Universidade Estadual de Campinas
Instituto de Computação

Caio Salvador Rohwedder

Accelerated pooling: devising a hardware accelerated implementation based on Im2col and Col2im

Pooling acelerado: criando uma implementação acelerada em hardware baseada em Im2col e Col2im

CAMPINAS
2021

Caio Salvador Rohwedder

Accelerated pooling: devising a hardware accelerated implementation based on Im2col and Col2im

Pooling acelerado: criando uma implementação acelerada em hardware baseada em Im2col e Col2im

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo

Este exemplar corresponde à versão final da Dissertação defendida por Caio Salvador Rohwedder e orientada pelo Prof. Dr. Guido Costa Souza de Araújo.

CAMPINAS
2021

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

R636a

Rohwedder, Caio Salvador, 1996-
Accelerated pooling : devising a hardware accelerated implementation based on Im2col and Col2im / Caio Salvador Rohwedder. – Campinas, SP : [s.n.], 2021.

Orientador: Guido Costa Souza de Araújo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Redes neurais (Computação). 2. Arquitetura de computador. I. Araújo, Guido Costa Souza de, 1962-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Pooling acelerado : criando uma implementação acelerada em hardware baseada em Im2col e Col2im
Palavras-chave em inglês:
Neural networks (Computer science)
Computer architecture
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Sandra Eliza Fontes de Avila
Bruno de Carvalho Albertini
Data de defesa: 05-07-2021
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-0636-2809
- Currículo Lattes do autor: http://lattes.cnpq.br/2913450321072290

Universidade Estadual de Campinas
Instituto de Computação

Caio Salvador Rohwedder

Accelerated pooling: devising a hardware accelerated implementation based on Im2col and Col2im

Pooling acelerado: criando uma implementação acelerada em hardware baseada em Im2col e Col2im

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo
  IC/Unicamp
- Profa. Dra. Sandra Eliza Fontes de Avila
  IC/Unicamp
- Prof. Dr. Bruno de Carvalho Albertini
  EPUSP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 05 de julho de 2021

To my Mom and Dad,
For your love, support, and for giving me the priviledge of completing this degree.

Acknowledgements

I would like to thank my supervisor, Dr. Guido Araújo, for all his advice and support during, and also before, this master's degree. I would also like to thank Dr. Nelson Amaral, for all his help and guidance during my visit at the University of Alberta. It was an honor to work with you both, thank you again for all the opportunities presented to me during our time together.

I would like to thank Amy Wang and her team at Huawei Canada for working with me and leading me to produce this manuscript. Lastly, I thank Dr. João Carvalho, for helping and teaching me in so many steps of this work.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and in part by The Brazilian National Council for Scientific and Technological Development (CNPq), grant #130626/2019-8.

Resumo

Convolução é uma operação crucial para aplicações de *Deep Learning*. Como tal, ela tem sido o foco de muitos esforços de otimização neste domínio de aplicação. *Image-to-column* (Im2col) e *column-to-image* (Col2im) são transformações amplamente utilizadas para mapear convolução para multiplicação de matrizes. Essas transformações reorganizam as entradas da convolução para evitar seu padrão de acesso não contínuo à memória, fornecendo assim um arranjo de dados mais amigável para CPUs e GPUs. Em aceleradores de inteligência artificial (IA), essas transformações permitem que a convolução seja executada em unidades multiplicadoras de matriz. Implementadas em *software*, no entanto, elas impõem um aumento significativo em tempo de computação que deve ser compensado pelos ganhos de eficiência dos multiplicadores de matriz. DaVinci é uma arquitetura de acelerador de IA que apresenta instruções para otimizar Im2col e Col2im, reduzindo assim a sobrecarga de execução da convolução em seu multiplicador de matriz. Outra camada central para redes neurais convolucionais que apresenta um padrão de acesso semelhante à convolução é *pooling*. A execução de *pooling* é tipicamente direcionada para unidades de computação vetorial. Contudo, implementações baseadas nas transformações Im2col e Col2im podem ser utilizadas para melhorar a sua execução. Este trabalho explora o uso das instruções Im2col e Col2im do DaVinci para acelerar camadas de *pooling*. A abordagem proposta usa uma unidade de computação vetorial de propósito geral e instruções projetadas principalmente para convolução. Uma avaliação experimental revela que as implementações de *pooling* propostas podem produzir ganhos de velocidade de até 5,8 vezes em comparação com implementações base que não usam essas instruções especializadas. Os ganhos de velocidade são obtidos a partir de uma melhoria no arranjo de dados das entradas do *pooling*, pois esse arranjo leva à melhor vetorização de suas instruções.

Abstract

Convolution is an operation crucial to Deep Learning applications. As such, it has been the focus of many optimization efforts on this application domain. 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, thus lowering the overhead of executing convolution in its matrix-multiplier. Another core layer of convolutional neural networks that presents a similar memory access pattern to convolution is pooling. The execution of pooling is typically targeted to vector computational units. Nevertheless, implementations based on the Im2col and Col2im transformations can be leveraged to improve its execution. This work explores the use of Im2col and Col2im instructions of DaVinci to accelerate pooling layers. The proposed approach uses a general-purpose vector computational unit and instructions primarily designed for convolution. An experimental evaluation reveals that the proposed pooling implementations can yield up to 5.8x speedup compared to baseline implementations that do not use these specialized instructions. The speedups follow from an improved memory layout in the inputs of pooling, as this layout leads to better vectorization of its instructions.

List of Figures

List of Tables

List of Abbreviations

AD	Automatic Differentiation
AI	Artificial Intelligence
AKG	Auto Kernel Generator
BLAS	Basic Linear Algebra Subprograms
CCE	Cube-Based Compute Engine
CNN	Convolutional Neural Network
Col2im	Column-to-Image
CPU	Central Processing Unit
DDR	Double Data Rate
DL	Deep Learning
DSL	Domain-Specific Language
FPGA	Field Programmable Gate Arrays
GEMM	General Matrix-Matrix Multiplication
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
Im2col	Image-to-Column
ML	Machine Learning
MXU	Matrix-Multiplier Unit
OWA	Ordered Weighted Average
SCU	Storage Conversion Unit
TLB	Translation Lookaside Buffer
TPU	Tensor Processing Unit
TVM	Tensor Virtual Machine
UB	Unified Buffer

Contents

Chapter 1

Introduction

With the increasing adoption of convolutional neural networks (CNNs) in everyday applications, high-efficiency CNN execution has become fundamental. Convolution has been the main target of optimization because it is the most used and computationally expensive layer in CNNs architectures [36]. Targeting convolutions led to special computing units in GPUS (Tensor Cores) and hardware accelerators based on systolic arrays to be designed. This type of accelerator has become one of the most efficient ways to execute CNNs training and inference.

DaVinci [31] is a hardware 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 [23]. 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 [54] and Eigen [17] 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 close 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 a 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 of DaVinci at a low overhead.

For modern CNN architectures, however, solely optimizing convolutional and fully connected layers is not enough. Many other operations are required and they should be able to run in hardware accelerators such as DaVinci to avoid transferring computation between devices. Some of these operations are activation and loss functions, batch normalization, and pooling. Pooling layers are present in most CNNs to extract translation-invariant features and to perform subsampling of images. 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 [30].

Max-pooling has strided memory access patterns similar to convolution, however, the calculation of a maximum value cannot take place in a matrix-multiplier. Even so, its

implementation can leverage specialized `Im2Col` and `Col2Im` instructions. Thus, this work proposes two key ideas to accelerate pooling: to produce an improved data layout by applying `Im2Col` instructions to the input of forward pooling, and to apply `Col2Im` instructions to backward pooling instead of traditional vector instructions. Previous attempts to accelerate CNNs using FPGAs proposed pooling-specific instructions and computational units [16, 45]. In contrast, the proposed approach uses a general-purpose vector computational unit and instructions primarily designed for convolution. Earlier work on improving pooling also overlooked its backward implementation [16,27,42], which is essential for training. Many previous works on pooling focus on improving network accuracy and avoiding overfitting by improving the sampling quality of pooling. Stochastic Pooling [57] and Fractional Pooling [14] are examples of these works. However, such works are not as practical as Max-pooling and neglect execution performance. Lastly, operation fusion, which effectively improves pooling paired with convolution [42,46], is independent of the Im2col/Col2im based implementation presented in this work. Both optimizations can be applied in conjunction.

The main contributions of this work are:

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

The remaining chapters are organized as follows: foundational concepts for this work appear in Chapter 2. Chapter 3 presents an overview of the DaVinci architecture, focusing on its `Im2Col` and `Col2Im` instructions, and the software stack used to implement pooling operators for DaVinci through the TVM framework. Chapter 4 describes the Im2col and Col2im based pooling implementations showing details of their high-level and low-level code. Chapter 5 presents a performance comparison of implementations of pooling and discusses the results. Finally, Chapter 6 presents related works, and Chapter 7 concludes this work.

The work developed in this thesis has been submitted for publications as **Caio S. Rohwedder**, João P. L. de Carvalho, José Nelson Amaral, Guido Araújo, Giancarlo Colmenares, and Kai-Ting Amy Wang, "Pooling Acceleration in the Davinci Architecture Using Im2col and Col2im Instructions", *Proceedings of the Thirty-Fifth IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, in May 2021.

Chapter 2

Background

2.1 Deep Learning

Artificial intelligence (AI) is a computer science field that studies algorithms capable of learning and modeling tasks that commonly would require human intelligence to be done. Machine learning (ML) is a sub-field of AI that studies learning algorithms, such algorithms learn patterns in datasets. A neural network is a technique of ML that tries to mimic the processing patterns of human brains with neuron structures organized in layers to learn patterns. Deep neural networks are neural networks composed of many layers, they are capable of learning more complex patterns [3]. Convolution is a filtering operation used in image processing. Deep convolutional neural networks (CNNs) are networks that use the convolution operation as their main building block, to better process image inputs. These deep networks are part of the subfield of ML called Deep Learning.

Deep learning (DL) started to regain attention in 2012 when the ImageNet [40] classification competition was won by a group that used a deep CNN called AlexNet [24] that obtained error rates considerably better than the previous state-of-the-art models. Today, big tech companies, such as Google, Apple, Tesla, and others employ CNNs in many applications, from self-driving cars to smartphone cameras that can identify what they are seeing. CNNs have two distinct modes of operation: training and inference. Training is the act of tuning weights to minimize an error function, it takes a tremendous amount of data and it is usually computed on computer clusters [47]. Inference is the use of an already-trained network to produce an output. Usually, inference is the only focus of edge devices but there are areas such as Reinforcement Learning where inference and training are intertwined on edge devices. Regardless of the mode of operation, the increased use of CNNs highlights the need for hardware and software solutions to optimize them, and their main operation, convolution, is at the center of such solutions.

