ACCEPTED VERSION AT IPDPSW 2021 – DOI 10.1109/IPDPSW52791.2021.00016

Pooling Acceleration in the DaVinci Architecture Using Im2col and Col2im Instructions

Caio S. Rohwedder[+*], João P. L. de Carvalho[+*], José Nelson Amaral[*],
Guido Araújo[+], Giancarlo Colmenares[†], Kai-Ting Amy Wang[†]

[+]*Institute of Computing*
University of Campinas (UNICAMP)
Campinas, SP, Brazil
{joao.carvalho,guido}@ic.unicamp.br
{c157754}@dac.unicamp.br

[*]*Dept. of Computing Science*
University of Alberta
Edmonton, AB, Canada
{jamaral}@ualberta.ca

[†]Heterogeneous Compiler Lab.
Huawei Canada Research Centre
Markham, ON, Canada
{kai.ting.wang}@huawei.com
{giancarlo.colmenares}@huawei.com

***Abstract*—Image-to-column (Im2col) and column-to-image (Col2im) are data transformations extensively used to map convolution to matrix multiplication. These transformations rearrange the inputs of convolution to avoid its strided memory access pattern, thus providing a friendlier data layout for CPUs and GPUs. In artificial intelligence (AI) accelerators, these transformations allow convolution to be computed in matrix-multiplier units. Implemented in software, however, they impose a significant overhead that must be compensated by the efficiency gains of matrix multipliers. DaVinci is an AI accelerator architecture that introduces instructions to optimize Im2col and Col2im. Another core layer of convolutional neural networks that presents a strided memory access pattern is pooling. This paper explores the specialized Im2col and Col2im instructions to accelerate pooling layers in DaVinci. An experimental evaluation reveals that the proposed pooling implementations can yield speedups of up to 5.8 times compared to a baseline that does not use these specialized instructions. The speedups follow from an improved memory layout in the inputs of pooling, as this layout leads to better utilization of the vector processing unit in DaVinci.**

***Index Terms*—CNN, AI Accelerator, Maxpool, Gradient, TVM**

I. Introduction

With the increasing adoption of convolutional neural networks (CNNs), the optimization of each of its components has become fundamental. Convolution has been the main target of optimization because it is the most used and expensive layer in CNNs. However, many modern CNN architectures also use pooling to extract translation-invariant features and to perform subsampling. Max-pooling is the main variant of pooling that subsamples using the maximum value. While the performance impact of pooling is low compared to convolution, a naive implementation can hinder the overall performance of a CNN [1].

DaVinci [2] is an AI accelerator architecture that implements scalar, vector, and matrix multiplier units. The matrix multiplier unit allows efficient computation of convolution and other CNN layers, such as the fully connected, that can be mapped to matrix multiplication [3]. Convolution is mapped to matrix multiplication through the Im2col and Col2im data transformations. These transformations are memory-intensive and add significant performance overhead to convolution. However, highly optimized solutions for matrix multiplication both in software (*e.g.*, OpenBLAS and Eigen libraries) and in hardware (*e.g.*, matrix multipliers) overcome this overhead. Still, DaVinci introduced instructions to optimize Im2col and Col2im. First, Im2col is performed during a load instruction (`Im2Col`) just before data reaches the memory buffers closest to DaVinci's computational units. As such, this operation uses no temporaries and its memory overhead is only seen in these buffers. Second, `Col2Im` is vector instruction capable of better vectorizing over the scattered access pattern of Col2im. By using these instructions, convolution is computed in the matrix multiplier unit at a low overhead.

Max-pooling also has a strided access pattern, but unlike convolution it cannot be mapped to the matrix multiplier. Even so, its implementation can leverage the specialized `Im2Col` and `Col2Im` instructions. This paper thus proposes two key ideas to accelerate pooling in DaVinci: to produce an improved data layout by applying `Im2Col` instructions to the input of forward pooling, and to apply `Col2Im` instructions to the backward pooling instead of traditional vector instructions. Previous attempts to accelerate CNNs using FPGAs proposed pooling-specific instructions and computational units [4], [5]. Whereas the proposed approach uses a general-purpose vector computational unit and instructions primarily designed for convolution. Earlier work on improving pooling also overlooks its backward implementation [4], [6], [7], which is essential for training. Lastly, operation fusion, which effectively improves pooling paired with convolution [6], [8], is independent of the Im2col/Col2im based implementation presented in this work. Both optimizations can be applied in conjunction.

The main contributions of this paper are:

- A description of DaVinci's `Im2Col` and `Col2Im` instructions, showing how they are executed and how they integrate into DaVinci's datapaths (Section III).
- An approach to accelerate pooling with an Im2col-based forward implementation and a Col2im-based backward implementation using the DaVinci-specific `Im2Col` and `Col2Im` instructions (Section V).
- A rigorous evaluation of multiple pooling implementations in DaVinci, revealing speedups of up to 5.8 times on the Im2col/Col2im based implementations (Section VI).

The remaining sections are organized as follows: foundational concepts for this work appear in Section II. Section IV presents the software stack used to implement pooling operators for DaVinci. Finally, Section VII presents related works, and Section VIII concludes this paper.

II. Background

A. Convolution, Im2col and Matrix Multiplication

Convolution is a filtering operation used in image processing. This operation is the main building block of CNNs [9]. In them, convolution repeatedly applies a kernel — a multi-channel filter composed of trainable weights — over patches of the input image. Patches are regions of the input that have the same size as the kernel. They are selected based on the stride parameters S_h and S_w, and given these parameters may or may not overlap. In an application of a kernel, its weights multiply a patch of the input. The multiplied results are summed together to generate a single output. A kernel is applied over each patch to generate a two-dimensional output, which is called a feature map. Convolution uses multiple kernels to produce multiple feature maps that are stacked as channels into a three-dimensional output.

The memory layout for the input of a convolutional layer is commonly described as $NCHW$, where each character represents a dimension of a four-dimensional input: the number of images stacked together (N), channels (C), height (H), and width (W). The character's order specifies the order in which each dimension is arranged in memory. For simplicity, the dimension N has a length equal to one throughout the paper.

Convolution unrolling, also known as Image-to-Column (Im2col[1]), is a data transformation that allows the mapping of convolution into matrix-matrix multiplication [11]. This transformation, illustrated in Figure 1, consists of creating two matrices, Out_{In} and Out_{Ker}, based on the input image and the kernels, respectively. Each row of matrix Out_{In} contains all the input needed to compute one element of an output feature map linearized into one dimension. Each column of matrix Out_{Ker} contains the weights of a kernel similarly linearized. Thus, multiplying Out_{In} and Out_{Ker} is equivalent to performing convolution with its original inputs.

