OH \times OW$
  $dst[mb] \leftarrow$ blas_gemm(wei, im_buf)
Return dst ; // dst: $MB \times OC \times OH \times OW$

The implementation described in Algorithm 5 can be useful when other layers of the CNN require tensors with format $MB \times IC \times IH \times IW$. However, the im2col convolution suffers from poor performance because it calls the gemm routine on smaller tensors. As a result, we will only consider the im2row convolution in the performance evaluation section.

A drawback of the `im2row`-based algorithms is that the im2row buffer contains a lot of data redundancy between successive lines. These redundancies cause the buffer to be very large compared to the source tensor. For example, the most common convolution layer has $KH = KW = 3$, $SH = SW = 1$, $KH = KW = 1$ and $IC = OC$, so the buffer has a size $MB \times OH \times OW \times IC \times KH \times KW$ which is $KH \times KW = 9$ times bigger than the source tensor.

5.3 Memory Efficient Convolution

A State-of-the-art work describes several ways to reduce the memory consumption of im2row or im2col based convolutions. A good way to reduce the memory growth of the im2row buffer is to generate the buffer in a compact way and compute GEMMs within slices of the buffer. Cho *et al.* described an algorithm using such a method to reduce the size of the buffer by a factor KH. This algorithm is called MEC for Memory Efficient Convolution [21].

The idea of MEC is to notice that the rows of the source tensor can be reused among IH for multidimensional kernels (for example, when $KH = KW = 3$, MEC buffer is created without redundancies among IH and GEMMs are computed on slices of size $KH \times KW$ with a step of KH over the IH dimension. Figure 9 describes the creation of MEC buffer and how it is accessed to compute the appropriate GEMMs.

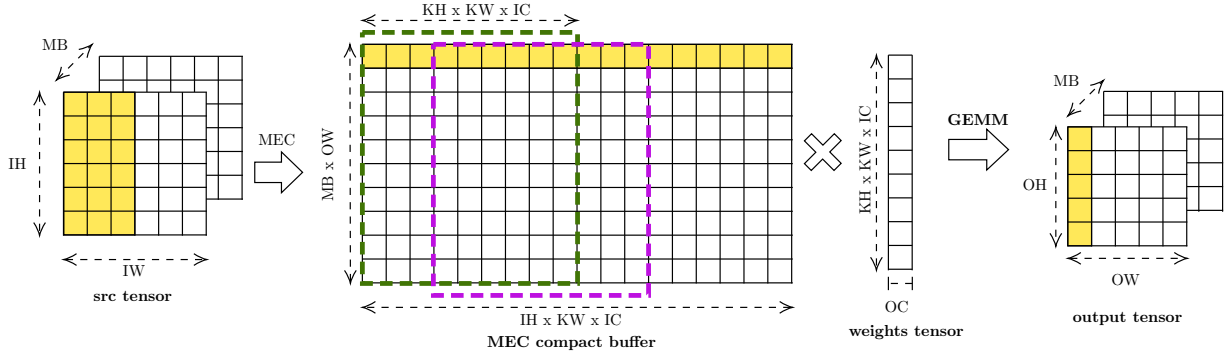

Figure 9: MEC algorithm using compact buffer.

In order to perform slicing on the IH dimension, IH should be the outer-most dimension of the multiplied tensor. In order to achieve this, a solution is to compute MEC buffer with OH-last memory format and perform a reshape on the result of the GEMMs. Algorithm 6 presents an implementation of MEC algorithm with this solution.

Algorithm 6: MEC algorithm.

1 **Input tensor** *src*: $MB \times IC \times IH \times IW$
2 **Weights tensor** *wei*: $(KH \times KW \times IC) \times OC$
3 $mec\_buf \leftarrow mec\_im2row(src)$; `// mec_buf:` $(IH \times KW \times IC) \times MB \times OW$
4 **for** $oh \leftarrow 0$ *to* OH **do**
5 $\quad start \leftarrow (SH \times KW \times IC) \times oh$
6 $\quad end \leftarrow buf\_start + KH \times KW \times IC$
7 $\quad dst[oh, :, :, :] \leftarrow \text{blas\_gemm}(mec\_buf[start : end, :]^T, wei)$; `// dst:` $OH \times MB \times OW \times OC$
8 reshape(dst, $MB \times OC \times OH \times OW$)
9 **Return** *dst*

In algorithm 6, we suppose that we have a function `mec_im2row` which transforms the *src* tensor into a compact version of the im2row buffer. This compact buffer should have $(IH \times KW \times IC) \times MB \times OW$ format to allow slicing on its first dimension. The loop in line 4 performs OH call to BLAS' GEMM between appropriate slices of mec_buf and weights. An important detail is that the slice of mec_buf is transposed in order to perform the GEMM with the weights tensor. Indeed, mec_buf slice has $(KH \times KW \times IC) \times MB \times OW$ format so it needs to be transposed to $MB \times OW \times (IH \times KW \times IC)$ format in order to perform the GEMM with the weights tensor, which has $(KH \times KW \times IC) \times OC$ format. The result of each GEMM has $MB \times OW \times OC$ format so at the end of the OH loop, destination tensor has $OH \times MB \times OW \times OC$ size. Finally, when $MB > 1$, a reshape is needed on *dst* to output it in $MB \times OH \times OW \times OC$ format.

Despite reducing im2row memory overhead, slicing the buffer and using smaller GEMMs does not allow MEC to fully benefit from the high performance GEMM implementation. As a result, our MEC implementation showed poor results compared to im2row and im2col. Moreover, the next section explains another way of taking profit from data redundancy in convolution kernels which is always better than MEC because it also reduces arithmetic complexity.

