# Automatic Program Generation for Convolutional Neural Networks on Resource Constrained Devices

by

Cormac David Keane

**Thesis**

Submitted to the School of Computer Science and Statistics
in partial fulfillment of the requirements for the degree of

Master in Science
(Computer Science)

School of Computer Science and Statistics

TRINITY COLLEGE DUBLIN

June 2022

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

## Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Cormac David Keane

Dated: September, 2022

# Permission to Lend and/or Copy

# Abstract

Convolutional Neural Networks (CNNs) are both arithmetically and memory intensive when performing inference. This is a problem when executing CNNs on resource constrained machines, such as small embedded devices. This thesis proposes domain-specific program generators (DSPG), and automatic program optimizers (APO) to improve the resource usage (execution time, memory usage, energy usage) of CNN convolution on ARM devices.

We extend previous work on a DSPG and APO for direct CNN convolution to create *Genvolution*. Genvolution can automatically generate optimized implementations for CNN convolution on Intel and ARM NEON devices. Genvolution implementations outperforms vendor library *im2col* implementations for 33% of tested CNN convolutions. Genvolution was also used to investigate the use of *Flyte*, a reduced precision floating-point storage datatype, on CNN convolution on ARM devices. We demonstrate that generated code using the Flyte datatype improves energy usage while maintaining execution speed for 60% of tested CNN convolutions.

We also propose Winogen, a second DSPG and APO created to produce Winograd CNN convolution implementations. Winogen implementations outperform vendor library Winograd implementations for 90% of tested CNN convolutions. Winogen is also used to investigate a novel Winograd CNN convolution algorithm. Our proposed algorithm reduces the memory overhead of standard Winograd CNN convolution, while still leveraging the problem complexity reduction Winograd convolution allows. We found our new algorithm outperforms standard Winograd convolution for 33% of tested CNN convolutions.

We demonstrate that automatic program generation can be used to improve the resource usage of CNNs on ARM devices. All CNN resource reduction is significant when embedded devices will run the same CNN countless times over their lifespan.

# Acknowledgements

Many people helped me directly and indirectly throughout the course of this project. I would like to thank my family for support, especially Jacinta for all the meals she made for me over the last few months. I'd like to thank my supervisor Dr. David Gregg. His advice throughout this work has been invaluable. Finally, I'd like to thank everyone who I shared an office space with for keeping me company throughout this research.

This work was supported, in part, by Science Foundation Ireland grants 12/IA/1381 and 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Convolution neural networks (CNNs) are a type of neural network present in machine learning. A CNN is a directed acyclic graph where each node represents a data transformation (LeCun et al. 1989). Inputs are fed into the bottom of the graph, and move up through the graph. The input is transformed by the nodes as it passes through them. The leaf nodes of the graph contain the outputs of the CNN, generated from a given input. CNNs were originally designed for image classification (LeCun et al. 1989), but they have also been used to solve other vision (Gatys, Ecker, and Bethge 2016), and non-vision problems such as text processing (Jacovi, Shalom, and Goldberg 2018).

The main operation used by CNNs is CNN convolution. CNN convolution takes in an input tensor, and an input kernel. The input kernel is applied to each point in the input tensor to produce a new output tensor. There are multiple common methods for implementing CNN convolution. CNN convolution can implemented using a nest of loops that moves across the input tensors. This is called *direct convolution*. CNN convolution can also be mapped to matrix multiplication by producing a patch matrix from the input tensor (Vasudevan, Anderson, and Gregg 2017). This is called *im2col convolution*. There are also so-called ′fast′ convolution algorithms that transform the inputs to another representation. One ′fast′ method is *Winograd Convolution* (Lavin and Gray 2016).

CNNs are extremely resource intensive, requiring a large amount of computational work and memory. This becomes an issue when CNNs are used on resource constrained machines, such as embedded devices. The majority of resources are used performing CNN convolution. By reducing the resource usage of CNN convolution, CNNs can be more easily ported to resource constrained devices.

We investigated methods for reducing the execution time, memory size, and energy usage of CNN convolution, with an emphasis on reducing resource usage on low-power single-threaded ARM devices. The principal contributions of this thesis are as follows:

- We extend previous work on a domain-specific program generator to develop *Genvolution* (Chapter 4). Genvolution is a program generator and automated program optimizer that can be used to generate optimized direct CNN convolution implementation for Intel, ARMv7, and ARMv8 architectures. We found that the implementations generated by Genvolution outperformed vendor library direct convolution implementations on tested ARM devices for all tested input sizes (section 4.12). The generated code also outperformed vendor library Im2Col convolution implementations for a number of input sizes (Section 4.12). Also, all Genvolution-generated implementations required only a fraction of the temporary memory that im2col convolution requires which is important on memory constrained devices.

- Genvolution was also used to automatically generate and optimize matrix multiplication implementations on ARM devices (Chapter 7). We found that the implementations generated by Genvolution outperformed vendor library matrix multiplication implementations for a small number of input sizes (Section 7.6).

- We developed *Winogen*, a domain-specific program generator and automated program optimizer for generating optimized Winograd convolution implementations for ARMv7 and ARMv8 architectures (Chapter 6).

We found that the implementations generated by Winogen outperformed vendor library Winograd convolution implementations for a number of input sizes (Section 6.15).

- We propose a new Winograd CNN convolution algorithm. Our method uses a vertical 1D convolution to accumulate multiple 1D Winograd CNN convolutions (Section 6.12). The result of the vertical 1D convolution is same as a regular CNN convolution. This method requires less memory than the standard Winograd convolution algorithm, but still reduces the problem complexity of convolution using the Winograd algorithm. We found our new method outperformed vendor library Winograd convolution implementations for a number of input sizes (Section 6.15).
- We investigated the usage of the *Flyte* floating-point datatype for reducing the energy and memory usage of direct convolution implementations (Chapter 8). We developed an ARMv7, and ARMv8 library for transforming multiple values to and from the Flyte datatype using ARM SIMD vector intrinsics (Section 8.3). We found that the total energy usage of direct convolution implementations could be reduced, with no effect on execution time, for a number of input sizes by using the Flyte datatype with our developed libraries (Section 8.7).

While in most of our research topics, we only improve resource usage for some input sizes, our results are still significant. The input sizes of CNN convolutions in CNN networks are static and known in advance. This means that it can be found empirically which layers our produced code performs better on. While this adds some extra work for the network implementer, even a small increase in performance can be very significant if the network is to be run many times on many devices. This is especially true for CNN networks running on mass-produced embedded devices.

The remainder of this thesis is structured as follows.

In Chapter 2, we describe background information referenced throughout the rest of the thesis. This includes a review of CNN convolution, and necessary terminology related to it. This chapter also introduces SIMD vector architectures which are referenced throughout this thesis. Data blocking to improve data cache usage is also explained.

In Chapter 3, we cover tools used and experimental set-up. The libraries used for collecting results, as well as the vendor libraries used for baseline comparisons of results are outlined. The SIMD libraries used are also defined. Finally, the specifications of the three test machines are given.

In Chapter 4, the design of Genvolution, our domain-specific program generator for direct convolution is explained. The general design of Genvolution is given. How the search space for program optimizations is built and traversed is also described. This chapter also covers how Genvolution generates C++ code for direct convolution from a set of program parameters using an intermediate representation abstract syntax tree.

In Chapter 5, we describe optimizations that can exist in the C++ code generated by Genvolution. We present a number of different ways SIMD vectorization can be used to implement CNN convolution, and the different strengths and weaknesses associated with each way.

In Chapter 6, we first explain the theory of Winograd convolution, and the standard algorithm for implementing Winograd convolution outlined by Lavin et al. (Lavin and Gray 2016). We then cover different ways that Winograd convolution can be optimized further. Finally, we outline our new Winograd convolution algorithm in detail.

In Chapter 7, we describe how CNN convolution with a $(1 \times 1)$ input kernel can be implemented as matrix multiplication without transforming the input or output tensors. We then show how Genvolution can be used to generate optimized matrix multiplication implementations for ARM devices.

In Chapter 8, we first introduce the Flyte datatype, a reduced precision floating-point storage datatype. We then outline our ARM NEON libraries for transforming multiple IEEE-754 binary32 floating-point values to and from the Flyte datatype in parallel.

In Chapter 9, we present our conclusions. We evaluate the results of the previous chapters as a whole, and mention possible topics of further research.

In the next chapter, we move from giving an overview of this thesis to describing background information needed throughout the rest of the thesis.

# Chapter 2

# Background

## 2.1 Convolution and CNNs

### 2.1.1 Overview of the CNN Convolution Operation

Convolution neural networks (CNNs) are a type of deep neural network (DNNs) used within machine learning. CNNs are a type of multilayer perceptron, and are constructed as a graph of operations that can be conceptually split into multiple layers (Vasudevan, Anderson, and Gregg 2017). Each layer is constructed from one or more CNN nodes. To perform inference with a CNN (i.e. to use it to estimate an output given an input), we feed the input values into the bottom layer of the graph. The CNN nodes of each layer transforms the data fed into them, and the node's output propagates up the graph to the next layer. Note that a CNN graph must be acyclic, and that every CNN node can have an accompanying state used during input transformation (LeCun et al. 1989). In theory, a CNN node can perform any transformation that is valid for its given input. However, in practice there are a small number of CNN node types that are shared across the majority of CNN designs.

The most common transformation performed by a CNN node is CNN convolution. CNN convolution takes an input image tensor and applies an input kernel tensor to it to produce a new output image tensor. The simplest form of CNN convolution is *single-channel* CNN convolution, where we take

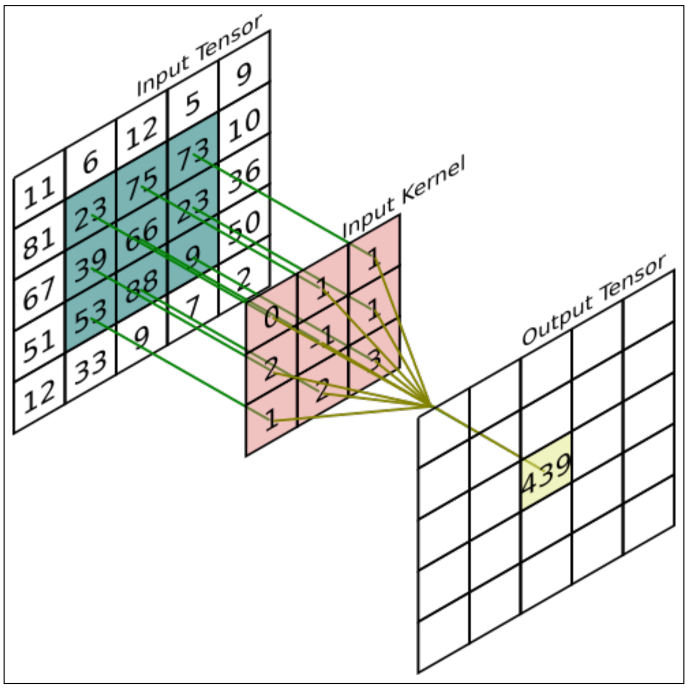


Figure 2-1: Single Channel CNN convolution. The value of an output point P is the sum of products of the overlapping input tensor and input kernel points centred at P.

a two-dimensional input image $img$ (with height $H$ and width $W$), and a two-dimensional input kernel $ker$ (with height $X$ and width $Y$), and produce a two-dimensional output tensor $out$ (also with height $H$ and width $W$). To calculate the value of each point $out(h, w)$ we overlay our kernel $ker$ on top of $img$ so that $ker$ is centered on the point $img(h, w)$. We then perform point-wise multiplication between the overlapping points of $img$ and $ker$, and finally sum the $X \times Y$ produced values to get the final value for the point $out(h, w)$. Figure 2-2 gives the equation for *single-channel* CNN convolution. Also, if the input kernel is square (i.e. $X = Y$), then the length of the kernel's dimensions are equal, and we refer to them as having length $K$ (where $K = X = Y$).

$$out(h, w) = \sum_{x=0}^{X} \sum_{y=0}^{Y} img(h + (y - \lfloor Y/2 \rfloor), w + (x - \lfloor X/2 \rfloor)) * ker(x, y)$$

Figure 2-2: Equation for *single-channel* CNN convolution.

*Single-channel* CNN convolution can be extended to *multi-channel* CNN convolution by adding an extra dimension $C$ to both the input image tensor $img$ and the input kernel tensor $ker$. $img$ and $ker$ are now three-dimensional tensors with shapes $(H \times W \times C)$ and $(X \times Y \times C)$ respectively. However, it is often conceptually more helpful to think of them as two-dimensional tensors where every point contains a vector of $C$ scalar values. To calculate the value of $out(h, w)$ we again overlay $ker$ to be centered on the point $img(h, w)$. However, we now perform a vector dot product between the vectors in the overlapping points, and then sum those values together to get the output value. Figure 2-3 gives the equation for *multi-channel* CNN convolution.

$$out(h,w) = \sum_{x=0}^{X}\sum_{y=0}^{Y}\sum_{c=0}^{C} img(h + (y - \lfloor Y/2 \rfloor), w + (x - \lfloor X/2 \rfloor), c) * ker(x, y, c)$$

Figure 2-3: Equation for *multi-channel* CNN convolution.

Most CNN convolution nodes in a CNN perform *multi-channel* CNN convolution. However, instead of having a single input kernel tensor $ker$, they have a vector of $M$ input kernels tensors. Every kernel tensor in the vector has the same dimensions, and each kernel is applied in turn to the input image tensor to create $M$ separate 2D output tensors. The $M$ 2D outputs are concatenated to create a final output image tensor with shape $(H \times W \times M)$.

### 2.1.2 Dealing with Edge Points in CNN Convolution

When we are calculating the edge values for an output tensor $out$ using CNN convolution, we may need values from outside the boundaries of the input image tensor (Sewak, Karim, and Pujari 2018). For example, if we are performing *single-channel* CNN convolution and we want the value for the point $out(0, 0)$ with $X = 3$, $Y = 3$, then we will need to calculate the value of $img(-1, -1) * ker(0, 0)$. However, $img(-1, -1)$ is outside the boundaries of $img$ so we must select some method which will allow us to synthesis these needed values. Figure 2-4 shows another example where an input kernel tensor is centered on the top-right point of an input image tensor. This causes the top row and right column of the input kernel tensor to lay outside the bounds of the input image tensor.

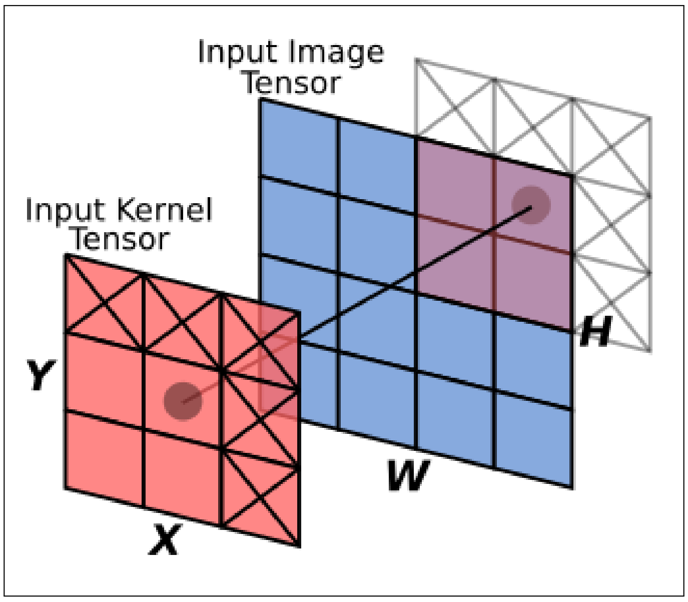


Figure 2-4: Input Kernel tensor overlayed on an input image tensor such that some values outside the image are required..

There are a number of common methods of synthesizing the extra values required:

- Zero-Padding: Assume all values outside the boundaries of $img$ have the value 0. This is the simplest method and most commonly used one (Khan et al. 2018).

- Mirroring: Assume that $img$ is conceptually mirrored at every edge. This is equivalent to taking the absolute value of all $img$ indices before using them (i.e. $img(\mid h \mid, \mid w \mid)$). For example, the value of $img(-2, 4)$ is mapped to $img(2, 4)$ (Sewak, Karim, and Pujari 2018).

- Extension: Assume that all the border points of $img$ extend infinitely outwards from $img$. This is equivalent to mapping every synthesized value to the nearest valid value (using Euclidean distance) (Sewak, Karim, and Pujari 2018).

- Cropping: Do not calculate the value of any output points that require values from outside the boundaries of $img$. This will result in an output tensor $out$ that is slightly smaller than $img$. In this case $out$ will have the shape $(H - (X - 1)) \times (W - (Y - 1))$ (Khan et al. 2018).

The same synthesizing methods are used for *single-channel* and *multi-channel* convolution, with the *multi-channel* tensors being treated as two-dimensional tensors where every point contains a vector of $C$ values.

### 2.1.3 Direct CNN Convolution Implementation

The simplest method for implementing CNN convolution is the direct convolution algorithm. Direct convolution consists of six nested loops that iterate over the dimensions of the input tensors to perform the CNN convolution. The six loops are independent and can be reordered in any manner. The dimensions of the input and output tensors can also be arranged in any order. Figure 2-5 shows an example pseudo-code implementation of direct convolution.

```
in_tensor[height][width][channels]
in_kernels[kernels][X][Y][channels]
out_tensor[height][width][kernels]
for (m = 0; m < kernels; m++) {
  for (h = 0; h < height; h++) {
    for (w = 0; w < width; w++) {
      for (x = -(X/2); x <= X/2; x++) {
        for (y = -(Y/2); y <= Y/2; y++) {
          //skipping points outside in_tensor is the
          //same as multiplying by zero, i.e. zero
          //padding
          if ((h+x >= 0 && h+x < height) &&
              (w+y >= 0 && w+y < width)) {
            for (c = 0; c < channels; c++) {
              i = in_tensor[h+x][w+y][c];
              k = in_kernels[m][x][y][c]
              r = i * k
              out_tensor[h][w][m] += r
            }
          }
        }
      }
    }
  }
}
```

Figure 2-5: Pseudo-code implementation of direct convolution using zero-padding.

### 2.1.4 Im2Col CNN Convolution Implementation

CNN convolution can also be implemented by transforming the problem to matrix multiplication (Vasudevan, Anderson, and Gregg 2017). To do this, the input tensor is transformed into a 2D tensor of size $((H \times W) \times (C \times K \times K))$, and the input kernel is transformed into a 2D tensor of size $((C \times K \times K) \times M)$. A matrix multiplication is performed between the two transformed input tensors to create a 2D output tensor of size $((H \times W) \times M)$. The output tensor contains all the output values of the wanted CNN convolution. Performing CNN convolution using matrix multiplication is called *im2col convolution*, because the input (image) tensor is transformed into a 2D matrix where each $(C \times K \times K)$ column contains all the values needed to calculate one value in the output tensor.

Im2Col CNN convolution is often used in practice, because it allows programmers to make use of pre-existing highly optimized matrix multiplication routines. This saves the time needed to optimize a CNN convolution implementation from scratch. However, im2col convolution introduces a very large memory overhead, because memory for the transformed $((H \times W) \times (C \times K \times K))$ input tensor must be allocated.

## 2.2 SIMD Vectors

### 2.2.1 Flynn's Taxonomy and SIMD

Flynn's taxonomy classifies the possible types of computer architectures into four categories. One of these categories is *Single Instruction Multiple Data*, or SIMD, which consists of any architecture that applies a single stream of operations/instructions to multiple data streams in parallel (Flynn 1972). An example of this architecture is general purpose GPUs. General purpose GPUs typically have multiple arithmetic-logic-units (ALUs) all being fed with different data streams, but every ALU applies the same operations to their data streams in parallel as issued by a single instruction stream. Another example of a SIMD architecture are SIMD vectors.

### 2.2.2 SIMD Vector

Typically, we visualize the data in a processor being stored in scalar registers, i.e. registers that hold a single numeric value. A stream of scalar instructions operates on the scalar registers, and each instruction produces a single result (excluding edge-case instructions, such as division instructions that also produce a remainder). We can parallelize this design by extending our scalar registers so that they each hold several values at once. These are called SIMD vector registers as they hold a vector of scalar values in a single register (Hennessy and Patterson 2012). Logically each SIMD register is considered to be split into an ordered list of scalar values, where each scalar value occupies one *lane* of the vector.

SIMD instructions take a pair of SIMD vector registers as input, and produce a SIMD vector register of values as output, where each lane of the output is calculated using the matching lanes of the two inputs. For example, an add instructions between two SIMD vector registers containing {1,2,3,-1} and {2,4,6,8} respectively would produce a SIMD vector register containing {3,6,9,7} (Hennessy and Patterson 2012). This matches the definition of SIMD in Flynn's taxonomy, because given a single instruction stream, we operate on multiple streams of data spread across the lanes of the vector registers.

Often SIMD instructions will have similar latency and throughput as their scalar counterparts, meaning that vectorizing scalar code can have a significant impact on performance. For example, on the Intel Skylake architecture, the *VMULPS* instruction performs a SIMD multiplication between two SIMD register holding eight 32-bit floats with a latency of 4 cycles and a throughput of 1 cycle (Intel 2019). *FMUL*, the scalar Skylake instruction for multiplying two scalar 32-bit floats has the same latency and throughput (Fog 2018). This means the vectorized *VMULPS* instruction can perform eight times more computation than the scalar *FMUL* instruction in the same amount of time.

### 2.2.3 Data Layout

A limitation of modern general purpose processor SIMD architectures is that it introduces some restrictions on data layouts. Most SIMD architectures only implement performant vector register load and store operations, where all the scalar values being accessed are contiguous in memory. For example, the Intel *VMOVAPD* instruction for loading 256-bit SIMD vectors from memory effectively just copies 256 bits starting at a given memory address into a vector register (Intel 2019). This means data layout choices are restricted when using SIMD vectorization, because the data being vectorized must be kept packed and contiguous.

### 2.2.4 Horizontal Instructions

SIMD vector architectures are usually designed to prioritize operations that work on pairs of values in matching lanes of *distinct* vector registers. These include the obvious pairwise arithmetic and bitwise operations. However, as each vector register already contains multiple values, we can apply operations that work across the lanes of one register to produce a new value. These are called *horizontal* operations.

Common horizontal operations are reordering the lanes in a vector register, or summing all the values across the lanes of a vector register. Every major Intel SIMD architectures offer some range of different vector reordering instructions. Although, in general for the Intel architectures, as the size of the vector registers increases more restrictions are placed on the reordering operations. For example, in the Intel AVX architecture, the *_mm_permute_ps* intrinsic allows a 128-bit register to be freely permuted as four 32-bit lanes. However the 256-bit equivalent intrinsic, *_mm256_permute_ps*, includes the restriction that values cannot be moved from the high 128-bits to the low 128-bits of the register or vice versa (Intel 2019). The ARM NEON architectures offer a single set of reordering instructions that treat the input vectors as tables. They can be used to perform any reordering operation, including duplicating lanes (ARM 2018).

## 2.3 Data Blocking

All tensor-based operations will exhibit data access characteristics inherent to the operation being performed. For example, if we are performing *single-channel* CNN convolution and wish to calculate the output point value at position $(h, w)$ in the output tensor, then we will need to access the values surrounding the point $(h, w)$ in the input tensor. If we then calculated the value for $(h + 1, w)$, there would be a large overlap with the values needed for calculating $(h, w)$. CNN convolution can exhibit strong temporal locality in it's data accesses. Data caches and the memory hierarchy rely on exploiting temporal and spatial locality to improve performance. Following the calculation of the value at $(h, w)$ with $(h + 1, w)$ will have good cache usage as the second calculation reuses many values from the first that should still be stored in the fastest cache.

When implementing a tensor-based operation, it is important to take into account the data access patterns of the problem to improve cache usage. However, the obvious approach to implementing tensor based operations can lead to poor cache usage. Returning to the CNN convolution example outlined above, the simplest implementation for calculating all the values of the output tensor would be to have two nested for loops that run across the image height $H$ and image width $W$ dimensions of the output image tensor. This would lead to the output points being calculated in the order $(0, 0)$, $(0, 1)$, ..., $(0, W - 1)$, $(1, 0)$, $(1, 1)$, ..., $(H - 1, W - 1)$. However, calculating the output points in this order would mean that there is a large gap between the calculation of $(h, w)$ and $(h + 1, w)$. If the values used in the calculation of $(h, w)$ are evicted from the cache by the intermediary points calculated before reaching $(h + 1, w)$, the values would need to be refetched from a slower level of memory hierarchy leading to worse performance.

Figure 2-6 shows how data can be shared between two convolutions when using a 3x3 kernel. The convolution centered on the circle needs six of the same data points as the convolution centered on the star. If these two convolutions were performed sequentially much of the data required for the convolution

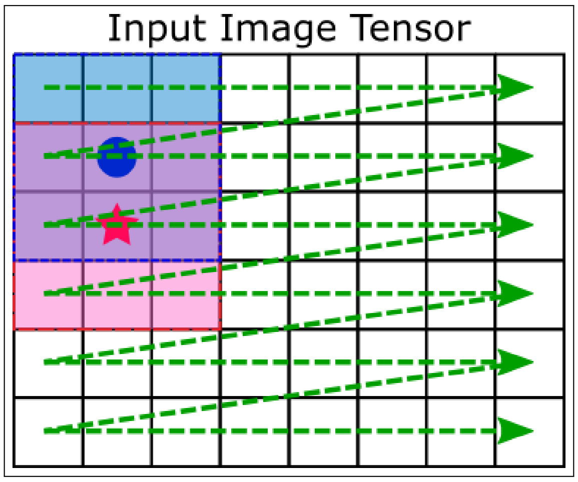


Figure 2-6: Overlapping memory usage from two convolutions (centered on the circle, and the star) using a 3x3 kernel. The dotted line shows the order the convolutions are performed.

centered on the star would already be in the L1 cache. However, the dotted line in figure 2-6 shows the order the convolutions will be performed in when using a pair of for loops to iterate through the needed convolutions. All the other convolutions in the same row as the circle will be calculated before the convolution centered on the star will be performed. By this time, the data overlapping with the convolution centered on the circle may have been evicted from the faster caches.

The current loop ordering means that the point calculated after $(h, w)$ would be $(h, w + 1)$. These two points also have a large overlap in data usage. However, we can improve cache usage more by trying to perform all calculations that share common data sequentially. This is the basis of 'data blocking', which attempts to improve cache usage by breaking larger loops into smaller sections that iterate over small groups of values before moving to the next group.

### 2.3.1 Implicit Tile Blocking

Implicit tile blocking is a form of data blocking. Implicit tile blocking is based around splitting the loops that iterate across the dimensions of a problem into pairs of inner and outer loops. Each inner loop covers a small section of a given dimension and the outer loops select what section the inner loops will iterate over. When multiple dimensions are blocked like this, then the iteration space is split into smaller blocks of data. Each block is completely iterated across before moving onto the next block (Daydé, Marques, and Nakajima 2013). For example, if we used implicit tile blocking on figure 2-6 to create data blocks of $(3 \times 3)$ we would then iterate through the output points $(h, w)$, $(h, w + 1)$, $(h, w + 2)$, $(h + 1, w)$, $(h + 1, w + 1)$, $(h + 1, w + 2)$, $(h + 2, w)$, $(h + 2, w + 1)$, $(h + 2, w + 2)$ sequentially for each data block. As the calculations for all these points share many input data points, we could expect to see excellent cache data usage.