If the stride sizes (S_h, S_w) are smaller than the kernel's height and width (K_h, K_w), patches will overlap. The overlapping elements will be copied to multiple rows of matrix Out_{In}, resulting in a bigger memory footprint. This is the main drawback of the Im2col technique when contrasted with direct-convolution based approaches. An example with a single channel is shown in Figure 2. The two patches are highlighted and they overlap on the elements $\{3, 8, 13\}$. As a result, these elements appear in both rows of the output of Im2col (on the right). Nonetheless, Im2col is used across AI frameworks to implement convolution because matrix multiplication offers an input with a friendlier memory layout to CPUs and GPUs, making it easier to apply vectorization techniques [11]. The availability of optimized linear algebra libraries such as OpenBLAS [12], ATLAS [13], and Eigen [14], and AI accelerator designed around matrix multiplier units, further incentivizes such a transformation.

[1]The transformation of the input image can also be an image-to-row transformation if the multiplication is transposed $(AB)^T = B^T A^T$ [10]. The Im2col name will be used to refer to all variants of this transformation.

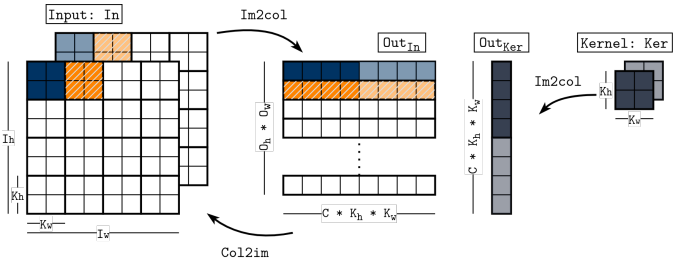


Fig. 1. Im2col: In (C, I_h, I_w) is transformed into matrix Out_{In} $(O_h \times O_w, C \times K_h \times K_w)$ and a single kernel (C, K_h, K_w) is transformed into the matrix Out_{Ker} $(C \times K_h \times K_w, 1)$, where O_h and O_w represent the number of patches in the height and width of the input. The bold squares $(2, 2)$ in In represent patches of the image to which the kernel is applied. Col2im: the backward operator of Im2col, from Out_{In} to In.

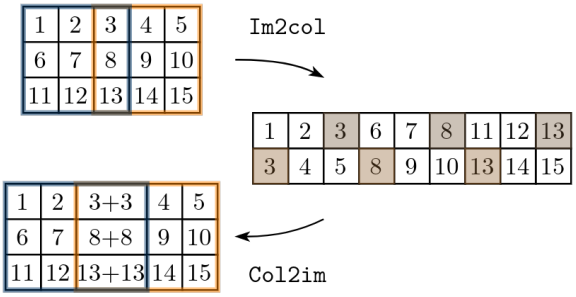


Fig. 2. Im2col and Col2im performed on two overlapping patches.

B. Backward Operators and Col2im

To train a neural network, the input values are first propagated forward to produce an output. Then, the error between this generated output and the expected output is calculated through a loss function. The gradient of this loss function is propagated backward towards the input so that the network can be tuned. Thus, every forward operator has a dual-operator applied in the backward pass, namely its *backward operator* [15].

The backward operator of Im2col is called Col2im, and it is also illustrated in Figure 1. Col2im is used in the backward propagation pass of convolutional layers implemented with Im2col. The incoming gradients in the shape of the matrix Out_{In} are propagated back to the original $NCHW$ layout. If there is no overlap, as in the example of Figure 1, Col2im simply returns the matrix to its original shape. But when patches do overlap, gradients that refer to the same position in the output are summed, as shown in Figure 2.

C. Pooling Operators

Spatial feature pooling subsamples images to obtain translation-invariant feature maps in computer-vision archi-

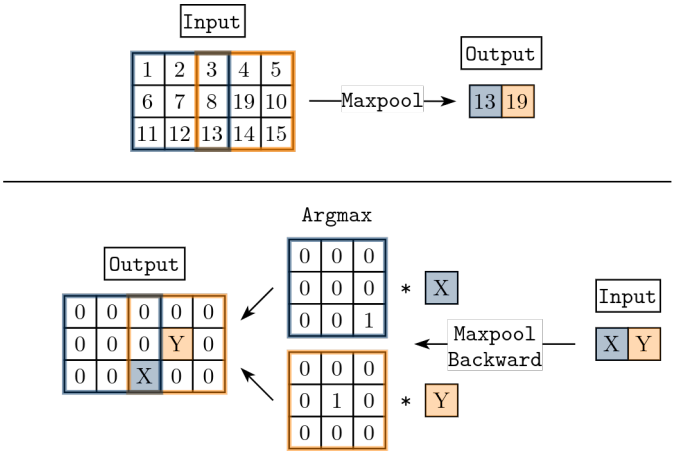


Fig. 3. Forward and backward computation of `MaxPool` for two overlapping patches.

tectures [16]. Similar to convolution, pooling is one of the building blocks of CNNs. Pooling layers are commonly used in modern CNN architectures such as Resnet [17], Inception [18], and Xception [19]. Pooling also applies a kernel over patches of its $NCHW$ input. But unlike convolution, the kernel has no weights, it only selects patches based on the stride parameters. A reduction function is applied to the selected patches to subsample the input. This reduction is typically applied to the height (H) and width (W) dimensions of the input, operating on the channels independently. As a result, the output of pooling has the same number of channels as the input. Different reduction functions can be chosen: the `max` function selects the maximum value (`MaxPool`), and the `avg` function computes the average of the patch (`AvgPool`). `MaxPool` is preferred among CNNs as it looks at the maximal activation of features, rather than diluting them with an average [20]. Figure 3 (on top) shows an example of `MaxPool` forward.

The implementation of backward pooling depends on the reduction function utilized. For `Maxpool`, each input is multiplied by its corresponding `Argmax` mask, where the position of the maximum element in the original patch is set to 1 and all the other positions are set to 0. Next, as with Col2im, the masks return to the original $NCHW$ shape and the overlapping elements are summed together. This output correlates how much a change in each input element of `MaxPool` forward affects its output elements [21]. Figure 3 shows an example of `Maxpool` backward on two overlapping patches. In summary, the gradients are only propagated backward to the maximum elements [22].

III. The DaVinci Architecture

DaVinci [2] is an AI accelerator architecture used by Huawei's Ascend chips. The following subsections describe components of the Ascend 910 chip.