6 Winograd's method for convolutions

6.1 Winograd's minimal filtering algorithm

Schmuel Winograd has described in 1980 a method to reduce the arithmetic complexity of any Finite Impulse Response (FIR) filter [22]. However, a FIR filter behaves exactly as an unidimensional convolution, so recent works have focused on adapting Winograd's method to deep learning convolutions. A FIR filter is described as $F(m, r)$ where m is the number of outputs computed by the filter and r is the size of the kernel. Figure 10 represents the FIR filter $F(2, 3)$ as a 1D convolution.

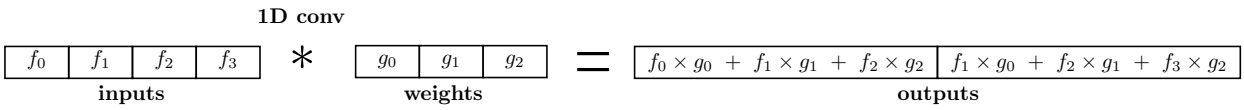

Figure 10: FIR filter F(2,3) seen as a 1D convolution

A very important result of Winograd's work is a lower bound on the number of multiplications needed to compute the output of an FIR filter. In his work, Winograd proves that computing the output of a $F(m, r)$ filter requires at least $\mu(F(m, r)) = m + r - 1$ multiplications. Winograd's work also provides an algorithm to achieve the minimal number of multiplications $\mu(F(m, r)) = m + r - 1$ when computing the output of an $F(m, r)$ filter. For the sake of comparison, the naive version requires $m \times r$ multiplications.

In his work, Winograd introduces an algorithm which computes $F(2, 3)$ using 4 multiplications instead of 6. This algorithm is described in Figure 11.

$$Y = \begin{pmatrix} f_0 & f_1 & f_2 \\ f_1 & f_2 & f_3 \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{pmatrix}, \quad \text{with} \quad \begin{array}{ll} m_1 = (f_0 - f_2)g_0, & m_2 = (f_1 + f_2)\dfrac{g_0 + g_1 + g_2}{2}, \\ m_4 = (f_1 - f_3)g_2, & m_3 = (f_2 - f_1)\dfrac{g_0 - g_1 + g_2}{2}. \end{array}$$

Figure 11: Winograd's algorithm for $F(2, 3)$.

In the algorithm described in Figure 11, we only consider 1 multiplication by term m_i because $\frac{1}{2}(g_0 + g_1 + g_2)$ and $\frac{1}{2}(g_0 - g_1 + g_2)$ can be computed for each matrix $\begin{pmatrix} g_0 & g_1 & g_2 \end{pmatrix}$.

The previous algorithm can be written in matrix form using an element-wise product written $\odot$:

$$Y = A^T \big[(Gg) \odot (B^T d) \big], \tag{3}$$

where:

$$B^T = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}, \qquad G = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}, \qquad A^T = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{pmatrix},$$

and:

$$g = \begin{pmatrix} g_0 & g_1 & g_2 \end{pmatrix}^T, \qquad d = \begin{pmatrix} f_0 & f_1 & f_2 & f_3 \end{pmatrix}^T.$$

This matricial formalism has the advantage of presenting the multiplications required by Winograd's algorithm as the element-wise product $\odot$. Indeed, the application of the transformation matrices A^T, G and B^T can be considered as simple sums.

6.2 Winograd's method applied to deep learning convolutions

The idea in Winograd's method is to take profit of the redundancy patterns of the convolution operation to reduce the number of multiplications. In the previous section, we already tried to take profit of this redundancy to reduce the storage overhead of the im2row buffer. In this section, we take this idea even further by reducing the arithmetic complexity of the convolution, following the work of Andrew Lavin and Scott Gray [23].

In their Work, A. Lavin and S. Gray explained that a minimal algorithm for a 2-dimensional filter $F(m \times m, r \times r)$ can be created by nesting two minimal algorithms for a 1-dimensional filter $F(m, r)$. With the previous matricial notation, we can write a minimal algorithm for the 2-dimensional filter as:

$$Y = A^T \big[(GgG^T) \odot (B^T dB) \big] A. \tag{4}$$

This algorithm can compute a tile of $m \times m$ outputs of a convolution with a kernel of size $r \times r$ among 1 channel with a minimal number of multiplications. It can be used to compute a convolution in a general case by dividing the input tensor into $P = MB \times \lceil \frac{IH}{m} \rceil \times \lceil \frac{IW}{m} \rceil$ tiles of size $\alpha = m + r - 1$ with an overlap of $r - 1$ between neighboring tiles. The idea is to apply $F(m \times m, r \times r)$ on each tile of size α and to use the channels dimension to compute GEMMs instead of element-wise products in the formula (4). The resulting algorithm is described in Algorithm 7.