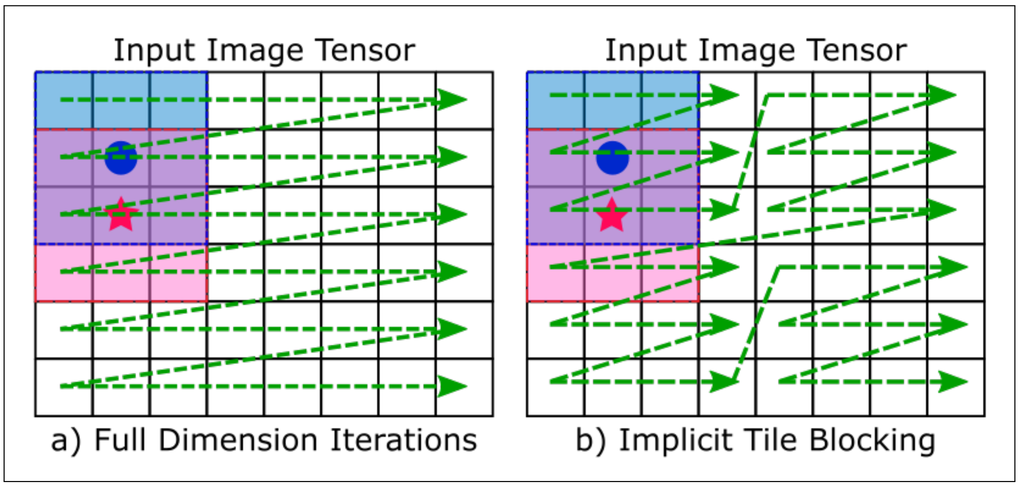


Figure 2-7: Overlapping memory usage from two convolutions (centered on the circle, and the star) using a $(3 \times 3)$ kernel. The dotted line shows the order the convolutions are performed.

Figure 2-7 shows the effect implicit tile blocking can have on the order the convolution are performed in. In part (b) of the figure, the dimensions of the input image tensor have been broken into blocks of size $(3 \times 4)$. This means that there are only three other convolutions performed between the calculation of the convolution centered on the circle and the convolution centered on the star, rather than the seven convolutions between them in part (a). This means it is more likely that the overlapping values from the circle convolution will still be in a fast cache when the star convolution is performed.

Implicit tile blocking does not guarantee better performing cache usage. There may be high data reuse between points at the edge of a data block and the points of a adjoining data block. However, implicit tile blocking does not attempt to try and ensure that the cached data used by one set of edge points will not be evicted before reaching adjoining edge points. More advanced blocking techniques attempt to exploit data reuse between adjoining blocks of data. Hilbert curve blocking uses the recursive Hilbert curve to create a hierarchy of data blocks. It attempts to retain data shared between smaller data blocks in larger blocks stored in the L3 cache (Böhm, Perdacher, and Plant 2016). However, calculating the next point to iterate to is significantly more expensive than implicit tile blocking. Implicit tile blocking attempts to improve the usage of spatial locality in a problem with the minimum overhead possible, even if this means not optimally designing for data cache reuse.

When performing implicit tile blocking, selecting the correct block size is very important. A larger data block allows for more points that share common data to be iterated across sequentially, and reduces the number of points that share data being split across different blocks. However, if the block is too large the common data will be evicted from the data cache before it can be reused, returning data usage to how it was pre-blocking (Park, Hong, and Prasanna 2003). To reduce unwanted capacity cache misses inside the blocks, the data blocks should be smaller then caches available. Lam et al prove that in theory the optimal size for a square data block to reduce capacity misses is $\sqrt{aso \times csc / aso + 1}$ where $aso$ is the associativity of the cache and $csc$ is the size of the cache (Lam, Rothberg, and Wolf 1991). This assumes that the program does not need to share the cache hierarchy with any other program, and in practice the optimal block size may be much smaller due to this and other practical constraints.

Implicit tile blocking will create data blocks that contain non-contiguous data, which can increase the number of conflict cache misses that occur depending on the alignment of data and the associativities of the caches in use (Athanasaki, Koziris, and Tsanakas 2005). Typically, cache systems use the lower bits of each memory address to select which cache set the address belongs to (excluding the lowest bits that index into each cache line). If an inner loop iterates across a non-contiguous dimension, a large number of addresses that map to the same cache set may be sequentially accessed, because all the addresses may share the same lower bits. This is more likely to occur if the inner dimension sizes are powers of two. This leads to a large number of avoidable cache misses as a disproportionate amount of accessed addresses map to the same cache set which will need to evict entries while other cache sets still had room.

The non-contiguous jumps caused by implicit tile blocking can also cause performance issues if the jumps move across different virtual memory pages. These jumps can cause poor translation look-ahead buffer (TLB) usage (Park, Hong, and Prasanna 2003). Standard virtual memory pages each cover a 4KB range of addresses, which can hold 1024 32-bit float values contiguously. This means if the inner-most dimension of a matrix contain over 1024 values, when we make a non-contiguous jump by iterating on one of the outer dimensions, we will be moving to a new virtual page. This will occur every time we iterate an outer dimension which occurs significantly more often when using implicit tile blocking. If the chosen data blocks have more page-size jumps than there are entry slots in the L1 TLB, then it can cause severe TLB thrashing. This is because as we iterate through the final portion of any block $blk$, we will evict the older TLB entries that mapped the pages used by the earlier portions of $blk$ from the TLB. However when we move to block $blk+1$ we will need the earlier mappings again so the new TLB entries will be evicted even though they'll be needed next. This will repeat for all the data blocks causing very poor L1 TLB usage, where we must continually load virtual-physical page mappings from slower levels of the memory hierarchy. This problem can be mitigated by only selecting data blocks that contain few non-contiguous jumps, however this can lead to worse data reuse when compared to more square data blocks (Park, Hong, and Prasanna 2003).

## 2.3.2 Explicit Tile Blocking

Explicit tile blocking attempts to solve the problems that can arise using implicit tile blocking. Explicit tile blocking does this by copying all data in the current data block into a contiguous data buffer before the data is used (Daydé, Marques, and Nakajima 2013). By moving the data block into a contiguous buffer, we can expect the pressure each data block applies on the TLB to be reduced significantly (Park, Hong, and Prasanna 2003). This is because the contiguous buffer will only occupy the minimum number of virtual memory pages required to fit it in, unlike in implicit data blocking where each data block can be spread across a large number of virtual pages. Explicit tile blocking also removes the possibility of many data block values all mapping to the same cache sets thus creating avoidable conflict cache misses. This is because if the data is stored in a contiguous buffer, then as iterate across the data block the addresses will increment sequentially meaning the lower bits of the addresses will be the one changing most often. As the lower bits of the memory addresses select which cache set the addresses maps to, we can expect to see a much better distribution of the cache sets being used.

The main drawbacks of explicit tile blocking is that it adds an extra memory and computational overhead for the allocation of the memory buffer, and the time spent copying each block into the buffer before it's data can be used. Although, the size of the memory buffer is usually many times smaller than the size of the input and output data structures, therefore it's inclusion usually has very little impact on the total memory footprint of the problem. It is hoped that the computational cost of the data copying is counteracted by increased cache performance caused by data blocking, while also avoiding the possibly performance impediments that implicit tile blocking can cause.

Explicit data blocks can also be used to represent the output data structure as well. Using an intermediate output buffer as an accumulator can significantly reduce the number of writes required to the full output data structure, with most output writes instead writing to a buffer that should be fully contained in one of smaller faster caches.

Further background information required for specific chapters is introduced at the start of each chapter. We now move from general background information to general information about tools used during research and test set-ups.

# Chapter 3

# Tools and Experimental Set Up

## 3.1 Tools and Libraries

### 3.1.1 Intel SIMD Libraries

The Intel Streaming SIMD Extensions(SSD) are a set of instruction extensions for Intel x86 and Intel x64 architectures that implements SIMD vector functionally. We used the FM3 SSD extension during experimentation. AVX1 is an SSD extension that introduced 256-bit floating-point SIMD registers to Intel architectures, and AVX2 expanded the number of operations available, especially vector reordering operations (Intel 2016). AVX2 also introduced 256-bit integer SIMD registers (Intel 2016). FMA3 is a minor extension of AVX2 that adds a number of tertiary operand fused-multiple-add instructions.

GCC gained full FMA3 support with GCC version 4.7.0 in June 2012 (Team 2019). Every instruction extension exposes a set of new x86 assembler instructions, however the SIMD functionality can be more easily accessed through the Intel SIMD intrinsic libraries that Intel produces (Intel 2019). Each intrinsic library maps the SIMD x86 instructions to C functions, and allows a programmer to use the Intel SIMD functionality, while removing unwanted micromanagement like controlling register allocation, which the C compiler will perform instead (Intel 2019). The intrinsic libraries were exclusively used to implement SIMD functionality on Intel architectures during research.

### 3.1.2 ARM Neon Libraries

SIMD functionality was introduced into ARM devices with the ARMv6 SIMD extension, but this was fully replaced with the NEON extension in ARMv7 in 2009. The ARMv7 NEON extension included a set of 32 128-bit programmer-facing vector registers along with a SIMD instruction set to manipulate them. With the release of ARMv8 and AARCH64, the NEON architecture was extended with a number of new instructions and many other instructions were improved so they could also act on 128-bit registers, rather than just 64-bit registers. It also introduced 64-bit scalar values to NEON, as well as making NEON a standard extension rather than an optional one. Like Intel, ARM has also produced a number of C intrinsic libraries that allow low-level access to SIMD functionally while removing the need to manually allocate registers. ARMv7 NEON was used during testing on ARMv7 devices, and ARMv8 NEON was used when testing ARMv8 or AARCH64 devices.

### 3.1.3 PAPI

The Performance Application Programming Interface (PAPI) API was used for gathering performance metrics on ARM processors (Müller et al. 2010). PAPI exposes an API for measuring any performance metric, however the actual metrics available depend on the hardware being used.

### 3.1.4 ARMCL: ARM Compute Library

The ARM Compute Library (ARMCL) was used as the baseline library for comparing the performance of different methods on ARM processors. ARMCL is Machine Learning Library developed by ARM for the ARM Cortex-A family of CPU processors. It contains optimized implementations for a number of standard CNN convolution implementations, such as direct convolution, im2col convolution, and Winograd CNN convolution (ARM 2019b).

### 3.1.5 MKLDNN Library

The Micro-Kernel Library Deep Neural Network (MKLDNN) library was used as a baseline library for comparing the performance of different methods on Intel processors. MKLDNN is a open-source performance library for deep learning applications developed by Intel for Intel CPU and GPUs (Intel 2018).

### 3.1.6 TriNNity Library

The TriNNity library was used as a baseline library for comparing the performance of different methods on Intel processors. TriNNity is a low-level machine learning library developed at Trinity College Dublin (Anderson 2019). TriNNity contains optimized implementations for a number of standard CNN convolution implementations, such as direct convolution, and im2col convolution.

### 3.1.7 Miscellaneous Tools and Libraries

All generated code was written in C++17. GCC 9.1 was used to compile all generated source code for Intel devices, and GCC 9.0.0 was used to cross-compile all generated code for ARM devices. GHC 8.5.6 was used to compile all Haskell source code used. The thesis was written in Latex using TeXworks and TexStudio as Latex IDEs. GnuPlot was used to produce all graphs shown in the thesis. Inkscape 0.9.3 was used to draw all diagrams shown in the thesis.

## 3.2 Test Machines

### 3.2.1 ARM

Two ARM test machines were used during research to collect metrics between the baseline implementations and the implementations created during research.

The first test machine was the ODROID XU3 produced by HardKernel. It has an ARM armv7l big.LITTLE CPU arrangement with a Samsung Exynos5422

Cortex-A15 2.0Ghz quad-core 'big' CPU and a Cortex-A7 quad-core 'little' CPU. All experiments were pinned to a single core on the A15 CPU. The device has 2GB of LPDDR3 RAM. The device has the ARMv7 NEON SIMD instruction extension. The device also has energy monitoring hardware which can collect energy readings for each CPU and DRAM in parallel. We used ARCH Linux 2018.08.1 4.14.127 as the operating system on this machine during experimentation. This machine is referred to as *ARM Target 1* in all results section.

The second test machine was the Nvidia Jetson TX1 Developer Kit. It has a Quad-core ARM Cortex-A57 MPCore CPU ARMv8 Processor. The device has 4GB of LPDDR4 RAM. The device has the AARCH ARMv8 NEON SIMD instruction extension. All experiments were pinned to a single core on the A57 CPU. We used Ubuntu 18.04.2 LTS as the operating system on this machine during experimentation. This machine is referred to as *ARM Target 2* in all results section.

### 3.2.2 Intel

To collect metrics for Intel Architecture performance, the following target machine was used.

The target machine had a 64-bit Sandybridge Intel Core i7-2600 eight-core CPU. All cores ran at 3.4GHz. Each core had a 32KB private instruction L1 cache, a 32KB private data L1 cache, a private 256KB L2 cache, and a shared 8192KB L3 cache. The machine used Linux Arch-Linux 2019.06.01 x86_64 as an operating system with the 5.0.4-arch1-1-ARCH Linux kernel. This machine is referred to as *Intel Target 1* in all results section.

We have now completed covering general information, and are ready to begin examining the research conducted for this thesis. The next chapter introduces Genvolution, an automatic program generator for direct CNN convolution.

# Chapter 4

# Automating the Search for CNN Convolutions

## 4.1 Chapter Motivation

Most CNNs spend the majority of their execution time performing CNN convolution. Therefore, by improving the performance of CNN convolution, we improve the performance of many CNNs. However, optimizing any operation is a time consuming task, this includes CNN convolution. Also, optimizing CNN convolution for one machine architecture does not guarantee that it will run efficiently on other machine architectures.

Direct CNN Convolution implements CNN convolution directly (hence the name), using a nest of loops to move across the input tensors. However, it is very common to instead implement CNN convolution by mapping it to other problems that already have optimized solutions on most machine architectures. The *im2col* algorithm maps CNN convolution to matrix multiplication (Vasudevan, Anderson, and Gregg 2017), and CNN convolution is also often mapped to fast Fourier transform (FFT) operations (Abtahi et al. 2018). Optimized libraries for performing matrix multiplication and the FFT exist for most machine architectures. This produces fast CNN convolutions that can be more easily ported without spending a large amount of time optimizing CNN convolution directly.

However, mapping CNN convolution to other methods has two major drawbacks. Firstly, it introduces the computational overhead of transforming the inputs and outputs of the CNN convolution to and from the intermediate problem domain. Secondly, it introduces a large memory overhead to store the inputs and outputs in the intermediate problem format. For example, the im2col algorithm requires $(H \times W \times C \times K \times K)$ extra values to store a transformed version of the input tensor (Vasudevan, Anderson, and Gregg 2017).

We investigate the use of a domain-specific program generator that automatically generates and optimizes direct CNN convolution implementations. Direct CNN convolution does not require the memory overhead of indirect CNN convolution implementations, like im2col. The generator can automatically optimize direct CNN convolution implementations for major modern Intel architecture (e.g. SandyBridge, Haswell, SkyLake), and any ARMv7 or ARMv8 architecture with the NEON SIMD hardware extension. Our aim is to automate the production of efficient CNN convolution code that does not require a large temporary memory overhead. Also, automatic code generation allows us produce a unique code implementation for every layer in a network, opening the possibility to tailor optimizations for all expected input tensor sizes.

## 4.2 Previous Work during Final Year Project

### 4.2.1 General Overview

Our undergraduate final year project focused on the creation of a program generator to try and optimize direct CNN convolution on Intel architectures. A large part of work for this thesis was spent addressing the weaknesses of the previously created program generator, as well as adding significant modifications and more advanced features. The previous program generator viewed direct CNN convolution in two sections: an outer macro-program, and an inner microkernel. The macro-program would implement data transformations and large-scale control flow, while the microkernel would be used by the macro-program to implement the hottest sections of the generated convolution implementation. Structuring convolution like this was based off the BLIS framework developed at the University of Texas, which is used to optimize similar matrix-matrix mathematical problems (Zee and Geijn 2015). The generator used a brute-force method to find performant CNN convolution implementations. Given the dimensions of the convolution to optimize, the generator would create a large number of varied implementations. Each implementation would use different macro-program techniques, different microkernels, different data layouts, and other different alterations to try and find performant implementations. The performance of each implementation would be measured to find the most optimal.

### 4.2.2 Strengths and Weaknesses

The previous program generator was able to generate a large search space of possible CNN convolution implementations. It had the ability to produce valid convolution implementations with arbitrary loop ordering, arbitrary data layouts, and was able to generate convolution implementations that used implicit loop blocking and explicit loop blocking with arbitrary block sizes. However, it also had a large number of limitations. As the microkernels used for the innermost loops were hand-written, the generator had no ability to manipulate and permute the most important code used in it's generated implementation. Also, the generator could produce implementations that only worked with the Intel AVX2 architecture. This was due to the fact that the hand-coded microkernels used AVX2 intrinsics to implement SIMD functionality. The fact that the micro-kernels were mostly immutable limited the number of implementation variations that could be generated.

### 4.2.3 Results

The original program generator was mostly successful. Using it, we were able to find a number of direct CNN convolution implementations that used dramatically less memory than a normal im2col CNN convolution, while also outperforming an im2col CNN convolution for a number of input sizes.

## 4.3 New Generator Overview

The new program generator (from here on referred to as 'Genvolution') addresses the problems present in the old generator. Genvolution is also much more flexible, and can be used in a number of new ways (such as matrix multiplication generations (section 7), and using irregular datatypes (section 8)). Most importantly is that Genvolution produces the entire generated solution in an intermediate representation first. this allows Genvolution to permute and modify any part of the implementation before it is converted into a concrete C++ implementation. This is in contrast to the old generator, which could manipulate only parts of the outer loops, and none of the microkernel inner loops in any code it generated. By representing the entire problem in the intermediate representation first, Genvolution can be used to investigate the optimization potential of a number of techniques. Genvolution can also produce C++ code for any SIMD architecture given a mapping that maps symbols in the intermediate representation to concrete SIMD architecture instructions. The other main difference is that Genvolution does not use solely brute force to find good optimizations. Instead, Genvolution will build a model of the entire search space of possible optimizations. Genvolution can use this model to perform different optimization strategies, fine grain search space control, and pruning optimizations based on heuristics or past performances.

## 4.4 Genvolution Workflow

Genvolution breaks the process of optimizing a given CNN convolution into a number of steps.

1. Genvolution collects a set of values that describes the problem to be implemented. The set members are taken from command line arguments

and/or configuration files. Example set members include the sizes of the input and output tensors, and the target architecture. This set is referred to as the *problem description*.

2. Genvolution constructs a model of the entire search space of possible valid implementations. The search space is represented as a lazily evaluated tree.

3. The problem description tells Genvolution what traversal algorithm it should use to traverse the search space. With the given traversal algorithm, Genvolution will select a set of implementation parameters from the search space.

4. The set of implementation parameters describes one concrete implementation of the CNN convolution that we wish to optimize. An intermediate representation of the implementation is generated using the set of parameters.

5. Genvolution applies a set of optimizations to the intermediate representation. An example of this is inserting software prefetching operations. The optimizations to apply can be set in the problem description.

6. A concrete C++ implementation is generated from the intermediate representation.

7. The generated C++ implementation is benchmarked on the target machine, and the collected performance statistics are stored with the set of implementation parameters used to generate the implementation.

8. Genvolution returns to step three of the workflow, and selects another set of implementation parameters. This is repeated for a fixed number of iterations.

9. Based on the performance of previous implementations, Genvolution may manipulate the search space data structure. An example of this is reduc-

ing the search space to parameters that have only appeared in fast implementations so far.

## 4.5 Traversing the Search Space

### 4.5.1 Building the Search Space

After producing the *problem description*, Genvolution creates a search space that represents all possible valid implementations of the problem to optimize. The search space is dependent on the type of convolution being requested and certain input parameters (such as target architecture). The range of valid values for a search space parameter can be given as a command line argument (such as: *-layout image data {HWC, CHW}*), which would limit the possible layouts for the input image tensor to either $(H \times W \times C)$ layout or $(C \times H \times W)$ layout. Parameter ranges can also be listed in a configuration file. For parameters where no explicit value/range of values is requested, Genvolution will use heuristics to estimate 'reasonable' ranges (such as default possible data layouts, or calculating 'reasonable' blocking factors based on SIMD lane lengths and tensor dimension lengths). For a standard CNN convolution optimization, there are between twenty and thirty parameters in the search space to be selected for each implementation.

### 4.5.2 Optimizing using the Search Space

To represent the search space, Genvolution uses a lazily evaluated decision tree. The tree is $pn$ nodes deep, where $pn$ is the number of parameters to be chosen before an implementation can be generated. Every layer on the tree represents a different parameter to select. For example, the sixth layer may represent the different data layouts that can be used for the input image tensor. Every node in the fifth layer has a child for each possible selection in the sixth layer, and every node in the sixth layer will have a child for every value in the seventh layer. A path from the root of the tree to a leaf node represents one full set of parameters that can be used to generate a unique implementation of the given problem. Figure 4-1 shows a simple search space tree that represents the search space for the input and output tensor data layouts.

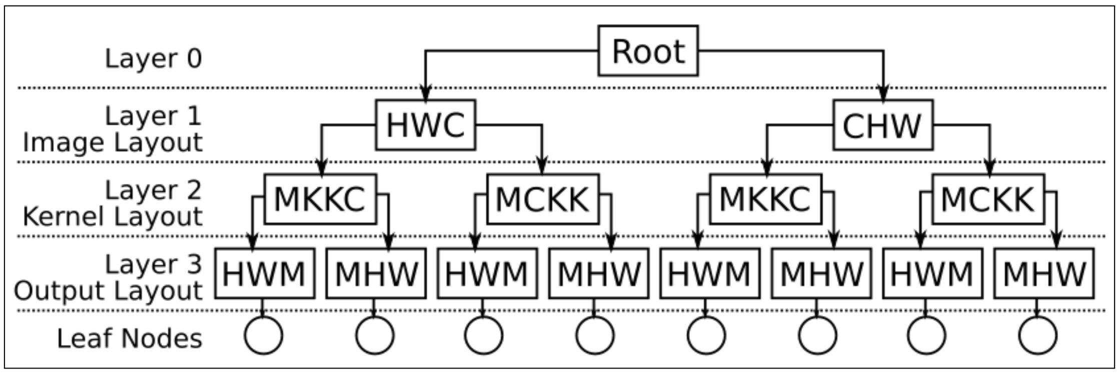


Figure 4-1: Simplified tree representing the implementation search space. Each path from the root to a leaf is a unique set of parameters.

Genvolution continually traverses the created search space to generate a variety of implementations for the given problem. However, the total search space is extremely large, even when we restraint many parameters of the search space (e.g. fixing the input and output data layouts). For example, there are six independent for loops in direct CNN convolution that can be nested in any configuration. This creates $6!$ possible orderings. There also $3!$ possible data layouts for the input image tensor, $4!$ possible layouts for the input kernel tensor, and $3!$ for the output image tensor. This gives us 622,080 ($6! * 3! * 4! * 3!$) possible convolution implementations only from selecting the data layouts and the order that dimensions are traversed. When we include more parameters such as data blocking sizes, unrolling factors, and SIMD strategies, it is common to have over ten million theoretically valid convolution implementations to select from.

To limit how many implementations must be generated to meaningfully traverse the search space, Genvolution makes use of parameter dependent heuristics that apply weights to the search space parameters. These weights influence how the search space is traversed. Parameter dependent heuristics are heuristics that effect the selection of future parameters based on the parameters that have already selected. For example, if the image input tensor layout chosen for a given implementation is $(H \times W \times C)$, but the loop ordering parameters have yet to be chosen, then the loop ordering parameters that work well with a $(H \times W \times C)$ data structure are weighted more favourably. Parameters that contradict the heuristics' suggestions can still be chosen, but it is less likely to occur. Search space parameters can also be pruned if they contradict a parameter that has already been selected. For example, certain SIMD vector parameters only work with certain input data layouts.

The decision tree representing the entire search space tree is extremely large and constructing it would be very expensive. The tree is instead lazily evaluated, and only the nodes of the tree that have been traversed to are allocated. The weight of a node edge is calculated only when it is needed. Each leaf node contains the collected performance information of the implementation that the path to the leaf node represents.

## 4.6 Creating Parameter Sets

Every path from the root node of the search space tree to a leaf node will contain all the parameters needed to generate a unique direct CNN convolution implementation. Genvolution will continually create paths through the tree from the root to a leaf, and generate the corresponding CNN convolution. Genvolution has a number of different algorithms for traversing the search space to try and find the most performant implementations for the given model.

#### Pure Random

The search space tree is traversed randomly (excluding the node weighting from the heuristics) for a fixed number of iterations .

#### Genetic Selection

1. The search space tree is traversed randomly (excluding the node weighting from the heuristics) for a fixed number of iterations.

2. A subset of the best performing implementations is collected. A new search space is created only using parameters that appeared in the implementations in the created subset. The size of the subset is configurable.

3. The first two steps are repeated a set number of times.

The number of iterations in step 1 and step 3 are configurable.

**Narrowing Search**

1. The search space tree is traversed randomly (excluding the weighting from the heuristics) for a fixed number of iterations.

2. A subset of the best performing implementations is collected. For each implementation in the subset, a fixed number of new implementations are produced that are locally similar (i.e. they share many of the same parameters). How many parameters must be at least shared between 'similar' sets is set in the problem description.

3. Step 2 is repeated for a fixed number of iterations.

## 4.7 Intermediate Representation AST

When a path from the root to a leaf node of the search space tree has been selected, the set of parameters contained in the path are used by Genvolution to generate a unique CNN convolution implementation. The implementation is first constructed in an intermediate representation. The intermediate representation is represented with a simplified abstract syntax tree (AST). Each *AST node* of the AST represents a basic C++ statement (e.g. a variable declaration), C++ construct (e.g. a for loop), or generic SIMD operation (e.g. loading a SIMD vector). There is also a global data table shared between the nodes in the AST which is used to share information between *AST nodes*, such as the names and datatypes of variables that exist in the code the AST represents.

Figure 4-2 shows an simplified example of the AST. Line 1 shows a *DefinitionNode*. *DefinitionNodes* represent all C++ variable declaration and definition

```
1 DefinitionNode("sum", global.baseType, 0),
2 ForNode(global.dimens.C, global.baseType, unroll.C {
3   UnrollNode(unroll.C, {
4     DefinitionNode("ti", global.baseType),
5     DefinitionNode("tk", global.baseType),
6     LoadNode("ti", locals.image),
7     LoadNode("tk", locals.kernel),
8     FMANode("sum", "ti", "tk"),
9   })
10 })
11 DefinitionNode("sumScalar", type.scalar),
12 HorizontalAdd("sumScalar", "sum"),
13 StoreNode("sumScalar", locals.output, type.scalar)
14
```

Figure 4-2: Simplified example of the AST.

statements. They also create an entry in the global table for the variable they declare, which can be accessed using the variable's name. *DefinitionNodes* take the name of the new variable, the type of the variable, and optionally a value to initialize it to. The value *global.baseType* represents the variable type the convolution is based around (e.g. scalar floats, 4-lane SIMD vectors). *ForNodes* represent for loops. Line 2 of figure 4-2 shows a specialized *ForNode* constructor which takes the dimension it will loop over as it's first parameter (in the figure, this is the input channels dimension). The second parameter is used to calculate how large the increment step should be. the third parameter (*unroll.C*) is a reference to the unrolling table. The unrolling table is part of the global table and is used to control loop unrolling. Line 3 shows an *UnrollNode*. *UnrollNodes* are used to unroll loops by duplicating it's child nodes. It also manipulates the duplicated nodes so there are no variable name clashes and memory accesses behave correctly. Lines 6 and 7 show *LoadNodes*. These are used to generate memory reads. In reality, they also take a small tree structure representing the equation needed for the memory access, but this has been removed from figure 4-2 for readability. *LoadNodes* determine if a scalar load, or a SIMD load is needed based off the type of the variable they are writing to. An *FMANode* is used to generate code that multiplies parameter 1 and parameter 2, and sums the result with parameter 0. *HorizontalAdd* nodes generate code that sums all the values across the lanes of SIMD vector, and then store the results in the scalar variable passed in as parameter 0 to the *HorizontalAdd* node. *StoreNodes* are the memory writing equivalent of *LoadNodes*. Figure 4-3 and figure 4-4 show two simplified examples of code that figure 4-2 could generate depending on other parameters.

```
float sum = 0;
for (signed c = 0; c < CHANNELS; c++) {
  float ti;
  float tk;
  ti = image[h-(y/2)][w-(x/2)][c];
  tk = kernel[m][y][x][c];
  sum += ti * tk; //FMA
}
float sumScalar;
sumScalar = sum; //horizontal Add
output[m][h][w] = sumScalar;

```

Figure 4-3: Scalar code generated from figure 4-2 with no unrolling.

```
float sum = 0;
for (signed c = 0; c < CHANNELS; c+=4*2) {
  //unroll 0
  float32x4 ti_0;
  float32x4 tk_0;
  load_32x4(ti_0, &(image[h-(y/2)][w-(x/2)][c+4*0]));
  load_32x4(tk_0, &(kernel[m][y][x][c+4*0]));
  fma_32x4(sum, ti_0, tk_0);
  //unroll 1
  float32x4 ti_1;
  float32x4 tk_1;
  load_32x4(ti_1, &(image[h-(y/2)][w-(x/2)][c+4*1]));
  load_32x4(tk_1, &(kernel[m][y][x][c+4*1]));
  fma_32x4(sum, ti_1, tk_1);
}
float sumScalar;
horz_add_32x4(sumScalar, sum);
output[m][h][w] = sumScalar;

```

Figure 4-4: SIMD code generated from figure 4-2 with the loop unrolled one time.