A. AI Core

Figure 4 shows a closer view of DaVinci's main component, the AI Core, and its corresponding data paths. The AI Core is

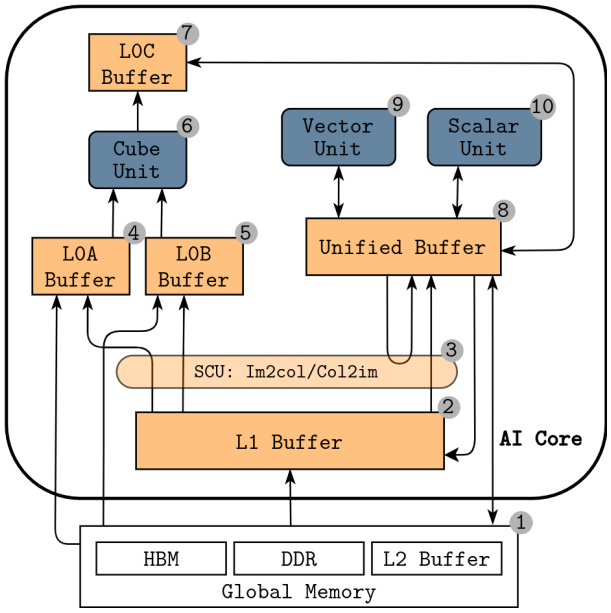


Fig. 4. Data paths of the AI Core.

composed of three processing units (Cube, Scalar, and Vector Unit), a set of private buffers (`L0A`, `L0B`, `L0C`, `L1`, and `Unified` Buffer), and a Storage Conversion Unit (SCU). Outside of the AI Core sits the Double Data Rate (DDR) and High Bandwidth Memory (HBM) memories and an `L2` Buffer, all of which are shared among the AI Cores of a chip.

Both Scalar and Vector Units operate on data loaded from/-stored to the `Unified` Buffer. The Vector Unit performs basic arithmetic and logic vector operations (*e.g.*, subtracting two vectors). It uses a 128-bit mask register in which every bit represents one element of a vector instruction that may be processed or not. The Scalar Unit has both general and special-purpose registers, which are used to execute control-flow and scalar arithmetic operations, as well as index and address calculations.

The Cube Unit is based on a multidimensional systolic array [23], it implements matrix multiplication using an array of processing elements that perform multiply-accumulate operations. This unit acts similarly to the Matrix Multiplier Unit (MXU) of Google's Tensor Processing Unit [24]. Buffers `L0A` and `L0B` store the inputs of the Cube Unit, and the `L0C` buffer stores its output. While the operands for the Vector Unit are vectors, the Cube Unit receives *data-fractals* from its input buffers. A *data-fractal* is a small matrix of a constant size of 4096 bits. The Cube Unit can multiply two data-fractals per clock cycle.

The private buffers of the AI Core (`L0A`, `L0B`, `L0C`, `L1`, and `Unified` Buffer) are organized as scratch-pad memories [25]. Data movement between these buffers must be explicitly managed by the application, in contrast, hardware-managed caches are transparent to the application and ensure consistency by hardware protocols. Thus, the programmer needs to specify which data should be brought to each buffer, and also needs

to maintain data consistency. In a scratch-pad memory, each buffer has its own address space, which is separated from the address space of the memory. With this organization, more complexity is placed upon the application's code, but it comes with the benefit of not requiring tag bits, dirty bit, and the comparison logic that transparent caches need in the hardware. From the AI Core's perspective, all shared memories (DDR, HBM, and L2) are considered global memory and are represented as (1) in Figure 4. Given that their data-paths are the same, they are drawn only once.

The Storage Conversion Unit (SCU) may perform many data-layout transformations when data is transferred between buffers. This unit implements Im2col, Col2im, and other transformations, out of the scope of this work, such as padding, matrix-tile transposition, and sparse-matrix decompression. The SCU enables instructions, such as `Im2Col`, to perform fast layout transformations while data is transferred between buffers. As a result, the memory overhead that these transformations may imply appears only on the target buffers. Such instructions were specifically designed to operate on the memory layout described next.

B. Fractal Memory Layout

To avoid memory alignment and padding problems in the Cube Unit, DaVinci includes the constant-length dimension C_0 in the representation of an input image. As a result, a slight variation of $NCHW$ is used, called the *fractal memory layout*. This format is represented by NC_1HWC_0, in which C_0 represents part of a split in the channel dimension (C) of $NCHW$. To make the conversion from $NCHW$ to NC_1HWC_0, C is split into C_1 and C_0, where $C_1 = \lceil C/C_0 \rceil$. If the original number of channels (C) is not divisible by C_0, the C_0 dimension must be zero-padded to reach its required length. Given a data type, the length of C_0 makes the inputs of the Cube Unit (data-fractals) always have 4096 bits of data. A data-fractal has $16 * C_0$ elements, thus for `Float16`, C_0 has a length of 16. For `Unsigned8`, C_0 has a length of 32. The data type `Float16` is adopted in this paper.

C. Im2Col Instruction

`Im2Col` is a data-transformation instruction executed in the SCU that acts as a load instruction. It may be applied to a data-fractal that is loaded from `L1` to `L0A` (2)→(4) and `L0B` (2)→(5), so as to prepare data for computation in the Cube Unit. It may also be applied to a data-fractal that is loaded from `L1` to the `Unified` buffer (2)→(8), to prepare data for computation in the Vector and Scalar Units.

There are two main differences when comparing the `Im2Col` instruction to the Im2col transformation shown in Figure 1. First, `Im2Col` is a single instruction, it is only able to load and transform one fractal of an image at a time. Even if it could operate on a whole image, its target buffers (`L0A`, `L0B`, `Unified` Buffer) may not be capable of storing the transformed image. For this reason, `Im2Col` instructions can be used to load and transform a tile of an input. Second, `Im2Col` is designed to load an input that is in the fractal memory layout NC_1HWC_0. Therefore, its output will also have a different memory layout when compared to the one shown on the right of Figure 1. The advantage of performing Im2col as load instruction is that the increase in memory overhead from duplicated elements only appears in the target buffers (`L0A`, `L0B`, and `Unified` buffer), which are the buffers closest to the Cube and Vector Units.

`Im2Col` needs the following parameters related to the input image (or tile), which are constant for all instructions loading the same input:

- Height (I_h) and width (I_w) of the input image;
- Left (P_l), right (P_r), top (P_t), and bottom (P_b) zero padding;
- Stride in the height (S_h) and width (S_w) directions;
- Kernel height (K_h) and width (K_w).

Based on these parameters, the number of patches (O_h, O_w) in the input's height and width can be calculated by Equation 1. Furthermore, each `Im2Col` instruction needs the three following positional parameters to choose which elements of the input it will load, in which the parameters (x, y) are coordinates in the height and width (HW) dimensions of the input.

- The starting position in the image (x, y);
- Relative position inside of a patch (x_k, y_k);
- Access index of the C_1 dimension (c_1).

$$\begin{aligned} O_h &= \left\lfloor \frac{I_h + P_b + P_t - K_h}{S_h} \right\rfloor + 1 \\ O_w &= \left\lfloor \frac{I_w + P_l + P_r - K_w}{S_w} \right\rfloor + 1 \end{aligned} \tag{1}$$

To load a fractal (16 rows of C_0 elements) to a buffer, `Im2Col` performs the following tasks: (i) process each element of dimension N individually; (ii) access the element c_1 of dimension C_1; (iii) select the next 16 consecutive patches starting from position (x, y); (iv) select the elements in the (x_k, y_k) position, relative to each of the 16 patches; (v) load the C_0 dimension for the 16 selected elements; (vi) store the loaded elements as a fractal into the target buffer.