Algorithm 7: Winograd convolution using $F(m \times m, r \times r)$

1 $P = MB \times \lceil \frac{IH}{m} \rceil \times \lceil \frac{IW}{m} \rceil$ is the number of tiles
2 $\alpha = m + r - 1$ is tile size
3 $d_{ic,b}$: image tile b in channel ic ; // $d_{ic,b} : \alpha \times \alpha$
4 $g_{oc,ic}$: weights of kernel (oc,ic) ; // $g_{oc,ic} : r \times r$
5 G, B^T, A^T are static transformation matrices
6 $Y_{oc,b}$: output tile b for kernel oc ; // $Y_{oc,b} : m \times m$
7 // transform weights: GgG^T
8 **for** *oc = 0 **to** OC* **do**
9 **for** *ic = 0 **to** IC* **do**
10 $u \leftarrow G g_{oc,ic} G^T$; // u: $\alpha \times \alpha$
11 Scatter u to U: $U[\xi, \nu, oc, ic] \leftarrow u_{\xi,\nu}$; // $U : \alpha \times \alpha \times OC \times IC$
12 // transform input tensor: $B^T dB$
13 **for** *b = 0 **to** P* **do**
14 **for** *ic = 0 **to** IC* **do**
15 $v \leftarrow B^T d_{ic,b} B$; // v: $\alpha \times \alpha$
16 Scatter v to V: $V[\xi, \nu, ic, b] \leftarrow v_{\xi,\nu}$; // $V : \alpha \times \alpha \times IC \times P$
17 // compute GEMMs
18 **for** ξ *= 0 **to** α* **do**
19 **for** ν *= 0 **to** α* **do**
20 $M[\xi, \nu, :, :] = U[\xi, \nu, :, :] V[\xi, \nu, :, :]$; // $M : \alpha \times \alpha \times OC \times P$
21 // transform output: $A^T yA$
22 **for** *oc = 0 **to** OC* **do**
23 **for** *b = 0 **to** P* **do**
24 Gather y from M: $y_{\xi,\nu} = M[\xi, \nu, oc, b]$; // $y : \alpha \times \alpha$
25 $Y_{oc,b} \leftarrow A^T yA$; // $Y_{oc,b} : m \times m$
26 (mb, oh, ow) are coordinates of tile b in *dst*
27 $dst[mb, oh + mh, ow + mw, oc] \leftarrow Y_{oc,b}[mh, mw]$; // $dst : MB \times OH \times OW \times OC$
28 **Return** *dst*

Our implementation of the Winograd algorithm uses $F(2 \times 2, 3 \times 3)$ in order to work on 3×3 convolution kernels. In Algorithm 7, we can consider that the transformations of the weights, inputs and outputs are only performing sums. As a result, the major part of the computational cost comes from the GEMMs computed at line 20. Moreover, we can avoid using a costly call to the GEMM primitive to compute the static transformations GgG^T,$B^T dB$ and $A^T yA$ by using generated code implementing corresponding sums. For this purpose, we used the python library sympy to perform the symbolic computations for each transformation and we generated the appropriate C code.

6.3 Arithmetic cost analysis

Winograd's algorithm aims at reducing the number of multiplications in the convolution operator. `direct` and `im2row` implementations require a number of multiplications equal to the number of elements in the kernel multiplied by the number of elements in the output tensor. Winograd's algorithm works for square convolutions kernels of size r. However, when $KH = KW = r$ `direct` and `im2row` performs the following number of multiplications:

$$\begin{aligned}\#mul(direct) = \#mul(im2row) &= MB \times OH \times OW \times OC \times IC \times KH \times KW \\ &= MB \times OH \times OW \times OC \times IC \times r^2.\end{aligned}$$

Winograd's algorithm performs $\alpha \times \alpha$ GEMMs between matrices of sizes $OC \times IC$ and $IC \times P$ so the total number of multiplications is $\alpha \times \alpha \times OC \times IC \times P$. By considering $SH = SW = 1$, and if we assume that $IH = OH$ and $IW = OW$, the number of multiplications of Winograd's algorithm is:

$$\begin{aligned}\#mul(wino) &= \alpha^2 \times OC \times IC \times P \\ &= (m + r - 1)^2 \times OC \times IC \times MB \times \left\lceil \frac{OH}{m} \right\rceil \times \left\lceil \frac{OW}{m} \right\rceil \\ &= MB \times OH \times OW \times OC \times IC \times \frac{(m + r - 1)^2}{m^2}.\end{aligned}$$

At this point, a remark we can make is that Winograd's algorithm does not reduce the number of multiplications when $KH = KW = r = 1$: $\frac{(m+r-1)^2}{m^2} = r^2 = 1$. Indeed a 1×1 convolution does not involve any data redundancy between successive kernels so Winograd's optimization is not relevant in this case.

Our implementation of the Winograd algorithm considers 3×3 kernels, so it requires a number of multiplications proportional to $\frac{(m+2)^2}{m^2}$. This quantity is strictly decreasing on $]0; +\infty[$ and $\lim_{m\to\infty} \frac{(m+2)^2}{m^2} = 1$ (see Figure 12). As a consequence, we can theoretically expect reducing by ~ 9 the number of multiplications of the convolutions. However, this would require m to be arbitrarily big, causing poor performance due to poor locality and an eventual large overflow of the tiles over the dimensions IH and IW of the input tensor.

At this point, our best candidates are $m = 2$, $m = 4$ and $m = 8$. Corresponding algorithms require a number of multiplications respectively proportional to 4, 2.25 and 1.5625.

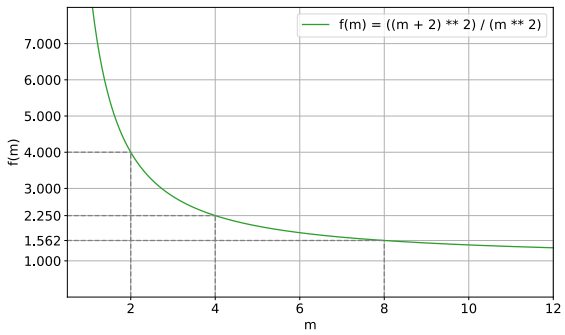

Figure 12: Multiplications required depending on m.