## 4.8 Constructing the AST

### 4.8.1 Microkernels

The convolution implementations generated by Genvolution are based off the program design used by the BLIS framework (Zee and Geijn 2015). The BLIS framework implements matrix-matrix operations by splitting program implementations into highly optimized microkernels that implement the hottest loops of an implementation, and macro-programs that implement the outer loops of an implementation and make use of the microkernel.

Genvolution can generate a number of different microkernels to implement CNN convolution. The microkernels do not all perform the same action (e.g. some microkernels implement a dot product between two vectors, some perform an outer product between two small fixed size vectors). However, Genvolution is still able to generate a valid direct CNN convolution implementation no matter what microkernel is selected. The differences between the microkernels available are covered in detail in chapter 5.

All the microkernels are hand-written as a Genvolution AST. During the traversal of the search space tree, a microkernel will be selected to use for the current implementation. The AST of the selected microkernel becomes the first part of the AST that will represent the entire CNN convolution implementation.

### 4.8.2 Dimension Loops

The algorithm for direct CNN convolution involves a number of loops that iterate across the dimensions of the input tensor and the input kernel. These loops need to be accounted for in any direct CNN convolution. After selecting the microkernel to use as the start of the implementation AST, Genvolution inserts the necessary loops into the AST. The *loop list* contains all the loops that still need to be inserted into the AST to fully implement direct CNN convolution. The loops are inserted through the following steps:

1. There are six major loops in total (one for the $M$, $H$, $W$, $Y$, $X$, and $C$ dimensions). These loops are added as the first elements to the *loop list*.

2. Genvolution can produce blocked and unblocked convolution implementations. Dimensions that have been blocked must be split into an *outer loop* that moves between blocks, and an *inner loop* that iterates inside the current block. Which dimensions to block is selected when traversing the search space tree. The dimensions that are blocked are removed from the *loop list* and replaced with two loops, one for the inner loop and the outer loop of the blocked dimension.

3. The selected microkernel may contain loops that duplicate loops in the *loop list*. For example, a microkernel may already iterate across the *input channels* dimension of the input tensors, so this loop already exists in the implementation and does not need to be inserted again. The AST of the microkernel is traversed and any loops that also exist in the *loop list* are removed from the loop list. If a dimension has been split into an *inner* and *outer loop*, then only the *inner loop* is removed. The loop nodes in the AST of the microkernel are updated according to what loops they replaced from the loop list (i.e. did they replace an *inner loop* that only iterates across a block, or a *full loop* that iterates across an entire dimension).

4. The *loop list* is reordered to match how the loops will be nested when added to the AST (and how they'll be nested in the generated code). The ordering of the loop is selected when traversing the search space tree. The edges of the search space tree are weighted by a heuristic such that the ordering of the loops are more likely to match the data layout of the input and output tensors. For example, if the input image tensor has the layout $(H \times W \times C)$, then it will be more likely that the loop that moves across $W$ is nested inside the loop that moves across $H$. Nesting dependencies between inner and outer loops are also preserved during reordering, as an inner loop must be nested within it's corresponding outer loop. Also, while not technically needed, having any outer loops nested inside an

inner loop (even if the loops move across different dimensions) leads to very poor performance, so this is also disallowed.

5. The *loop list* is converted into a nest of AST *ForNodes*, with the root node of the AST for the microkernel connected as a child node to the innermost loop from the *loop list*.

### 4.8.3 Inserting Extra Constructs

After the major loops have been added to the convolution AST, a number of other constructs are inserted into the AST. Genvolution is used to generate zero-padded CNN convolution. Zero padding implementations assumes if we try to access an out-of-bounds point from the input tensor, that point has the value zero. A simple way of implementing this in practice is to skip all computations which involves data from outside the dimensions of the input image tensor, because convolution is built from multiplication and multiplication with zero produces zero.

Genvolution inserts up to four conditional statements that check if the implementation is currently performing calculations outside the dimensions of the input image tensor. The four statements check if 1) The implementation is underflowing on the $H$ dimension, 2) The implementation is overflowing on the $H$ dimension, 3) The implementation is underflowing on the $W$ dimension, and 4) The implementation is overflowing on the $W$ dimension. Each condition is inserted into the AST as a *IfNode*. Each conditional statement is inserted at the outermost valid point in the AST, so that it lead to the largest amount of code being skipped when it is false. To do this, Genvolution traverses the AST and tracks what information is available at any point in the AST. It then inserts each *IfNode* at the earliest possible point.

Not all four conditional statements must be inserted into the AST. Before inserting the *IfNodes*, Genvolution inspects the AST of the microkernel to check if the microkernel already handles any of the four conditional statements. Also, if the $H$ dimension or the $W$ dimension are being blocked using explicit tile blocking, then the conditional statements relating to them are unneeded. This is because the generated code for explicit tile blocking already handles padding by explicitly padding data in the created tile buffers.

If explicit tile blocking is being used, this is also the point in AST construction when the AST nodes for copying data to the tile buffers is inserted. The nodes for copying data are also inserted at the outermost point in the AST, so the data copying is performed the minimal number of times.

### 4.8.4 AST Manipulators

The final step of AST construction is to apply a list of AST manipulators that transform the AST. Each manipulator walks the entire AST, and checks each node against a predicate. If that predicate is true for a given node, it performs some transformation to the AST at the node's position. An example manipulator is the prefetching manipulator that walks the AST, and inserts software prefetching AST nodes before some or all memory access nodes. The set of AST manipulators is decided by the parameters selected when traversing the search space tree, and/or the *problem description*.

## 4.9 Generating C++

To convert the AST to concrete C++ code, two transformation stages are used. The first stage uses a set of node translators that perform lowering operations on the AST nodes. Lowering operations replace complex nodes with trees of simpler nodes. The second stage convert the nodes in the AST to C++ snippets. The C++ snippets are then concatenated to produce the final C++ implementation.

The first transformation stage used a set of AST transformers called the *lowering set*. The *lowering set* contains a number of *lowering transformers*. Each *lowering transformer* contains a predicate which checks if an AST node can be lowered by said transformer, and a function to perform a given lowering operation. The predicate stored in each *lowering transformer* checks the type of the AST node, as well as the data stored inside, and variables it references (e.g.

```
DefinitionNode("sum", global.baseType, 0)

```

(a) AST node for a variable definition.

```
DeclarationNode("sum", global.baseType),
AssignmentNode("sum", 0)

```

(b) DefinitionNode from (a) lowered by a SIMD transformer.

```
//DeclarationNode("sum", global.baseType)
ConcatNode({
  TypeNode(global.baseType),
  VarNode("sum"),
  SemicolonNode()
}),
//AssignmentNode("sum", 0)
ConcatNode({
  FunctionNameNode(global.baseType, funcs.simd_set),
  BracketsNode({
    CommaListNode({
      VarNode("sum"),
      LiteralNode("0")
    })
  }),
  SemicolonNode()
})

```

(c) AST nodes from (b) lowered by a SIMD transformer.

Figure 4-5: Lowering a DefinitionNode twice using a SIMD transformer.

does the node reference SIMD or scalar variables). There are general *lowering transformers*, and specific *lowering transformers* for each hardware architecture. If we have a *FMA Node* that acts on three 4-lane SIMD variables, and we are generating C++ code for the *ARMv7 NEON* architecture, then we will use the NEONv7 *lowering transformer* to lower the *FMA Node*. If the *FMA Node* acted on scalar variables instead, then we would use the scalar *lowering transformer*.

The lowering operation transforms an AST node into a tree of simpler AST nodes. For example, the *DefinitionNode* in part (a) of figure 4-5 is transformed into two simpler AST nodes. However, the lowering operation can also transform AST nodes into tree of extremely simpler nodes. For example, each AST node in part (b) of figure 4-5 represent a full C++ statement, but in part (c) they are transformed into AST nodes which each represent much simpler constructs. The lowering operations is also be used to synthesize generic SIMD operations that are not available on all architectures. For example, there is no single instruction in the *Intel AVX* SIMD architecture to perform a horizontal add of a 256-bit vector register containing 8 32-bit floats. Instead, when a *HorizontalAdd Node* representing this operation is lowered, it is replaced with an AST representing a block of instructions that performs the operation in software.

The AST is repeatedly traversed, and every node in the AST is checked by every *lowering transformer* to see if it can be lowered. If the AST is fully traversed with no lowering operations performed, then Genvolution moves onto the next step of transformation.

When the lowering phase is complete, the AST will no longer contain AST nodes that represent full statements or operations, instead the AST will be built from much simpler nodes that each represent only a fragment of code. Part (c) of figure 4-5 shows an simplifed example of these nodes. The *TypeNode* represents a datatype type, the *ConcatNode* represents the idea concatenating the code snippets of it's child nodes, the *LiteralNode* represents a literal value (e.g. a string literal, an integer value, etc). The simpler nodes are transformed into C++ code snippets, and concatenated in a depth-wise first order to construct the final C++ implementation. The transformations are usually very simple. For example, the *FunctionNameNode* node is a set of table look-ups to find the name of the function it represents.

## 4.10 Evaluation of Results

To evaluate the effectiveness of Genvolution, Genvolution was used to generate direct convolution implementations for a set of CNN convolution input sizes for three target machines (ARM Target 1, ARM Target 2, and Intel Target 1). The CNN convolution input sizes were taken from five commonly used CNN networks: AlexNet (Krizhevsky, Sutskever, and Hinton 2012), Inception V4 (Singh and Markovitch 2017), MobileNet V2 (Howard et al. 2017), ResNET-

152 (He et al. 2016), and VGG ILSRVC (Bengio and LeCun 2015). A unique convolution implementation was generated and optimized for each input size on each target machine. The average execution time and cache miss rate for each generated implementation was collected on each target machine. The execution time and cache miss rates are the mean average of 20 runs. The average execution time and cache miss rate of a relative vendor library were also collected on each target machine to act as a baseline.

Execution time is given as a relative speed-up against a baseline method in the graphs in section 4.12. This means that higher is better for execution time graphs. Cache miss rates are given as the average measured rate. A rate of 0.0 denotes no cache misses, and a miss rate of 1.0 denotes that all cache accesses were cache misses. This means that lower rates are better in cache miss rate graphs.

### 4.10.1 ARM Target 1

The ARMCL library (see section 3.1.4) was used as the baseline library for evaluating the performance of direct convolution implementations generated by Genvolution for ARM Target 1.

For all input sizes, Genvolution-generated code (labelled 'Genvolution' in all graphs in section 4.12) executed faster than the direct CNN convolution implementation given by ARMCL (labelled as 'ARMCL-Direct' in all graphs in section 4.12). For 10 of the 29 input sizes, Genvolution-generated code also executed faster than the im2col CNN convolution (see section 2.1.4) given by ARMCL (labelled as 'ARMCL-im2col' in all graphs in section 4.12).

Genvolution-generated code outperformed ARMCL-Im2Col by the largest margin in the first input size of figures 4-7(a) (where $H$=224, $W$=224, $C$=3, $M$=32, $K$=3). Figure 4-13(a) and figure 4-19(a) show that the Genvolution-generated code had a much lower L1 and L2 cache miss rate than ARMCL-Im2Col for this input size. The lower cache miss rates may explain the large difference in performance. A similar pattern of Genvolution-generated code outperforming ARMCL-Im2Col can be seen in the first input sizes of figures

4-6(a) (where $H$=229, $W$=229, $C$=3, $M$=32, $K$=3), and figure 4-8 (where $H$=224, $W$=224, $C$=3, $M$=64, $K$=3). For all three of these input sizes, the number of input channels $C$ is very small. It is possible that ARMCL-Im2Col is optimized for larger $C$ values, and the matrix multiplication is not as efficient when the input matrices are very thin (the rows of the first matrix in the Im2Col matrix multiplication will only have 9 values ($K \times K \times C = 9$) for these input sizes).

Genvolution-generated code has the smallest relative speed-up compared to ARMCL-Direct in the 8th input size of figure 4-8 (where $H$=28, $W$=28, $C$=256, $M$=512, $K$=3). This may be due to fact that there is a larger relative number of synthesized values to handle when the height and width of an input tensor (i.e. $H$ and $W$) are small. An input tensor needs $H \times W$ explicit points and $H \times (K - 1) + W \times (K - 1)$ synthesized implicit edge points to produce a $(H \times W)$ output tensor using a $(K \times K)$ input kernel. This means the ratio between synthesized and explicit points is $(H \times (K-1) + W \times (K-1)) : (H \times W)$, which grows as $H$ and $W$ shrink. For example, $(112 \times 1 + 112 \times 1) : (112 \times 112)$ is 0.0179, while $(28 \times 1 + 28 \times 1) : (28 \times 28)$ is 0.0714. This means as $H$ and $W$ shrink, the less optimal edge-case code for handling synthesized values takes up more of the runtime, which effects performance. Figure 4-8 may support this idea as it somewhat shows a relationship between the relative performance of Genvolution-generated code and the size of $H$. However, this trend is not replicated in figure 4-7(a) or figure 4-6(b) so other factors may also be involved.

Table 4-1 shows that for all input sizes, Genvolution-generated code used significantly less memory than ARMCL-Im2Col. In many cases, Genvolution-generated code required no temporary memory, and in the worst case (where $H$=14, $W$=14, $C$=256, $M$=256, $K$=3), it only only required 0.7% of the temporary memory im2col requires.

### 4.10.2 ARM Target 2

The ARMCL library was used as the baseline library for evaluating the performance of direct convolution implementations generated by Genvolution for ARM Target 2.

For all input sizes, Genvolution-generated code executed faster than the direct CNN convolution implementation given by ARMCL. For 7 of the 29 input sizes, Genvolution-generated code also executed faster than the im2col CNN convolution given by ARMCL.

Similar to ARM Target 1, Genvolution-generated code outperforms ARMCL-Im2Col by a large margin in the first input size from Inception V4 (figure 4-9(b)), MobileNet V2 (figure 4.10(a)), and VGG ILSRVC (figure 4.11). The relevant L1 and L2 cache miss rate graphs show that Genvolution-generated code had lower L1 and L2 cache miss rates than ARMCL-Im2Col for the three mentioned input sizes. It is possible that the poor performance of ARMCL-Im2Col for these input sizes is again the fact that $C$ is very small, which ARMCL-Im2Col may not be optimized for.

Genvolution-generated code has the smallest relative speed-up compared to ARMCL-Direct in the first input size of figure 4-9(a) (where $H$=27, $W$=27, $C$=96, $M$=256, $K$=5) on ARM Target 2. Figure 4-21(a) shows that the Genvolution-generated code has a slightly higher L2 cache miss rate than ARMCL-Direct for the first input size. This may explain why the performance between the two is similar. Figure 4-15(a) also shows there is not a very large difference in L1 cache miss rates between the Genvolution-generated code and ARMCL-Direct. The first input size of figure 4-9(a) is also the only input where $K = 5$. A larger $K$ value will lead to more synthesized values being needed. Handling synthesized values requires less optimal edge-case code being run, which could effect the performance of Genvolution-generated code.

Table 4-2 shows that for all input sizes, Genvolution-generated code used significantly less memory than ARMCL-Im2Col on ARM Target 2. In many cases, Genvolution-generated code required no temporary memory, and in the worst case (where $H$=112, $W$=112, $C$=128, $M$=128, $K$=3), it only only required 0.7% of the temporary memory Im2Col requires.

### 4.10.3 Intel

The TriNNity library (see section 3.1.6), and the MKLDNN library (see section 3.1.5) were used as the baseline libraries for evaluation the performance of direct convolution implementations generated by Genvolution for Intel Target 1. In the figures in section 4.12.2, 'Direct' refers to the direct CNN convolution implementation given by the TriNNity library, 'Im2Col' refers to the Im2Col CNN convolution implementation given by the TriNNity library, and 'MKL' refers to the direct CNN convolution implementations given by the MKLDNN library.

Genvolution-generated code outperformed TriNNity-Direct for all 29 input sizes, it outperformed TriNNity-Im2Col for 3 inputs sizes, and it outperformed MKLDNN for 8 input sizes. Genvolution-generated code was the fastest method for 3 of the 29 input sizes. Two of the input sizes where Genvolution-generated code was the fastest were input sizes where $C$ was 3. This is similar to the results on ARM Target 1 and 2, and may be explained for the same reason. For every input where Genvolution-generated code outperformed MKLDNN, $C$ and $M$ were both relatively small values. Figures 4-25(a), 4-25(b), and 4-26 all suggest there being some relationship between the relative speed of MKLDNN and the ratio between $H \times W$ and $C \times M$. In these figures, as $H \times W$ shrinks and $C \times M$ grows, the relative performance of MKLDNN to the baseline grows. This suggests that MKLDNN may be better optimized for large input kernels than other methods, or that it can handle the synthesized values better (which become more common relatively as $H \times W$ shrinks).

Table 4-2 shows that for all input sizes, Genvolution-generated code used significantly less memory than ARMCL-Im2Col on Intel Target 1. In many cases, Genvolution-generated code required no temporary memory, and in the worst case (where $H$=28, $W$=28, $C$=256, $M$=512, $K$=3), it only only required 4.3% of the temporary memory Im2Col requires.

## 4.11 Conclusions

We believe our research into the use of automatic code generation to improve the performance and memory usage of CNN convolution has been successful. For 10 inputs on ARM Target 1 and 7 inputs on ARM Target 2, Genvolution-generated code outperformed a vendor library Im2Col implementation while needing less than one percent of the Im2Col's temporary memory. While we were not able to match the performance of Im2Col for every input size, we believe our results are still significant. Reducing the execution time and memory usage of just some CNN convolutions in a CNN network can have a major impact if that network is to be run many times. While extra work is required by the network programmer up front to find empirically which input sizes Genvolution performs well on, over the lifetime of the network's use, a small increase in performance can have a big impact.

It can also be difficult fitting larger networks on small resource-constrained devices (while still executing the network quickly enough). Therefore, even reducing the memory overhead of some CNN convolutions in a CNN network can make the difference in deciding if the network can fit on a resource-constrained device or not.

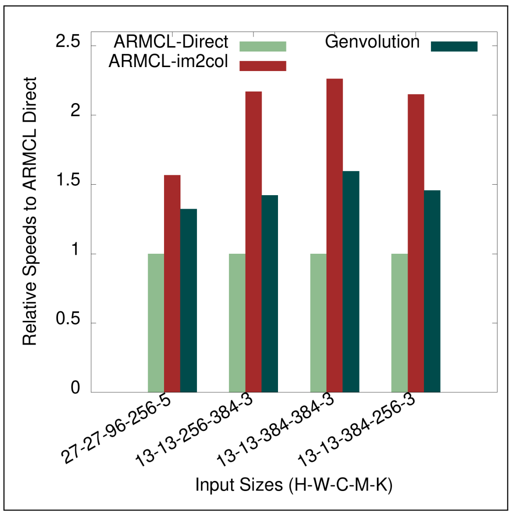
We believe that the success of Genvolution also suggests that the performance of other problems may be improved by using automatic code generation and optimization. For example, Genvolution was only used to optimize "standard" CNN convolution. However, there are many CNN convolution variants (such as dilated CNN convolution) that need unique implementations. Producing a hand optimized implementation for every CNN convolution variant would be very time consuming. Therefore, it may be advantageous to investigate automating the generation and optimization of CNN convolution variants.

Having completed outlining the design of Genvolution, the following chapter covers how the code generated by Genvolution is optimized using SIMD vectorization.

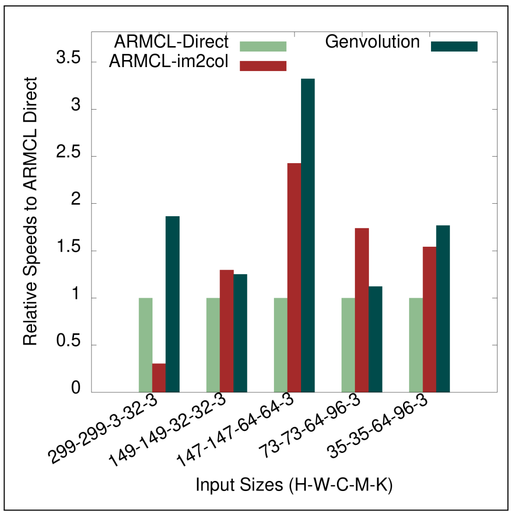
## 4.12 Results

### 4.12.1 ARM

#### Relative Execution Speed-Up

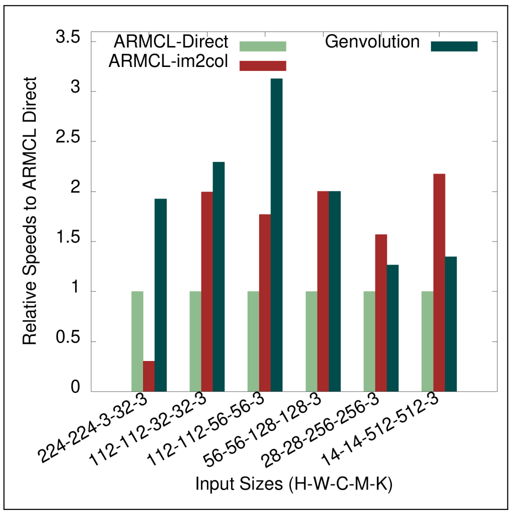


(a) AlexNet convolutions.

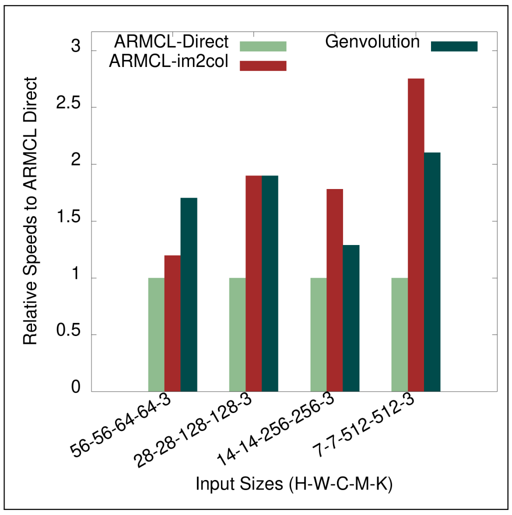


(b) Inception V4 convolutions.

Figure 4-6: Execution Time on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet V2 convolutions.



(b) ResNet-152 convolutions.

Figure 4-7: Execution Time on MobileNet V2 and ResNet-152 convolutions on ARM Target 1.

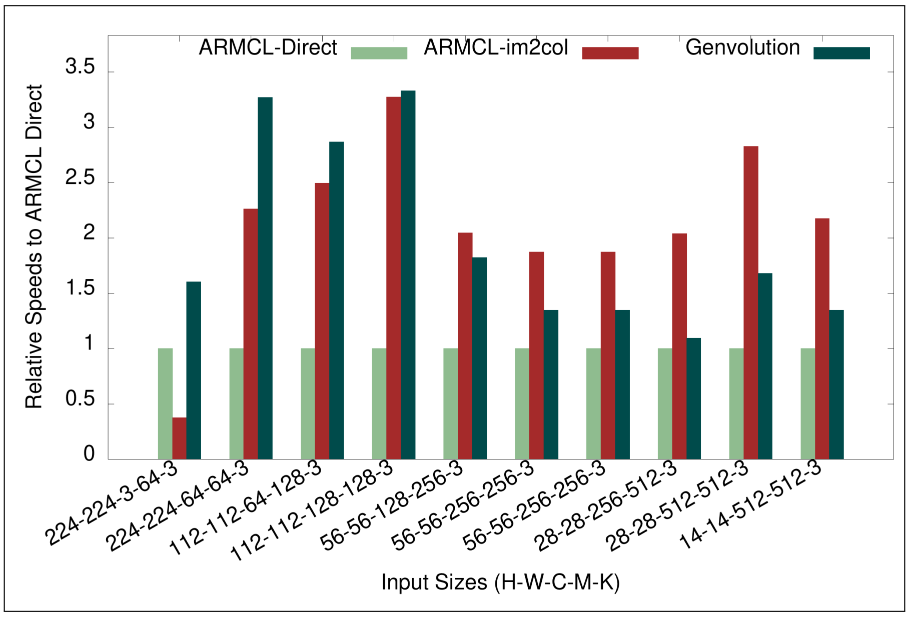
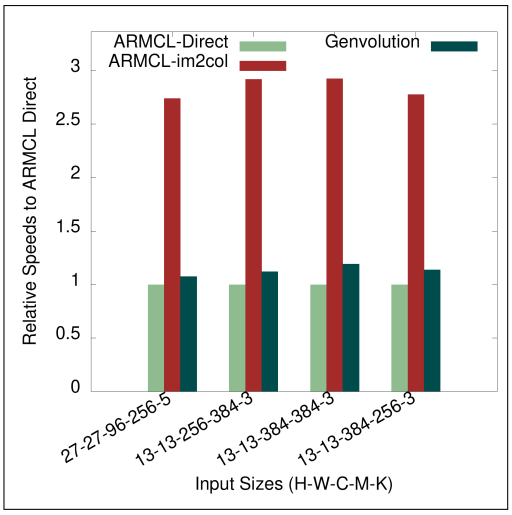
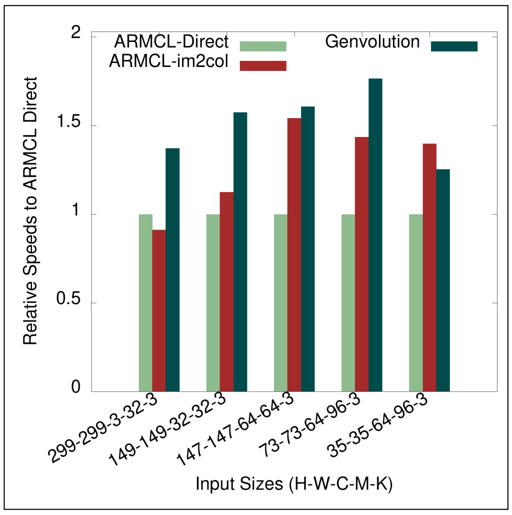


Figure 4-8: Execution Time of Genvolution implementations on VGG ILSRVC convolutions on ARM Target 1.

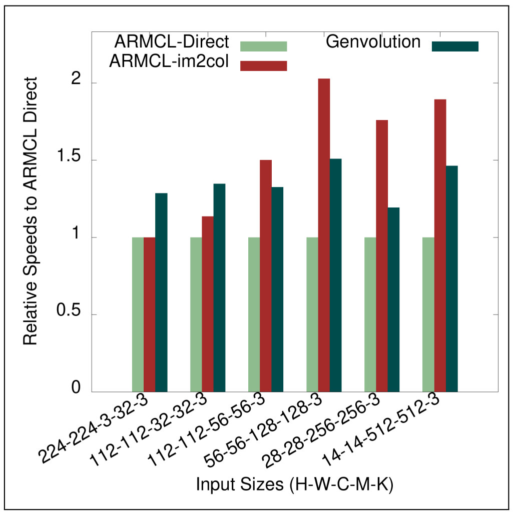


(a) AlexNet convolutions.