Figure 5 exemplifies a small image loaded using four `Im2Col` loads. The input image is in the fractal layout NC_1HWC_0, but the lengths of N and C_1 are 1, so they are not shown. The parameters used in this example correspond to: $(I_h, I_w) = (8, 8)$, $(K_h, K_w) = (2, 2)$, $(S_h, S_w) = (2, 2)$, and $(O_h, O_w) = (4, 4)$. Notice that there is no padding. The input has exactly 16 patches (bold squares), so (x, y) is set to the first position $(0, 0)$ and is not changed afterward. For the first `Im2Col` (blue squares), $(x_k, y_k) = (0, 0)$, while for the second (orange squares), $(x_k, y_k) = (0, 1)$. Two more `Im2Col` instructions are issued, corresponding to (x_k, y_k) equal to $(1, 0)$ and $(1, 1)$. This results in four fractals concatenated side by side. If there were more patches in the image, (x, y) would be changed to another position to create a new row of fractals in the output. Bigger inputs are loaded by issuing multiple `Im2Col` instructions while iterating the positional parameters sequentially. This iteration can be seen as if it composed a

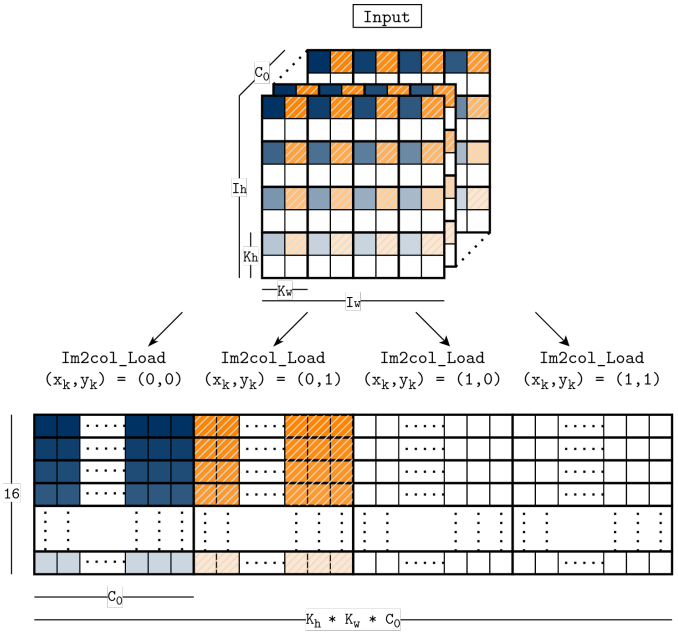


Fig. 5. Four `Im2Col` loads. The input is HWC_0. On the bottom, are the four resulting fractals of size $16 \times C_0$. The difference between the loads is the position relative to the patch (x_k, y_k), which is $(0, 0)$ for the first load (highlighted in blue) and $(0, 1)$ for the second (highlighted in orange), $(1, 0)$ for the third, and $(1, 1)$ for the fourth. The resulting fractals are concatenated in the output buffer.

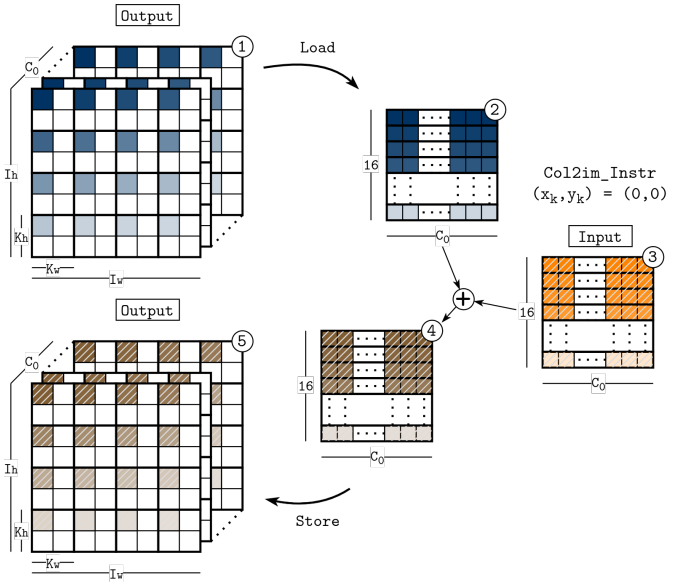


Fig. 6. Single Col2im load with parameters $(x, y) = (0, 0)$ and $(x_k, y_k) = (0, 0)$.

triple-nested loop with iterator vector in the form of $[(x, y), c1, (x_k, y_k)]$, from the outermost to the innermost loop.

As with most instructions in the DaVinci architecture, `Im2Col` supports a repetition parameter that causes an instruction to be reissued automatically. For `Im2Col` there are two possible repetition modes. Mode 0 repeats `Im2Col` for the next positions inside the kernel (x_k, y_k), from $(0, 0)$ to $(0, 1)$, for example. If the length of C_1 is bigger than 1, `Im2Col` in repetition mode 0 will continue to the next c_1 index and iterate over (x_k, y_k) again. This repetition mode acts as the loops of $[c1, (x_k, y_k)]$, but multiple `Im2Col` are needed to also change (x, y). Therefore, the input in Figure 5 can be fully loaded by issuing a single `Im2Col` starting at $(x_k, y_k) = (0, 0)$ with repeat mode 0 to repeat four times, changing (x_k, y_k) from $(0, 0)$ to $(0, 1)$, $(1, 0)$ and $(1, 1)$. Mode 1 reissues `Im2Col` for the next (x, y) position after skipping the 16 currently selected patches. In this mode, one `Im2Col` instruction acts as the loop of $[(x, y)]$, and multiple instructions are needed to change $c1$ and (x_k, y_k), thus, (x, y) becomes the innermost loop of the iterator vector. If the nesting order of these loops changes, so does the order in which fractals are stored in memory. By changing the order from $[(x, y), c1, (x_k, y_k)]$ to $[c1, (x_k, y_k), (x, y)]$ in mode 1, `Im2Col` will store fractals in a transposed order resulting in an output matrix of shape $(C_1 \times K_h \times K_w \times 16, (O_h \times O_w)/16 \times C_0)$. This shape can also be considered as a tensor of dimensions $(C_1, K_h, K_w, O_h, O_w, C_0)$, which is the shape used in the accelerated forward pooling implementation in Section V.

D. Col2Im Instruction

`Col2Im` is an instruction that is used as the backward operator of `Im2Col`. It acts as a vector instruction that loads data from and stores data to the `Unified Buffer` (8)→(8) (Figure 4). `Col2Im` takes fractals as inputs and stores them in the NC_1HWC_0 format. Because of this, `Col2Im` receives the same parameters as `Im2Col` referring to its output. Besides the change in memory layout, if two patches overlap in the output, input elements that refer to the same output position need to be summed. This sum is shown in Figure 2, but it is performed at an instruction level. For that, `Col2Im` requires its output to be initialized with zeros.