2.2 Convolution and Im2col

CNNs use convolutional layers to process images as their inputs. They are used because they explore the spatial nature of images and obtain a built-in invariance against small changes in images [28]. Convolutional layers also process images more efficiently as they

use shared weights, rather than having a weight for each pair of input and output elements. This allows CNNs to have pixels as inputs so that they learn to extract features themselves, rather than needing preprocessed features as inputs [28]. Such advantages result in convolutional layers being the main operation utilized to process images in neural networks.

Although numbers vary due to the utilized hardware and CNN architecture, several studies demonstrate that the convolutional layer is the most computationally expensive layer in a CNN. Park *et al.* [36] show that convolution accounts for at least 40% of computation in multiple CNNs during training in a GPU, at least 60% in most cases. Jung *et al.* [22] show similar results for training in a CPU, however, their work indicates that non-convolutional layers are gaining importance for newer CNN architectures. Li *et al.* [30] also show at least 40% of computation is spent on convolution during inference on different CNNs in mobile CPUs. But similar to the previous work, they point out that non-convolutional layers can be obstacles for mobile CPU execution time. CNNs can be composed of many operations besides convolution, such as batch normalization, pooling, fully connected layers, activation functions, concatenation, etc., but due to its computational impact, convolution is the main focus for optimization.

Convolutional layers work by repeatedly applying 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 stride parameters, 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 of the input 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 this work. One of the challenges in convolution is its memory access pattern because the kernel has a strided movement in the height and width of an input while accessing all of the input's channels. Such access pattern can lead to cache misses and poor vectorization in CPUs, and it can cause uncoalesced memory accesses in GPUs.

Convolution unrolling, also known as Image-to-Column (Im2col[1]), is a data transformation that allows the mapping of convolution into matrix-matrix multiplication [5]. This transformation, illustrated in Figure 2.1, 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 perform-

[1]The transformation of the input image can also be an image-to-row transformation if the multiplication is transposed $(AB)^T = B^T A^T$ [51]. The Im2col name will be used to refer to all variants of this transformation.

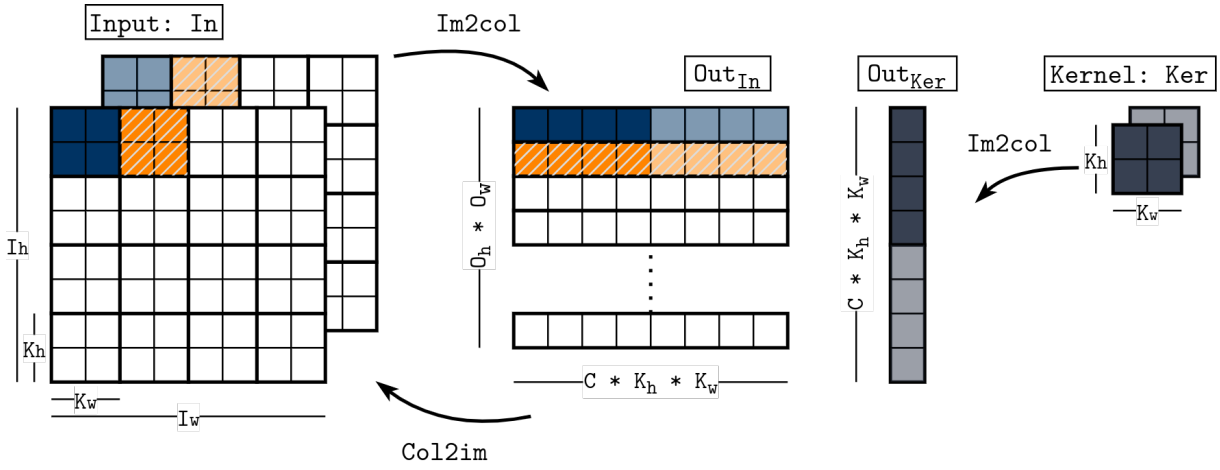


Figure 2.1: Im2col: In (C, I_h, I_w) is transformed into matrix Out_{In} $(O_h \times O_w, C \times K_h \times K_w)$ and a single kernel (C, K_h, K_w) is transformed into the matrix Out_{Ker} $(C \times K_h \times K_w, 1)$, where O_h and O_w represent the number of patches in the height and width of 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.

ing convolution with its original inputs. This is one of the main optimizations used to implement convolution.

The stride parameters S_h and S_w define how many elements must be moved in height and width after the kernel is applied on a patch of the input. First, the kernel will move S_h elements to the right until it reaches the end of the row, then, it moves S_h elements down and starts from left to right again. If the stride sizes (S_h, S_w) are smaller than the kernel's height and width (K_h, K_w), patches will overlap during convolution. 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 2.2. 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 [5]. Such advantages of matrix multiplication are discussed in more detail in the next section.

2.3 Matrix Multiplication

General matrix-matrix multiplication (GEMM) is the term used to refer to the matrix multiplication of matrices A and B resulting in matrix C in the form described in Equation 2.1, where α and $beta$ are constants. The most common form of matrix multiplication $(C = AB)$ is contained in this definition $(\alpha = 1,\ \beta = 0)$.

$$C = \alpha * AB + \beta * C \tag{2.1}$$

As seen in Listing 2.1, matrix multiplication can be implemented with ease, how-

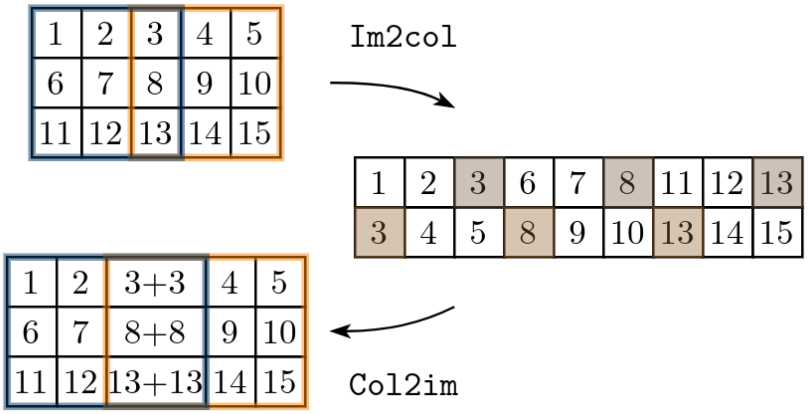


Figure 2.2: Im2col and Col2im performed on two overlapping patches. Im2col: patches overlap on the elements $3, 8, 13$, these elements are present in both rows of the output of Im2col. Col2im: Duplicated elements need to be summed when returning to the original shape.

```
1  for(i=0; i<M; i++)
2      for(j=0; j<N; j++)
3          for(k=0; k<P; k++)
4              C[i][j] += A[i][k]*B[k][i];
```

Listing 2.1: Naive matrix multiplication

ever, this is a naive implementation that misses on many opportunities for improvement. The structure of this computation is very flexible, it has loops with no dependencies between iterations, known as DOALL loops [53]. As a consequence, these loops can easily be interchanged and transformed to optimize execution targeting specific hardware. For cache-based CPUs, GEMM can obtain improved performance by using techniques such as blocking, optimal loop ordering, and packing. Loop ordering and blocking work together to break the input matrices into smaller chunks of data, and packing takes these chunks and reorders them in memory to optimize the memory accesses of the computation. Architecture-specific inner computations are used to operate on these chunks of data, they are often implemented in a lower-level language. Such optimizations improve cache hits, lower pressure on the Translation Lookaside Buffer (TLB), and allow architecture-specific tuning depending on cache size, number of cache levels, and vectorization features. These optimizations on GEMM have been extensively studied [13], and linear algebra libraries such as OpenBLAS [54], ATLAS [52], and Eigen [17] implement highly optimized versions of GEMM. These libraries are based on Basic Linear Algebra Subprograms (BLAS) [4], which is the most used standard for linear algebra routines in computer programs. GEMM is a key routine in these libraries because most other matrix-matrix routines can be implemented based on it with simple modifications [12].

Other types of hardware, such as GPUs also have highly optimized implementation of GEMM in libraries as cuBLAS [35]. Newer models of GPUs even have Tensor Cores [33] units that perform matrix-matrix multiply-accumulate of small matrices. As a conse-

quence of these hardware and software based optimizations, and given the possibility to map convolution to GEMM, GEMM is often used to implement convolution. The recent advances of AI accelerators that are designed around matrix-multiplier units further incentivize such a transformation.

2.4 AI Accelerators

Google's Tensor Processing Unit (TPU) [21] was one of the first examples of a new trend of specialized AI accelerators. The TPU was designed to meet the computational demands of Google's data centers, which were being overwhelmed with AI applications [20]. Many other companies built their own AI accelerators for their specific targets, such as Apple's Bionic and Amazon's Inferentia chips. The Ascend 910 and Ascend 310 chips are Huawei's AI accelerator designs based on the DaVinci architecture [31].

Both Google's and Huawei's AI accelerators were designed around a core processing unit based on a systolic array architecture that executes matrix and tensor operations. Such a unit is called Matrix-Multiplier Unit (MXU) on the TPU core and Cube Unit on the DaVinci architecture. A systolic array is an architecture designed to simultaneously exploit pipelined spatial/temporal parallelism and data reuse to achieve high throughput [26]. This architecture works by passing its input through an array of processing elements before storing it back to memory [25]. As defined by Kung, "In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart" [25]. The structure of this array can be multidimensional, thus enabling the processing of two-dimensional inputs, such as matrices. Both the MXU and the Cube Unit implement matrix multiplication by creating an array of processing elements capable of performing multiply-accumulate operations [21]. Convolutional and fully connected layers, which are core components of CNNs, can be easily implemented through matrix multiplication [23].

Even though the matrix-multiplier is the core of AI Accelerators, CNNs may require many other operations besides convolution. To avoid expensive offloading operations from a host to an accelerator, operations such as pooling, batch normalization, activation functions should also be supported by accelerators. Further, backpropagation is needed for training in an accelerator, and thus, backward operators should also be supported.

2.5 Backward Operators and Col2im

To train a neural network, the input values are first propagated in a forward pass 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. To propagate the gradient towards the input on a backward pass, every forward operator must have a dual-operator, namely its *backward operator* [39].