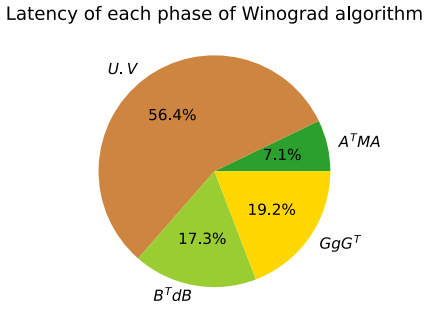

Figure 13: Profiling of our `winograd` implementation.

Figure 13 shows the profiling of our Winograd implementation for the 3×3 convolutions of Resnet50-v1.5, evaluated on the CPU of a NVIDIA Jetson AGX Orin. According to this profiling, we consider that reducing the number of multiplications is not the only factor to optimize our `winograd` implementation so we chose to stay with $F(2 \times 2, 3 \times 3)$. However, we believe that implementing Winograd algorithm with bigger values of m could yield better performance in future work.

7 Specific convolutions

7.1 Pointwise convolution

Previous algorithms were designed to optimize convolutions in the general case, supporting $KH \geq 1$ and $KW \geq 1$. However, we observe that convolutions with $KH = KW = SH = SW = 1$ are very common in most of the popular neural networks. Moreover, such a convolution becomes very simple and allows a lot of optimizations. It is often worth it to develop a custom primitive to implement efficient pointwise convolutions.

When $KH = KW = SH = SW = 1$, padding can be considered to 0 and the convolution formula can be written as such:

$$\begin{aligned} &dst(n,oc,oh,ow) \\ &\quad = \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} src(n,ic,oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L) \cdot weights(oc,ic,kh,kw) \\ &\quad = \sum_{ic=0}^{IC-1} src(n,oh,ow,ic) \cdot weights(ic,oc,0,0). \end{aligned}$$

In this formula, we suppose that the source tensor is stored with format $MB \times IH \times IW \times IC$ and the weights tensor is stored with format $IC \times OC \times KH \times KW$. As a result, we can express the result of the convolution as a simple matrix multiplication:

$$dst = src \cdot weights. \tag{5}$$

Finally, our implementation of pointwise convolutions is simply a call to the optimized GEMM featured in OneDNN. Compared to `im2row`, `pointwise` does not use a buffer to store intermediate transformations. Moreover, in contrast to direct, `pointwise` can benefit from the use of the optimized GEMM.

7.2 OneDNN implementation: implicit im2row

Besides having good performance in most cases, the `im2row` algorithm studied in section 5 is not adopted in most of commercial deep learning libraries. Indeed an approach called *implicit* is often preferred to the algorithm we presented, which can be qualified explicit. In our `im2row` implementation, we compute *explicitly* the output tensor before giving it to the GEMM primitive. Implicit approaches perform the im2row transformation on-the-fly on small tiles of the tensor and compute the output result for each tile in parallel [24]. These methods are particularly efficient on highly parallel architectures such as GPUs or TPUs [25].

A first advantage of implicit im2row methods is to allow a better adaptability for the parallelism of the convolution. Designing an implicit im2row algorithm requires the design of a blocking strategy to determine the tiles assigned to each processing unit. Moreover, performing the im2row transformations on small tiles can reduce considerably the memory overhead of the implementation, because the transformed data can be immediately reused in the GEMM. This improvement of data locality often grants performance improvement.

In this work, we consider the implementation `gemm:ref` of OneDNN [26] which implements convolutions with an implicit im2row algorithm. However, this implementation is made for CPUs, an architecture usually limited in terms of parallelism capacity compared to GPUs or TPUs. As a result, the tiles considered by OneDNN implementation are relatively big, which is not optimal regarding the performance. Designing an efficient implicit im2row implementation for CPU would be a good perspective for future work on optimizing deep learning CPU inference.

8 Performance evaluation of convolutions

8.1 Evaluation platform

The ALSOC team has designed an experiment platform called The Monolithe [1], in reference to Stanley Kubrick's movie "2001: A Space Odyssey". This platform is composed of several single board computers (SBCs), the objective being to test the suitability of embedded applications. In this context, energy consumption is a crucial factor and require an appropriate measurement tool. For this purpose, each SBC is alimented by a laboratory power supply, with a measurement board plugged in between. The energy measurements can be observed from a front-end computer, communicating with the measurement boards via USB.

Before the beginning of my internship, each board only had an independent OpenSSH server, which was not optimal to test applications over several targets. Inspired by my previous experience in high performance computing, I set up a slurm [27] environment and a Network File System (NFS) to share the files among the different targets. Slurm is a popular tool used in HPC for resource management. In concrete terms, users can now connect to a slurm frontend, upload their code on the directory shared via NFS and submit slurm jobs for compiling and testing on any SBC of the Monolithe.

In order to benchmark our implementations, we had to choose between several architectures available in the Monolithe. A good candidate was the NVIDIA Jetson AGX Orin, which has a 12-core ARM processor and 64GB of RAM. We position ourselves in the context of embedded systems so we will only run the implementations on its ARM CPU.

8.2 Evaluation methodology

All the algorithms described in the previous sections were implemented into Intel's OneDNN framework [26]. This framework provides an abstraction level to implement deep learning primitives, and comes up with a very strong benchmarking tool called BenchDNN, allowing us to evaluate the correctness and the performance of our implementations. In addition, OneDNN is used as backend for several well-known deep learning frameworks such as Pytorch or Tensorflow. Therefore, implementations could be easily reused for further study of CNN inference.