Figure 6 shows how a single `Col2Im` instruction works with an already initialized output. This example uses the same parameters as the first (blue) `Im2Col` shown in Figure 5. In Figure 6, `Col2Im` loads the initialized output (1) in an `Im2Col` manner (2). Then, it sums the loaded fractal with the input fractal (3). Finally, it stores the resulting fractal (4) back to its corresponding positions in the output (5). This example could not be loaded using a repetition because the only repetition mode available for `Col2Im` is mode 1. It works as in `Im2Col` by changing the (x, y) parameters and thus requires an input with more than 16 patches.

IV. Software Stack

A C-like language called CCE (Cube-based Compute Engine) C is used to write code for DaVinci chips. Because it is a very low-level language, implementing and optimizing multiple AI operators manually is a cumbersome and error-prone task. The Automatic Kernel Generator (AKG), a tool for operator design and also a library of operators, is used to enable the design of AI operators in CCE C. AKG uses TVM's [26] domain-specific language (DSL) to design its

operators, which are lowered to CCE C by its compiler passes. For every operator that is defined with AKG, its backward operator is also needed to allow training.

A. Scheduling for DaVinci

TVM's DSL is based on the Halide language [27]. The main idea of both languages is to decouple the execution definition (the algorithm) from the execution strategy (the algorithm's schedule). With this separation, the programmer is free to test multiple optimization strategies by rewriting a schedule without changing the algorithm. The schedule allows the use of techniques such as function inlining and loop transformations (*e.g.*, tiling, fusion, unrolling, and loop vectorization). The decoupling of the algorithm from its schedule is possible because Halide's and TVM's DSLs are tailored respectively for image processing and deep learning algorithms. There is a high degree of data parallelism in applications from these fields [27] as their algorithms are mainly composed of loops with no dependencies between iterations, known as DOALL loops [28]. In this scenario, the loop transformations previously mentioned are trivial.

TVM allows code generation for other backends besides CPUs. Hence, schedules can explicitly refer to a backend-specific construct. For example, schedules allow binding loops in the algorithm to blocks and threads, which are constructs found in GPUs. AKG uses the same principle to generate code for DaVinci devices. A DaVinci-specific schedule is responsible for controlling the movements of data between the scratch-pad buffers and for specifying computations that are local to a buffer. Together with the backend-specific schedule primitives, it is possible to apply other optimization techniques (*e.g.*, tiling) to improve the locality of memory accesses. Between all the possible primitives, two are handled automatically by AKG: vectorization and parallelization. First, the inner loops of computations are vectorized (minimally on the C_0 dimension) so that the Vector Unit is utilized automatically. When possible, the vector instructions are also issued with repeat factors. Second, the outer loops are parallelized between the AI Cores available on the target device. These default behaviors are similar to those taken by Halide's auto-scheduler [29]. AKG also has a polyhedral framework that automatically schedules computation on DaVinci, but it does not support all instructions (*e.g.*, `Col2Im`).

V. Im2col/Col2im Based Pooling

The Vector Unit computes pooling in DaVinci. The performance of vector instructions running in it depends mostly on two factors. First, the vector mask should be saturated so that all vector lanes are utilized and parallelism is maximized. Second, the repetition parameter should be employed, thus removing loops and barriers around vector instructions, and taking pressure off instruction fetching. Ideally, a single instruction should operate over an entire tensor (or tile) present in the `Unified` buffer. This Section describes the Im2col/Col2im based pooling implementations in comparison to their standard implementations in TVM. Lowered CCE C code is used to highlight the above-mentioned factors in each implementation.

```
1  input = placeholder((N, C1, Ih, Iw, C0),
2                      name="input")
3  red_h = reduce_axis((0, Kh), "red_h")
4  red_w = reduce_axis((0, Kw), "red_w")
5  output = compute((N, C1, Oh, Ow, C0),
6                   lambda n, c1, h, w, c0:
7                   max(input[n, c1,
8                             h*Sh+red_h,
9                             w*Sw+red_w,
10                            c0],
11                  axis=[red_h, red_w]))
```

Listing 1. `MaxPool` defined with TVM's DSL

A. Maxpool Forward

A standard TVM implementation of Maxpool forward is represented in Listing 1. It describes its input and output shapes (Lines 1 and 5, respectively) and its 2D `max` reduction of each patch (Lines 6 to 10). Using TVM's schedule, this computation is divided in the C_1 dimension so that a tile of size (I_h, I_w, C_0) is computed at a time, producing an output tile of size (O_h, O_w, C_0) unless further tiling is needed. Each input tile is sent from global memory to the `Unified` Buffer of an AI Core (1)→(8). If multiple AI Cores are available, multiple tiles can be processed in parallel. After the computation is finished, the output tile is brought back to global memory (8)→(1).

This implementation is lowered to CCE C code where the `vmax` instruction is executed. `vmax` computes the maximum between elements of the output and input tiles and writes back to the output tile. For that, the output tile is initialized with the minimum value of the data type in use. In this setting, only 16 of 128 elements of the vector mask are set, accounting for the innermost dimension C_0 of the tiles. Additionally, each `vmax` uses repetition to obtain the maximum value across the width of a patch K_w (the innermost reduction axis red_w). The `vmax` instruction is issued $O_h * O_w * K_h$ times to complete the computation. These suboptimal parameters result from the strided access pattern seen in Lines 8 and 9 of Listing 1.

The Im2col based implementation is described in Listing 2. Lines 1 and 3 represent the Im2col transformation. In this implementation, the input starts in the global memory with its original shape, which is tiled along the C_1 dimension. Next, the input is first loaded to the `L1` buffer of an AI Core (1)→(2) and then loaded with `Im2Col` to its `Unified` buffer using the repeat mode 1 (2)→(8). The transformed input has the shape shown in Line 3. Its tiles have a shape $(K_h, K_w, O_h, O_w, C_0)$ in the `Unified` buffer. The `max` reduction now occurs in the outer (K_h, K_w) dimensions, as shown in Lines 10 and 11. Considering the input and output tiles, $(K_h, K_w, O_h, O_w, C_0)$ and (O_h, O_w, C_0) respectively, the lowered CCE C code is able to set all 128 elements of the vector mask, and, in conjunction with the repetition parameter, a single `vmax` computes the `max` between the entire output tile and the three innermost dimensions of the input tile, which are identical. This instruction is only issued $K_h * K_w$

```
1  input = placeholder((N, C1, Ih, Iw, C0),
2                      name="input")
3  input-ub = placeholder((N, C1, Kh, Kw, Oh, Ow, C0),
4                      name="input-ub")
5  red_h = reduce_axis((0, Kh), "red_h")
6  red_w = reduce_axis((0, Kw), "red_w")
7  output = compute((N, C1, Oh, Ow, C0),
8                   lambda n, c1, h, w, c0:
9                   max(input-im2col[n, c1,
10                                  red_h,
11                                  red_w,
12                                  h, w, c0],
13                  axis=[red_h, red_w]))
```

Listing 2. `MaxPool` performed on an input with the resulting shape of using the `Im2Col` load

```
1  argmax-mask = placeholder((N, C1, Kh, Kw, Oh, Ow, C0))
2  gradients = placeholder((N, C1, Oh, Ow, C0))
3  mask-gradient = compute((N, C1, Kh, Kw, Oh, Ow, C0),
4                   lambda n, c1, kh, kw, oh, ow, c0:
5                   argmax-mask(b, c1, kh, kw, oh, ow, c0)
6                   * gradient(b, c1, oh, ow, c0)
7                   )
```

Listing 3. Part of `MaxPool` backward defined with TVM's DSL

times to finish the computation, effectively improving upon the standard implementations of Listing 1.