The gradient is a vector of partial derivatives that points to the steepest ascend on

the loss function. In order to minimize the loss, the tunable parameters of a network, its weights, are updated in the opposite direction of the gradient. This process is known as gradient descent [38]. The gradient descent uses the gradient of the loss function with respect to the weights. But the gradient of the loss function is initially calculated with respect to the output layer of the network. The backpropagation algorithm [39] is used to propagate the initial gradient backward, one layer at a time until it reaches all the weights. Reaching a weight x means that the gradient is computed with respect to x. Backpropagation explores the chain rule of derivatives to calculate and propagate the gradient to every layer. Backward operators are essentially the implementation of the partial derivatives to propagate the gradient from the outputs of one layer to its inputs.

The backward operator of Im2col is called Col2im, and it is also illustrated in Figure 2.1. 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 2.1, 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 2.2. Note that the term Col2im may be used to refer to a reshape operation that is applied to the resulting matrix of forward convolution implemented with GEMM. In this case, overlapping elements are not summed, however, in this work Col2im refers only to the backward operator of Im2Col.

2.6 Pooling Operators

Spatial feature pooling subsamples images to obtain translation-invariant feature maps in computer-vision architectures [11]. Similar to convolution, pooling is one of the building blocks of CNNs. Pooling layers are commonly used in modern CNN architectures such as Resnet [18], Inception [50], and Xception [7]. Pooling also applies a kernel over patches of its $NCHW$ input. But unlike convolution, a pooling 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 [9].

2.6.1 Maxpool

Figure 2.3 (on top) shows an example of `Maxpool` forward where 13 is the maximum value of the first patch, and 19 is the maximum of the second patch. This figure also shows an example of `Maxpool` backward on the same overlapping patches. The implementation of backward pooling depends on the reduction function. For `Maxpool` backward, each input is multiplied by its corresponding `Argmax` mask. In this mask, the position of the

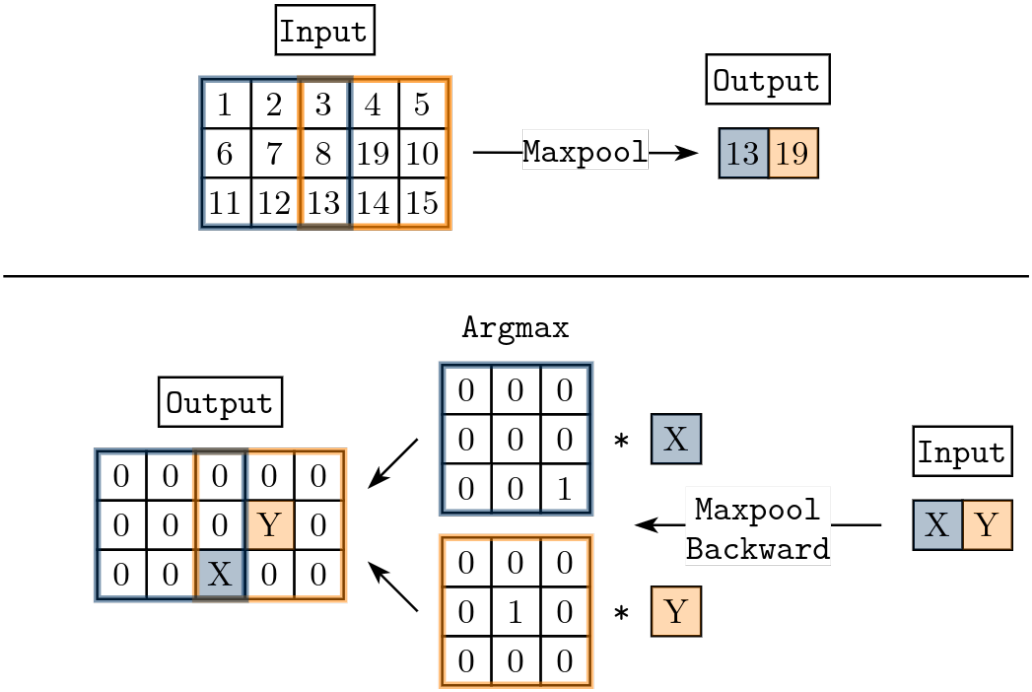


Figure 2.3: Forward and backward computation of `Maxpool` for two overlapping patches. Forward: the output is the maximum value of each patch. Backward: the gradients are propagated only to the position of the maximum value of their corresponding patch.

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 are returned to their original $NCHW$ shape and the overlapping elements are summed together. The output correlates how much a change in each input element of `Maxpool` forward affects its output elements [2]. In summary, the gradients are only propagated backward to the maximum elements, as they are the only elements that affect the output [41].

While Figure 2.3 shows the intuition of backward `Maxpool`, Figure 2.4 shows a deeper look into the working of this backpropagation step in a single patch. As shown in Figure 2.4, the inputs to `Maxpool` backward are the gradients with respect to the output of `Maxpool` forward ($\frac{\partial L}{\partial Y_j}$). The desired output of `Maxpool` backward is a matrix containing the gradients of the loss function with respect to the inputs of `Maxpool` forward ($\frac{\partial L}{\partial X_i}$). The chain rule $\frac{\partial L}{\partial X_j} = \frac{\partial Y_1}{\partial X_j} * \frac{\partial L}{\partial Y_1}$ (`Argmax` multiplied by the input) is used to obtain the output, where Y is the max function [39]. Equations 2.3 and 2.4 show the partial derivatives of max based on its definition in Equation 2.2. These equations can be applied to calculate $\frac{\partial Y_1}{\partial X_j}$ in the same way, the only difference is that `Maxpool`'s `max` takes multiple inputs. The results for these derivatives are shown below the `Argmax` matrix in Figure 2.4.

$$max(x, y) = \begin{cases} x, & \text{if } x \geq y \\ y, & \text{if } x < y \end{cases} \tag{2.2}$$

$$\frac{\partial max(x, y)}{\partial x} = \begin{cases} 1, & \text{if } x > y \\ 0, & \text{if } x < y \end{cases} \tag{2.3}$$

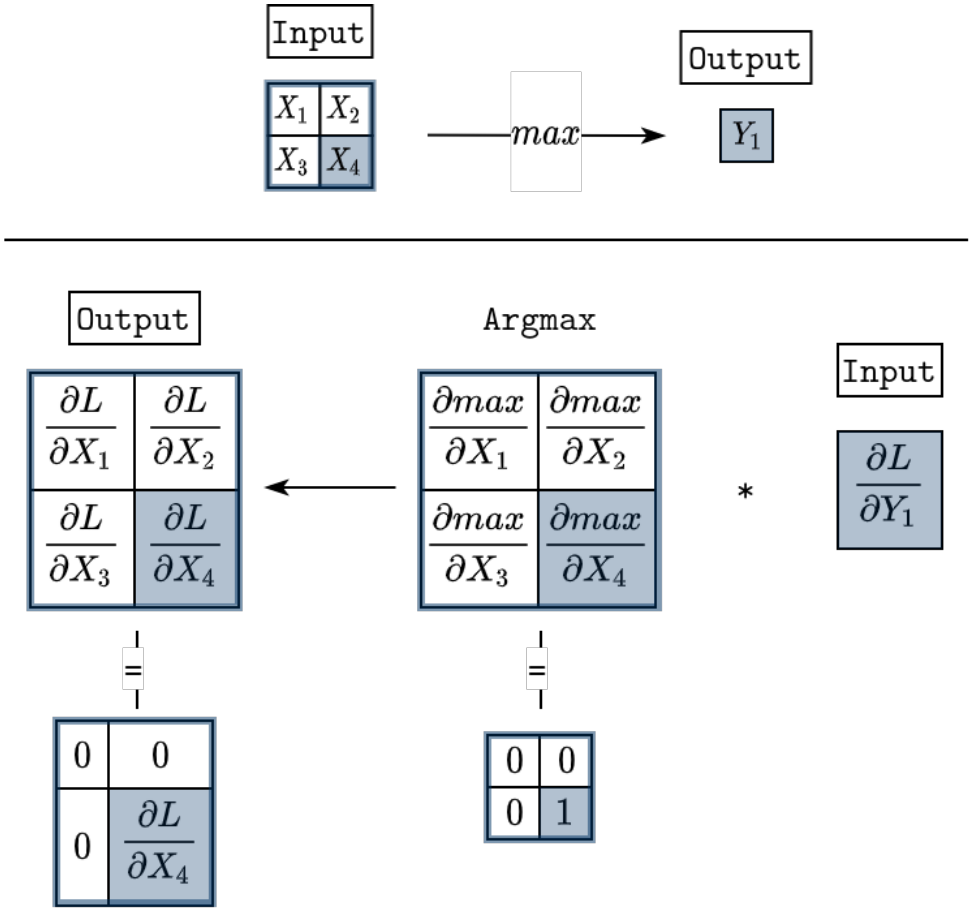


Figure 2.4: Forward and backward computation of a single patch where X_4 is the maximum element. The backward pass shows its derivatives and their resulting values below them.

$$\frac{\partial max(x,y)}{\partial y} = \begin{cases} 0, & \text{if } x > y \\ 1, & \text{if } x < y \end{cases} \tag{2.4}$$

The derivatives of `max` (Equation 2.2) are undefined when $x = y$. Equation 2.5 demonstrates that the left and right one-sided limits of the derivatives have different values when calculating the derivative of Equation 2.3. This is also the case for `Maxpool` when there are multiple maximum values in a single patch. In the case of an undefined derivative, it is up to the system designer to define a reasonable behavior. Most popular AI frameworks assign 1 to the first maximum in a patch and 0 to the others, regardless if they are also maximum values. Two other implementations are possible: (i) assign 1 to all maximum values in a patch and (ii) assign $1/m$ to all m maximum values in a patch (*e.g.*, 0.5 for 2 maximum values). There are no studies, as of the time of writing, that compare these different implementations and their impact on a network's performance. The `Maxpool` implementations in this work assign 1 to all maximum values in a patch as this is the simplest option and it is enough for the goal of the comparisons presented in Chapter 5. The other options would need an extra computation to identify the position of the first maximum value or to count how many maximum values exits in a patch.

$$\begin{aligned} \lim_{h \to 0^+} \frac{max(a, a+h) - max(a, a)}{h} &= 1 \\ \lim_{h \to 0^-} \frac{max(a, a+h) - max(a, a)}{h} &= 0 \end{aligned} \tag{2.5}$$

2.6.2 Avgpool

For `Avgpool`, the forward computation can be divided into two steps. First, a summation of the elements of each patch. Second, the division of the results by the patch size (number of elements in a patch). `Avgpool` backward starts by propagating the division step, which requires dividing all input gradients by the patch size. This is a trivial case because the division of a patch of size 2 for example, $f(x) = 0.5 * x$, has a derivative of 0.5. Hence, the constant value is multiplied by the gradient to complete the chain rule. The summation step is similar to `Maxpool` but changing the `max` function for a `sum`. Equation 2.6 defines the `sum` function of two values and Equations 2.7 and 2.8 define its partial derivatives. As opposed to `max`, in which only the maximum element contributes to the output, `sum` utilizes all of its input elements, thus, it propagates back to all elements. Thus, the same scheme in Figure 2.4 can be used, however, the masks for `sum` have the value 1 in all its positions.

$$sum(x, y) = x + y \tag{2.6}$$

$$\frac{\partial sum(x, y)}{\partial x} = 1 \tag{2.7}$$

$$\frac{\partial sum(x, y)}{\partial y} = 1 \tag{2.8}$$

2.6.3 Global Pooling

Another type of pooling present in the CNNs discussed in this work is Global Pooling [32], which is often implemented as Global Average Pooling. Table 2.1 shows a few CNNs and their use of this type of pooling with its input sizes. The implementation does not change from `Avgpool`, however, Global Average Pooling averages the whole height and width dimensions to a single value, that is, every channel has only one patch. This layer was proposed to substitute the fully connected layers at the end of CNNs so that the resulting vector is fed directly to the softmax layer. It can be advantageous to make this substitution because Global Pooling has no trainable parameters and thus has no possibility of overfitting. Another good outcome is that a relation is enforced between the final features of the network and classification categories. This type of pooling has a much simpler implementation than traditional Pooling because there is no strided reduction function, but only a 2-dimensional single reduction.