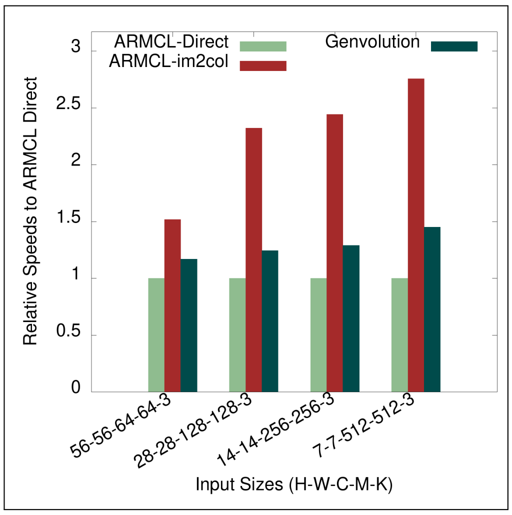


(b) Inception V4 convolutions.

Figure 4-9: Execution Time on AlexNet and Inception V4 convolutions on ARM Target 2.



(a) MobileNet V2 convolutions.



(b) ResNet-152 convolutions.

Figure 4-10: Execution Time on MobileNet V2 and ResNet-152 Convolutions on ARM Target 2.

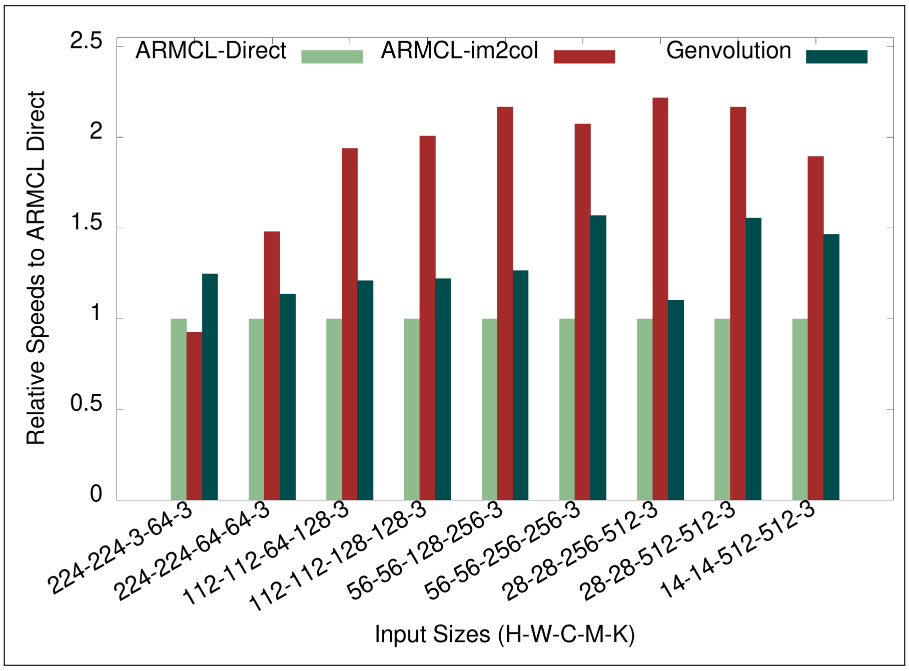
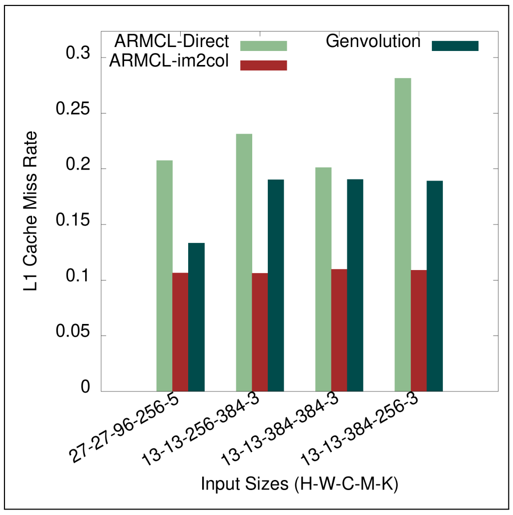
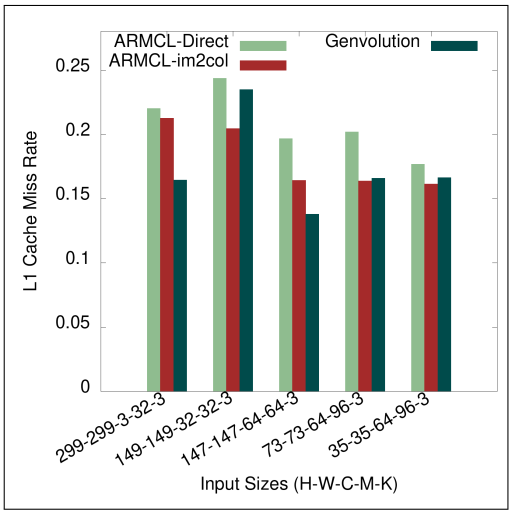


Figure 4-11: Execution Time of Genvolution implementations on VGG ILSRVC convolutions on ARM Target 2.

## L1 Cache Miss Rate

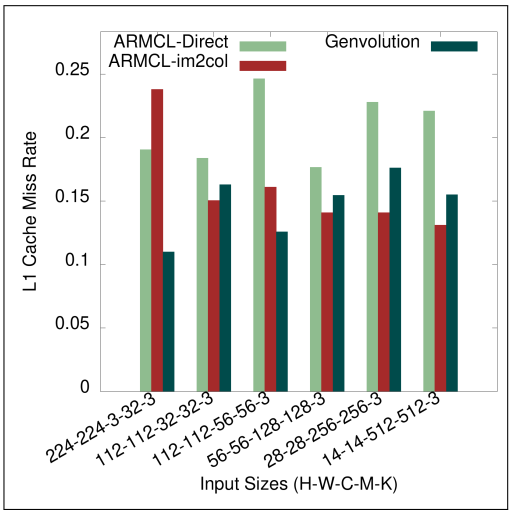


(a) AlexNet convolutions.

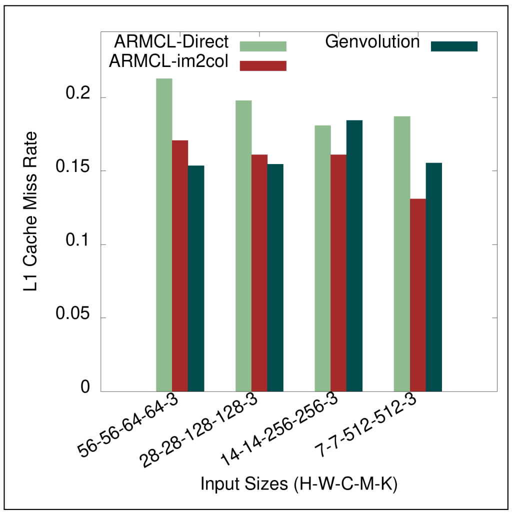


(b) Inception V4 convolutions.

Figure 4-12: L1 cache miss rate on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet V2 convolutions.



(b) ResNet-152 convolutions.

Figure 4-13: L1 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

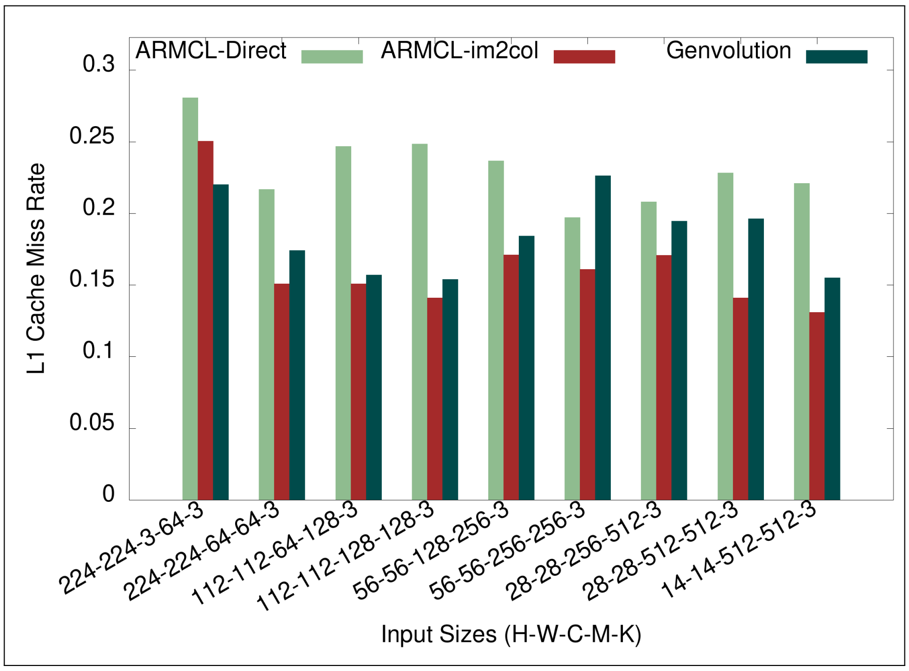
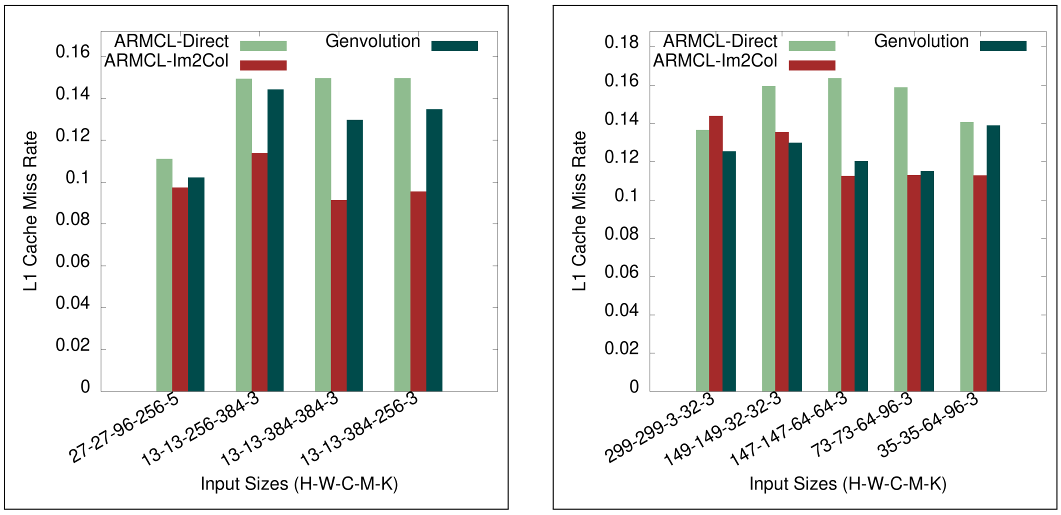
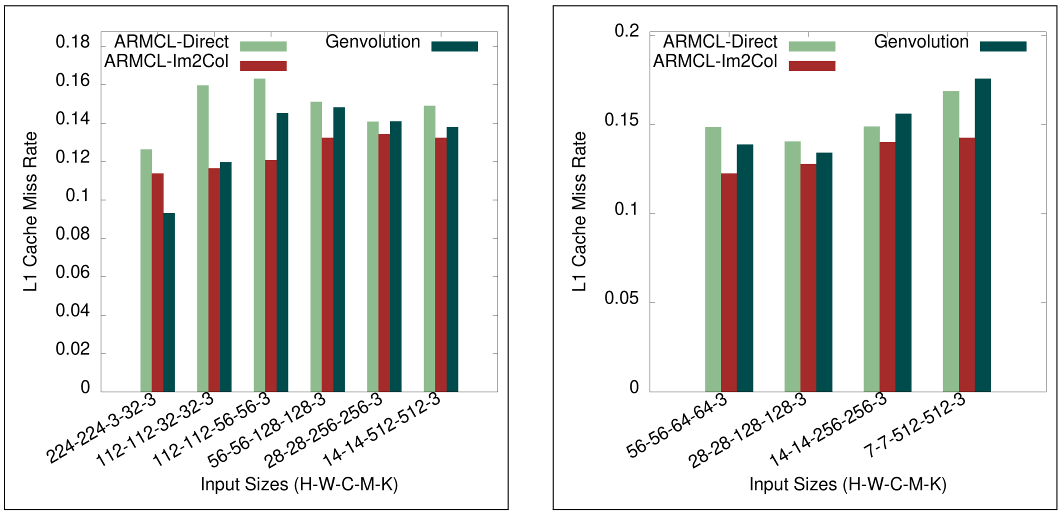


Figure 4-14: L1 Cache Miss Rate of Genvolution implementations on VGG ILSRVC convolutions on ARM Target 1.



(a) AlexNet convolutions. (b) Inception V4 convolutions.

Figure 4-15: L1 cache miss rate on AlexNet and Inception V4 convolutions on ARM Target 2.



(a) MobileNet V2 convolutions. (b) ResNet-152 convolutions.

Figure 4-16: L1 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 2.

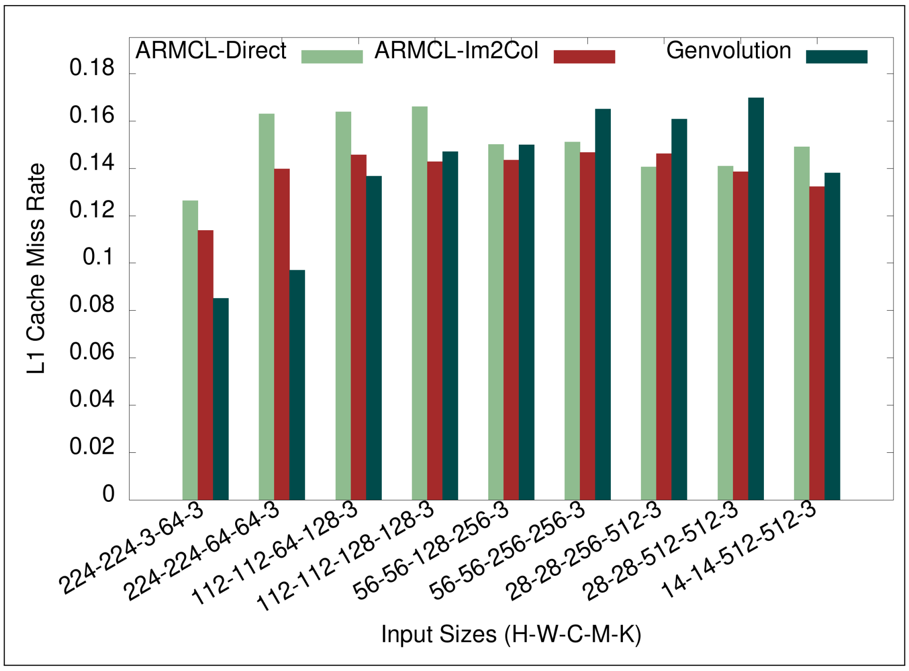
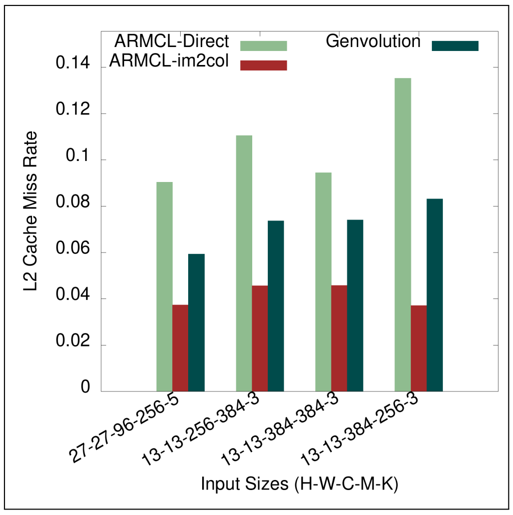
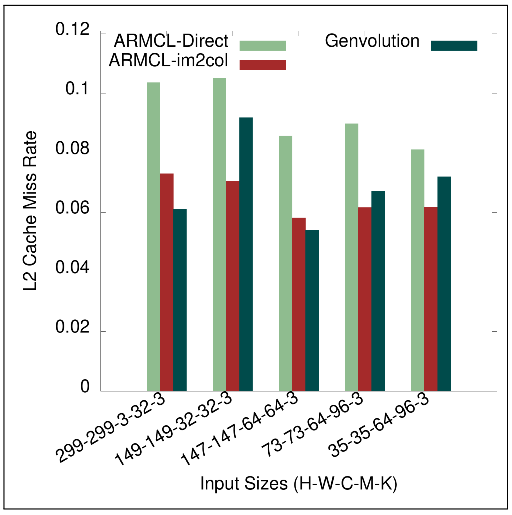


Figure 4-17: L1 Cache Miss Rate of Genvolution implementations on VGG ILSRVC convolutions on ARM Target 2.

## L2 Cache Miss Rate

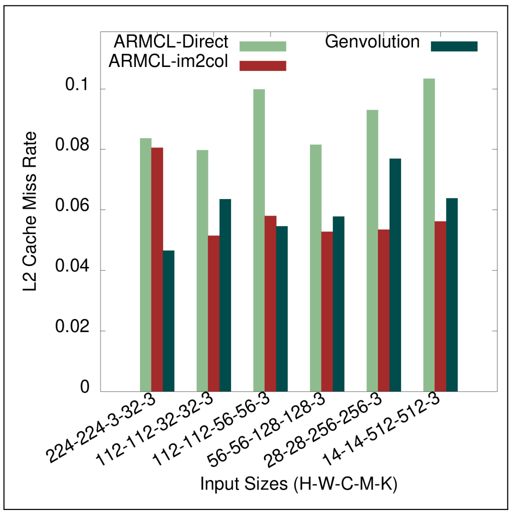


(a) AlexNet convolutions.

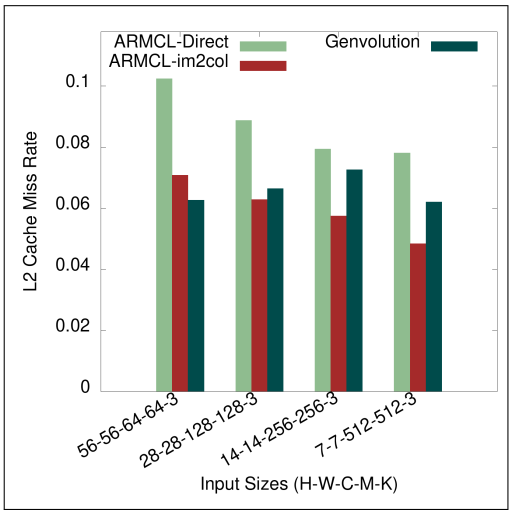


(b) Inception V4 convolutions.

Figure 4-18: L2 cache miss rate on AlexNet and Inception V2 convolutions on ARM Target 1.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 4-19: L2 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

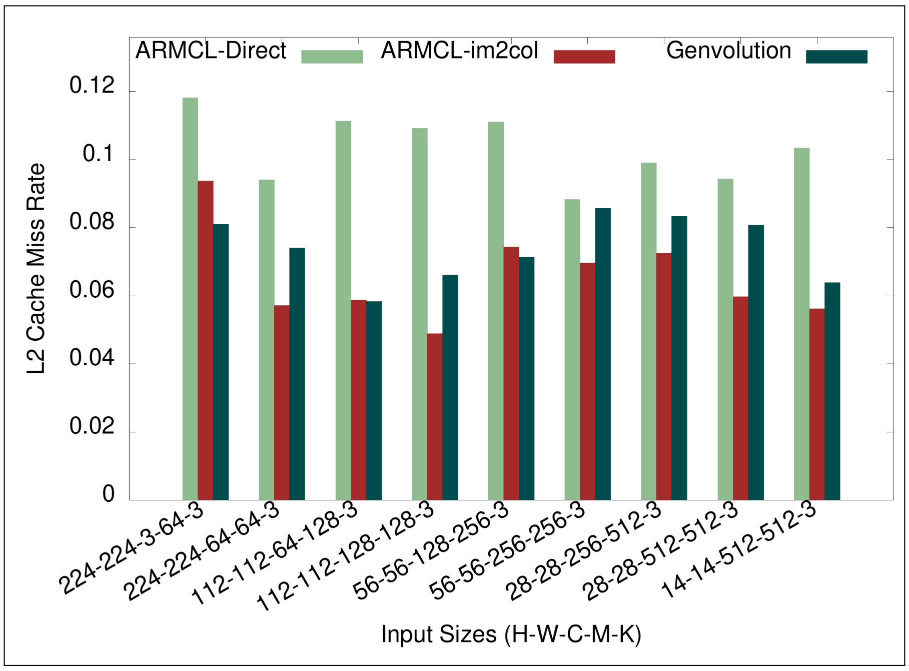
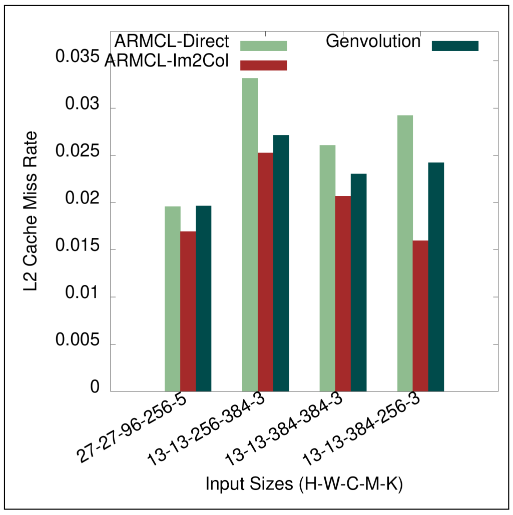
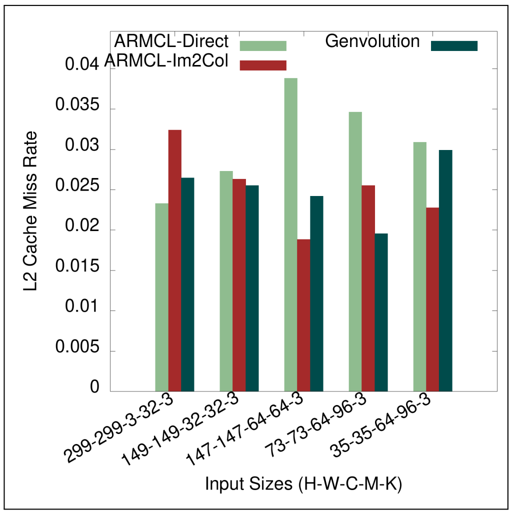


Figure 4-20: L2 Cache Miss Rate of Genvolution implementations on VGG ILSRVC Convolutions on ARM Target 1.

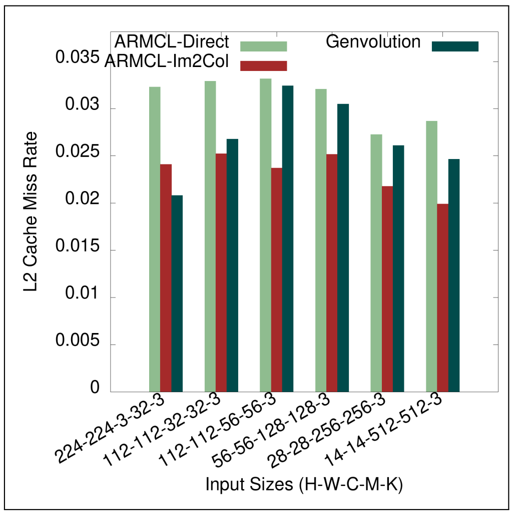


(a) AlexNet convolutions.

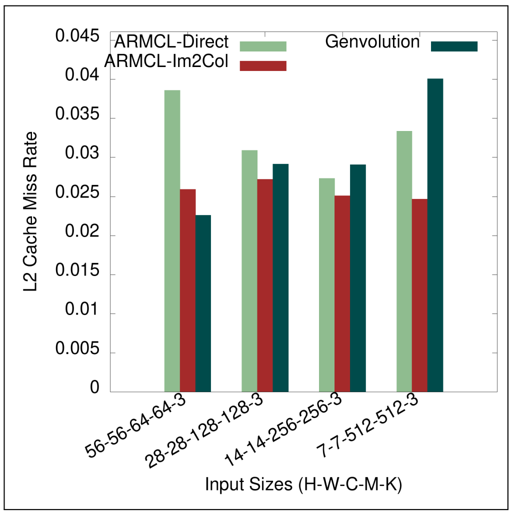


(b) Inception V4 convolutions.

Figure 4-21: L2 cache miss rate on AlexNet and Inception V2 convolutions on ARM Target 2.



(a) MobileNet V2 convolution.



(b) ResNET-152 convolutions.

Figure 4-22: L2 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 2.

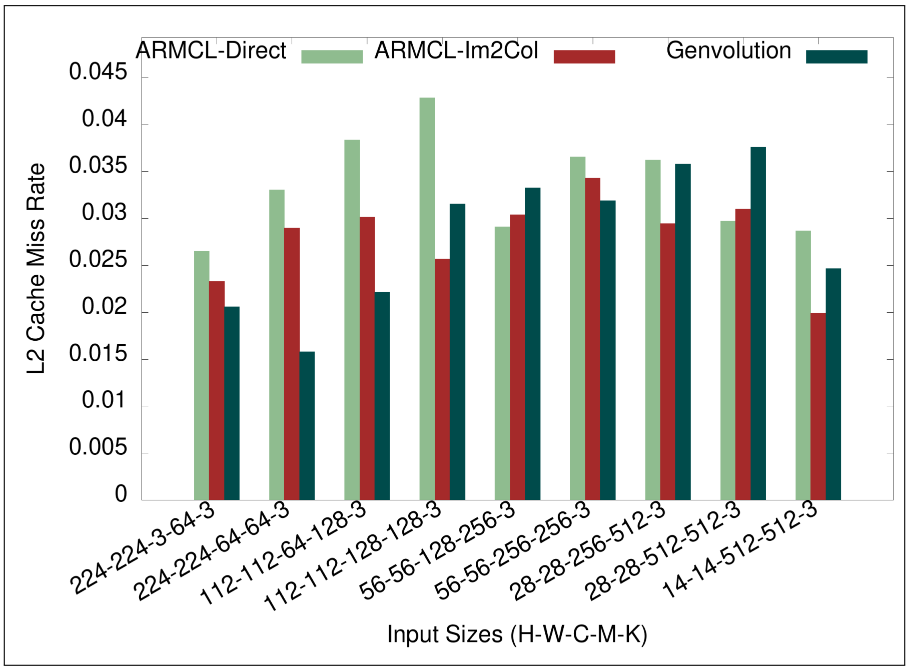


Figure 4-23: L2 Cache Miss Rate of Genvolution implementations on VGG ILSRVC Convolutions on ARM Target 2.