For training, it is useful to save an additional result in the forward implementation of `Maxpool`: the `argmax` mask. This mask is used by `Maxpool`'s backward operator to store the position of the maximum element of each patch, as shown in Figure 3. This result is obtained by comparing each patch of the input with its maximum value. Saving this mask is independent of the use of `Im2Col` instructions. Still, the `Im2Col` output shape of Line 3 in Listing 2 is used to store it, as it keeps overlapping patches separated. This shape also enables `Maxpool` backward to use `Col2Im` instructions, which is described next.

B. Maxpool Backward

`Maxpool` backward receives two inputs: the `argmax` mask and the incoming gradients. Listing 3 shows part of its implementation. Line 3 defines a computation that multiplies the patches in the `argmax` mask with their corresponding gradients. This multiplication is represented on the bottom of Figure 3. Next, the multiplied patches need to be merged back into the original (N, C_1, I_h, I_w, C_0) shape by summing values in the overlapping areas, which is not shown in Listing 3 for brevity. This merge step is critical for performance and it is depicted on the bottom-left of Figure 3. Its TVM implementation requires expanding `mask-gradient` to a shape of $(N, C_1, I_h, I_w, O_h, O_w, C_0)$, where each patch is copied only once in its correct position in I_h and I_w, and other elements are set to zero. The expanded representation is then reduced with `sum` on dimensions O_h and O_w, effectively summing up the overlapping areas in every patch and obtaining the final shape of (N, C_1, I_h, I_w, C_0). This expansion would be incredibly costly due to its size, however, TVM allows it to be inlined using a schedule. As a consequence, the patches are merged, and the overlapping regions are summed directly to the final output shape (from the shape $(N, C_1, K_h, K_w, O_h, O_w, C_0)$ to (N, C_1, I_h, I_w, C_0)). Besides the mentioned inlining, the schedule works similarly to `Maxpool` forward, loading both inputs from the global memory to the `Unified` buffer ①→⑧ so that the multiplication is computed in the Vector Unit ⑨, and tiling the computation on C_1.

The lowered code uses `vmul` for the multiplication step, and `vadd`, for the merge step. These instructions work in the same way as `vmax`, but for multiplication and addition. While `vmul` works well in multiplying tiles of the gradient with the mask, the scattered access pattern of the merge step leads to very poor usage of the Vector Unit. That is because the `vadd` instructions only set 16 elements of the vector mask (vectorizing on C_0) and repetition is not used.

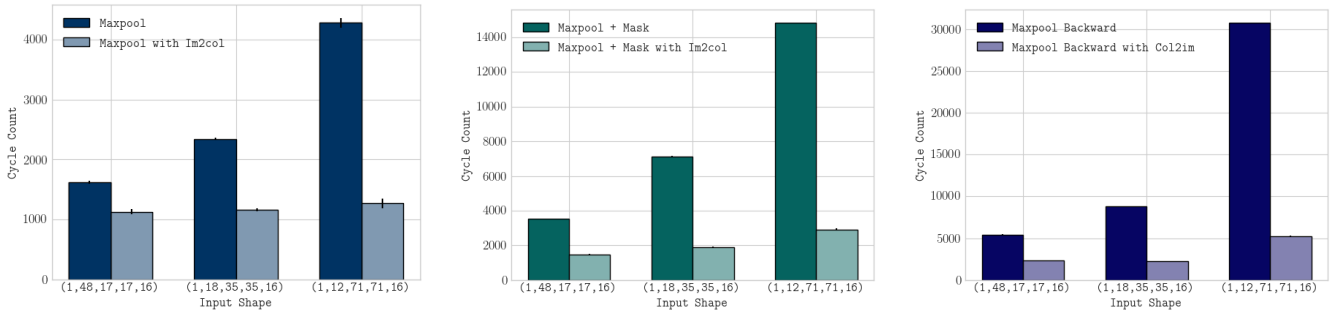
The Col2im based implementation comes from the observation that the merge step computes exactly the Col2im operation. As mentioned before, `Col2Im` uses the `Unified` buffer to load its input and to store its output ⑧→⑧. Therefore, it is possible to use `Col2Im` instead of `vadd`. The `Col2Im` instruction is able to load and store to the scattered elements of the output, summing two fractals at a time, as shown in Figure 6. In comparison with `vadd` that had 16 (C_0) elements of the vector mask set, `Col2Im` enables vectorization over $16 * 16$ elements (a fractal) at a time, and its repetition mode can be used to operate over the entire tile in the `Unified` buffer. A `Col2Im` instruction needs to be issued $K_h * K_w$ times to complete the merge step of a tile. Therefore, switching `vadd` for `Col2Im` presents a good opportunity for performance gains.

C. Avgpool

The forward and backward operators of `Avgpool` are similar to those described before. But opposed to `Maxpool`, the forward implementation reduces using `sum` instead of `max`. Consequently, its CCE C code uses `vadd` instead of `vmax`. But regardless of this change, the access pattern stays the same and can benefit from using `Im2Col`. Additionally, a new operation is needed to compute an element-wise division before saving the final output. As for the backward operator, there is no need to use the `Argmax` mask as an input. The equivalent mask for `Avgpool` contains 1 in all its positions, given that all input elements contribute to the output of a `sum`. Besides the mask, the backward implementation is the same and it can also use `Col2Im` instructions.

VI. Experimental Evaluation

This evaluation compares the performance of the Im2col/Col2im based `Maxpool` with the standard TVM `Maxpool` implementation described in Section V. All the experiments were run on an Ascend 910 chip, which contains 32 AI Cores. The cycle count numbers were obtained using the hardware counters of the chip, and they refer to the on-chip execution time running at a frequency of 100 MHz. The cycle count is currently the only metric that could be obtained from the chip. Each evaluation was repeated ten times, and the graphs show the average value and a 95% confidence interval. To use the `Im2Col` and `Col2Im` instructions in



(a) `Maxpool` (b) `Maxpool` and `Argmax` Mask (c) `Maxpool` Backward

Fig. 7. Comparison of `Maxpool` implementations with and without `Im2Col` and `Col2Im` instructions. The graphs show the cycle count in the Ascend 910 chip by the size of the input. The input sizes are from InceptionV3. All tests use a kernel size of (3,3) and a stride of (2,2) with no padding.