Table 2.1: Global Avgpool Input Sizes in CNNs (height, width, channels).

CNN	Input 1
InceptionV3	8,8,2048
Xception	10,10,2048
Resnet50	7,7,2048
VGG16	-

Chapter 3

The DaVinci Architecture

DaVinci [31] is an AI accelerator architecture used by Huawei's Ascend chips. Its main computational component, the AI Core, has a scalable design that tailors resources such as memory size, matrix-multiplier throughput, and the interconnection bandwidth to meet the requirements of multiple application domains, from wearable devices to DL training clusters. The following sections describe components of the Ascend 910 chip along with its software stack.

3.1 AI Core

Figure 3.1 shows a closer view of DaVinci's main component, the AI Core, and its corresponding datapaths. The AI Core is 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 each 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 [26], 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 [21]. 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 [19]. Data movement between these buffers must be

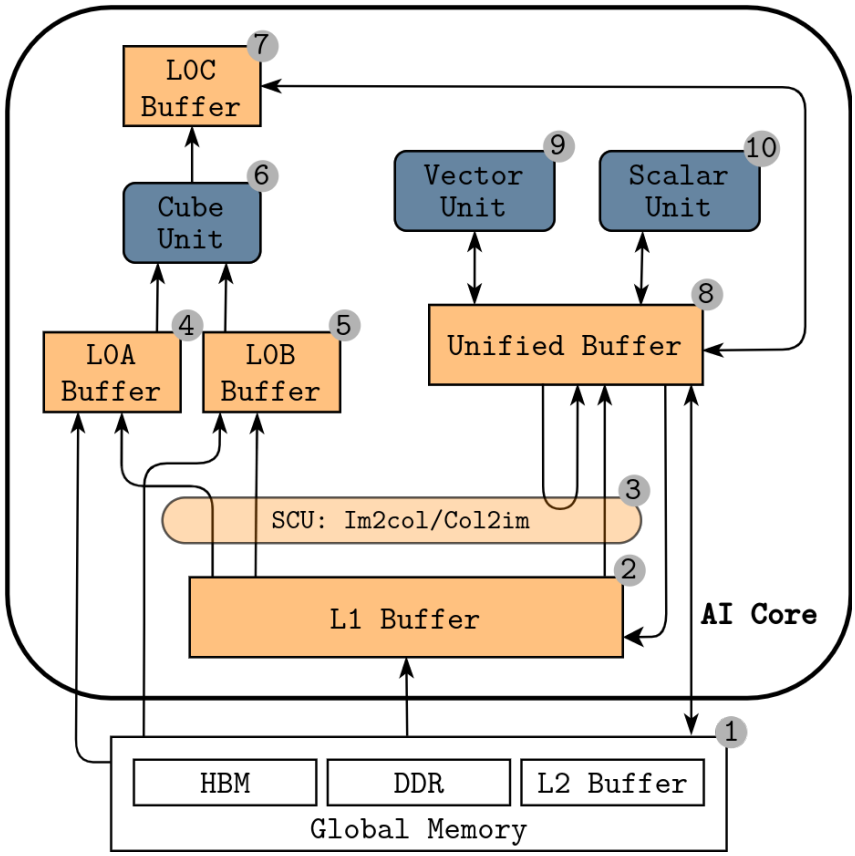


Figure 3.1: Data paths of the AI Core.

explicitly managed by the application, and thus, the programmer needs to specify which data should be brought to each buffer, and also needs to maintain data consistency. In contrast, hardware-managed caches are transparent to the application and ensure consistency by hardware protocols. 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 3.1. Given that their datapaths 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 introduce appears only on the target buffers. Such instructions were specifically designed to operate on the memory layout described next.

3.2 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, C_0 has a length of:

- 16, for `Float16`
- 32, for `Unsigned8`

The data type `Float16` is adopted in this work.

3.3 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 2.1. 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 2.1. The advantage of performing Im2col during a 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 3.1. 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).

$$\begin{aligned} O_h &= \left\lfloor \frac{I_h + P_b + P_t - K_h}{S_h} \right\rfloor + 1 \\ O_w &= \left\lfloor \frac{I_w + P_l + P_r - K_w}{S_w} \right\rfloor + 1 \end{aligned} \tag{3.1}$$

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 3.2 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 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 3.2 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

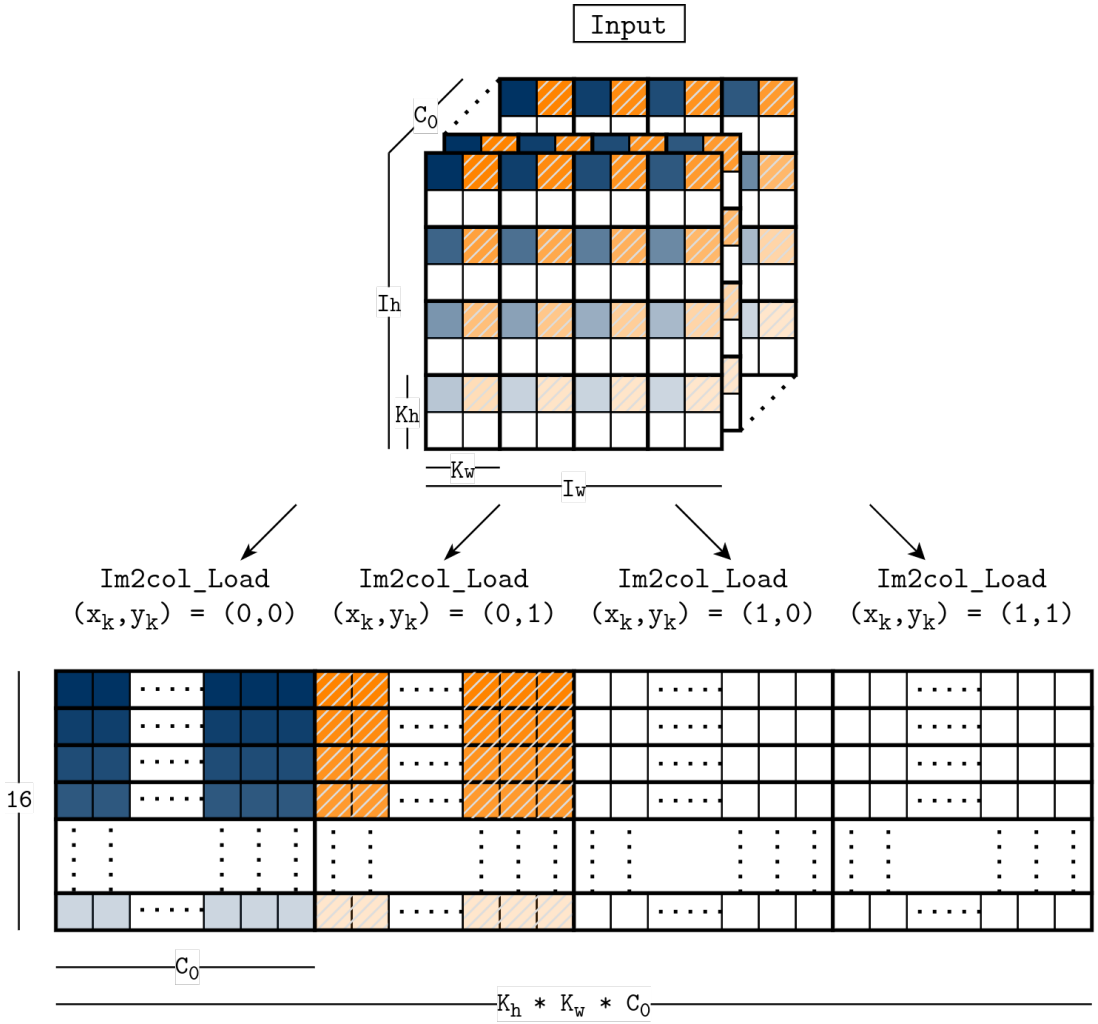


Figure 3.2: Four `Im2Col` loads. The input is HWC_0. At the bottom, are the four resulting fractals of size $16 \times C_0$. The difference between the loads is the position relative to the patch (x_k, y_k), which is $(0, 0)$ for the first load (highlighted in blue) and $(0, 1)$ for the second (highlighted in orange), $(1, 0)$ for the third, and $(1, 1)$ for the fourth. The resulting fractals are concatenated in the output buffer.

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 Chapter 4.

3.4 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 3.1). `Col2Im` takes fractals as inputs and stores them in the NC_1HWC_0 format.