A first metric we consider is the latency of a convolution. The latency is given by BenchDNN, each time measured on several executions of the same convolution. OneDNN implementations are compiled with the C++ optimization flags `-fopenmp -O3`, which means that our implementations will support OpenMP parallelism and be compiled with optimization level 3. Moreover, the framework only measures the computational part of the convolution, without taking into account tensors allocation and primitive declaration in the measure. This allow to focus only on algorithmic comparison of the different implementations.

A second metric we consider is energy consumption. Indeed, this metric is critical for system-on-chip targets, which are often power-limited. The ALSOC team has designed high frequency boards to measure electric power on the SBCs of the monolithe. These boards allow us to measure electric power with a sampling frequency of 5000Hz, which is way more precise than the 10 Hz of NVIDIA power rails embedded in the Jetson AGX Orin. Moreover, an important feature of the measure boards is the implementation of General Purpose Inputs/Outputs (GPIOs) which allows us to isolate the measures corresponding to a specific part of the code. It is important to note that we give our performance results in terms of energy, obtained by multiplying the average power by the latency of an implementation.

In some cases, we want to evaluate the performance of a convolution layer depending on its parameters. For this purpose, we introduce a quantity called pb_size defined as:

$$pb\_size = MB \times OC \times OH \times OW \times KH \times KW \times IC$$

This quantity is proportional to the number of floating point operations performed by `direct` and `im2row` implementations. It allows us to predict in most cases if a convolution is expected to be computed faster than another one.

[1] A detailed description is available at `https://monolithe.proj.lip6.fr/`.

8.3 Benchmarking pointwise convolutions

When we introduced CNNs, we described the inference phase as the successive application of different convolution layers. These layers can be very different but the most common CNN models do not require convolution kernels that have dimensions different from 1×1 (called pointwise) and 3×3.

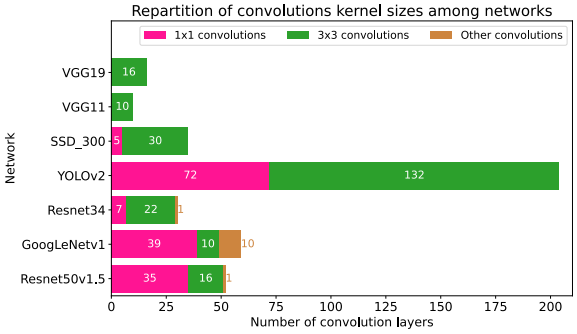

Figure 14: Structure of studied networks.

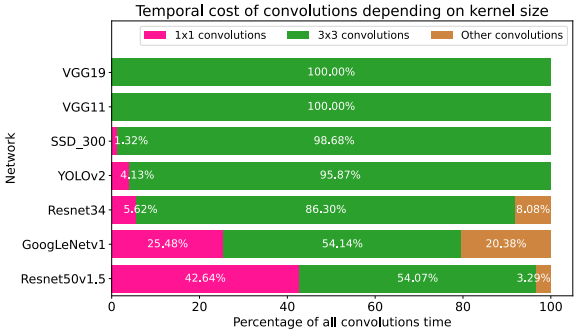

Figure 15: Temporal cost of 1×1 convolutions.

Figure 14 shows the distribution of convolution layers in seven popular CNN models, according to BenchDNN test files. We observe that a major part of the convolution layers are indeed 3×3 and 1×1 convolutions. Using Figure 15, we can compare the computational cost of these two kinds of convolution layers. We then remark that despite consisting in a large fraction of convolution layers, pointwise convolutions are not very costly in terms of performance.

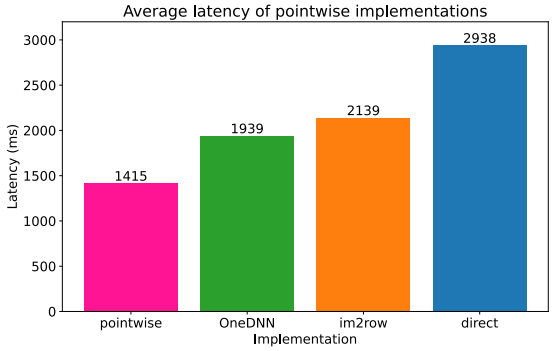

Figure 16: Latency of 1×1 implementations.

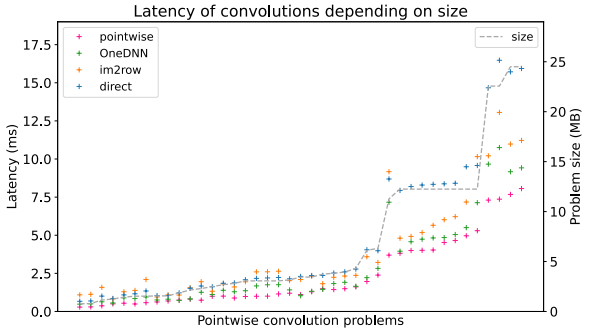

Figure 17: Latency of 1×1 convolution layers.

Figure 16 shows the average latency of convolution layers of height popular networks. We observe that our specific implementation of pointwise convolutions is, as expected, faster than other implementations. Figure 17 illustrates the behavior of the different implementations of pointwise convolutions depending on problem size. More generally, we observe that in a multithread context, `pointwise` still provides this kind of speedups compared to the other implementations.

8.4 Mono-thread latency of 3×3 convolutions

As observed in Figure 15, 3×3 convolutions are the most critical layers in terms of latency. In this subsection, we evaluate the implementations `direct`, `im2row`, `wino` and `onednn` described previously. Figure 18 and Figure 19 presents the latency of 3×3 convolutions on height different CNNs. In both figures, we observe that `wino` is often the most efficient implementation. However, the case of Densenet in Figure 18 shows that for a network containing only very small convolutions (i.e. small values of *pb_size*), the implementations using additionnal memory computations such as `im2row` and `wino` are slower than `onednn` and `direct`.