## Temporary Memory Usage

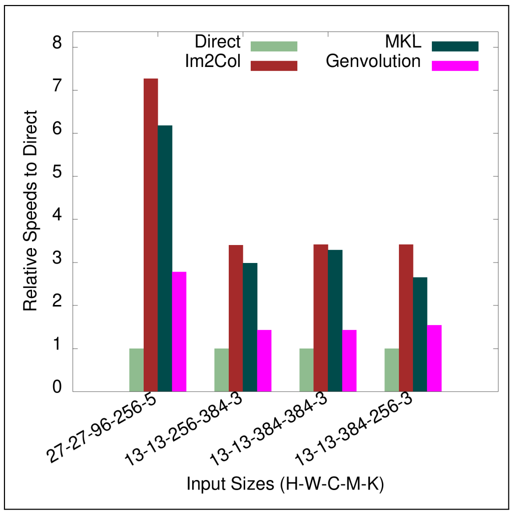
| Network | H | W | C | M | K | Genvolution Temporary Bytes | Im2Col Temporary Bytes | Temporary Memory Ratio |
|---|---|---|---|---|---|---|---|---|
| AlexNet | 27 | 27 | 96 | 256 | 5 | 0 | 6998400 | 0 |
| AlexNet | 13 | 13 | 256 | 384 | 3 | 0 | 1557504 | 0 |
| AlexNet | 13 | 13 | 384 | 384 | 3 | 0 | 2336256 | 0 |
| AlexNet | 13 | 13 | 384 | 256 | 3 | 0 | 2336256 | 0 |
| Inception V4 | 299 | 299 | 3 | 32 | 3 | 0 | 9655308 | 0 |
| Inception V4 | 149 | 149 | 32 | 32 | 3 | 0 | 25575552 | 0 |
| Inception V4 | 147 | 147 | 64 | 64 | 3 | 0 | 49787136 | 0 |
| Inception V4 | 73 | 73 | 64 | 96 | 3 | 0 | 12278016 | 0 |
| Inception V4 | 35 | 35 | 64 | 96 | 3 | 0 | 2822400 | 0 |
| MobileNetV2 | 224 | 224 | 3 | 32 | 3 | 0 | 5419008 | 0 |
| MobileNetV2 | 112 | 112 | 32 | 32 | 3 | 50176 | 14450688 | 0.003472 |
| MobileNetV2 | 112 | 112 | 56 | 56 | 3 | 7168 | 25288704 | 0.000283 |
| MobileNetV2 | 56 | 56 | 128 | 128 | 3 | 0 | 14450688 | 0 |
| MobileNetV2 | 28 | 28 | 256 | 256 | 3 | 6272 | 7225344 | 0.000868 |
| MobileNetV2 | 14 | 14 | 512 | 512 | 3 | 1416 | 3612672 | 0.000392 |
| ResNet-153 | 56 | 56 | 64 | 64 | 3 | 3584 | 7225344 | 0.000496 |
| ResNet-153 | 28 | 28 | 128 | 128 | 3 | 0 | 3612672 | 0 |
| ResNet-153 | 14 | 14 | 256 | 256 | 3 | 1416 | 1806336 | 0.000784 |
| ResNet-153 | 7 | 7 | 512 | 512 | 3 | 392 | 903168 | 0.000434 |
| VGG ILSVRC | 224 | 224 | 3 | 64 | 3 | 0 | 5419008 | 0 |
| VGG ILSVRC | 224 | 224 | 64 | 64 | 3 | 0 | 115605504 | 0 |
| VGG ILSVRC | 112 | 112 | 64 | 128 | 3 | 0 | 28901376 | 0 |
| VGG ILSVRC | 112 | 112 | 128 | 128 | 3 | 0 | 57802752 | 0 |
| VGG ILSVRC | 56 | 56 | 128 | 256 | 3 | 0 | 14450688 | 0 |
| VGG ILSVRC | 56 | 56 | 256 | 256 | 3 | 7296 | 28901376 | 0.000252 |
| VGG ILSVRC | 28 | 28 | 256 | 512 | 3 | 904 | 7225344 | 0.000125 |
| VGG ILSVRC | 28 | 28 | 512 | 512 | 3 | 2592 | 14450688 | 0.000179 |
| VGG ILSVRC | 14 | 14 | 512 | 512 | 3 | 1416 | 3612672 | 0.000392 |

Table 4.1: Temporary Memory Overhead on ARM Target 1.

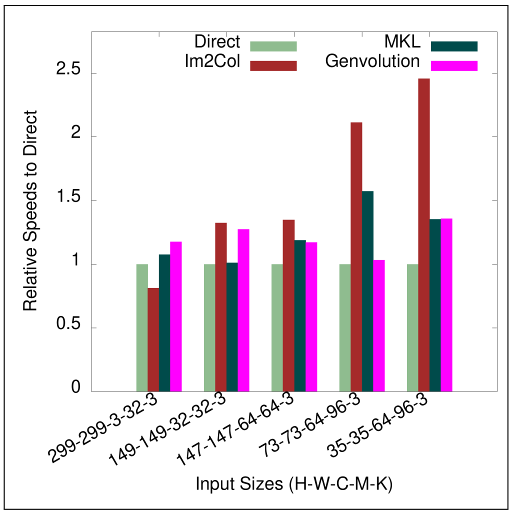
| **Network** | **H** | **W** | **C** | **M** | **K** | **Genvolution Temporary Bytes** | **Im2Col Temporary Bytes** | **Temporary Memory Ratio** |
|---|---|---|---|---|---|---|---|---|
| AlexNet | 27 | 27 | 96 | 256 | 5 | 0 | 6998400 | 0 |
| AlexNet | 27 | 27 | 96 | 256 | 5 | 0 | 6998400 | 0 |
| AlexNet | 13 | 13 | 256 | 384 | 3 | 0 | 1557504 | 0 |
| AlexNet | 13 | 13 | 384 | 384 | 3 | 0 | 2336256 | 0 |
| AlexNet | 13 | 13 | 384 | 256 | 3 | 0 | 2336256 | 0 |
| Inception V4 | 299 | 299 | 3 | 32 | 3 | 0 | 9655308 | 0 |
| Inception V4 | 149 | 149 | 32 | 32 | 3 | 0 | 25575552 | 0 |
| Inception V4 | 147 | 147 | 64 | 64 | 3 | 2480 | 49787136 | 0.000050 |
| Inception V4 | 73 | 73 | 64 | 96 | 3 | 0 | 12278016 | 0 |
| Inception V4 | 35 | 35 | 64 | 96 | 3 | 6528 | 2822400 | 0.002313 |
| MobileNetV2 | 224 | 224 | 3 | 32 | 3 | 0 | 5419008 | 0 |
| MobileNetV2 | 112 | 112 | 32 | 32 | 3 | 25088 | 14450688 | 0.001736 |
| MobileNetV2 | 112 | 112 | 56 | 56 | 3 | 100352 | 25288704 | 0.003968 |
| MobileNetV2 | 56 | 56 | 128 | 128 | 3 | 32256 | 14450688 | 0.002232 |
| MobileNetV2 | 28 | 28 | 256 | 256 | 3 | 6528 | 7225344 | 0.000903 |
| MobileNetV2 | 14 | 14 | 512 | 512 | 3 | 6528 | 3612672 | 0.001807 |
| ResNet-153 | 56 | 56 | 64 | 64 | 3 | 25088 | 7225344 | 0.003472 |
| ResNet-153 | 28 | 28 | 128 | 128 | 3 | 101920 | 3612672 | 0.028212 |
| ResNet-153 | 14 | 14 | 256 | 256 | 3 | 6528 | 1806336 | 0.003614 |
| ResNet-153 | 7 | 7 | 512 | 512 | 3 | 6528 | 903168 | 0.007228 |
| VGG ILSVRC | 224 | 224 | 3 | 64 | 3 | 0 | 5419008 | 0 |
| VGG ILSVRC | 224 | 224 | 64 | 64 | 3 | 1792 | 115605504 | 0.000016 |
| VGG ILSVRC | 112 | 112 | 64 | 128 | 3 | 25088 | 28901376 | 0.000868 |
| VGG ILSVRC | 112 | 112 | 128 | 128 | 3 | 426496 | 57802752 | 0.007378 |
| VGG ILSVRC | 56 | 56 | 128 | 256 | 3 | 3584 | 14450688 | 0.000248 |
| VGG ILSVRC | 56 | 56 | 256 | 256 | 3 | 1792 | 28901376 | 0.000062 |
| VGG ILSVRC | 28 | 28 | 256 | 512 | 3 | 392 | 7225344 | 0.000054 |
| VGG ILSVRC | 28 | 28 | 512 | 512 | 3 | 25344 | 14450688 | 0.001754 |
| VGG ILSVRC | 14 | 14 | 512 | 512 | 3 | 6528 | 3612672 | 0.001807 |

Table 4.2: Temporary Memory Overhead on ARM Target 2.

### 4.12.2 Intel

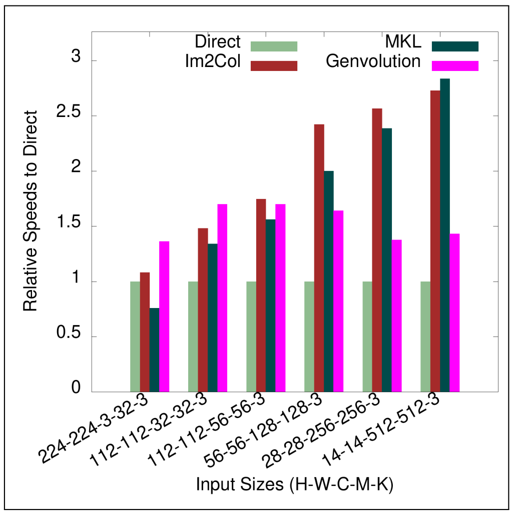


(a) AlexNet convolutions.

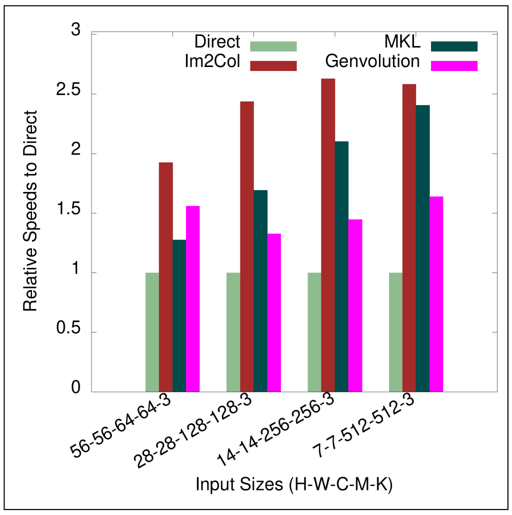


(b) Inception V4 convolutions.

Figure 4-24: Execution time on AlexNet and Inception V4 convolutions on Intel Target 1.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 4-25: Execution time on MobileNet V2 and ResNET-152 convolutions on Intel Target 1.

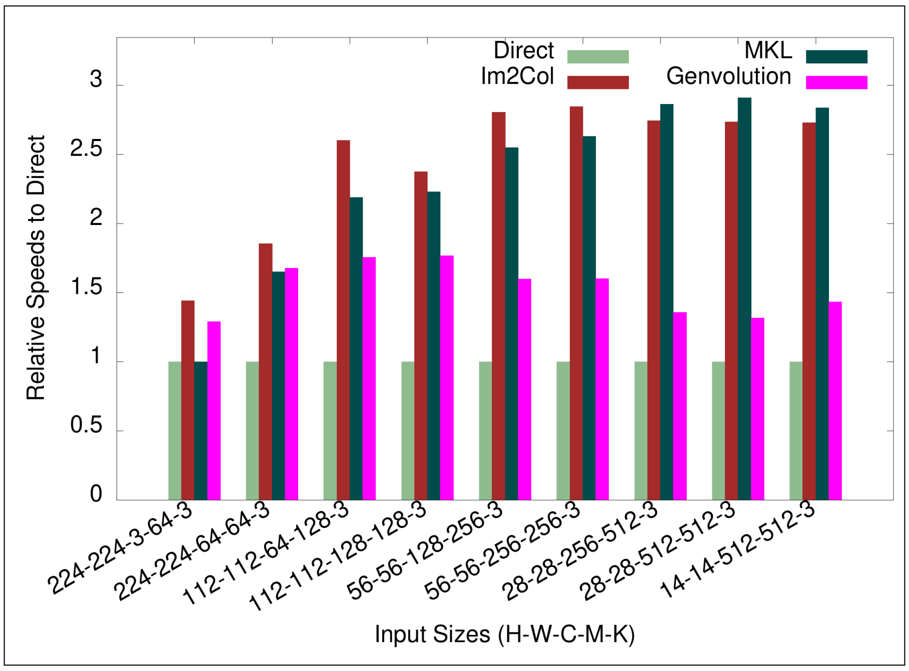


Figure 4-26: Execution Time of Genvolution implementations on VGG ILSRVC Convolutions on Intel Target 1.

## Temporary Memory Overhead

| Network | H | W | C | M | K | Genvolution Temporary Bytes | Im2Col Temporary Bytes | Temporary Memory Ratio |
|---|---|---|---|---|---|---|---|---|
| AlexNet | 27 | 27 | 96 | 256 | 5 | 0 | 6998400 | 0 |
| AlexNet | 27 | 27 | 96 | 256 | 5 | 0 | 6998400 | 0 |
| AlexNet | 13 | 13 | 256 | 384 | 3 | 0 | 1557504 | 0 |
| AlexNet | 13 | 13 | 384 | 384 | 3 | 0 | 2336256 | 0 |
| AlexNet | 13 | 13 | 384 | 256 | 3 | 0 | 2336256 | 0 |
| Inception V4 | 299 | 299 | 3 | 32 | 3 | 0 | 9655308 | 0 |
| Inception V4 | 149 | 149 | 32 | 32 | 3 | 0 | 25575552 | 0 |
| Inception V4 | 147 | 147 | 64 | 64 | 3 | 3136 | 49787136 | 0.000063 |
| Inception V4 | 73 | 73 | 64 | 96 | 3 | 0 | 12278016 | 0 |
| Inception V4 | 35 | 35 | 64 | 96 | 3 | 6272 | 2822400 | 0.002222 |
| MobileNetV2 | 224 | 224 | 3 | 32 | 3 | 0 | 5419008 | 0 |
| MobileNetV2 | 112 | 112 | 32 | 32 | 3 | 14336 | 14450688 | 0.000992 |
| MobileNetV2 | 112 | 112 | 56 | 56 | 3 | 109760 | 25288704 | 0.004340 |
| MobileNetV2 | 56 | 56 | 128 | 128 | 3 | 7168 | 14450688 | 0.000496 |
| MobileNetV2 | 28 | 28 | 256 | 256 | 3 | 54272 | 7225344 | 0.007511 |
| MobileNetV2 | 14 | 14 | 512 | 512 | 3 | 12544 | 3612672 | 0.003472 |
| ResNet-153 | 56 | 56 | 64 | 64 | 3 | 3584 | 7225344 | 0.000496 |
| ResNet-153 | 28 | 28 | 128 | 128 | 3 | 12544 | 3612672 | 0.003472 |
| ResNet-153 | 14 | 14 | 256 | 256 | 3 | 6272 | 1806336 | 0.003472 |
| ResNet-153 | 7 | 7 | 512 | 512 | 3 | 1568 | 903168 | 0.001736 |
| VGG ILSVRC | 224 | 224 | 3 | 64 | 3 | 0 | 5419008 | 0 |
| VGG ILSVRC | 224 | 224 | 64 | 64 | 3 | 71680 | 115605504 | 0.000620 |
| VGG ILSVRC | 112 | 112 | 64 | 128 | 3 | 57344 | 28901376 | 0.001984 |
| VGG ILSVRC | 112 | 112 | 128 | 128 | 3 | 25088 | 57802752 | 0.000434 |
| VGG ILSVRC | 56 | 56 | 128 | 256 | 3 | 25088 | 14450688 | 0.001736 |
| VGG ILSVRC | 56 | 56 | 256 | 256 | 3 | 50176 | 28901376 | 0.001736 |
| VGG ILSVRC | 28 | 28 | 256 | 512 | 3 | 309248 | 7225344 | 0.042800 |
| VGG ILSVRC | 28 | 28 | 512 | 512 | 3 | 31360 | 14450688 | 0.002170 |
| VGG ILSVRC | 14 | 14 | 512 | 512 | 3 | 12544 | 3612672 | 0.003472 |

Table 4.3: Temporary Memory Overhead on Intel Target 1.

# Chapter 5

# Genvolution Generated Code Design

This chapter covers optimization strategies that can be leveraged in the CNN convolution code generated by Genvolution. It mainly focuses on the different ways SIMD vectorization can be used to improve the execution time of direct CNN convolution.

## 5.1 Microkernels

As covered in chapter 4, all code generated by Genvolution can be split into two sections: a micorkernel, and a macro-program. The microkernel implements the hottest loops of the convolution implementation, and the macro-program implements the control flow of the convolution and makes use of the microkernel.

Genvolution has access to a number of different microkernels. Every microkernel makes use of SIMD vectorization in some way. These microkernels can be broken up into different classes of microkernel depending on how they use SIMD vectorization to implement direct CNN convolution. The different classes will be referred to as *SIMD strategies*. A number of the *SIMD strategies* had originally been developed as part of our final year project.

```
float sum = 0;
for (signed c = 0; c < CHANNELS; c++) {
  sum += input_image[h+y][w+x][c]
          * input_kernel[m][y+(y/2)][x+(x/2)][c]
}
output_image[m][h][w] = sum;

```

Figure 5-1: Scalar implementation of the dot product between two vectors of input channels, one from the input image tensor and one from the input kernel tensor.

## 5.2 Pairwise Channels SIMD Strategy

### 5.2.1 Strategy Outline

The *pairwise channels* SIMD strategy vectorizes the dot product between two vectors of input channels. This strategy was originally developed as part of our final year project. When performing *multi-channel* CNN convolution, we continually calculate the dot product between vectors of input channels from the input tensor and the input kernel. This is typically implemented as a for loop that iterates across the two vectors, multiplying each pair of scalar values, and accumulating the results of the multiplications in a running total. Figure 5-1 shows a code snippet implementing this.

We can vectorize the calculations in the for loop by replacing the scalar operations with their SIMD equivalent. Instead of loading scalar values from the two input tensors, we load a SIMD vector of values from each and perform multiple pairwise multiplications in parallel using a SIMD multiplication. We also store the running total in a vector register, and change the increment of the loop so it matches the number of lanes in the vector registers being used. After completing the vectorized for loop, the final value will be split across the lanes of the running total vector register. A horizontal add operation is used to sum the lanes of the running total register into a single scalar value, which is written out. Figure 5-2 shows a code snippet implementing the above. In figure 5-2, four-lane SIMD vectors that store 4 32-bit floats are used (*float_32x4*). The multiplication of the input values, and the accumulation with the running total

```
float_32x4 sum = 0;
for (signed c = 0; c < CHANNELS; c+=4) {
  float_32x4 i =
    load_32x4(&(input_image[h+y][w+x][c] ));
  float_32x4 k =
    load_32x4(&(input_kernel[m][y+(Y/2)][x+(X/2)][c]));
  sum = fused_mult_add_32x4(sum, i, k);
}
float result = horzitonal_add_32x4(sum);
output_image[m][h][w] += result;

```

Figure 5-2: Equivalent SIMD implementation of figure 5-1.

are performed with a single *fused multiple add (FMA)* that multiples two inputs together and adds the product to a third input.

### 5.2.2 Effect on Performance

By replacing the scalar operations with SIMD equivalent operations, we would hope to see a reduction in the number of cycles spent performing calculations. In modern general purpose processors (GPPs), SIMD instructions usually require a similar number of clock cycles as their scalar equivalent. By replacing scalar instructions with SIMD instructions, we may see up a $L$ times speed-up, where $L$ is the number of lanes in the SIMD vector registers being used. For example, on the Intel Skylake architecture, the *VFMADD132PS* instruction performs a fused multiply-add on a 256-bit AXV register holding eight 32-bit floats with a latency of 4 cycles and a throughput of 1 cycle (Intel 2019). *FMUL*, the scalar Skylake instruction for multiplying two scalar 32-bit floats has the same latency and throughput (Fog 2018). In theory, vectorizing using *VFMADD132PS* would give an 8X reduction in computational time. However, this increase in performance relies on the input values being read from the input tensors in a timely manner. If the memory reads continually stall, then the dot product will become memory bound rather than computational bound, and vectorizing it will have less of an impact on performance.

The *pairwise channels* SIMD strategy also relies on a horizontal add operation to sum the values across the lanes in the running total vector register. This

```
float32x4 input = /* ... input to horz add ... */
float32x2 t0 = vadd_f32(
  vget_high_f32(input),
  vget_low_f32(input));
float result = vget_lane_f32(
  vpadd_f32(t0. t0),
  0);

```

Figure 5-3: ARMv7 NEON Horizontal Add operation

operation often has poor performance on GPP SIMD architectures, or is not implemented in hardware, and must be performed using a sequence of other instructions. For example, the ARMv7 NEON SIMD architecture does not have an instruction to perform the horizontal add operation for a 128-bit vector register split into 4 32-bit floats. Instead, figure 5-3 shows an efficient horizontal add operation for a 128-bit vector register holding 4 32-bit floats on ARMv7 NEON implemented in software.

### 5.2.3 Pairwise Channels Microkernels

There are two microkernels that implement the *Pairwise Channels* SIMD strategy. The *CI_SSO* microkernel is roughly equivalent to the code snippet shown in figure 5-2. It can be used to generate code for all the tested architectures (Intel AVX2, ARMv7 NEON, ARMv8 NEON). The input channels for loop can be unrolled by Genvolution. If the length of the input channels dimension is not a perfect multiply of the number of lanes in the SIMD vectors being used, then Genvolution can insert a *clean-up loop* that covers the ragged end of the input channels dimension.

Figure 5-4 shows a simplified example of code produced by the *YXCI_SSO* microkernel. The *YXCI_SSO* microkernel implements the *pairwise channels* SIMD strategy. However, it also includes for loops that move across the input kernel's height $Y$ and input kernel's width $X$. The same running total is used for the entire microkernel. This reduces the number horizontal adds required by $Y * X$. It also reduces the number of memory writes to the output image tensor by $Y * X$. However, conditional statements that control when zero-padding is re-

```
float_32x4 sum = 0;
for (signed y = -(Y/2); y <= Y/2; y++) {
  for (signed x = -(X/2); x <= X/2; x++) {
    if (/*h+y and w+x in the bounds of input_image*/) {
      for (signed c = 0; c < CHANNELS; c+=4) {
        float_32x4 i =
          load_32x4(
            &(input_image[h+y][w+x][c] ));
        float_32x4 k =
          load_32x4(
            &(input_kernel[m][y+(Y/2)][x+(X/2)][c]));
        sum = fused_mult_add_32x4(sum, i, k);
      }
    }
  }
}
float result = horzitonal_add_32x4(sum);
output_image[m][h][w] += result;

```

Figure 5-4: Pairwise channels strategy that also includes for loops across the kernel height $Y$ and the kernel width $X$. This reduces the number of horizontal adds and writes to the output image.

quired (i.e. when computations should be skipped) must be inserted into the microkernel. This is because we need to know the values of the $X$ and $Y$ iterators to be able to perform the conditional checks. If the $X$ and $Y$ loops are in the microkernel, then the conditional statements must in the microkernel too. This is unwanted as having conditional statements in hot sections of code (e.g. the microkernels) can lead to a higher number of branch mispredictions.

## 5.3 Parallel Kernels SIMD Strategy

### 5.3.1 Implementation Outline

The *Parallel Kernels* SIMD strategy uses SIMD vectorization to convolve the input tensor with multiple input kernels in parallel. This strategy was originally developed as part of our final year project. During CNN convolution, the same input tensor is convolved with $M$ input kernel tensors to produce an output tensor with $M$ output channels. We can convolve the input tensor with $L$ kernels in parallel by storing the same point from $L$ kernels in a $L$ width SIMD vector. This SIMD vector is then used with a second SIMD vector that has the same value from the input image tensor stored in every lane. By multiplying these two vectors together, we perform $L$ convolutions in parallel.

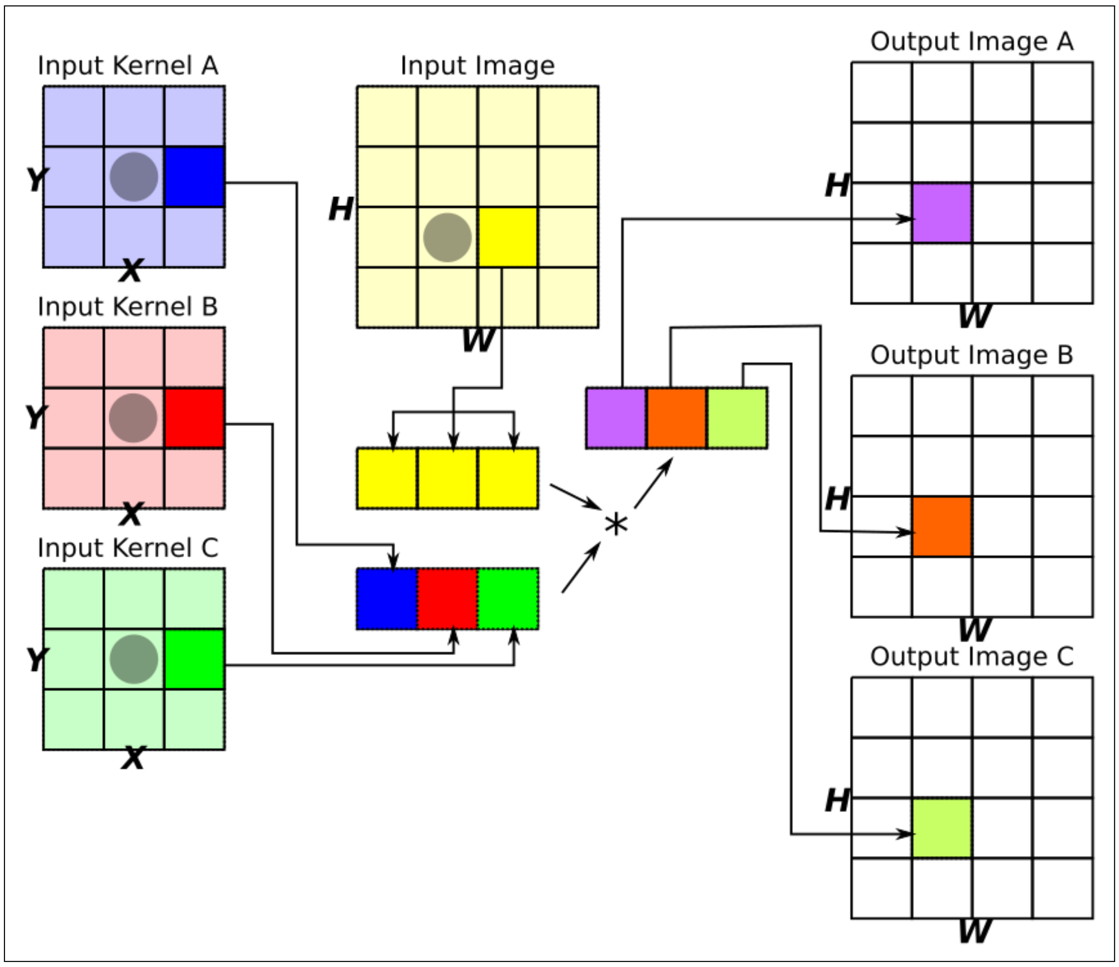


Figure 5-5: Convolving 3 input kernels with an input image in parallel using 3-lane SIMD vectors.

Figure 5-5 shows this strategy in a graphical form. The value at the point $(2, 1)$ is taken from *input kernels A, B, and C* and stored in a SIMD vector together. The point $(2, 2)$ from the *Input Image* is *broadcasted* to all the lanes of a second SIMD vector. The two SIMD vectors are multiplied, and each lane of the result is summed with the same point in the three output images. This process would be repeated with every point in the three input kernels to fully perform the convolution centered at $(1, 2)$ (the point marked by the circle in the *input image*). While figure 5-5 may suggest that a gather operation is used to build the SIMD vector of input kernel values, in practice the tensor of input kernel tensors is ordered so that the $M$ dimension is the innermost contiguous dimension (e.g. ordered $(Y \times X \times C \times M)$). This means the SIMD vector of input kernel values can be created using a normal SIMD load operation. The output image tensor is also stored with the output channels dimension being the innermost contiguous dimension (e.g. ordered $(H \times W \times M)$), so that values can be written to it using a normal SIMD store operation.

### 5.3.2 Broadcast Versus Scale

In the previous subsection, the *Parallel Kernels* SIMD strategy was described as using a *broadcast* operation to create SIMD vectors, where every lane contains the same value from the input tensor. This SIMD vector is then multiplied with a second vector of input kernel values. This step can be replaced with a *vector-scale* operation that would scale all the values in the vector of input kernel values by a value taken from the input tensor. Both these approaches produce the same result, with the difference in performance between them relying only on the difference in performance between a broadcast+multiply versus a vector-scale. None of the tested Intel SIMD architectures contain any form of floating-point SIMD vector-scale, so the broadcasting approach must be used for them. ARMv7 NEON and ARMv8 NEON are able to implement both approaches. Our empirical tests found that the vector-scale approach was marginally faster, however both approaches were used to produce the final performance results.