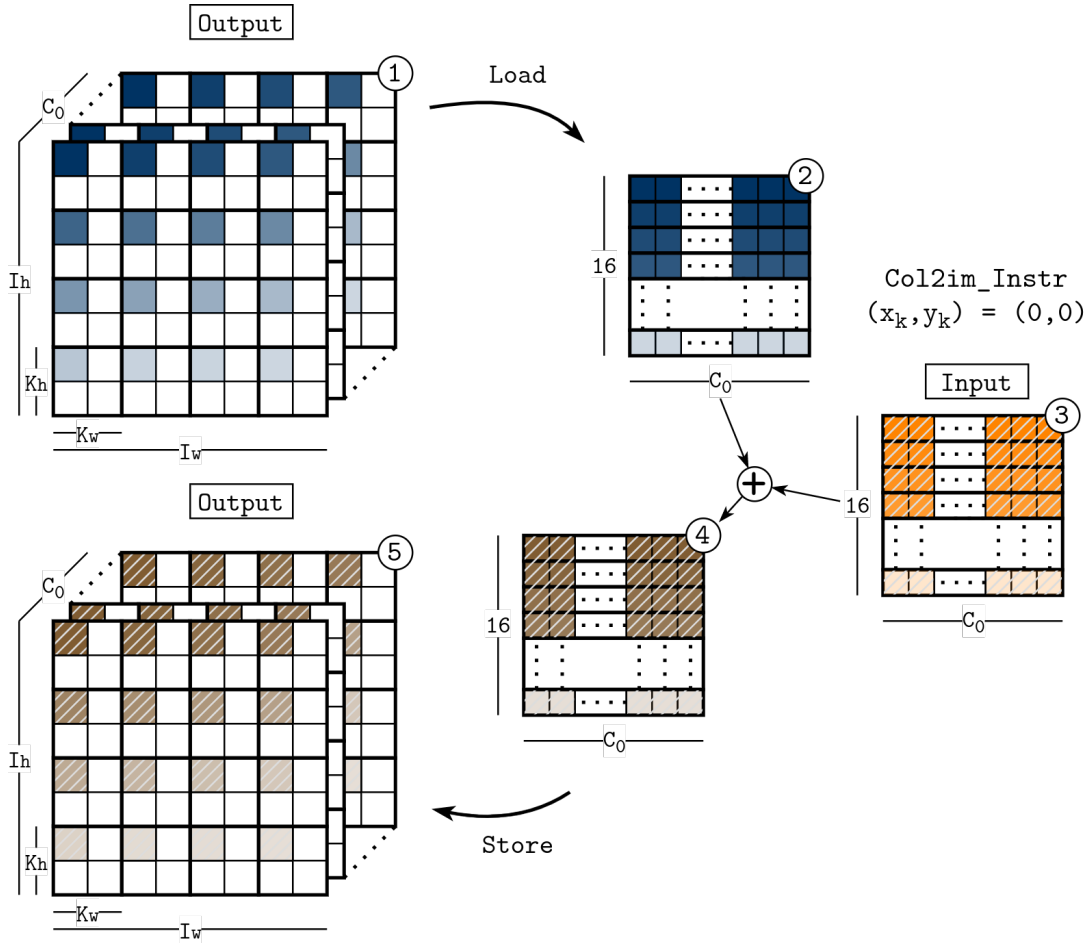


Figure 3.3: Single Col2im load with parameters $(x, y) = (0, 0)$ and $(x_k, y_k) = (0, 0)$.

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 2.2, but it is performed at an instruction level. For that, `Col2Im` requires its output to be initialized with zeros.

Figure 3.3 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 3.2. In Figure 3.3, `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.

3.5 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, enables code generation for DaVinci. AKG uses TVM's [6] 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.

3.5.1 Scheduling for DaVinci

TVM's DSL is based on the Halide language [37]. 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 [37] as their algorithms are mainly composed of DOALL loops. 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 [34]. AKG also has a polyhedral framework that automatically schedules computation on DaVinci, but it does not support all instructions (*e.g.*, `Col2Im`).

Listing 3.1 shows an example of TVM's DSL computing the sum of two tensors and scaling them by a factor of two. The variables `m` and `n` define the shape of the tensors `A` and `B`, which is `(m,n)`. Lines 3 and 4 define the tensors `A` and `B`. Line 5 defines a new tensor `C` with the same shape of `A` that will store the result of a computation. This computation is defined by a lambda function in Line 6, where every `C[i,j]` in `C` should be calculated as a sum of `A[i,j]` and `B[i,j]`. Line 8 defines a second computation that scales every element of C by a factor of 2 and stores the result in `D`. Listing 3.2 exemplifies a schedule that allows code generation of the algorithm defined in Listing 3.1 for a DaVinci device. Inputs and outputs start in the global memory. In Listing 3.2, Lines 2 and 3 create a copy of the tensors `A` and `B` in the `Unified` Buffer ("local.UB"), these local copies are inputs to

```
1  m = var("m")
2  n = var("n")
3  A = placeholder((m,n), name='A')
4  B = placeholder((m,n), name='B')
5  C = compute(A.shape,
6              lambda i,j: A[i,j] + B[i,j],
7              name="C")
8  D = compute(C.shape,
9              lambda i,j: 2 * C[i,j],
10             name="D")
```

Listing 3.1: Example of computation with TVM's DSL

```
1  s = create_schedule(D.op)
2  A_ub = s.cache_read(A, "local.UB", [C])
3  B_ub = s.cache_read(B, "local.UB", [C])
4  s[C].set_scope("local.UB")
5  D_ub = s.cache_write(D, "local.UB")
```

Listing 3.2: Example of schedule for the DaVinci architecture

the computation of tensor `C`. Line 4 defines that tensor `C` is created as a temporary in the `Unified` Buffer. Lastly, Line 5 has two effects, first, it defines that tensor `D` is also local to the `Unified` Buffer, and second, it stores `D` back to the global memory. Looking at Figure 3.1, the inputs are loaded from the global memory ①→⑧, the computation takes place mostly in the Vector Unit ⑩, which loads data from the `Unified` Buffer ⑧→⑨. After the output is computed, it is stored back to the global memory ⑧→①.

3.5.2 Backward Operators for DaVinci

As previously mentioned, backward operators are needed to train a neural network. AKG has an Automatic differentiation (AD) module that works as a source-to-source compiler from an operator definition to its backward operator, both in TVM's DSL. AD is also known as algorithmic differentiation [15], it is a set of techniques that can be applied to functions to obtain their derivatives. A computation, like the one shown in Listing 3.1, is received as input to the AD module and its backward operator is the output. AD allows new operators to be added to Mindspore without requiring their manually implemented backward counterpart. However, manually writing backward computations may be necessary for unsupported operators or to improve their performance. Such is the case of the backward operators described in the next Chapter.

Chapter 4

Im2col/Col2im Based Pooling

This Chapter describes the Im2col/Col2im based pooling implementations in comparison to their standard implementations in TVM. To use the `Im2Col` and `Col2Im` instructions in TVM, they are declared and manually added to the code as custom intrinsics through TVM's `decl_tensor_intrin` function and appear in the following listings as `im2col()` and `col2im()`. 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 and to use repetition parameters, thus operating on a full tile of the input. The Listings in this chapter show a simplified version of the implementations used in Chapter 5 for better readability, they start with the computation in the top part and are followed by their schedule.

The Vector Unit computes pooling in DaVinci, its utilization by vector instructions is key for the following implementations. 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. Lowered CCE C code is used to highlight the above-mentioned factors in each implementation presented.

4.1 Maxpool Forward

A standard TVM implementation of Maxpool forward is represented in Listing 4.1. The input and output tensors and their shapes ((N, C_1, I_h, I_w, C_0) and (N, C_1, O_h, O_w, C_0)) are defined in Lines 4 and 11, respectively. Next, the reduction axes are defined in Lines 7 and 8, together, they range from $(0, 0)$ to the size of a patch (K_h, K_w). Finally, Lines 12 to 17 defines `Maxpool` by computing each output element as a `max` reduction of a patch of the input. In each reduction, the patches of the input are accessed (in height and width) by their base address $(h * S_h, w * S_w)$, which is the height and width position of the output multiplied by the stride of the operation. To access all elements within a reduction, this base address is then summed to `red_h` (from 0 to K_h) in height, as well as `red_w` in width (from 0 to K_w). This computation simply defines `Maxpool` as it was shown in Chapter 2.

```
1  # Computation -------------
2
3  # input shape
4  input = placeholder((N, C_1, I_h, I_w, C_0))
5
6  # max reduction dimensions
7  red_h = reduce_axis((0, K_h), "patch_height")
8  red_w = reduce_axis((0, K_w), "patch_width")
9
10 # maxpool computation
11 output = compute((N, C_1, O_h, O_w, C_0),
12             lambda n, c1, h, w, c0:
13               max(input[n, c1,
14                         h*S_h+red_h,
15                         w*S_w+red_w,
16                         c0],
17                   axis=[red_h, red_w])
18           )
19
20 # Schedule ----------------
21
22 # input: global memory -> unified buffer
23 input_ub = cache_read(input, "local.UB", [output])
24
25 # output: unified buffer -> global memory
26 output_ub = cache_write(output, "local.UB")
27
28 # get computation axes and reorder them
29 b, c1, h, w, c0 = output_ub.op.axis
30 kh, kw = output_ub.op.reduce_axis
31 s[output_ub].reorder(b, c1, kh, kw, h, w, c0)
32
33 # tile computation in c1
34 b, c1, h, w, c0 = output.op.axis
35 s[output_ub].compute_at(s[output], c1)
36 s[input_ub].compute_at(s[output], c1)
```

Listing 4.1: `Maxpool` in TVM's DSL

The schedule section of code in Listing 4.1 allows code generation for DaVinci by issuing the necessary memory movement and the loop reordering of the `Maxpool` computation. It also tiles the computation so that the input and output can fit in the `Unified` Buffer, and achieve parallel execution on multiple AI Cores. As described in the previous chapter, the directives `cache_read` and `cache_write` move the input and output between memory buffers in the schedule. In Lines 23 and 26, these directives move the input from global memory to the `Unified` Buffer of an AI Core, and they move the output computed in the `Unified` Buffer back to global memory. Lines 29 to 31 get a reference to all the axes of the `Maxpool` computation in the `Unified` Buffer (its loops) and reorder them from the default ordering `(b, c1, h, w, c0, kh, kw)` to `(b, c1, kh, kw, h, w, c0)`, from outermost to innermost axis. This ordering is necessary to allow CCE C code generation for DaVinci, as `c0` has to be the last dimension. It also yields the best vectorization for this computation, which will be discussed later by mentioning the gen-

erated CCE C code. Lines 34 to 36 tile the computation. Again, this code section takes a reference to the computation axes, but now from `output` instead of `output_ub`. The variable `output_ub` is a reference to the `Maxpool` computation in the Vector Unit that utilizes the `Unified` Buffer, and `output` is a reference to the memory movement operation that stores the results in the Global Memory. In the same way, `input_ub` is a reference to the memory movement that loads the input to the `Unified` Buffer. The last two lines of the schedule tile the computation by directing it to occur inside of the `c1` axis of `output` with the `compute_at` directive. In other words, an input tile of size (I_h, I_w, C_0) is loaded, `Maxpool` is computed and a tile of size (O_h, O_w, C_0) is stored at a time. The two outer axes (N, C_1) are automatically parallelized if multiple AI Cores are available. This schedule accommodates smaller inputs, large inputs may need further tiling of the height and width dimensions to fit the (I_h, I_w, C_0) tile the `Unified` Buffer.