In order to study this phenomenon, we described in Figure 20 the average latency of 3×3 convolutions for small *pb_size*. In this Figure, we observe that `im2row` latency is proportional to *pb_size* due to the memory accesses performed by im2row transformation. However, the other implementations can still benefit from cached data access, accelerating the computations and reducing the cost of memory accesses. For convolutions with small *pb_size*, the latency is minimal for the implicit lowering approach `onednn`.

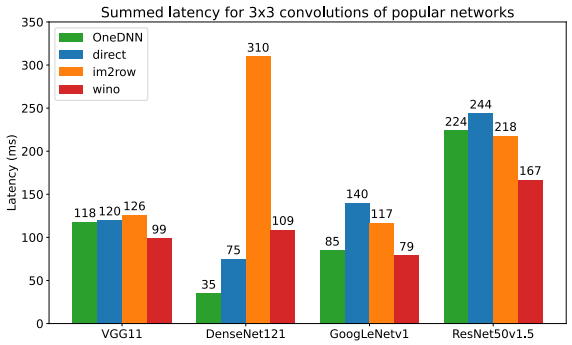

Figure 18: Total latency of 3×3 convolutions on smallest networks.

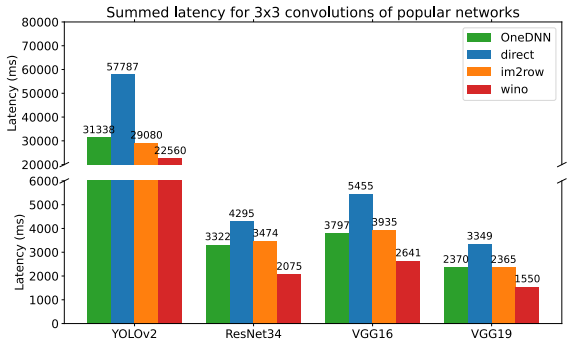

Figure 19: Total latency of 3×3 convolutions on biggest networks.

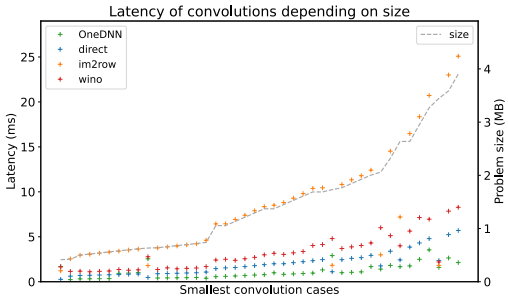

Figure 20: Average latency of 3×3 convolutions for small *pb_size*.

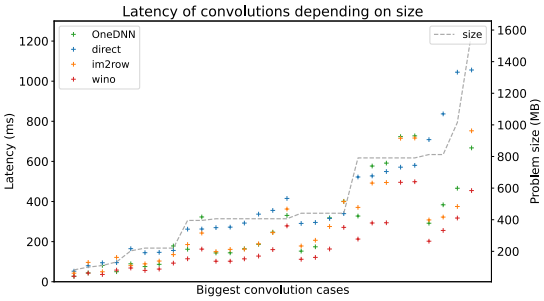

Figure 21: Average latency of 3×3 convolutions for big *pb_size*.

Figure 21 shows that the approaches `im2row` and `wino` are often more efficient for the biggest convolution layers. An explanation is that the relative cost of the transformations becomes low enough to get a lower latency compared to `direct` and `onednn`. Moreover, we can observe in Figure 23 that despite having the lowest latency on big convolutions, `wino` executes less floating-point operation per seconds (Flop/s). This illustrates the reduction of arithmetic complexity achieved by Winograd's algorithm. Figure 22 illustrates the fact that on small cases, `onednn` implementation is the only one to perform a high number of Flop/s. Indeed, `wino` and `im2row` are slowed down by their transformations and `direct` by the succession of memory accesses and floating point multiplications.

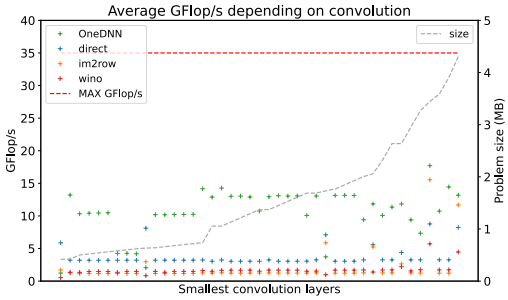

Figure 22: Average GFlop/s of 3×3 convolutions on smallest networks.

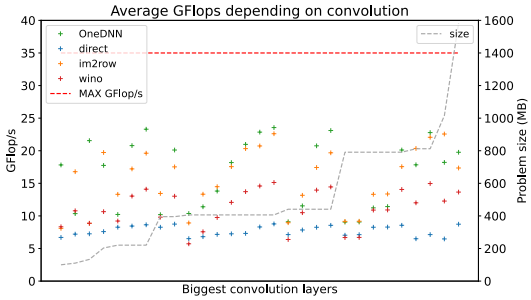

Figure 23: Average GFlop/s of 3×3 convolutions on biggest networks.

To conclude, mono-thread performance results show that `wino` is the most efficient implementation on big convolutions due to its low arithmetic complexity. However, for small convolution layers, an implicit approach such as `onednn` is often more efficient.