### 5.3.3 Effects On Performance

Similar to the *pairwise channels* SIMD strategy, we can expect to see a roughly $L$ reduction in the number of cycles needed to perform the numeric arithmetic of the CNN convolution as the arithmetic has been vectorized. The *Parallel Kernels* SIMD strategy also allows us to reduce the number of memory reads required from the input tensor, because we read from the input tensor before entering the innermost loop. However, this is counteracted by the fact that we

```
float32x4 i = broadcast_32x4(input_image[h+y][w+x][c])
for (signed m = 0; m < KERNELS; m+=4) {
  float32x4 k =
    load_32x4(
    &(input_kernel[y+(Y/2)][x+(X/2)][c][m]));
  float32x4 old =
    load_32x4(
    &(output_image[h][w][m]));
  float32x4 current =
    fused_mult_add_32x4(old, i, k);
  store_32x4(current, &(output_image[h][w][m]));
}

```

Figure 5-6: Parallel Kernels Pseudocode Microkernel

```
for (signed m = 0; m < KERNELS; m+=4) {
  float32x4 sum = {0,0,0,0};
  for (signed c = 0; c < CHANNELS; c++) {
    float32x4 i = broadcast_32x4(input_image[h+y][w+x][c])
    float32x4 k =
      load_32x4(
      &(input_kernel[y+(Y/2)][x+(X/2)][c][m]));
    fused_mult_add_32x4(sum, i, k);
  }
  float32x4 old =
  load_32x4(
  &(output_image[h][w][m]));
  float32x4 current = add_32x4(old, sum);
  store_32x4(current, &(output_image[h][w][m]));
}

```

Figure 5-7: Parallel Kernels SIMD strategy with the input channels as the innermost loop.

must continually read and write to the output tensor while using this strategy. This is due to the fact that we are iterating across one of the dimensions of the output tensor in the innermost loop (the $M$ dimension). Using explicit tile blocking on the output tensor can have very positive effects on the performance of the *Parallel Kernels* SIMD strategy, because it now only reads and writes to a much smaller buffer which can hopefully be fully stored in the L1 cache.

An alternative approach to reducing the number of reads and writes to the output image tensor is to make the input channels loop the innermost loop as shown in figure 5-7. This reduces the number of reads and writes to the output image tensor required. However, it does mean that the input tensor must be read from in the innermost loop now. It also means that we are now no longer iterating contiguously along the tensor of input kernel tensors which can lead to poorer cache performance.

### 5.3.4 Parallel Kernels Microkernel

There are six *Parallel Kernels* microkernels. Two implement the strategy similarly to figure 5-6, with the main difference being one uses broadcasting, and one uses a vector-scale. Two microkernels implement the strategy similarly to figure 5-7. they are also differentiated by their use of broadcasts, and vector-scales respectively. Finally, there are two other microkernels that are similar to figure 5-7, but the for loops across the kernel height $Y$, and kernel width $X$ dimensions are moved inside the $M$ loop as well. This reduces the number of reads from both input image tensor and the output image tensor. There is a version that uses broadcasting, and a version that uses vector-scales. All the microkernels can handle a ragged end on the $M$ dimension by inserting scalar code if necessary. The unrolling factor of the $M$ loop (and the $C$ loop when it applies) can be controlled by Genvolution for all the microkernels.

## 5.4 Outer Product SIMD Strategy

### 5.4.1 Implementation Outline

The *Outer Product* SIMD strategy uses an outer product between two vectors to perform CNN convolution. The Intel AVX version of this strategy was originally developed as part of our final year project.

Convolving a single-channel 1D input image tensor by a single-channel $(1 \times 1)$ kernel (i.e. a kernel that is a single value) is equivalent to scaling all the values in the input image tensor by the kernel. If we have multiple $(1 \times 1)$ kernels stored in a vector, performing an outer product between the 1D input image tensor and the vector of kernels is equivalent to convolving the input image tensor by all the kernels.

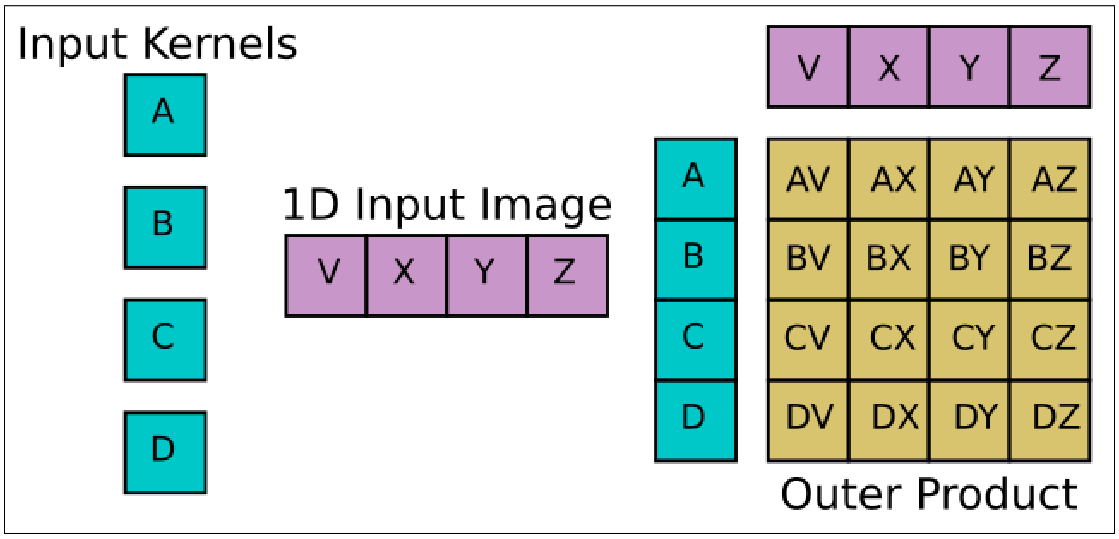


Figure 5-8: Convolving $(1x1)$ kernels with a 1D input image using an outer product.

Figure 5-8 shows an example of this. In the figure, the input kernels are packed into a vector, and an outer product is performed with the 1D input image. Each row of the outer product is a convolution between the input image and one of the kernels. Multi-channel convolution between a $(H \times W \times C)$ input image tensor and a $(M \times Y \times X \times C)$ tensor of input kernel tensors can be performed as a sum of the outer product method detailed above. This is the approach taken by the *Outer Product* SIMD strategy.

In the *Outer Product* SIMD strategy, two SIMD vectors registers are produced. One register contains the same value from multiple kernels (like in the *Parallel Kernels* SIMD strategy), the other contains consecutive values across the $W$ dimension of the input image tensor. An outer product is then performed between the two vector registers, with the resulting matrix being stored in a running total matrix made from a set of SIMD vector registers.

### 5.4.2 Intel

the *Outer Product* SIMD strategy is based upon performing an outer product between two SIMD vector registers. Due to differences between Intel SIMD architectures and ARM SIMD architectures, how the outer product is performed differs greatly. On Intel Architectures, the outer product is performed by continually multiplying two SIMD registers while reordering their contents after each multiplication. For the remainder of this subsection, we will consider only 256-bit eight-lane Intel SIMD registers, although the 128-bit four-lane implementation is very similar.

The outer product of two AVX registers, $op\_img\_vec$ (which contains data from the input image tensor), and $op\_ker\_vec$ (which contains data from the tensor of input kernels), will be a $(8 \times 8)$ matrix where each column of the matrix is $op\_ker\_vec$ scaled by one of the lanes of $op\_img\_vec$. However, as the Intel AVX architecture has no vector-scale operations, we must use a different approach to calculate the values of the outer product. We can calculate all the values required for the outer product by multiplying $op\_img\_vec$ and $op\_ker\_vec$ eight times, and barrel-shifting the lanes of $op\_ker\_vec$ after each multiplication. This produces all the needed multiplication pairs, however barrel-shifting an AXV register is very expensive. While the AVX registers are logically 256-bits wide, in many cases the lower 128-bits and the upper 128-bits are isolated from one another. For example, the *_mm256_permute_ps* intrinsic allows us to swap values in AXV register lanes, however values cannot be moved across the 128-bit divide. The *_mm256_permute2f128_ps* intrinsic allows us to swap the upper and lower 128 bits of an AVX register, however it has three times the latency of *_mm256_permute_ps* because it crosses the 128-bit divide (Intel 2019). Barrel-shifting the lanes of $op\_ker\_vec$ would be very expensive, because we would move values across the 128-bit divide on every shift. Instead of this, figure 5-9 shows a cheaper list of transformations that also allow us to produce all the values required for the outer product.

The transformation in figure 5-9 only requires a single movement across the 128-bit divide. The results of the eight multiplications in figure 5-9 are each stored in a separate AVX regsiter. The result registers act as a running total matrix, and multiple outer products can be accumulated on top of one another without having to perform any memory writes. The main issue with this ap-

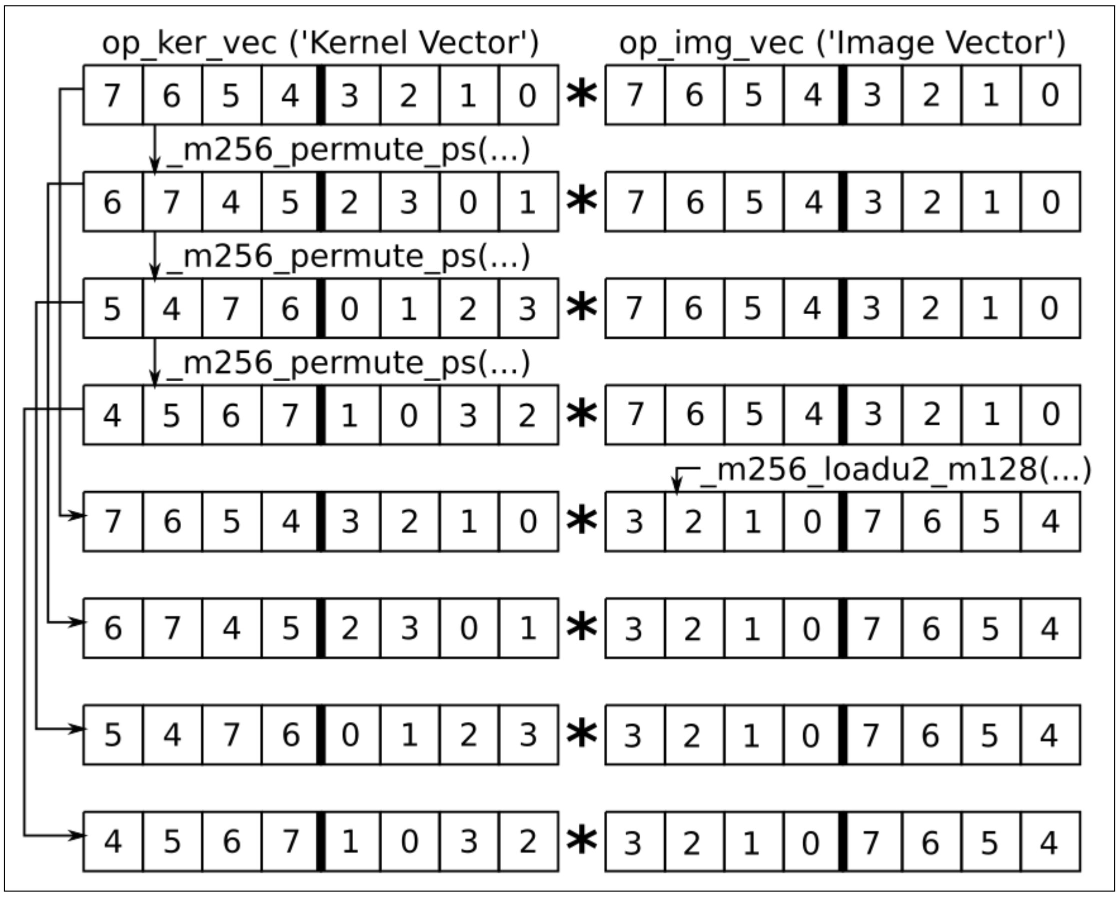


Figure 5-9: Efficient permutations of two AVX registers to calculate outer product values.

proach is that the values in the running total matrix will be in the completely wrong order. This means that before they can be written out to the output tensor, they must be reordered. However, this does not occur nearly as often as the outer product itself. With the correct loop ordering, the matrix of registers only needs to reordered and written out once for each $Y \times X \times C$ outer products calculated.

A second minor issue is that it is possible that we will have an input SIMD vector of input image values, where some values need to be synthesized and some don't. To solve this issue, there is conditional code that will mask out any lanes that need to be synthesized from the SIMD vector containing values from the input tensor. This replaces the incorrect/junk data in the lanes with zeroes, i.e. it fills the lanes with zero-padded synthesized values.

### 5.4.3 ARM

Both ARMv7 and ARMv8 NEON have access to vector-scale operation, therefore for these architectures we calculate outer products by scaling $op\_ker\_vec$ multiple times by the lanes of $op\_img\_vec$.

In theory, this means that $op\_img\_vec$ does not actually need to be created, as we can just use scalar values from the input tensor to scale $op\_ker\_vec$. However, in practice it is worth it so we can make use of the fused-scale-add operations NEON offers. Both NEON architectures contain a fused-scale-add operation, accessed as *vmlaq_lane_f32* in the NEON intrinsic libraries. *vmlaq_lane_f32* takes a lane from a 64-bit 2-lane vector register, scales a 128-bit 4-lane register by the lane, and adds the scaled register to a second 128-bit vector register. By splitting $op\_img\_vec$ across two 64-bit vector registers we can make use of this faster operation. Figure 5-10 shows a simplified example of the *outer product* SIMD strategy on the ARMv7 NEON architecture. The outer products are still accumulated in a matrix of vector registers. The matrix of registers will also be in correct order with this approach, so no reordering is required when writing them out.

### 5.4.4 Register Blocking The Running Total Matrix

As mentioned in sections 5.4.3 and 5.4.2, there is a running total matrix of outer products that have been calculated. The running total matrix is stored in a set of registers. As covered in section 2.3, memory blocking can be used to reduce pressure on the memory hierarchy, and reduce the number of cache misses by splitting large data structures into smaller blocks that fit in the faster data caches. Section 2.3 only considers blocking on the cache level, but we can also consider blocking on the register level. While values in the L1 cache can be accessed very quickly, there is still a delay in accessing them of several cycles. For example, there is an approximately 4 cycle delay for accessing the data in the L1 cache on an Intel Core i7 Sandybridge processor (Fog 2018). However, values already stored in registers incur no access cost. Also, SIMD register files can hold many values, with many registers to store data. For example, the ARMv7

```
for (signed m = 0; m < KERNELS; m+=4) {
  for (signed w = 0; w < WIDTH; w+=4) {
    //outer product running total matrix
    float32x4 mat_row_0 = {0,0,0,0};
    float32x4 mat_row_1 = {0,0,0,0};
    float32x4 mat_row_2 = {0,0,0,0};
    float32x4 mat_row_3 = {0,0,0,0};
    for (signed y = -(Y/2); y <= Y/2; y++) {
      for (signed x = -(X/2); x <= X/2; x++) {
        for (signed c = 0; c < CHANNELS; c++) {
          float32x4 op_ker_vec =
            load_32x4(
            &(input_kernel[y+(Y/2)][x+(X/2)][c][m]));
          float32x2 op_img_vec0 =
            load_32x4(
            &(input_image[c][h+y][w+x+0] ));
          float32x2 op_img_vec1 =
            load_32x4(
            &(input_image[c][h+y][w+x+2] ));
          //fused_scale_add(a,b,c,d) => (a += b * c[d])
          fused_scale_add(mat_row_0,
            op_ker_vec, op_img_vec0, 0);
          fused_scale_add(mat_row_1,
            op_ker_vec, op_img_vec0, 1);
          fused_scale_add(mat_row_2,
            op_ker_vec, op_img_vec1, 0);
          fused_scale_add(mat_row_3,
            op_ker_vec, op_img_vec1, 1);
        }
      }
    }
    store_32x4(mat_row_0, &(output_image[h][w][m]));
    store_32x4(mat_row_1, &(output_image[h][w+1][m]));
    store_32x4(mat_row_2, &(output_image[h][w+2][m]));
    store_32x4(mat_row_3, &(output_image[h][w+3][m]));
  }
}

```

Figure 5-10: ARMv7 NEON Pseudocode for a *outer product* microkernel.

NEON SIMD architecture has a register file of 32 128-bit vector registers. This allows up to 128 32-bit floats to be stored in registers simultaneously. In the *Outer product* SIMD strategy, we use register blocking to store a running total of all the outer products that have been performed in the innermost loop of the strategy. Figure 5-11 shows an example of how registers from the ARMv7 NEON register file can be allocated to store the running total matrix, and the current input vectors. Sixteen 128-bit registers are used to store the $(8 \times 8)$ running total matrix in figure 5-11, and 4 registers are used to store op_ker_vec and op_img_vec.

### 5.4.5 Effects On Performance

The crux of the *outer product* SIMD strategy is the ratio between computations and memory reads. The *outer product* SIMD strategy allows us to read in two vectors of length $L$, and use them to calculate $L^2$ values. The large ratio between memory accesses and computations means there is significantly reduced pressure on the memory hierarchy. The *outer product* SIMD strategy is often computational-bound in terms of performance, and does not become stalled waiting for values to be retrieved from caches.

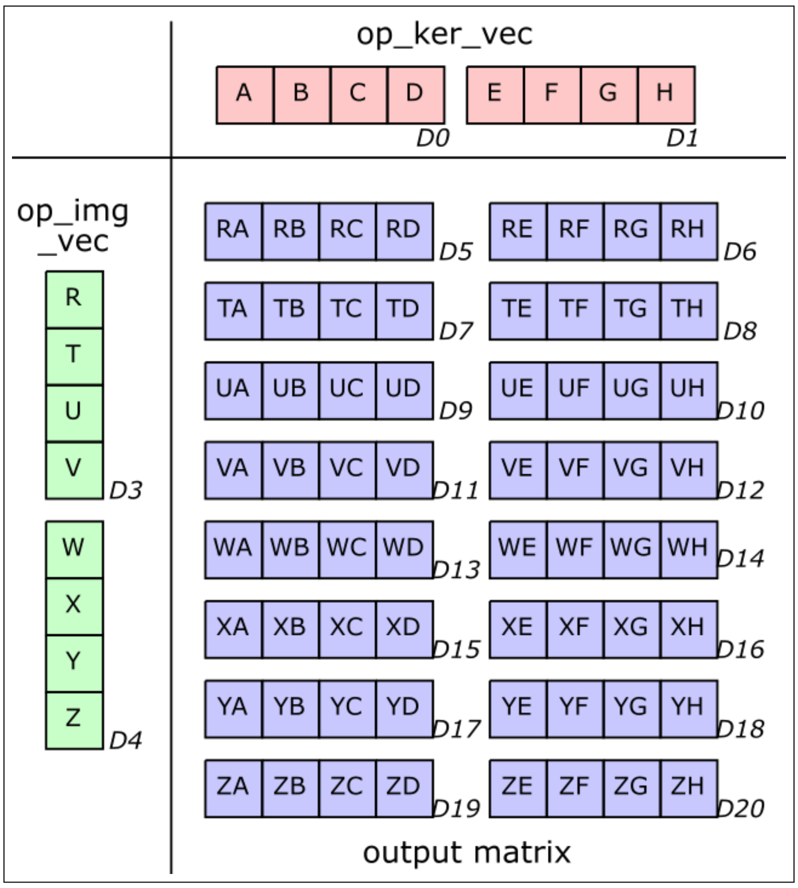


Figure 5-11: Using 2 4-lane NEON registers to construct logically eight-lane $op\_ker\_vec$ and $op\_img\_vec$. 16 registers are used to store an $(8 \times 8)$ output matrix.

The larger the size of $L$, the greater the ratio between computational operations and memory operations. By choosing vector registers with more lanes, we can increase $L$. However, we can also increase $L$ by using multiple vectors to represent the conceptional input vectors to the outer product. Figure 5-11 shows an example of this, $op\_ker\_vec$ and $op\_img\_vec$ are made from 2 NEON registers each. The size of $L$ is limited by the number of registers available in the architecture being used, because we need enough registers to store the output of the outer products efficiently. The Intel AVX architecture exposes 16 256-bit YMM vector registers. The largest output matrix we can create with this is an $(8 \times 8)$ output matrix, with 8 AVX regsiters used for the matrix, one to store $op\_ker\_vec$, and one to store $op\_img\_vec$, and six left unused. The ARMv7 and ARMv8 NEON architecture offer 32 128-bit NEON registers. Using 29 NEON registers we can create an output matrix of $(12 \times 8)$, with twenty-four NEON registers for the output matrix, two registers for a logically eight-lane $op\_ker\_vec$, and three registers for a logically twelve-lane $op\_img\_vec$.

Special data layouts for the input tensors are required to improve cache performance when accessing the input tensor and kernel. Figure 5-10 shows pseudo-code for implementing a NEON *outer product* microkernel. The code in the figure minimizes the number of writes to the output image required by nesting the $Y$, $X$, and $C$ loops inside the $M$ and $W$ loops. As we can loop across the $Y$, $X$, and $C$ without changing what points we are calculating for the output tensor, we can keep accumulating to the registers that store the outputs of the outer products without having to write them out often. However, in figure 5-10, the code does not move across the input tensor or kernel contiguously. The input image tensor, *input_image*, is a $(C \times H \times W)$ array, while the tensor of input kernel tensors, *input_kernel*, is a $(Y \times X \times C \times M)$ array. They have these data layouts so a normal SIMD load operation can be used to load the values for $op\_ker\_vec$, $op\_img\_vec1$, and $op\_img\_vec2$ quickly, without having to perform an expensive gather operation. However, we iterate across the $C$ dimension in the innermost loop of figure 5-10. This will most likely reduce cache performance as we making large jumps every time we access the input tensors.

However, this can be solved by rearranging the input tensors so that there are $L$ contiguous values from the vectorized dimension as the innermost dimension of the input tensors, but the entire vectorized dimension is dispersed across the entire input tensors. For example, the new input tensors for figure 5-10 would have the layouts $(H \times (W/4) \times C \times 4)$, and $((M/4) \times Y \times X \times C \times 4)$ for the input image tensor, and the tensor of input kernel tensors respectively. With these data layouts, we can iterate across the $C$ dimension as the innermost loop of the microkernel, while still being able to use a normal SIMD load instruction to create $op\_ker\_vec$, $op\_img\_vec1$, and $op\_img\_vec2$ quickly, and have better cache performance as we would now be iterating across the input tensors contiguously. However, for our evaluation we only allowed the input kernel to use the special data layout, because in practice we can preprocess kernel data beforehand for free, however the input image tensor is only available at runtime, so would need to reorder it during the actual inference.

### 5.4.6 Outer Product Microkernels

There are four *outer product* SIMD strategy microkernels. There are two for ARM NEON, which use vector-scaling to perform the outer product, and two for Intel AVX, which use vector reordering to calculate the outer product values. Each architecture has a microkernel that contains the smallest number of loops, but will need to write out the outer product matrix often, and a larger microkernel that contains more loops, allowing it write out the outer product matrix less often. Loop unrolling is used to search for the best performing length for $op\_ker\_vec$ and $op\_img\_vec$. Genvolution will also investigate if $op\_ker\_vec$ and $op\_img\_vec$ should be of equal length or not for better performance. The microkernels can be generated to use the standard input data layouts, or the special layouts covered in the previous section.

## 5.5 W-M-C SIMD Strategy

### 5.5.1 Implementation Outline

The *W-M-C* SIMD strategy is inspired by an implementation for SGEMM (binary32 GEneral Matrix Multiplication) in the QNNPack CNN Library (Facebook 2018). Like the *outer product* SIMD strategy, the *W-M-C* SIMD strategy also implements an outer product along the $M$ dimension of the input kernels, and the $W$ dimension of the input image tensor. However, instead of vectorizing along the $W$ dimension of the input image tensor, the *W-M-C* SIMD strategy vectorizes along the $C$ dimension of the input image tensor.

The *W-M-C* SIMD strategy is an ARM NEON only SIMD strategy. This is because its design relies heavily on the vector-scale-and-add *vmlaq_lane_f32* operation detailed in section 5.4.3. On many ARM devices, *vmlaq_lane_f32* is significantly faster than using a separate scale and add operations. For example, on the Cortex A-7 processor, the result of *vmlaq_lane_f32* is available after 5 cycles, while the result of a separate scale and add operation is available after 8 cycles (Rullgard 2014).

Figure 5-12 shows the major steps of the *W-M-C* SIMD strategy. The innermost loop of the *W-M-C* SIMD strategy iterates across the $C$ dimension in increments of two. Inside this loop, two vectors of length $L$ are read from the input kernels. The two vectors are vectorized along the $M$ dimension, however one vector takes values from the current position in the $C$ dimension (see line 21 of figure 5-13), and the other takes values from the current position in the $C$ dimension plus one (see line 24 of figure 5-13). In part (ii) of figure 5-12, the two vectors from the input kernels contain {A, C, E} and {B, D, F} respectively. The {A, C, E} vector takes values from the $C[0]$ dimension, and the {B, D, F} vector takes values from the $C[1]$ dimension.

While the two vectors from the input kernels are being loaded, $L$ 2-lane vectors from the input image tensor are also created. The $L$ vectors take values from consecutive points in the $W$ dimension. For example, in figure 5-12, there are 3 vectors taken from the input image tensor. The first vector takes the first

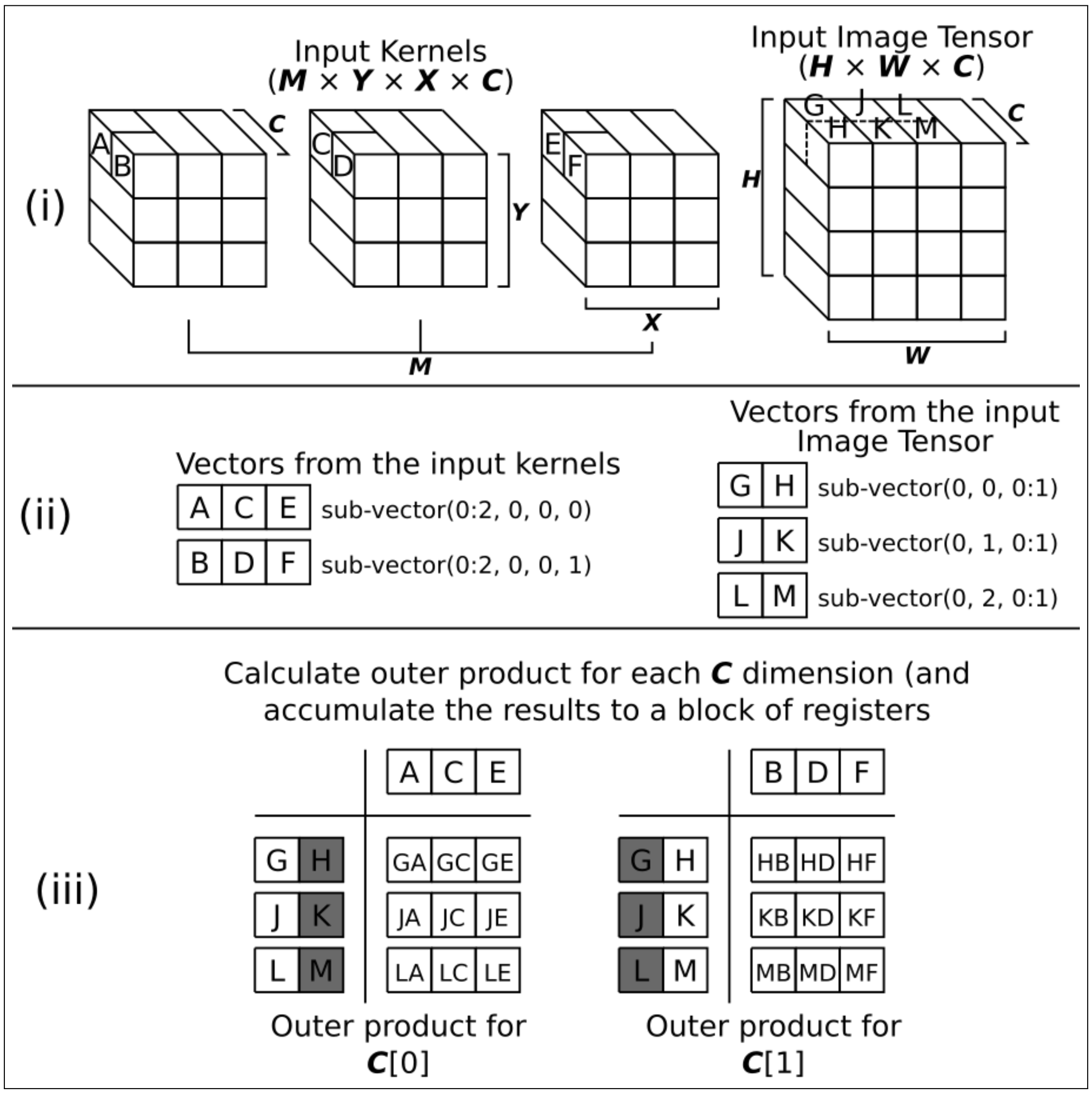

Input Kernels
(M × Y × X × C)
Input Image Tensor
(H × W × C)
(i)
A
B
C
D
E
F
G
H
J
K
L
M
C
Y
X
M
H
W
(ii)
Vectors from the input kernels
A C E sub-vector(0:2, 0, 0, 0)
B D F sub-vector(0:2, 0, 0, 1)
Vectors from the input Image Tensor
G H sub-vector(0, 0, 0:1)
J K sub-vector(0, 1, 0:1)
L M sub-vector(0, 2, 0:1)
(iii)
Calculate outer product for each C dimension (and accumulate the results to a block of registers
A C E
GA GC GE
JA JC JE
LA LC LE
B D F
HB HD HF
KB KD KF
MB MD MF
Outer product for C[0]
Outer product for C[1]

Figure 5-12: The major steps in the *W-M-C* SIMD strategy.

two input channels from the point at $W[0]$ (the {G, H} vector), the second vector takes from the $W[1]$ point (the {J, K} vector), and the third vector takes from the $W[2]$ point (the {L, M} vector). These three vectors are vectorized along the $C$ dimension, and take values from the same position in the $C$ dimension as the two vectors taken from the input kernels.