Note that as AKG utilizes an auto-vectorizer instead of a vectorization directive in the schedule, later vectorization may modify the axes of a computation. The reordering in Line 31 is utilized to allow the compiler to vectorize the computation in concordance with the rest of the schedule. This implementation is lowered to CCE C code by AKG's compiler where, among other instructions, `vmax` is executed. The `vmax` instruction 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 the `vmax` execution of this computation, 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 K_w of a patch (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 14 and 15 of Listing 4.1.

The Im2col based implementation is described in Listing 4.2, where the main differences compared to the previous implementation are highlighted. It has an extra load of the input tiles with `Im2Col` instruction through the `im2col` intrinsic. The schedule has different memory movements, as `Im2Col` instructions require their input to be in the L1 Buffer. The last change happens on the `Maxpool` computation, it operates on the input transformed by `Im2Col` instructions. In Listing 4.2 `input` is loaded by the schedule in Line 32 to the L1 Buffer. This allows the computation in Line 10 that generates `input_im2col` by loading `input` from the L1 Buffer to the `Unified` Buffer through `im2col` intrinsic calls. This intrinsic utilizes `Im2Col` instructions with repeat mode 1, resulting in the shape $(N, C_1, K_h, K_w, O_h, O_w, C_0)$ shown in Line 7. The `Maxpool` computation operates on this shape and the `max` reduction now occurs in the (K_h, K_w) dimensions, as shown in Lines 23 and 24. The schedule also does the same reordering operation for `output_ub` and tiling for the whole computation in the C_1 dimension of `output`, as in the previous implementations. The tiles loaded by `Im2Col` instructions have a resulting shape of $(K_h, K_w, O_h, O_w, C_0)$ in the `Unified` buffer. 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 $K_h * K_w$

```
# Computation -------------

# input shape at global memory and L1 buffer
input = placeholder((N, C_1, I_h, I_w, C_0))

# input shape at unified buffer
im2col_shape = (N, C_1, K_h, K_w, O_h, O_w, C_0)

# im2col load intrinsic
input_im2col = compute(im2col_shape,
                  lambda n, c1:
                    im2col(input[n, c1, :, :, :])
                )

# max reduction dimensions
red_h = reduce_axis((0, K_h), "patch_height")
red_w = reduce_axis((0, K_w), "patch_width")

# maxpool on im2col shape
output = compute((N, C_1, O_h, O_w, C_0),
            lambda n, c1, h, w, c0:
              max(input_im2col[n, c1,
                               red_h,
                               red_w,
                               h, w, c0],
                  axis=[red_h, red_w])
          )

# Schedule ----------------

# input:  global memory -> L1 buffer
input_l1 = cache_read(input, "local.L1", [input_im2col])

# output: unified buffer -> global memory
output_ub = cache_write(output, "local.UB")

# set as local to unified buffer
s[input_im2col].set_scope("local.UB")

# get computation axes and reorder them
b, c1, oh, ow, c0 = output_ub.op.axis
kh, kw = output_ub.op.reduce_axis
s[output_ub].reorder(b, c1, kh, kw, oh, ow, c0)

# tile computation in c1
b, c1, h, w, c0 = output.op.axis
s[data_im2col].compute_at(s[output], c1)
s[data_l1].compute_at(s[output], c1)
s[output_ub].compute_at(s[output], c1)
```

Listing 4.2: `Maxpool` with `Im2Col` in TVM's DSL

times to finish the computation, effectively improving upon the standard implementations of Listing 4.1.

For training, it is useful to save an additional result in the forward implementation

of `Maxpool` to avoid redundant computations: the `Argmax` mask. This mask is used by `Maxpool`'s backward operator as it stores the position of the maximum element of each patch (shown in Figure 2.3). 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 7 in Listing 4.2 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.

4.2 Maxpool Backward

`Maxpool` backward receives two inputs: the `Argmax` mask and the incoming gradients. Listing 4.3 shows its implementation. The inputs are first initialized, then, Line 12 defines a computation that multiplies the patches in the `Argmax` mask with their corresponding gradients. This multiplication is represented at the bottom of Figure 2.3. Having the `Argmax` mask as an input simplifies this computation as the only other step needed is to merge the multiplied patches back to the original (N, C_1, I_h, I_w, C_0) shape by summing values in the overlapping areas. Critical for performance, this merge step is depicted on the bottom-left of Figure 2.3. The Col2im based implementation comes from the observation that this merge step computes exactly the Col2im operation, details on the implementation without Col2im are explained later. In Listing 4.3 Lines 19 to 22 show this computation using the `col2im` intrinsic. All instructions in this implementation use the `Unified` Buffer including `Col2Im`, thus, the schedule section loads both inputs from the global memory to the `Unified` buffer and writes the result back to global memory. The computation is also tiled on the C_1 dimension. For brevity, only the Col2im-based implementation is shown.

To implement this operation without `Col2Im` instructions, TVM 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 then must be 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 by the schedule, effectively bypassing the expansion while retaining the `sum` reduction operation. 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 for the implementation without `Col2Im` instructions, the schedule is the same for both implementations.

The lowered code uses `vmul` for the multiplication step and depending on the implementation, `vadd` or `Col2Im` for the merge step. Instructions `vadd` and `vmul` 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` instruction can substitute the `vadd` instruction as it is able to load and store to the scattered elements of the output, summing two fractals at a time, as shown in Figure 3.3. 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.

4.3 Avgpool

The forward and backward operators of `Avgpool` are similar to those described before. The forward implementation, however, applies a reduction to each patch with `sum` instead of `max`. Consequently, its CCE C code uses `vadd` instead of `vmax`. A new operation is also needed to compute an element-wise division before saving the final output. Regardless of these changes, the access pattern is the same for the `sum` computation and it has the same benefits as `Maxpool` by loading its input with `Im2Col` instructions. For training, there is no need to save any additional results as the `Argmax` mask in `Maxpool`. In the backward operator of `Avgpool`, the equivalent mask 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 for its merge step.

```
# Computation -------------

im2col_shape = (N, C_1, K_h, K_w, O_h, O_w, C_0)

# first input
argmax_mask = placeholder(im2col_shape)

# second input
gradients = placeholder((N, C_1, O_h, O_w, C_0))

# multiply each patch by its gradient
mask_gradient = compute(im2col_shape,
                  lambda n, c1, kh, kw, oh, ow, c0:
                    argmax_mask(b, c1, kh, kw, oh, ow, c0)
                    * gradient(b, c1, oh, ow, c0)
                )

# col2im intrinsic
backprop_output = compute((N, C_1, I_h, I_w, C_0),
                    lambda n, c1:
                      col2im(mask_gradient[n, c1, :, :, :]
 )
                  )

# Schedule ----------------

# argmax mask: global memory -> unified buffer
argmax_mask_ub = cache_read(argmax_mask_ub,
                            "local.UB",
                            [mask_gradient])

# gradients: global memory -> unified buffer
gradients_ub = cache_read(gradients,
                          "local.UB",
                          [mask_gradient])

# backpropagation output: unified buffer -> global memory
backprop_output_ub = cache_write(backprop_output,
                                 "local.UB")

# tile computation in c1
b, c1, h, w, c0 = output.op.axis
s[argmax_mask_ub].compute_at(s[backprop_output], c1)
s[gradients_ub].compute_at(s[backprop_output], c1)
s[backprop_output_ub].compute_at(s[backprop_output], c1)
```

Listing 4.3: `Maxpool` backward with `Col2Im` in TVM's DSL

Chapter 5

Experimental Evaluation

This evaluation compares the performance of the Im2col/Col2im based `Maxpool` with the standard TVM `Maxpool` implementation described in Chapter 4. `Maxpool` was run in isolation from other CNN layers receiving random input values. All the experiments ran on an Ascend 910 chip, which contains 32 AI Cores. The cycle count numbers were obtained using the hardware performance counters of the chip, and they refer to the on-chip execution time running at a frequency of 100 MHz. Each evaluation was repeated ten times, and the graphs show the average value and a 95% confidence interval. The cycle count is currently the only metric that could be obtained from the chip.

5.1 InceptionV3 Comparison

To display how complex and large modern CNN architectures are, an overview of the InceptionV3 architecture is shown in Figure 5.1. In this figure,the first layer of the CNN is the leftmost block, and the final layer is the rightmost block. Each block corresponds to an individual layer, where the layer type is identified by its color. Among the many layers of InceptionV3, four `Maxpool` layers are identified in green, and ten `Avgpool` layers are identified in blue, excluding the auxiliary classifier exit represented outside of the dashed box. The last `Avgpool` layer (rightmost) is, in fact, a Global `Avgpool` layer. Tables 5.1, 5.2, and 2.1 display the input sizes of these pooling layers along with examples from other CNN architectures. Table 5.1 shows the input sizes of `Maxpool` layers, Table 5.2 shows input sizes of `Avgpool` layers, and Table 2.1 shows input sizes of Global Pooling layers. In these tables, input sizes are shown in the HWC layout. Input sizes were gathered from the Keras framework [8].

For the `Maxpool` layers represented in Table 5.1, all configurations use a kernel size of $(3, 3)$ and a stride of $(2, 2)$, except for VGG16 [43], which has a kernel size and stride of $(2, 2)$. To test the proposed implementations of `Maxpool`, three configurations were selected from InceptionV3 [49] (highlighted in bold in Table 5.1). The selected layers are represented in Figure 5.1 as the three rightmost green blocks. No padding is needed for their implementation, 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 `Im2Col` and `Col2Im`, these configurations were chosen to display the effects of different input sizes

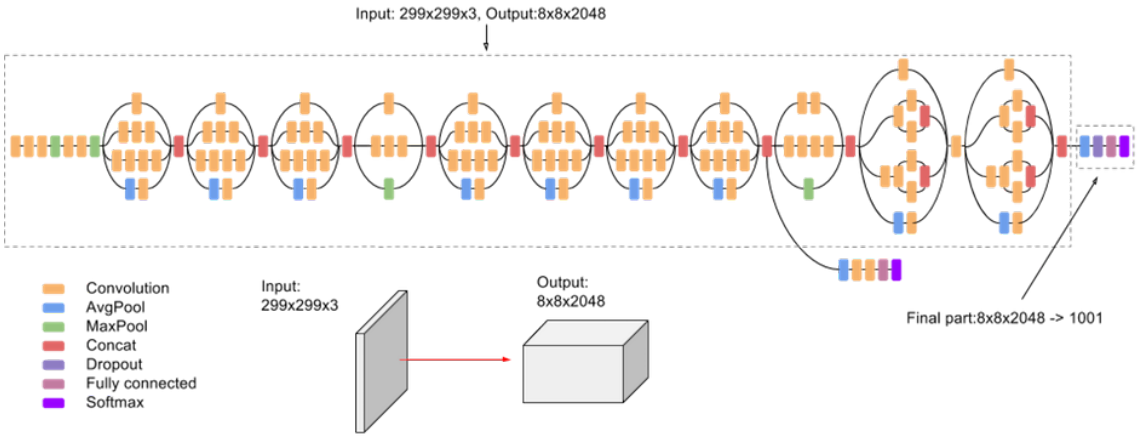


Figure 5.1: Overview of the InceptionV3 architecture [1].