8.5 Mono-thread energy consumption

By using the measurement boards described in section 8.1, we measured the electric power consumption during the execution of the convolutions. Figures 24 and 25 show the power consumption during the execution of several convolution layers depending on the size of the problem. We observe that the instantaneous power measures are relatively similar between the several implementations. Because the energy consumption is equal to $power \times latency$, the total energy consumed for computing several convolutions will be almost proportional to the sum of their latencies. Figure 26 and 27 show the total energy required by 3×3 convolutions of several popular CNNs. As expected, they look very similar to the latency measures (Figures 18 and 19).

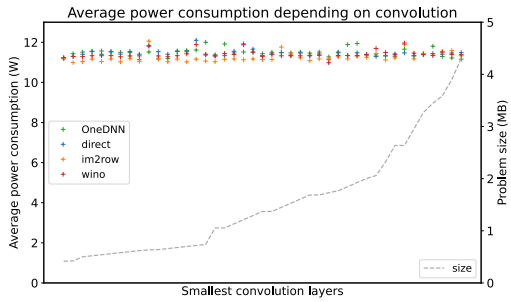

Figure 24: Average power of 3×3 convolutions on smallest networks.

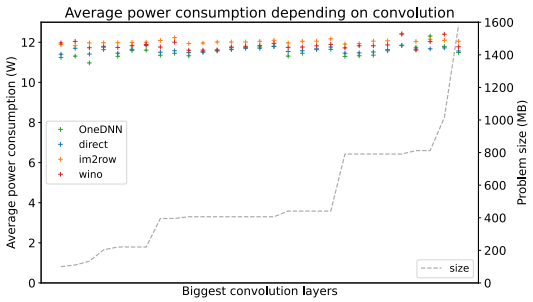

Figure 25: Average power of 3×3 convolutions on biggest networks.

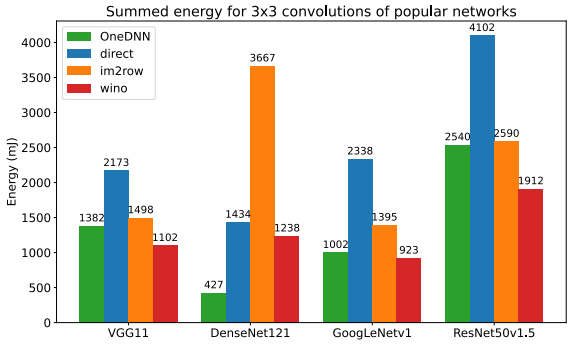

Figure 26: Summed energy of 3×3 convolutions on smallest networks.

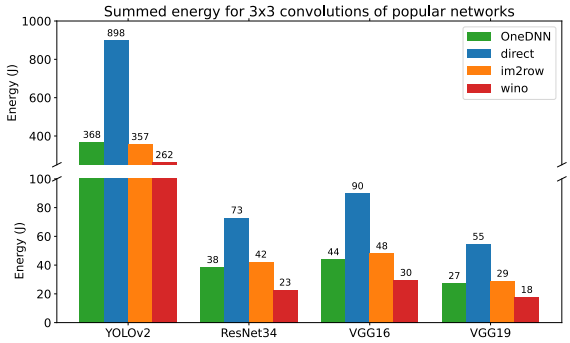

Figure 27: Summed energy of 3×3 convolutions on biggest networks.

8.6 Multi-threading scalability

In this section, we studied how our implementations behave while we increase the number of threads available. Our implementations are parallelized using OpenMP on outter-most loops, which are collapsed. We change the number of threads by modifying the environment variable `OMP_NUM_THREADS`.

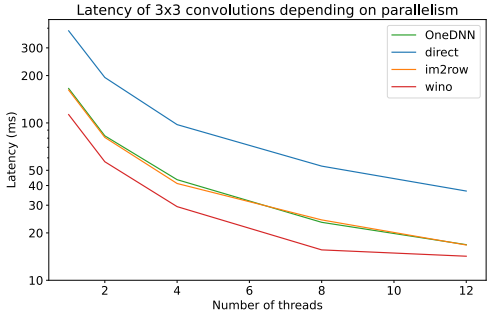

Figure 28: Latency of 3×3 convolutions depending on parallelism.

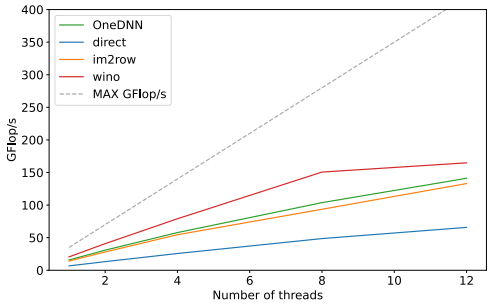

Figure 29: GFlop/s of 3×3 convolutions depending on parallelism.

Figure 28 shows the evolution of the latency depending on the number of threads. We observe that `wino` is always the fastest implementation but it gets almost no speedup between 8 threads and 12 threads. This phenomenon is also observed in Figure 29. Indeed, for 8 threads, `wino` seem to reach a limit in terms of GFlop/s. This limit is way below CPU's peak performance (dashed line), which indicates that our implementations are memory bounded.

Lastly, Figure 30 indicates the evolution of electric power consumption regarding the number of threads. Overall, we observe that when we increase the number of threads by a factor X, power consumption usually increases by a factor $Y < X$. A consequence of that is the fact that energy consumption of a convolution decreases when we increase the number of threads, as we can see it in Figure 31. Indeed, the energetic cost of increasing the number of threads is lower than the benefits obtained by increasing computational power. Moreover, we observe that `wino` consumes almost as much power for 8 threads than for 12 threads. A consequence of this can be shown in Figure 31: even if `wino` does not scale as well as the other implementations when the number of threads increases, `wino` is always the less energy-consuming implementation.