The first $L$-lane vector taken from the input kernels is then used with the first value from ech of the $L$ vectors taken from the input image tensor to calculate an $(L \times L)$ outer product. See the section labelled 'Outer product for $C$[0]' from part (iii) of figure 5-12 for an example. The same is then done with the second vector from the input kernels, and the other lane from the vectors taken from the input image tensor. The results from both these outer products are accumulated onto a set of registers like in the *outer product* SIMD strategy.

Each outer product is calculated using the *vmlaq_lane_f32* operation, using a vector from the input kernels, and a lane from each of vectors from the input image tensor (see lines 25 to 28 of figure 5-13). In figure 5-12, 3-lane vectors are taken from the input kernels, however in practice the vectors must be 4-lanes to match the parameter requirements of *vmlaq_lane_f32*. However, we can take more vectors from input kernels to create logically 8 or 12-lane input vectors for the outer product, building the larger vectors from many 4-lane vectors. The data layout of the input kernels shown in figure 5-12 is also incorrect. In practice, the special data layout $(M/L \times Y \times X \times C \times L)$ must be used for the input kernels (see line 2 of figure 5-13 for an example, where $L = 4$). The special data layout allows the code to iterate over the input kernels contiguously, while also being able to load vectors from the input kernels using a normal SIMD vector load operation (see line 21, 24 of figure 5-13).

### 5.5.2 Effects On Performance

The rational for the design of the *W-M-C* SIMD strategy is similar to the rational of the *outer product* SIMD strategy. The *W-M-C* SIMD strategy performs an outer product that requires $L$ data to calculate $L^2$ values, creating an excellent computational work to memory access ratio. This reduces pressure on the memory hierarchy, and on the bandwidth between the caches and the processors. It is hoped that this will keep any implementation computational bound, rather then memory bound from data stalls.

Also similarly to the *outer product* SIMD strategy, the result of the outer products are stored in a set of registers. A larger output matrix creates a better computational:memory ratio, but the size of the matrix is bound by the number of registers available. The max possible output matrix is a $(8 \times 12)$ output matrix, using 24 NEON registers, 4 NEON registers for values from the input image tensor, and 3 NEON registers for values from the input kernels, leaving 1 register unused. The registers storing data from the input image tensor also

```
//input_image[H][W][C]
//input_kernel[M/4][Y][X][C][4]
//output_image[H][W][M]
for (signed w = 0; w < W; w+=4) {
  for (signed m = 0; m < (M/4); mf++) {
    float32x4 mat_row_0  =  {0, 0, 0, 0};
    float32x4 mat_row_1  =  {0, 0, 0, 0};
    float32x4 mat_row_2  =  {0, 0, 0, 0};
    float32x4 mat_row_3  =  {0, 0, 0, 0};
    for (signed c = 0; c < CHANNELS; cf+=2) {
      float32x2 iV0;
      iV0 = load_32x2(&(input_image[h+y][w+x+0][c]));
      float32x2 iV1;
      iV1 = load_32x2(&(input_image[h+y][w+x+1][c]));
      float32x2 iV2;
      iV2 = load_32x2(&(input_image[h+y][w+x+2][c]));
      float32x2 iV3;
      iV3 = load_32x2(&(input_image[h+y][w+x+3][c]));
      float32x4_t kV0;
      kV0 = load_32x4(
        &(input_kernel[m][y+(Y/2)][x+(X/2)][c+0][0]));
      float32x4_t kV1;
      kV1 = load_32x4(
        &(input_kernel[m][y+(Y/2)][x+(X/2)][c+1][0]));
      vsum00 = vmlaq_lane_f32(mat_row_0, kV0, iV0, 0);
      vsum01 = vmlaq_lane_f32(mat_row_1, kV0, iV1, 0);
      vsum10 = vmlaq_lane_f32(mat_row_2, kV0, iV2, 0);
      vsum11 = vmlaq_lane_f32(mat_row_3, kV0, iV3, 0);

      vsum00 = vmlaq_lane_f32(mat_row_0, kV1, iV0, 1);
      vsum01 = vmlaq_lane_f32(mat_row_1, kV1, iV1, 1);
      vsum10 = vmlaq_lane_f32(mat_row_2, kV1, iV2, 1);
      vsum11 = vmlaq_lane_f32(mat_row_3, kV1, iV3, 1);
    }
    store_32x4(&(output_image[h][w+0][m*4]), mat_row_0);
    store_32x4(&(output_image[h][w+1][m*4]), mat_row_1);
    store_32x4(&(output_image[h][w+2][m*4]), mat_row_2);
    store_32x4(&(output_image[h][w+3][m*4]), mat_row_3);
  }
}

```

Figure 5-13: ARMv7 NEON Pseudocode for a *W-M-C* microkernel.

only need to be updated after every second outer product, as only a lane from each float_32x2 vector is used. (Note that a single 128-bit NEON register can store two float_32x2 vectors at the same time).

The most important difference between the *W-M-C* and the *outer product* SIMD strategy is that vectorizing along the $C$ means that the input image tensor can be in $(W \times H \times C)$ format. In the *outer product* SIMD strategy, the $W$ dimension must be the innermost dimension of the input image tensor, because we use vectors that vectorize across the $W$ dimension in the *outer product* SIMD strategy. To reduce the number of times that the outer product running total (stored in the set of registers) must be written out, we want the $C$ dimension as the innermost loop of the microkernel, because moving across the $C$ dimension does not change what output values we are currently calculating. With the input image tensor being stored in $(W \times H \times C)$ format for the *W-M-C* SIMD strategy, we can loop across the $C$ dimension in the innermost loop while still moving across the input image tensor contiguously, which is not possible with the *outer product* SIMD strategy.

### 5.5.3 ***W-M-C*** Microkernels

There are two *W-M-C* microkernels. One contains the minimal number of loops in the microkernel (i.e. loops for $M$, $W$, and $C$). The output matrix needs to written at the end of every $C$ loop. There is also a larger microkernel which nests a $Y$ and a $X$ loop inside the $W$ loop. This reduces the number of times the output matrix needs to be written out.

This chapter concludes the discussion of Genvolution as it relates to CNN convolution. The following chapter details our other largest topic of research: Winogen, a second automatic program generator for Winograd CNN convolution.

# Chapter 6

# Automatic Winograd Optimization and Generation

## 6.1 Chapter Motivation

The majority of the runtime of a CNN is spent performing CNN convolution. Therefore, by improving the performance of CNN convolution, we improve the performance of many CNNs.

Winograd CNN convolution is one method for implementing CNN convolution (Lavin and Gray 2016). Winograd convolution involves reducing the computational complexity of CNN convolution using minimal multiplication algorithms. It has been shown that Winograd convolution can be extremely performant, and outperform other CNN convolution methods, such as Im2Col (Maji et al. 2019).

However, implementing Winograd convolution efficiently can be a time consuming task. Winograd convolution involves three large data transformations that must be optimized to have an efficient implementation. There are also many different sets of transformations that offer different trade-offs in computational complexity reduction and additional memory required. Depending on the input sizes for a given CNN convolution, different transformation sets may perform more efficiently. There are also a number of other optimizations that can be applied to a Winograd convolution implementation.

To explore the impact of selecting different transformation sets and other optimizations, we developed Winogen. Winogen is a domain specific program generator that automatically generates and optimizes Winograd convolution implementations given a set of CNN convolution input sizes.

While Winograd convolution implementations can be very fast, they require a large temporary memory overhead to store the transformed data structures. This can be a problem on memory constrained devices, such as embedded ARM devices. We propose a novel CNN convolution algorithm which uses a sum of 1D Winograd convolution to perform normal CNN convolution. Our novel algorithm uses significantly less memory than the standard Winograd CNN convolution algorithm, while still reducing the computational complexity of CNN convolution.

## 6.2 Winograd Convolution

Convolution is a form of filtering, where the input kernel acts as a filter on the input tensor. Normally, performing a 1D convolution using an input kernel with $ker\_els$ elements to produce $out\_els$ output points requires $(ker\_els \times out\_els)$ multiplications. This is because for each output point we must perform $ker\_els$ pair-wise multiplications between the values in the input kernel, and values in the input tensor. However, it is possible the perform convolution using fewer multiplications.

Winograd showed that, given a kernel with $ker\_els$ values, the minimum number of multiplications required to calculate $out\_els$ output points is $(ker\_els + out\_els - 1)$ multiplications (Winograd 1980). $(in\_els)$ is the number of elements needed in the input tensor to perform the convolution, and is equal to the minimal number of multiplication required. In other words, $in\_els = (ker\_els + out\_els - 1)$. We use $F(out\_elements, ker\_elements)$ to refer to a minimal multiplication convolution that uses a input kernel with $ker\_elements$ to calculate $out\_element$ output values (and which needs an input tensor with $(ker\_els + out\_els - 1)$ elements).

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

where

$$m_1 = (d_0 - d_2)g_0$$
$$m_2 = (d_1 + d_2)(\frac{g_0 + g_1 + g_2}{2})$$
$$m_3 = (d_2 - d_1)(\frac{g_0 - g_1 + g_2}{2})$$
$$m_4 = (d_1 - d_3)g_2$$

Figure 6-1: Minimal multiplication algorithm for performing $F(2,3)$ using 4 multiplications.

Figure 6-1 shows an example of a minimal multiplication algorithm for $F(2,3)$ created by Winograd, which uses 4 multiplications to perform 1D convolution. A normal convolution would require 6 multiplications to calculate the same number of output points. The divisions required to calculate $m_2$ and $m_3$ in figure 6-1 are not included in multiplication count, because they are constants that only involve values from the input kernel $g$, and can be pre-calculated once in advance before performing any convolutions. Minimal multiplication algorithms for 1D convolution can also be written in a matrix form. Figure 6-2 gives the equation for the matrix form of the minimal multiplication algorithms. For example, the algorithm shown in figure 6-1 can be represented using the matrices shown in figure 6-3. Figure 6-4 shows an example of using the matrices in figure 6-3 to perform 1D convolution.

$$Y = A^T \times ((G \times g) \odot (B^T \times d))$$

where
$\odot$ is element-wise multiplication.
$g$ is the input kernel.
$d$ is the input tensor.
$A^T, G, B^T$ are the matrices representing the minimal multiplication algorithm.

Figure 6-2: Matrix form algorithm for 1D minimal multiplication algorithms.

The matrices $A^T$, $G$, and $B^T$ represent the minimal multiplication algorithm. The input vector $d$ is multiplied with $B^T$ to produce a new vector $trans\_in\_vec$, the input kernel $g$ is multiplied with $G$ to produce a new vector $trans\_ker\_vec$.

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

Figure 6-3: Matrices representing the minimal multiplication algorithm for $F(2,3)$ shown in figure 6-1.

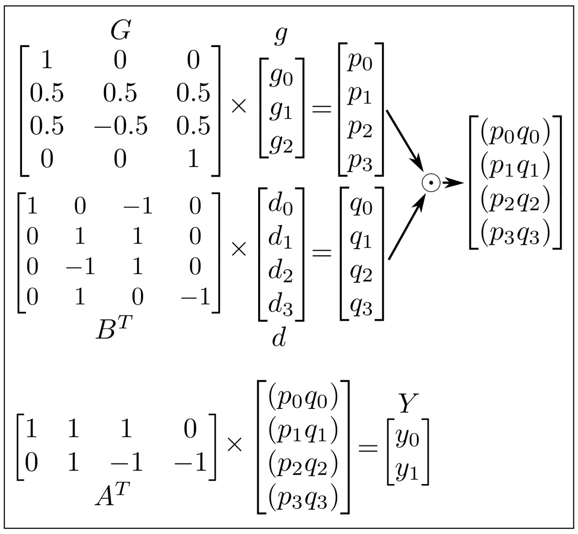


Figure 6-4: Performing the minimal multiplication algorithm for $F(2,3)$ shown in figure 6-1, using the matrices given in figure 6-3.

An element-wise multiplication is performed between $trans\_in\_vec$ and $trans\_ker\_vec$ to produce a new vector $trans\_product$. The vector $trans\_product$ is multiplied by $A^T$ to produce the output of a 1D convolution between $d$ and $g$. The triplet of $A^T$, $G$, and $B^T$ will be referred to as the *transformation matrices*. A different triplet of transformation matrices is needed to perform a minimal multiplication algorithm depending on the size of the input kernel and the number of output points wanted. For example, $F(2,3)$ will use a different triplet of transformation matrices than $F(4,3)$ or $F(2,5)$. Triplets of transformation matrices can be calculated using *Winograd's algorithm* or the *Toom-Cook algorithm* (Blahut 2010). In the CNN community, performing convolution using transformation matrices generated using either algorithm is usually called *Winograd convolution*, no matter which algorithm was used to generate the matrices. Also, there are an infinite number of different valid minimal multiplication algorithms (and matching *transformation matrices*) for all input sizes.

### 6.2.1 2D Winograd Convolution

It is valid to combine two 1D minimal multiplication convolution algorithms to create a 2D minimal multiplication algorithms. Given the 1D minimal multiplication algorithms $F(out\_els, ker\_els)$ and $F(out\_els', ker\_els')$, we can create the 2D minimal filtering algorithm $F(out\_els \times out\_els', ker\_els \times ker\_els')$ by nesting one 1D algorithm inside another 1D algorithm (Lavin and Gray 2016). The number of multiplications required for $F(out\_els \times out\_els', ker\_els \times ker\_els')$ is $(ker\_els + out\_els - 1)(ker\_els' + out\_els' - 1)$. The size of the input tensor required for $F(out\_els \times out\_els', ker\_els \times ker\_els')$ is also $(ker\_els + out\_els - 1)(ker\_els' + out\_els' - 1)$. We can also nest the same 1D minimal multiplication algorithm with itself to produce a 2D minimal multiplication algorithm. The matrix form equation for performing a 2D minimal multiplication algorithm created from nesting a 1D minimal multiplication algorithm with itself is shown in figure 6-5. The same transformation matrices are used for $F(out\_els, ker\_els)$ and $F(out\_els \times out\_els, ker\_els \times ker\_els)$. For example, $F(2, 3)$ and $F(2 \times 2, 3 \times 3)$ use the same transformation matrices.

$$Y = A^T \times ((G \times g \times G^T) \odot (B^T \times d \times B)) \times A$$

where
$\odot$ is element-wise multiplication.
$g$ is the input kernel.
$d$ is the input tensor.
$A^T, G, B^T$ are the matrices representing the minimal multiplication algorithm.

Figure 6-5: Matrix form algorithm for 2D minimal multiplication algorithms created from nesting the same 1D minimal multiplication algorithm.

## 6.3 Winograd CNN Convolution

Lavin et al. proposed implementing CNN Convolution using Winograd convolution. Lavin et al. also showed that CNN convolution implemented using Winograd convolution can outperform other CNN convolution algorithms (Lavin and Gray 2016). Winograd convolution reduces the number of multiplication required to perform convolution, while increasing the number of additions and requiring some kernel pre-processing. In general, the number of required multiplications is reduced by $((ker\_els^2)(out\_els^2))/(ker\_els + out\_els - 1)^2$ times when using a 2D Winograd convolution. For example, $F(2 \times 2, 3 \times 3)$ reduces the number of required multiplications by 2.25$\times$ (from 36 to 16 multiplications), and $F(4 \times 4, 3 \times 3)$ reduces it by 4$\times$ (from 144 to 36 multiplications). When performing floating-point arithmetic, floating-point multiplications are usually more expensive to perform then floating-point additions. For example, on the Intel Skylake architecture, a binary32 floating-point addition has 3 cycles of latency, while a binary32 floating-point multiplication has 5 cycles of latency (Fog 2018). By reducing the number of expensive floating-point multiplications required, we hope that performance of the CNN convolution increases. To implement CNN convolution using Winograd convolution, a number of issues must be accounted for that are covered in sections 6.4 through 6.5.

## 6.4 Winograd CNN Convolution Outline

Winograd CNN convolution takes in an input tensor $in\_tensor$ and an input kernel $in\_kernel$. It produces an output tensor $out\_tensor$ that is the CNN convolution between $in\_tensor$ and $in\_kernel$. A simple outline of Winograd CNN convolution is:

1. $G * in\_kernel * G^T$ is calculated. The result of this is a transformed kernel tensor, denoted by $trans\_ker$.

2. $B^T * in\_tensor * B$ is calculated. The result of this is a transformed input tensor, denoted by $trans\_in$.

3. Calculate the element-wise product between the elements of $trans\_ker$ and $trans\_in$. The result of this is denoted by $trans\_product$. $trans\_product$ contains the result of the convolution between $in\_tensor$ and $in\_kernel$, in a transformed format (sometimes referred to as "in the Winograd domain").

4. $A^T * trans\_product * A$ is calculated. The result of this is the CNN convolution between $in\_tensor$ and $in\_kernel$. The result is stored in $out\_tensor$.

## 6.5 Fixed Input Sizes

### 6.5.1 Upper Bound on Input Sizes

Winograd convolution algorithms function on inputs and output tensors of known sizes (Blahut 2010). For example, $F(2 \times 2, 3 \times 3)$ only accepts an input tensor of size $(4 \times 4)$, input kernel of size $(3 \times 3)$, and produces an output tensor of size $(2 \times 2)$. A Winograd convolution algorithm for any input size can be generated. For example, the Toom-Cook algorithm can be used to generate a triplet of transformation matrices for a $F(48 \times 48, 11 \times 11)$ 2D Winograd convolution algorithm, if desired. However, as the input sizes of a Winograd convolution increases, the inaccuracy of the result also increases. Barabasz et al. proved that the error introduced by the Toom-Cook algorithm grows at least exponentially with the size of the input tensor in the worst case (Barabasz and Gregg 2019). Empirically, we found that Winograd convolution algorithms that accepted an input tensor larger than $(8 \times 8)$ produced results that were consistently accurate to only two significant digits (i.e. the third digit of most produced results was wrong).

CNN convolutions often work on much larger input tensors. For example, half of the layers of the MobileNet V2 CNN use an input image tensor with an image height and width of at least 56 (Howard et al. 2017). We cannot generate a Winograd convolution algorithm that takes input tensors of this size directly while retaining accuracy. Lavin et al. propose using a tiling approach, where we split large input and output tensors into a number of smaller tensors, and perform convolution on the smaller tensors separately using Winograd convolution (Lavin and Gray 2016).

### 6.5.2 Tiling the Input and Output Tensor

Given a 2D Winograd convolution $F(out\_els \times out\_els, ker\_els \times ker\_els)$, we can perform a convolution with an input and output tensor of arbitrary size by splitting the input and output tensor into tiles. The size of the tiles will match the expected input and output sizes of $F(out\_els \times out\_els, ker\_els \times ker\_els)$. The output tensor $out\_tensor$ is split into non-overlapping tiles of size $(out\_els \times out\_els)$. The tiles covering $out\_tensor$ are referred to as *o-tiles*. The entire $out\_tensor$ needs to be covered by o-tiles, meaning we need $((\lceil H/out\_els\rceil) \times (\lceil W/out\_els\rceil))$ o-tiles in total, where $H$ is the height of $out\_tensor$ and $W$ is the width of $out\_tensor$. For example, in figure 6-6, we are using a $F(2 \times 2, 3 \times 3)$ Winograd convolution, and an $out\_tensor$ with size $(4 \times 3)$. The $out\_tensor$ is split into four o-tiles (labelled o-tiles 0 through 3).

We use a chosen 2D Winograd convolution algorithm $F(out\_els \times out\_els, ker\_els \times ker\_els)$ to produce the output for each o-tile that makes up $out\_tensor$. To do this, for each o-tile, we need a tile containing the relevant data from the input tensor. Each o-tile has a matching tile in the input tensor. Tiles from the input tensor are referred to as *i-tiles*. For example, in figure 6-6, o-tile 0 from $out\_tensor$ is matched with i-tile 0 from $in\_tensor$, o-tile 1 from $out\_tensor$ is matched with i-tile 1 from $in\_tensor$, and so on. The size of each i-tile will be $(in\_els \times in\_els)$, where $in\_els = (out\_els + ker\_els - 1)$. This is because the size of the input tensor required by a $F(out\_els \times out\_els, ker\_els \times ker\_els)$ 2D Winograd convolution algorithm is $(in\_els \times in\_els)$. Each i-tile is positioned at a different point on the $in\_tensor$, and the i-tile's position corresponds the position of its matching o-tile on the $out\_tensor$. Assuming the top-left element of a given o-tile was at position $(oth, otw)$ in the $out\_tensor$, then the top-left element of the o-tile's matching i-tile will be at position $(oth - \lfloor K/2\rfloor, otw - \lfloor K/2\rfloor)$, where $K$ is the height and width of the input kernel. For example, in figure 6-6,

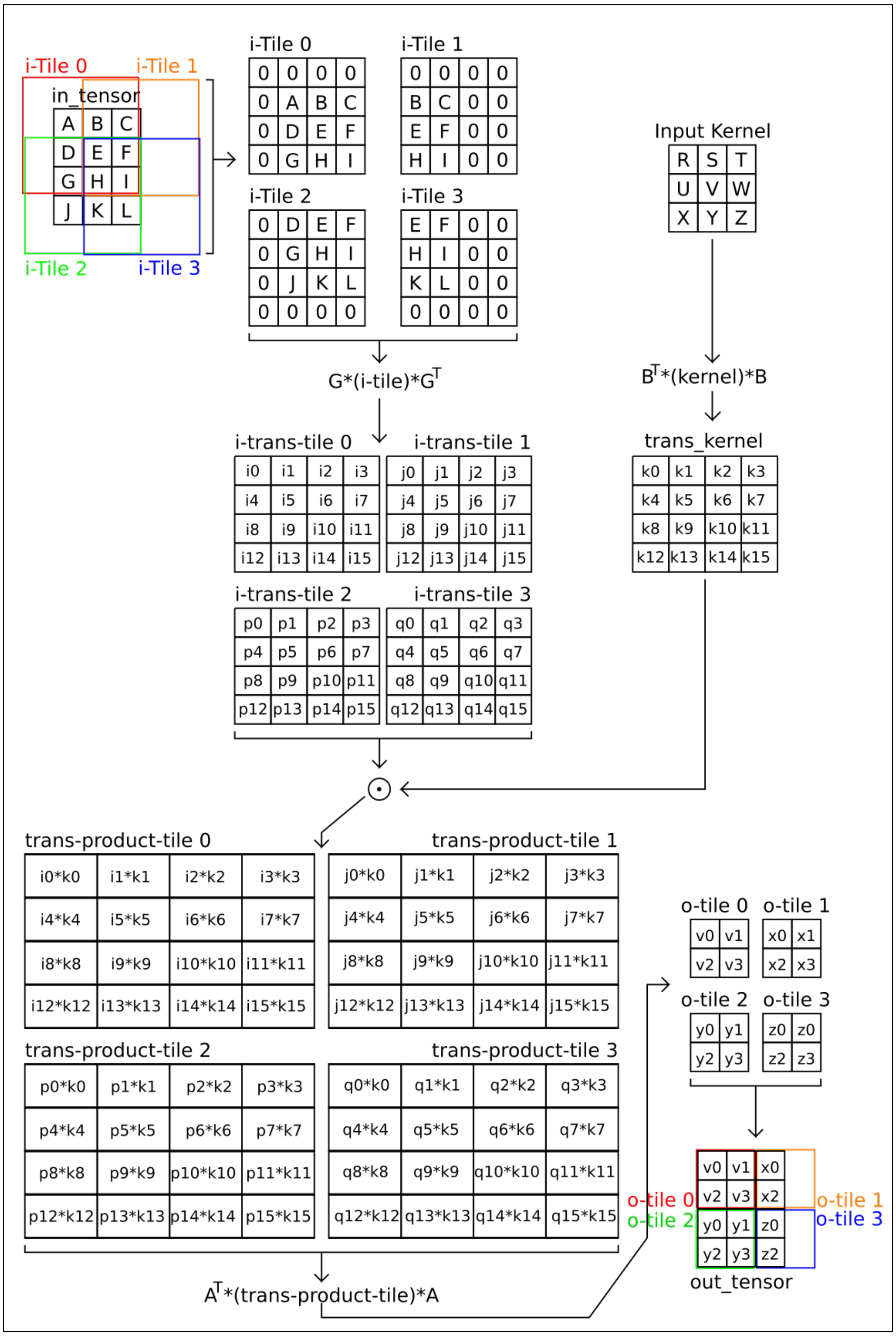


Figure 6-6: Performing a CNN Convolution by performing 4 smaller 2D Winograd convolutions on tiles from the input tensor.

the top-left element of o-tile 1 is at position $(0, 2)$, and the top-left element of i-tile 1 is at position $(-1, 1)$, because $(0 - \lfloor 3/2 \rfloor, 2 - \lfloor 3/2 \rfloor) = (-1, 1)$.

Some parts of an i-tile can lie outside the bounds of the input tensor. For example, in figure 6-6, the first row and the first column of i-tile 0 lie outside the bounds of $in\_tensor$. These values in the i-tiles need to be synthesized. They can be synthesized using any of the standard methods for synthesizing extra input tensor values, e.g. zero padding, mirroring values. In figure 6-6, zero padding is used. For example, the first row and column of i-tile 0 are filled with zeroes.

In CNN convolution, the height and width of the input kernels are often very small, usually either $(3 \times 3)$ or $(5 \times 5)$. The input kernels of CNN convolution are small enough that they do not need to be split into tiles to be used with a Winograd convolution algorithm. For example, the $F(2 \times 2, 3 \times 3)$ Winograd convolution algorithm accepts a $(3 \times 3)$ kernel. In figure 6-6, the input kernel is not tiled because it is only $(3 \times 3)$ in size.

### 6.5.3 Winograd Convolution per Tile

Once the input tensor $in\_tensor$ is split into i-tiles, a 2D winograd convolution is performed between each i-tile and the input kernel. The results of the convolutions are the o-tiles that cover the output tensor $out\_tensor$.

1. Each i-tile is transformed into a *i-trans-tile* using the equation $B^T \times d \times B$ where $d$ is one of the i-tiles. For example, in figure 6-6, i-tiles 0 through 4 are transformed into the i-trans-tiles 0 through 4.

2. The input kernel is transformed into the $trans\_kernel$, using the equation $G \times g \times G^T$, where $g$ is the input kernel.