Table 5.1: Maxpool Input Sizes in CNNs. Configurations selected for experimental evaluation are highlighted in bold.

CNN	Input 1	Input 2	Input 3	Input 4	Input 5
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	14,14,512

while using the most common parameters for kernel and stride.

The graphs in Figure 5.2 show the cycle count of the selected `Maxpool` configurations in the NC_1HWC_0 layout. Figure 5.2a shows both `Maxpool` forward implementations. The step of saving the `Argmax` mask is added in Figure 5.2b. This step adds to the computation, as shown by the different ranges in the graphs. For the evaluation in Figure 5.2b, 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 5.2c. In the largest input, the accelerated implementations achieve speedups of 3.2x, 5x, and 5.8x on the graphs in Figure 5.2, 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.

5.2 Stride Tests

This experiment investigates further different `Maxpool` forward implementations, and their interaction with the stride parameter, as shown in Figure 5.3. 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 5.2a. The input's

Table 5.2: Avgpool Input Sizes in CNNs.

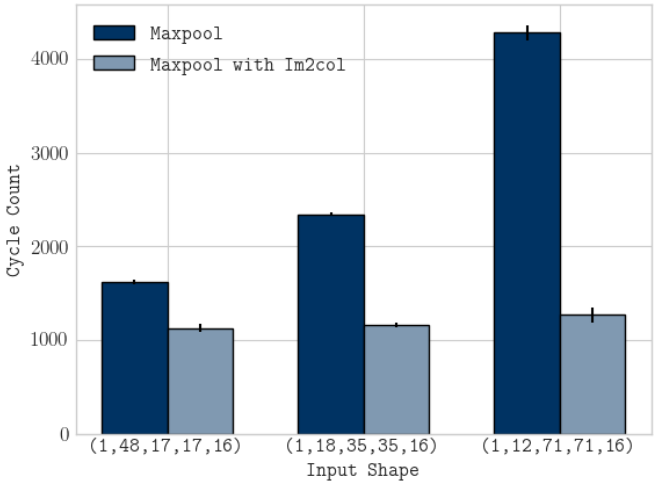
CNN	Input 1	Input 2	Input 3	Input 4	Input 5	Input 6
InceptionV3	35,35,192	35,35,256	35,35,288	17,17,768 (x4)	8,8,1280	8,8,2048
Xception	-	-	-	-	-	-
Resnet50	-	-	-	-	-	-
VGG16	-	-	-	-	-	-

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.

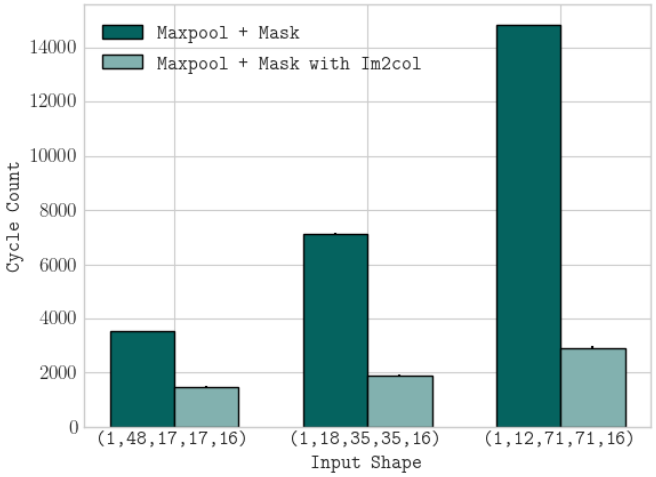
In the `Maxpool with expansion` implementation, regular vector instructions — instead of `Im2Col` instructions — transform the input to the `Im2Col` output shape. This transformation happens when the input is already in the `Unified` buffer, before computing `Maxpool`. `Maxpool with Im2col` and `Maxpool with expansion` achieve superior performance in Figures 5.3b and 5.3c. 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 5.3a 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 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.

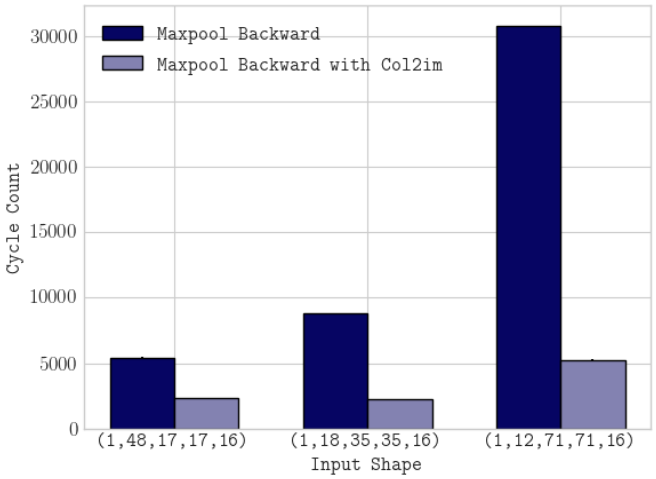
Pooling can also be implemented with an X-Y split by first calculating the reduction function on the width and then on the height of each patch. As a result, the first reduction is reused while computing the second. Lai *et al.* [27] use the X-Y split as a performant alternative to direct pooling. In 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, thus avoiding extra tiling steps needed because of the increase in memory use. Figure 5.3b 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.



(a) `Maxpool`

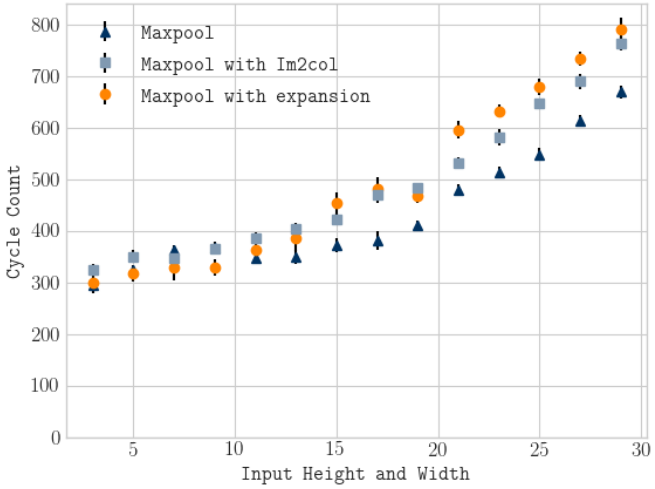


(b) `Maxpool` and `Argmax` Mask

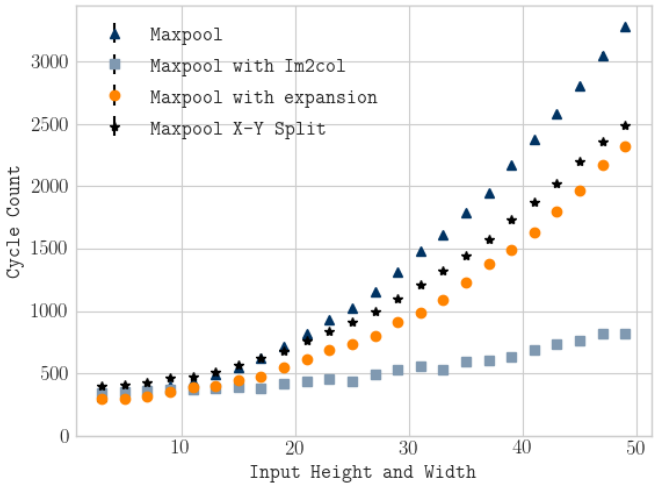


(c) `Maxpool` Backward

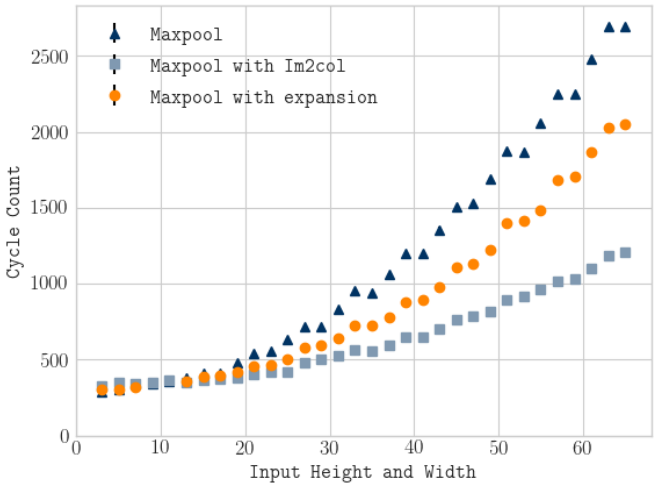
Figure 5.2: Comparison of `Maxpool` implementations with and without `Im2Col` and `Col2Im` instructions. The graphs show the cycle count in the Ascend 910 chip by the size of the input. The input sizes are from InceptionV3. All tests use a kernel size of (3,3) and a stride of (2,2) with no padding.



(a) `Stride = (1,1)`



(b) Stride = (2,2)



(c) Stride = (3,3)

Figure 5.3: 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).

Chapter 6

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. Many works focus on improving pooling and subsampling layers to avoid overfitting and to improve the accuracy of CNNs, instead of improving their performance. But when left unoptimized, these layers can become obstacles that lead to slowdowns in CNNs [30].

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 [45]. 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 [42]. 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 by AKG.

Kernel acceleration for CNNs. LightNet is a Matlab-based framework for Deep Learning [55]. 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 [27]. However, the results in Figure 5.3b 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 [29]. First, the use of the *CHWN* layout instead of *NCHW* to prevent un-coalesced 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 Chapter 5. 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 [46]. 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.