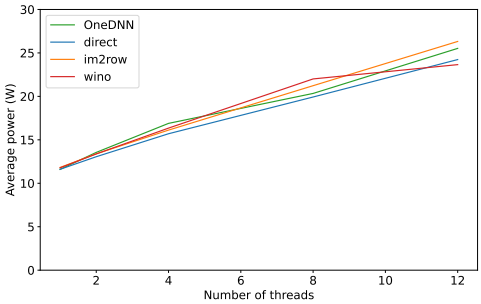

Figure 30: Instantaneous power consumption of 3×3 convolutions depending on parallelism.

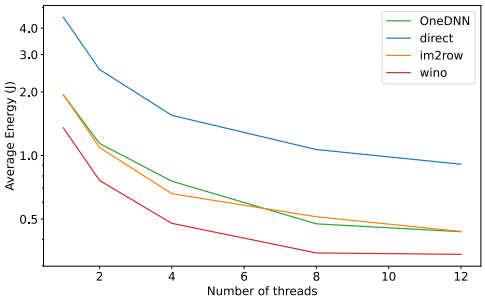

Figure 31: Energy consumption of 3×3 convolutions depending on parallelism.

Conclusion

This work gives a general overview of the most used algorithms to compute a forward pass through a convolution layer. Because of the complexity of the convolution operator, we could not determine one implementation being better than the others in any case. However, we determine that Winograd's method should be used on biggest convolution layers while implicit lowering approaches are better on small convolution layers. Moreover, optimizing the other convolution layers could be useful, especially the pointwise convolution which can be computed in one call to the GEMM implementation of the high-performance library BLAS. Our study also focuses on the energy consumption of the different implementations, which is a critical parameter for Systems-on-Chips. The conclusion to our performance analysis study is that the energy consumption of our implementations is almost proportional to their latency. As a consequence, we state the same guidelines to minimize energy consumption than to minimize latency. The only trade-off that we observed concerns multithreading: we do not observe any significant benefits in increasing the number of threads from 8 to 12 but it costs some energy and 4 process units that could be used for another job.

To summarize, this work states principles for implementing efficient forward convolution layers. However, deep neural networks are a wider structure composed of different layers. As a result, an interesting perspective for further work would be to add the dimension of the network, and to explore cross-layer optimizations.

References

[1] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022.

[2] N. Rambaux, J. Vaubaillon, L. Lacassagne, D. Galayko, G. Guignan, M. Birlan, P. Boisse, M. Capderou, F. Colas, F. Deleflie, F. Deshours, A. Hauchecorne, P. Keckhut, A. C. Levasseur-Regourd, J.-L. Rault, and B. Zanda, "Meteorix: a cubesat mission dedicated to the detection of meteors and space debris," in *1st ESA NEO and Debris Detection Conference*, (Darmstadt, Germany), p. 9 p., ESA Space Safety Programme Office, Jan. 2019.

[3] A. Al-Owais, M. E. Sharif, S. Ghali, M. Abu Serdaneh, O. Belal, and I. Fernini, "Meteor detection and localization using YOLOv3 and YOLOv4," *Neural Computing and Applications*, vol. 35, pp. 15709–15720, July 2023.

[4] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, pp. 541–551, 12 1989.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, p. 84–90, may 2017.

[6] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," 2013.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[9] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018.

[10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.

[11] 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," 2017.

[12] M. F. Dolz, S. Barrachina, H. Martínez, A. Castelló, A. Maciá, G. Fabregat, and A. E. Tomás, "Performance–energy trade-offs of deep learning convolution algorithms on arm processors," *The Journal of Supercomputing*, vol. 79, pp. 9819–9836, Jan. 2023.

[13] Stanford, "Cs231n: Convolutional neural networks for visual recognition." https://cs231n.github.io/.

[14] 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*, pp. 8024–8035, Curran Associates, Inc., 2019.

[15] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir, "A compiler framework for extracting superword level parallelism," *SIGPLAN Not.*, vol. 47, p. 347–358, jun 2012.

[16] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," Sept. 2018.

[17] 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.), (La Baule (France)), Université de Rennes 1, Suvisoft, Oct. 2006. http://www.suvisoft.com.

[18] B. Kågström, P. Ling, and C. van Loan, "Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, p. 268–302, sep 1998.

[19] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched gemm for gpus," in *High Performance Computing* (J. M. Kunkel, P. Balaji, and J. Dongarra, eds.), (Cham), pp. 21–38, Springer International Publishing, 2016.

[20] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," *CoRR*, vol. abs/1704.04428, 2017.

[21] M. Cho and D. Brand, "Mec: Memory-efficient convolution for deep neural network," June 2017.

[22] S. Winograd, *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, Jan. 1980.

[23] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," Sept. 2015.

[24] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "Delta: Gpu performance model for deep learning applications with in-depth memory system traffic analysis," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Mar. 2019.

[25] Y. Zhou, M. Yang, C. Guo, J. Leng, Y. Liang, Q. Chen, M. Guo, and Y. Zhu, "Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators," *CoRR*, vol. abs/2110.03901, 2021.

[26] Intel, "Oneapi deep neural network library." https://oneapi-src.github.io/oneDNN/.

[27] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing* (D. Feitelson, L. Rudolph, and U. Schwiegelshohn, eds.), (Berlin, Heidelberg), pp. 44–60, Springer Berlin Heidelberg, 2003.