TABLE I
MAXPOOL INPUT SIZES IN CNNS

CNN	Input 1	Input 2	Input 3	Input 4
InceptionV3	147,147,64	**71,71,192**	**35,35,288**	**17,17,768**
Xception	147,147,128	74,74,256	37,37,728	19,19,1024
Resnet50	112,112,64	-	-	-
VGG16	224,224,64	112,112,128	56,56,256	28,28,512

TVM, they are declared and manually added to the code as custom intrinsics through TVM's `decl_tensor_intrin` function. These intrinsics act in TVM's DSL as an inline assembly section in a C source. Instead of implementing a single instruction call, the custom intrinsics were defined to issue instructions multiple times. By also using the repetition parameters, they can operate on a full tile of the input.

A. *InceptionV3 Comparison*

Table I shows multiple CNNs and the input sizes of four of their `Maxpool` layers. The inputs are shown in the HWC layout and they were gathered on the Keras framework [30]. All configurations use a kernel size of $(3, 3)$ and a stride of $(2, 2)$, except for VGG16 [31], which has a kernel size and stride of $(2, 2)$. To test the implementations of `Maxpool`, three configurations were selected from InceptionV3 [32] (highlighted in bold). No padding is used in them, however, it is also possible to add padding during the `Im2Col` load, as the other CNNs would require. Given AKG's current limited support for the `Im2Col` and `Col2Im`, these configurations were chosen to display the effects of different input sizes while using the most common parameters of kernel and stride.

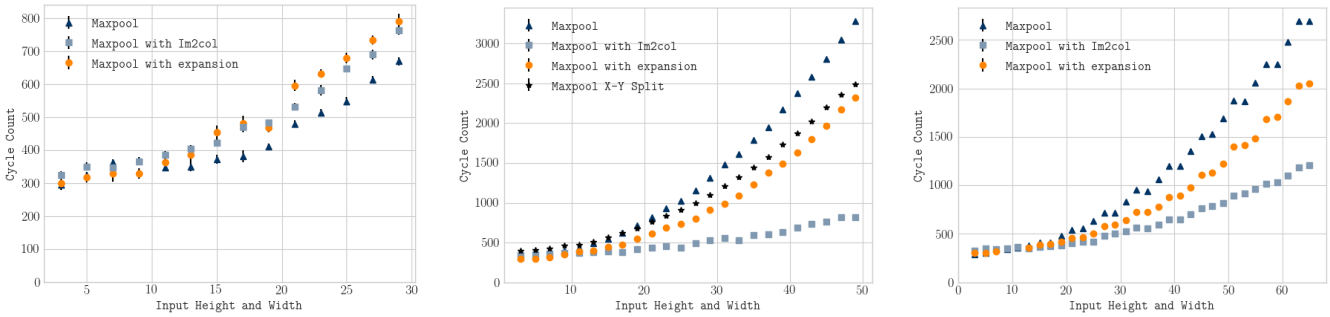
The graphs in Figure 7 show the cycle count of the selected `Maxpool` configurations in the NC_1HWC_0 layout. Figure 7a shows both `Maxpool` forward implementations. The step of saving the `Argmax` mask is added in Figure 7b. This step adds to the computation, as shown by the different ranges in the graphs. For the evaluation in Figure 7b, AKG's polyhedral framework schedules the computations, as it can better handle computations with multiple outputs of different shapes. Lastly, `Maxpool` backward is evaluated in Figure 7c. In the largest input, the accelerated implementations achieve speedups of 3.2x, 5x, and 5.8x on the graphs in Figure 7, respectively. The best improvement is on `Maxpool` backward. Its large speedup is expected, given the scattered access pattern of its merge step and how `Col2Im` can be used without any extra computations.

B. *Stride Tests*

This experiment investigates further different `Maxpool` forward implementations, and their interaction with the stride parameter, as shown in Figure 8. The stride size changes the amount of duplicated elements in Im2col. The kernel size was set at a constant size of $(3, 3)$. Given this kernel size, there is no duplication of data for the $(3, 3)$ stride, and the maximum duplication occurs for the $(1, 1)$ stride. In this experiment, `Maxpool` and `Maxpool with Im2col` are the same implementations shown in Figure 7a. The input's height and width increase in steps of two until the tiling threshold is reached, where this threshold is the maximum size before tiling is required. Bigger sizes would need individual tiling parameters and would trigger parallelization between AI Cores, which is out of the scope of this experiment. Moreover, dimensions N and C_1 are set to 1 so that only one AI Core is utilized.

In the `Maxpool with expansion` implementation, regular vector instructions — instead of `Im2Col` instructions — transform the input to the `Im2Col` output shape. This transformation happens when the input is already in the `Unified` buffer, before computing `Maxpool`. `Maxpool with Im2col` and `Maxpool with expansion` achieve superior performance in Figures 8b and 8c. These graphs confirm that the Im2col memory layout allows more efficient usage of the Vector Unit, producing speedups that compensate for the overhead of transforming the data. `Maxpool with Im2col` has the best performance in comparison to `Maxpool with expansion` due to Im2col occurring while the data is loaded into the `Unified` buffer, rather than in a separate step.

Figure 8a shows different results for a stride of $(1, 1)$. With this parameter, elements in consecutive patches of the original input appear consecutively in memory. This allows the `vmax` instruction to improve its use of the Vector Unit, combining the mask register set with all 128 elements and its repeat parameter



(a) `Stride = (1,1)` (b) Stride = (2,2) (c) Stride = (3,3)

Fig. 8. Comparison of different `Maxpool` implementations. The graphs show the cycle count in the Ascend 910 board and the height and width of the input. In all tests, the N and C_1 sizes are 1, kernel size is (3,3), with no padding. The x-axis goes up to the tiling threshold. An additional implementation of the X-Y split is shown for the stride of (2,2).

to compute the max between the (O_w, C_0) dimensions of the input and the initialized output. By also having no overhead to transform the data, and no data duplication, the direct `Maxpool` implementation is the fastest in this case.

Pooling can also be implemented with an X-Y split by first calculating the reduction function on the width and then on the height of each patch. As a result, the first reduction is reused while computing the second. Lai *et al.* [7] use the X-Y split as a performant alternative to direct pooling. In their work, the (undesirable) intermediate results are avoided by computing the result in-place. In TVM, all computations generate a new tensor, and thus the in-place approach is not possible. However, this experiment increases input sizes only until the tiling threshold is reached, therefore avoiding extra tiling steps needed because of the increase in memory use. Figure 8b shows the performance of a TVM version of the X-Y split (with intermediate results) compared to the other `Maxpool` implementations. Even though the X-Y split has a lower computational cost, it underperforms other implementations that use `Im2Col` because it does not overcome the scattered memory problems of pooling.

VII. Related Work

Convolutional layers have been the focus of extensive literature in optimizing CNN layers because they are responsible for most of the computation time of CNNs. Other layers such as pooling receive less attention, but when left unoptimized, they can be obstacles that lead to slowdowns in CNNs [1].