Other Pooling Methods. Although `Maxpool` and `Avgpool` are the two most commonly used subsampling layers in CNNs, many works try to improve this type of layer with different types of pooling. One of the techniques is to add randomness to address overfitting and to improve accuracy resulting from subsampling. Mixed Pooling randomly combines `Maxpool` and `Avgpool` in a single layer [56]. Stochastic Pooling randomly samples features from regions based on a probability distribution given to each element to allow non-maximal elements to be utilized [57]. Fractional Pooling randomizes how pooling regions are generated, as opposed to the fixed regions defined by kernel size and stride size in traditional pooling [14]. However, methods as batch normalization and dropout are more utilized to avoid overfitting together with the simpler `Maxpool` and `Avgpool` implementations that can be optimized by methods such as the Im2col/Col2im based implementations described in this work.

Springenberg *et al.* propose to eliminate pooling layers by performing a strided convolution (stride greater than 1) to subsample the input in an All Convolutional Network [44]. LEAP (LEArning Pooling) proposes a method to use a shared kernel for each channel of the input to act as their subsampling layer [48]. The weights of this kernel can be learned by the networks and `Avgpool` is a special case of this method where all the weights are equal. Forcen *et al.* use the Ordered Weighted Average (OWA) as an aggregator function to perform pooling, where the weights can be learned and `Maxpool` and `Avgpool` can be implemented with specific weights [10]. Such methods provide more flexible subsampling layers, however, the All Convolutional Network increases complexity and parameter number in CNNs and it does not operate on channels separately as `Maxpool`. LEAP and OWA improve upon the All Convolutional Network by having a learning component to their subsampling with fewer parameters and operating on channels independently. These solutions can improve network accuracy and avoid overfitting, but they are still more complex solutions than `Maxpool` and `Avgpool`. Even though LEAP claims to have the same complexity as `Maxpool`, it is only compared to the strided convolution method in their study.

Chapter 7

Conclusion

This work presented the current scenario of high demand for Deep Learning applications, CNNs in particular, which require tremendous amounts of data and computation to run. AI Accelerators are a solution for efficient execution of convolution and matrix multiplication, operations that contribute with a substantial part of the computation required for these applications. Still, many other operations are needed in modern CNNs and their lack of optimization can hinder the overall performance of a CNN. This work focused on the optimization of one of the building blocks of CNNs, pooling layers.

DaVinci's AI Accelerator architecture was described in great detail along with its dedicated `Im2Col` and `Col2Im` instructions. Such instructions are designed to optimize convolution by mapping it to matrix multiplication and enabling its computation in DaVinci's matrix-multiplier unit, the Cube Unit. It is shown that these instructions can be used to implement not only convolution, targeting the Cube Unit, but also pooling, targeting memory layout improvements, and improved execution in the Vector Unit. This is shown for the Im2col/Col2im based pooling implementations, which were described for the forward and backward operators of `Maxpool` and `Avgpool`. An experimental evaluation was run on the Ascend 910 chip to compare the proposed accelerated implementations to baselines that do not use the `Im2Col` and `Col2Im` instructions. The parameters and three input sizes used in InceptionV3 were used to represent common pooling configurations, and the results show speedups of up to 5.8x for the Im2col/Col2im based `Maxpool` implementations. 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.

The evaluation of this work targets the DaVinci architecture. However, the techniques presented to optimize pooling may benefit other architectures, even without the support of the `Im2Col` and `Col2Im` instructions. Future work could evaluate Im2col/Col2im based pooling in CPUs, GPUs, and other types of AI Accelerators. Such work would require a deep study of the characteristics of each architecture to adapt these pooling implementations.

For example, in CPUs, memory buffers are organized as a cache hierarchy instead of scratch-pad memories, and the memory layout of the input of pooling is not enforced. For pooling implementations in CPUs, the memory requirements for the Im2col and Col2im

transformations need to be carefully analyzed as duplicated elements would go through the entire cache hierarchy. Furthermore, the memory layout of the input could lead to slightly different Im2col and Col2im transformations. Such differences could affect the performance overhead of Im2col and Col2im, and how well the code is vectorized. Additionally, a direct version of pooling would also need to be optimized for a proper comparison. The best implementation can also be highly dependant on the parameters used, as shown in this work for a stride of $(1, 1)$. GPUs present another set of challenges such as memory offloading overhead, coalescing memory accesses, and their massively parallel hardware design. However, in more traditional architectures without computational units specialized for convolution, pooling can take many hints from the existing research in optimizing convolution.

Bibliography

[1] Advanced guide to inception v3 on cloud tpu. `https://cloud.google.com/tpu/docs/inception-v3-advanced`,. Online; accessed 22 July 2021.

[2] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, 2017.

[3] Monica Bianchini and Franco Scarselli. On the complexity of neural network classifiers: A comparison between shallow and deep architectures. *IEEE Transactions on Neural Networks and Learning Systems*, 25(8):1553–1565, 2014.

[4] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[5] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), 2006.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Operating Systems Design and Implementation (OSDI)*, page 579–594, Carlsbad, CA, USA, 2018.

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

[8] François Chollet et al. Keras. `https://keras.io`, 2015.

[9] Francois Chollet. *Deep Learning with Python*. Manning Publications Co., 1st edition, 2017.

[10] J.I. Forcén, Miguel Pagola, Edurne Barrenechea, and Humberto Bustince. Learning ordered pooling weights in image classification. *Neurocomputing*, 411:45–53, 2020.

[11] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002.

[12] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Transactions on Mathematical Software*, 35(1), July 2008.

[13] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, 2008.

[14] Benjamin Graham. Fractional max-pooling. *CoRR*, abs/1412.6071, 2014.

[15] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, USA, second edition, 2008.

[16] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. FP-DNN: an automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159, Napa, CA, USA, 2017.

[17] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`,, 2010. Online; accessed 28 August 2020.

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

[19] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[20] N. Jouppi, C. Young, N. Patil, and D. Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018.

[21] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture (ISCA)*, page 1–12, Toronto, ON, Canada, 2017.

[22] Wonkyung Jung, Daejin Jung, Sunjung Lee, Wonjong Rhee, Jung Ho Ahn, et al. Restructuring batch normalization to accelerate cnn training. In *SysML Conference (SysML)*, Palo Alto, CA, USA, 2019.

[23] Ahmed Khaled, Amir F. Atiya, and Ahmed H. Abdel-Gawad. Applying fast matrix multiplication to neural networks. In *Symposium on Applied Computing (SAC)*, page 1034–1037, Brno, Czech Republic, 2020.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, page 1097–1105, Red Hook, NY, USA, 2012.

[25] H. T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[26] Hsiang T Kung and Charles E Leiserson. Systolic arrays for (VLSI). Technical report, Carnegie-Mellon Univ Pittsburgh Pa Dept Of Computer Science, 1978.

[27] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: efficient neural network kernels for Arm Cortex-M CPUs. *CoRR*, abs/1801.06601, 2018.

[28] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[29] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang 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.

[30] Dawei Li, Xiaolong Wang, and Deguang Kong. DeepRebirth: accelerating deep neural network execution on mobile devices. In *AAAI Conference on Artificial Intelligence*, pages 2322–2330, New Orleans, LA, USA, 2018.

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

[32] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *International Conference on Learning Representation (ICLR)*, Banff, AB, Canada, 2014.

[33] Stefano Markidis, Steven Chien, Erwin Laure, Ivy Peng, and Jeffrey Vetter. Nvidia tensor core programmability, performance & precision. In *International Parallel and Distributed Processing Symposium Workshops(IPDPSW)*, pages 522–531, Vancouver, BC, Canada, 05 2018.

[34] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4), 2016.

[35] NVIDIA. cublas. `https://developer.nvidia.com/cublas`. Online; accessed 19 March 2021.

[36] Jay H Park, Sunghwan Kim, Jinwon Lee, Myeongjae Jeon, and Sam H Noh. Accelerated training for cnn distributed deep learning through automatic resource-aware layer placement. *arXiv preprint arXiv:1901.05803*, 2019.

[37] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Programming Language Design and Implementation (PLDI)*, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[39] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[40] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[41] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networks (ICANN)*, page 92–101, Thessaloniki, Greece, 2010.

[42] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to FPGAs. In *Microarchitecture (MICRO)*, pages 1–12, Taipei, Taiwan, 2016.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representation (ICLR)*, San Diego, CA, USA, 2015.

[44] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. In *International Conference on Learning Representation (ICLR)*, San Diego, CA, USA, 2015.

[45] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Field-Programmable Gate Arrays (FPGA)*, page 16–25, Monterey, California, USA, 2016.

[46] Shunsuke Suita, Takahiro Nishimura, Hiroki Tokura, Koji Nakano, Yasuaki Ito, Akihiko Kasagi, and Tsuguchika Tabaru. Efficient convolution pooling on the gpu. *Journal of Parallel and Distributed Computing*, 138:222 – 229, 2020.

[47] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Harikrishna Mulam. Revisiting unreasonable effectiveness of data in deep learning era. In *International Conference on Computer Vision (ICCV)*, pages 843–852, Venice, Italy, 10 2017.

[48] Manli Sun, Zhanjie Song, Xiaoheng Jiang, Jing Pan, and Yanwei Pang. Learning pooling for convolutional neural network. *Neurocomputing*, 224:96–104, 2017.

[49] 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)*, pages 2818–2826, Las Vegas, NV, USA, 2016.

[50] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, Boston, MA, USA, 2015.

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

[52] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 1–27, San Jose, CA, USA, 1998.

[53] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[54] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 BLAS performance optimization on loongson 3A processor. In *International Conference on Parallel and Distributed Systems (ICPADS)*, page 684–691, Singapore, 2012.

[55] Chengxi Ye, Chen Zhao, Yezhou Yang, Cornelia Fermüller, and Yiannis Aloimonos. LightNet: A versatile, standalone matlab-based environment for deep learning. In *Multimedia (MM)*, page 1156–1159, Amsterdam, The Netherlands, 2016.

[56] Dingjun Yu, Hanli Wang, Peiqiu Chen, and Zhihua Wei. Mixed pooling for convolutional neural networks. In Duoqian Miao, Witold Pedrycz, Dominik Ślezak, Georg Peters, Qinghua Hu, and Ruizhi Wang, editors, *Rough Sets and Knowledge Technology*, pages 364–375. Springer International Publishing, 2014.

[57] Matthew D. Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. In *International Conference on Learning Representation (ICLR)*, Scottsdale, Arizona, USA, January 2013.