FPGA implementations for CNNs. In their implementation of CNN layers for OpenCL-based FPGA accelerators, Suda *et al.* propose to unroll pooling at the hardware level so that multiple outputs are computed in a single cycle [5]. However, their optimizer chooses an unrolling factor of 1 for the CNNs evaluated, which is equal to no unrolling. Given an (FPGA, CNN) pair, Sharma *et al.* automatically synthesize a CNN accelerator where the computation of pooling modules overlaps with convolution modules. This overlap is used to hide latency and to take advantage of the fact that a pooling layer usually follows convolutional layers [6]. Sharma *et al.* do not consider the backward operators used in training. In contrast to these pooling-specific hardware solutions, Im2col/Col2Im based pooling in DaVinci leverages a general-purpose vector unit and the `Im2Col` and `Col2Im` instructions, which are primarily designed for convolution. The improvements to the pooling layer afforded by Im2col/Col2im could be combined with fusion in DaVinci, but this is not yet supported.

Kernel acceleration for CNNs. LightNet is a Matlab-based framework for Deep Learning [33]. Its `Maxpool` implementation uses Im2col to transform pooled regions into vectors to benefit from vector instructions. Their proposition is similar to the Im2col based forward pooling, however, no performance results are presented to justify their implementation. CMSIS-NN is a collection of efficient neural network layers targeting IoT edge devices that uses X-Y splitting for pooling [7]. However, the results in Figure 8b show that the X-Y split is not the best alternative for DaVinci. Additionally, CMSIS-NN does not consider backward operators because its target edge devices only perform inference. The Im2col/Col2im based pooling accelerates both inference and training devices, as DaVinci edge chips also feature `Im2Col` instructions.

Li *et al.* use two optimizations for pooling [34]. First, the use of the $CHWN$ layout instead of $NCHW$ to prevent uncoalesced strided memory accesses caused by HW as the innermost dimensions. Second, the reduction of the off-chip memory requests by tuning the number of outputs calculated by each thread during pooling. The memory layout used in DaVinci (NC_1HWC_0) is a variant of the $NCHW$ layout. However, the Im2col-based pooling transforms this layout into $NC_1K_hK_wO_hO_wC_0$, where the accesses can also be performed consecutively in memory, thus resulting in the performance speedups shown in Section VI. The outer loops are automatically parallelized in DaVinci among the available AI Cores, where each core calculates a share of the output.

Suita *et al.* focus on fusing convolution with pooling in GPUs [8]. They only consider `Avgpool` because it can be mapped to convolution where the kernel's weights are equal to $1/(K_h * K_w)$, and then further fused with its preceding convolution. As a result, the Im2col transformation can also

be used to implement the fused convolution-pooling. However, CNNs tend to use `Maxpool`, which cannot be fused in the same way.

VIII. Conclusion

This work presents DaVinci's `Im2Col` and `Col2Im` instructions. It is shown that they can be used to implement not only convolution, targeting the Cube Unit, but also pooling, targeting memory layout improvements for the Vector Unit. Accelerated Im2col/Col2im based implementations are described for the forward and backward operators of `Maxpool` and `Avgpool`. An experimental evaluation was run on the Ascend 910 chip with the parameters and three input sizes used in InceptionV3. The results show speedups of up to 5.8x for the accelerated `Maxpool` implementations, compared to baselines that do not use the `Im2Col` and `Col2Im` instructions. Although the stride parameter can impact the Im2col and Col2im operations drastically, the proposed acceleration approach achieved improved performance for all but $(1, 1)$ stride. The Im2col/Col2im based pooling also proves superior to other strategies of optimization, such as the X-Y split. Further work could evaluate the proposed approach in other architectures, and also consider the fusion techniques described by Suita *et al*. [8] to execute `Avgpool` together with convolution as matrix multiplication in the Cube Unit.

References

[1] D. Li, X. Wang, and D. Kong, "DeepRebirth: accelerating deep neural network execution on mobile devices," in *AAAI Conference on Artificial Intelligence*, New Orleans, LA, USA, 2018, pp. 2322–2330.

[2] H. Liao, J. Tu, J. Xia, and X. Zhou, "DaVinci: A scalable architecture for neural network computing," in *Hot Chips Symposium (HCS)*, Cupertino, CA, USA, 2019, pp. 1–44.

[3] A. Khaled, A. F. Atiya, and A. H. Abdel-Gawad, "Applying fast matrix multiplication to neural networks," in *Symposium on Applied Computing (SAC)*, Brno, Czech Republic, 2020, p. 1034–1037.

[4] Y. Guan *et al.*, "FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, USA, 2017, pp. 152–159.

[5] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Field-Programmable Gate Arrays (FPGA)*, Monterey, California, USA, 2016, p. 16–25.

[6] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Microarchitecture (MICRO)*, Taipei, Taiwan, 2016, pp. 1–12.

[7] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: efficient neural network kernels for Arm Cortex-M CPUs," *CoRR*, vol. abs/1801.06601, 2018.

[8] S. Suita *et al.*, "Efficient convolution pooling on the gpu," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 222 – 229, 2020.

[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] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *Application-specific Systems, Architectures and Processors (ASAP)*, Seattle, WA, USA, 2017, pp. 19–24.

[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*, La Baule (France), 2006.

[12] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 BLAS performance optimization on loongson 3A processor," in *International Conference on Parallel and Distributed Systems (ICPADS)*, Singapore, 2012, p. 684–691.

[13] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, San Jose, CA, USA, 1998, p. 1–27.

[14] G. Guennebaud, B. Jacob *et al.* (2010) Eigen v3. [Online]. Available: http://eigen.tuxfamily.org

[15] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Representations by Back-Propagating Errors*. Cambridge, MA, USA: MIT Press, 1988, p. 696–699.

[16] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016, pp. 770–778.

[18] C. Szegedy *et al.*, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA, 2015, pp. 1–9.

[19] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI, USA, 2017, pp. 1800–1807.

[20] F. Chollet, *Deep Learning with Python*, 1st ed. Manning Publications Co., 2017.

[21] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 5595–5637, 2017.

[22] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *International Conference on Artificial Neural Networks (ICANN)*, Thessaloniki, Greece, 2010, p. 92–101.

[23] H. T. Kung and C. E. Leiserson, "Systolic arrays for (VLSI)." Carnegie-Mellon Univ Pittsburgh Pa Dept Of Computer Science, Tech. Rep., 1978.

[24] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*, Toronto, ON, Canada, 2017, p. 1–12.

[25] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[26] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, USA, 2018, p. 579–594.

[27] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Programming Language Design and Implementation (PLDI)*. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530.

[28] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, p. 452–471, 1991.

[29] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, 2016.

[30] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[31] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representation (ICLR)*, San Diego, CA, USA, 2015.

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

[33] C. Ye, C. Zhao, Y. Yang, C. Fermüller, and Y. Aloimonos, "LightNet: A versatile, standalone matlab-based environment for deep learning," in *Multimedia (MM)*, Amsterdam, The Netherlands, 2016, p. 1156–1159.

[34] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, Utah, 2016.