3. An element-wise multiplication is performed between each i-trans-tile and the $trans\_kernel$. The result of these are the *trans-product-tiles*. For example, in figure 6-6, the element-wise multiplication between i-trans-tile 2 and the $trans\_kernel$ produces trans-product-tile 2.

4. Each trans-product-tile is transformed by the equation $A^T \times a \times A$ to produce an o-tile, where $a$ is a trans-product-tile. For example, in figure 6-6, trans-product-tile 2 is transformed into o-tile 2.

5. The results in the o-tiles are stored into the output tensor $out\_tensor$. Some values in the o-tiles are not required, and are discarded. For example, the values in the second column of o-tile 1 in figure 6-6 are not needed, because they are outside the boundaries of the output. The output tensor now contains the result of the CNN convolution between the input tensor and the input kernel.

Figure 6-7 shows simplified pseudocode that uses the tiled Winograd convolution method to perform single-channel CNN convolution.

## 6.6 Handling Input Channels

2D Winograd Convolution works on single-channel inputs to produce single-channel outputs. To perform multi-channel CNN convolution using 2D Winograd convolution, a separate single-channel convolution is performed for every input channel in the input tensor. The results of each single-channel convolution are then summed to produce the multi-channel convolution result. For example, in figure 6-8, the input tensor $in\_tensor$ and the input kernel $in\_kernel$ both have 5 input channels. The loop from lines 4 to 12 performs a single channel convolution for each input channel. The results of each convolution are accumulated in the output tensor $out\_tensor$ on line 11.

In figure 6-8, we must apply a transformation to the values in $trans\_product$ 5 times, once for each input channel. The results of the transformation are accumulated in $out\_tensor$. However, accumulating all the results in the $trans\_product$ tensor, and then only transforming it once at the end will yield the same results (Lavin and Gray 2016). Line 10 of figure 6-9 shows an example of this, where we accumulate the results of the element-wise multiplications between $trans\_in$ and $trans\_kernel$ in $trans\_product$. The values in $trans\_product$ are

```
in_tensor[H][W]
input_kernel[K][K]
out_tensor[H][W]
int out_els = 2; //size of the output tiles
int in_els = out_els + K - 1; //size of the input tiles
//number of i-tiles/o-tiles needed is TILED_H*TILED_W
int TILED_H = ceiling(H, out_els)
int TILED_W = ceiling(W, out_els)
i_tiles[TILED_H*TILED_W][in_els*in_els]
trans_i_tiles[TILED_H*TILED_W][in_els*in_els]
trans_kernel[in_els*in_els]
trans_product_tiles[TILED_H*TILED_W][in_els*in_els]
o_tiles[TILED_H*TILED_W][out_els*out_els]

//split image into i_tiles and transforms them
i_tiles = splitIntoTiles(input_image);
trans_i_tiles = transformITiles(i_tiles)
//transform kernel to winograd domain
trans_kernel = transformKernel(input_kernel)

for (signed i = 0; i < TILED_H*TILED_W; i++) {
  trans_product_tiles[i] = element_wise_mult(
      i_trans_tile[i], trans_kernel);
}
//transform result back to spatial domain
o_tiles = transformOutput(trans_product_tiles);
output_image = insertTiles(o_tiles);

```

Figure 6-7: Pseudo-code implementation of a single-channel CNN convolution using a tiled Winograd Convolution algorithm.

```
in_tensor[5][4][4]; //in_tensor[C][H][W]
in_kernel[5][3][3]; //input_kernel[C][K][K]
out_tensor[2][2] = {0}; //zero-out output
for (c = 0; c < 5; c++) {
  trans_in[4][4];
  trans_kernel[4][4];
  trans_product[4][4];
  trans_in = transform_input(in_tensor[c]);
  trans_kernel = transform_kernel(in_kernel[c]);
  trans_product = element_mult(trans_in, trans_kernel);
  out_tensor += transform_output(trans_product);
}

```

Figure 6-8: Performing multi-channel convolution using a sum of single-channel Winograd convolutions.

then only transformed once, on line 12. This method of handling input channels is much more efficient then the one shown in figure 6-8, because we need to only transform $trans\_product$ once, no matter how many input channels there are.

```
1   in_tensor[3][4][4]; //in_tensor[C][H][W]
2   in_kernel[3][3][3]; //input_kernel[C][K][K]
3   out_tensor[2][2];
4   trans_product[4][4] = {0}; //zero out
5   for (c = 0; c < 3; c++) {
6     trans_in[4][4];
7     trans_kernel[4][4];
8     trans_in = transform_input(in_tensor[c]);
9     trans_kernel = transform_kernel(in_kernel[c]);
10    trans_product += element_mult(trans_in, trans_kernel);
11  }
12  out_tensor = transform_output(trans_product);
13
```

Figure 6-9: Performing multi-channel convolution using a sum of single-channel Winograd convolutions.

## 6.7 Optimizing Element-Wise Multiplication

In section 6.5.2, the technique of splitting large input tensors into smaller *i-tiles* is introduced. When the i-tiles are transformed, we end up with the data structure $trans\_i\_tiles$. The transformed i-tiles $trans\_i\_tiles$ is a 3D tensor of size $(\lceil {}^{H}/_{out\_els} \rceil \times \lceil {}^{W}/_{out\_els} \rceil \times (in\_els \times in\_els))$. We also end up with the data structure $trans\_kernel$ after transforming the kernel. The transformed kernel $trans\_kernel$ is a 1D array of size $((in\_els \times in\_els))$. Figure 6-10 shows an example $trans\_i\_tiles$, and an example $trans\_kernel$, where $\lceil {}^{H}/_{out\_els} \rceil = 2$, $\lceil {}^{W}/_{out\_els} \rceil = 2$, and $in\_els = 4$.

The elements of $trans\_i\_tiles$ are reordered so that it was now a 2D tensor of size $((in\_els \times in\_els) \times (\lceil {}^{H}/_{out\_els} \rceil \times \lceil {}^{W}/_{out\_els} \rceil))$. In other words, we reordered the elements of $trans\_i\_tiles$ so that every element from the same position in the transformed i-tiles would be contiguous. For example, if we reordered the elements of $trans\_i\_tiles$ in figure 6-10, the elements would be or-

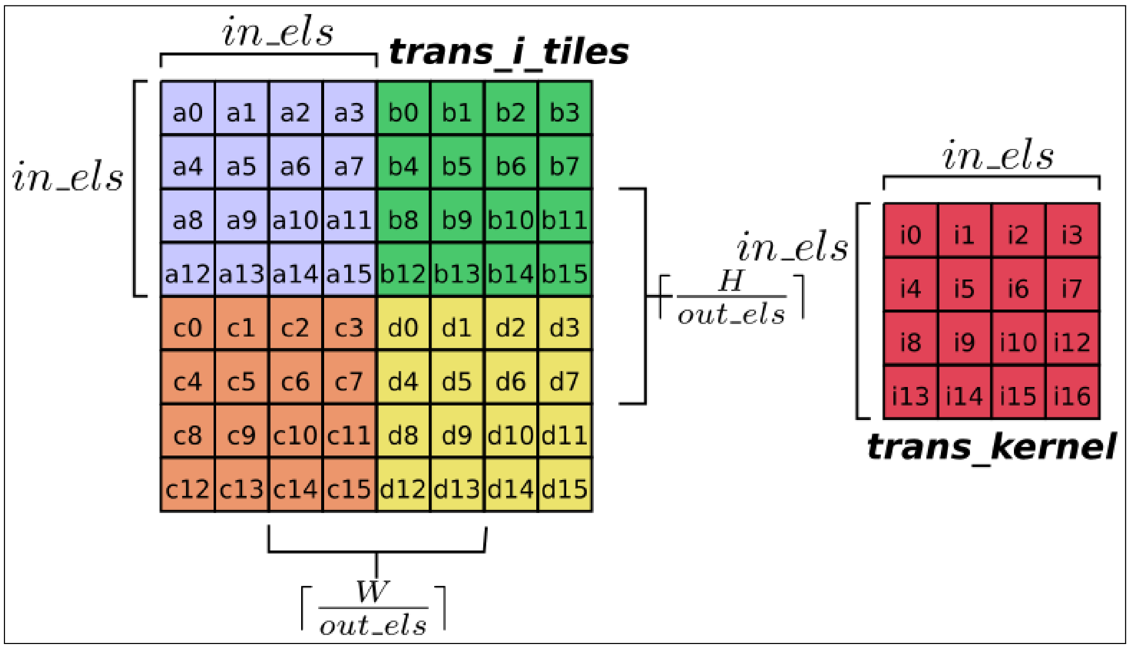


Figure 6-10: Example $trans\_i\_tiles$ and $trans\_kernel$ values.

dered $\{a0, b0, c0, d0, a1, b1, c1, d1, ...a15, b15, c15, d15\}$. In C code, the reordered $trans\_i\_tiles$ would be defined $(trans\_i\_tiles[in\_els{*}in\_els][(H/out\_els){*}(W/out\_els)])$.

The innermost dimension of the reordered $trans\_i\_tiles$ is a vector containing the values from the same position in all transformed i-tiles. For example, $(trans\_i\_tiles[2])$ would give an address pointing to an array containing the third value from every transformed i-tile. Figure 6-11 shows how we can use the reordered $trans\_i\_tiles$ to perform the element-wise multiplication between the transformed i-tiles and the transformed kernel. For each inner vector of $trans\_i\_tiles$ (i.e. $trans\_i\_tiles[i]$ in figure 6-11), we perform a vector scale with the value from the same position in $trans\_kernel$. Performing the vector scale for every point in a transformed i-tile performs all the needed element-wise multiplication between the transformed i-tiles and the transformed kernel for Winograd convolution. Lines 6 to 9 in figure 6-11 show a loop performing all the necessary vector scales.

An example vector scale between $(trans\_i\_tiles[0]$ and $trans\_kernel[0])$ is shown in figure 6-12, using values from figure 6-10. The figure shows the vector-scale between a vector containing the $(0, 0)$ point from every transformed i-tile and the $(0, 0)$ point from $trans\_kernel$.

```
//TILED_H = H/out_els;
//TILED_W = W/out_els;
//trans_i_tiles[in_els*in_els][TILED_H*TILED_W];
//trans_kernel[In_els*in_els]
//trans_product[in_els*in_els][TILED_H*TILED_W];
for (i = 0; i < in_els*in_els; i++) {
  trans_product[i] <= vector_scale(
    trans_i_tiles[i], trans_kernel[i]);
}

```

Figure 6-11: Performing Winograd Convolution using vector scales to calculate the necessary element-wise multiplications.

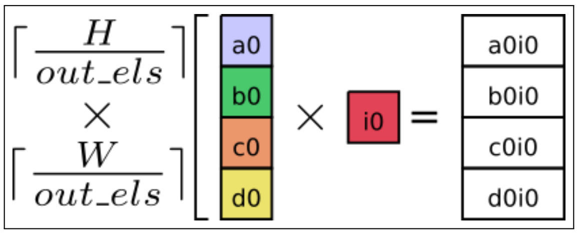

$\lceil \frac{H}{out\_els} \rceil \times \lceil \frac{W}{out\_els} \rceil$
a0
b0
c0
d0
×
i0
=
a0i0
b0i0
c0i0
d0i0

Figure 6-12: Performing a vector-scale between $trans\_i\_tiles[0]$ and $trans\_kernel[0]$.

In CNN convolution, we normally have multiple input kernels that are all applied independently to the input tensor to produce multiple 2D output tensors. Assume $M$ stands for the number of input kernels given. With multiple input kernels, $trans\_kernel$ gains another dimension. It is now a 2D tensor of size $((in\_els \times in\_els) \times M)$ or $(M \times (in\_els \times in\_els))$. Figure 6-13 is an extension to figure 6-10, where $trans\_kernel$ now contains $M$ different transformed kernels, where $M = 3$.

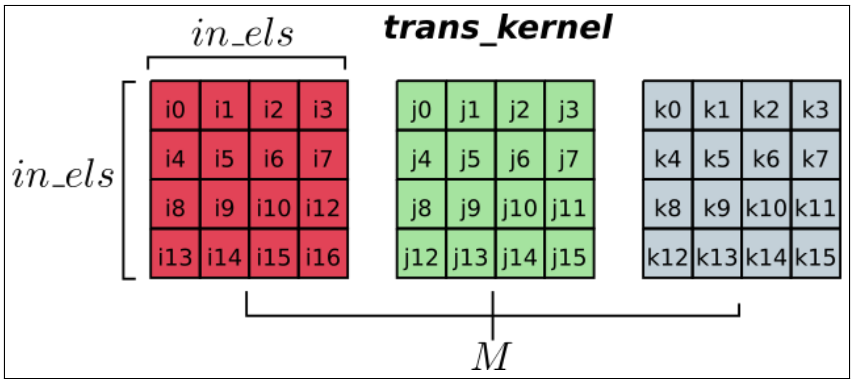


Figure 6-13: Example values for a $trans\_kernel$ constructed from $M$ different input kernels.

If we assume $trans\_kernel$ has the shape $((in\_els \times in\_els) \times M)$, a C code definition of it would be similar to $(trans\_kernel[In\_els * in\_els][M])$. The innermost dimension of this $trans\_kernel$ is a vector containing the values from the same position in all transformed kernels. For example, $(trans\_kernel[2])$ would be the address of an array containing the 3rd value from every transformed kernel. Assuming the values from figure 6-13, $(trans\_kernel[2])$ would reference a vector of length 3 containing $\{i2, j2, k2\}$. We can now calculate the element-wise multiplication between all the transformed i-tiles and the transformed kernels using outer products. Figure 6-14 shows an outer product between $(trans\_i\_tiles[0]$ and $trans\_kernel[0])$, assuming the values from figures 6-10 and 6-13. The outer product produces a matrix of size $((\lceil ^{H}/_{out\_els} \rceil \times \lceil ^{W}/_{out\_els} \rceil) \times M)$, which contains the multiplication between the $(0, 0)$ point from every transformed i-tile, and the $(0, 0)$ point from every transformed kernel. If we perform this outer product for every point in a transformed i-tile, we perform the entire element-wise multiplication required between every transformed i-tile and transformed kernel for performing Winograd convolution.

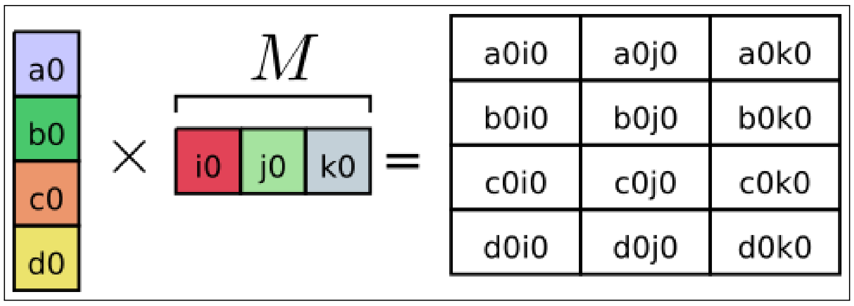


Figure 6-14: An Outer product between a vector of values taken from different i-tiles, and a vector of values taken from different transformed kernels.

Figure 6-15 shows pseudo-code using outer products to perform the element-wise multiplication. The $trans\_product$ data structure also gains an $M$ dimension, as there are now $M$ 2D output tensors. Lines 6 to 9 perform the necessary outer products.

As described in section 6.6, the input tensor and input kernels can have multiple input channels. In this case, we perform a separate convolution for every input channel, and then accumulate the results in $trans\_product$ before

```
//TILED_H = H/out_els;
//TILED_W = W/out_els;
//trans_i_tiles[in_els*in_els][TILED_H*TILED_W];
//trans_kernel[In_els*in_els][M]
//trans_product[in_els*in_els][TILED_H*TILED_W][M];
for (i = 0; i < in_els*in_els; i++) {
  trans_product[i] <= outer_product(
    trans_i_tiles[i], trans_kernel[i]);
}

```

Figure 6-15: Performing multi-channel convolution using a sum of single-channel Winograd convolutions.

transforming $trans\_product$ to produce the o-tiles. If the input tensor, and input kernels have an input channels dimension, denoted by $C$, the transformed i-tiles and the transformed kernels will also have a $C$ dimension. $trans\_i\_tile$ is now a 3D tensor with shape $((in\_els \times in\_els) \times (\lceil H/out\_els \rceil \times \lceil W/out\_els \rceil) \times C)$, and $trans\_kernel$ is now a 3D tensor with shape $((in\_els \times in\_els) \times C \times M)$. Other dimension orderings are valid, but we will assume these. Figure 6-16 shows example values for a $trans\_i\_tile$, and $trans\_kernel$ where $C = 2$, $M = 3$, $in\_els = 4$, $\lceil H/out\_els \rceil = 2$, and $\lceil W/out\_els \rceil = 2$.

We need to update the pseudocode in figure 6-15 to account for the extra dimension for input channels. To do this, the outer product is replaced with a matrix multiplication. Each matrix multiplication takes an innermost matrix from $trans\_i\_tile$ (i.e. a matrix of shape $((\lceil H/out\_els \rceil \times \lceil W/out\_els \rceil) \times C)$), and an innermost matrix from $trans\_kernel$ (i.e. a matrix of shape $(C \times M)$). Performing a matrix multiplication between them produces a matrix of shape $((\lceil H/out\_els \rceil \times \lceil W/out\_els \rceil) \times M)$, like the outer product in figure 6-15.

Figure 6-17 shows an example matrix multiplication between $trans\_i\_tiles[0]$ and $trans\_kernel[0]$, assuming the values from figure 6-16. The result matrix of the matrix multiplication contains the multiplication between the $(0, 0)$ point from every transformed i-tile, and the $(0, 0)$ point from every transformed kernel. However, the input channels have also been correctly paired together and accumulated. For example, in cell $(0, 0)$ of the result matrix in figure 6-17, $a0$ and $i0$ from the first input channel are correctly paired together, and $e0$ and $l0$

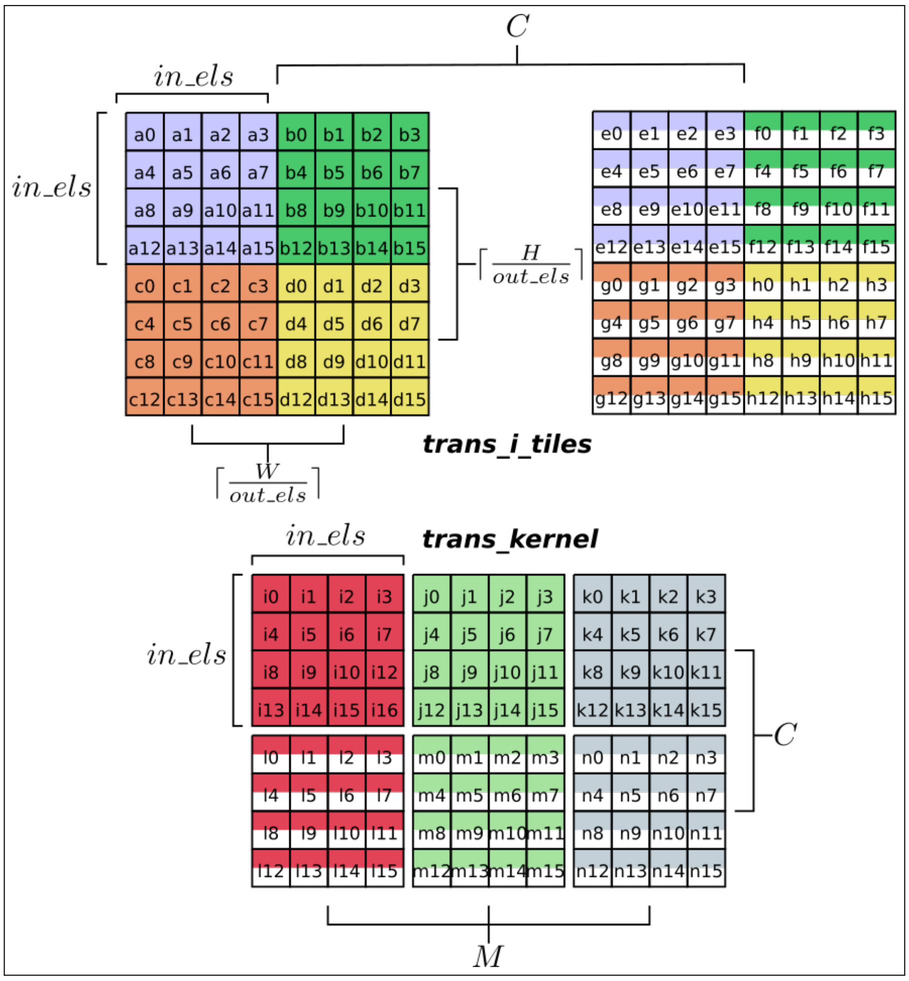

C
in_els
in_els
$\lceil \frac{H}{out\_els} \rceil$
$\lceil \frac{W}{out\_els} \rceil$
trans_i_tiles
in_els
trans_kernel
in_els
C
M

Figure 6-16: Example values for *trans_i_tiles* and *trans_kernel* with multiple input kernels and multiple input channels.

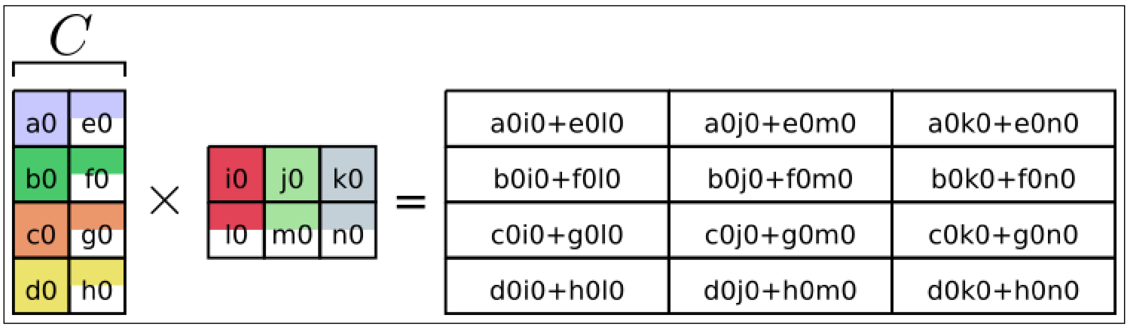

C
a0 e0
b0 f0
c0 g0
d0 h0
×
i0 j0 k0
l0 m0 n0
=
a0i0+e0l0 a0j0+e0m0 a0k0+e0n0
b0i0+f0l0 b0j0+f0m0 b0k0+f0n0
c0i0+g0l0 c0j0+g0m0 c0k0+g0n0
d0i0+h0l0 d0j0+h0m0 d0k0+h0n0

Figure 6-17: Performing a matrix multiplication between a matrix containing values from *trans_i_tiles*, and a matrix containing values from *trans_kernel*.

from the second input channel are correctly paired together, with the result of the two multiplications accumulated together.

We can now perform all the required element-wise multiplication between *trans_i_tiles* and *trans_kernel*, and correctly accumulated the results from the different input channels using $in\_els \times in\_els$ matrix multiplications. Figure 6-18 shows pseudocode that performs this.

```
//TILED_H = H/out_els;
//TILED_W = W/out_els;
//trans_i_tiles[in_els*in_els][TILED_H*TILED_W][C];
//trans_kernel[In_els*in_els][C][M]
//trans_product[in_els*in_els][TILED_H*TILED_W][M];
for (i = 0; i < in_els*in_els; i++) {
  trans_product[i] <= matrix_mul(
    trans_i_tiles[i], trans_kernel[i]);
}

```

Figure 6-18: Performing Winograd CNN convolution using $in\_els \times in\_els$ matrix multiplications.

Using the approach outlined above, we can perform all the calculations, except transforming the inputs and outputs, using $in\_els \times in\_els$ matrix multiplications. This is very advantageous, as there has been a very significant amount of work put into optimizing matrix multiplication, which can now be leveraged to implement Winograd CNN convolution. Lavin et al. were the first to propose implementing Winograd CNN convolution using $in\_els \times in\_els$ matrix multiplications as outlined above (Lavin and Gray 2016). Figure 6-19 shows a pseudocode implementation of Winograd CNN convolution using $F(2 \times 2, 3 \times 3)$ Winograd convolution, and the matrix multiplication method.

## 6.8 Winogen: Winograd Generator

When implementing a Winograd CNN convolution, selecting the correct minimal multiplication algorithm is important. This is because different minimal multiplication algorithms require different amounts of temporary memory overhead. For example, $F(2 \times 2, 3 \times 3)$ and $F(4 \times 4, 3 \times 3)$ requires different amounts of temporary memory overhead. $F(2 \times 2, 3 \times 3)$ requires less memory than $F(4 \times 4, 3 \times 3)$ when the input tensor is large and the input kernel is small, and $F(4 \times 4, 3 \times 3)$ requires less memory than $F(2 \times 2, 3 \times 3)$ when the input tensor is small and the input kernel is large. Larger minimal multiplication

```
//-- Data Structures --
in_tensor[H][W][C]
input_kernel[K][K][M][C]
out_tensor[H][W][M]
int out_els = 2; //size of the output tiles
int in_els = out_els + K - 1; //size of the input tiles
//number of i-tiles/o-tiles needed is TILED_H*TILED_W
int TILED_H = ceiling(H, out_els)
int TILED_W = ceiling(W, out_els)
trans_i_tiles[in_els*in_els][TILED_H*TILED_W][C]
trans_kernel[in_els*in_els][C][M]
trans_product_tiles[in_els*in_els][TILED_H*TILED_W][M]
o_tiles[out_els*out_els][TILED_H*TILED_W][M]

//-- perform all transformations in advance --
//split image into i_tiles and transforms them
//in a single step
trans_i_tiles = transformImage(in_tensor)
//transform kernel to winograd domain
trans_kernel = transformKernel(input_kernel)

//-- perform all element-wise multiplications --
for (i = 0; i < in_els*in_els; i++) {
    trans_product[i] = matrix_mul(
      trans_i_tiles[i], trans_kernel[i]);
  }
//transform trans_product_tiles to o_tiles and
//insert into out_tensor in one step
out_tensor = transformOutput(trans_product_tiles);

```

Figure 6-19: Pseudo-code implementation of Winograd CNN convolution using $in\_els \times in\_els$ matrix multiplications.

algorithms also require more complex transformations, but reduce the computational complexity of CNN convolution more. For inputs of a given size, it is not entirely clear what the correct minimal multiplication algorithm to use is. Also, producing transformation code for a minimal multiplication algorithm by hand is a time consuming process which deters experimentation.

In an attempt to tackle these problems, we developed Winogen. Winogen is a domain-specific program generator for Winograd CNN convolution. Given the input dimensions of a CNN convolution, Winogen can generate a working Winograd CNN convolution implementation. Winogen automatically optimizes Winograd CNN convolution code by producing many code variants to find well performing implementations. Using a bank of transformation matrices, Winogen can generate optimized transformation code for a large set of minimal multiplication algorithms. This allows it to evaluate the effect different minimal multiplication algorithms have on the convolution performance for a given input size, and select the correct minimal multiplication algorithm.

As well as investigating the effect of different minimal multiplication algorithms on performance, Winogen is also able to explore the effect of other optimization techniques on the performance of Winograd CNN convolution. During research, we explored the impact of and gave Winogen the ability to:

- Automatically transform the transformations of the inputs and outputs to a set of equations, and apply optimization rules to the equations (section 6.9).
- Explore ways to optimize the creation of the i-tiles (section 6.10).
- Explore how to handle tiles that are only partially used (section 6.11).
- Explore ways to perform Winograd CNN convolution as a sum of 1D Winograd convolutions to reduce the temporary memory needed (section 6.12).

We found that Winogen was able to produce code that outperformed the ARM Compute Library's Winograd convolution implementation across a number of input sizes on our test-bench ARM microprocessors.

## 6.9 Optimizing Winograd Transformations

Figure 6-20 restates the equation for performing 2D Winograd convolution in matrix form. To perform Winograd convolution, we perform three transformations represented using matrix multiplications: $(G \times g \times G^T)$ where $g$ is the input kernel, $(B^T \times d \times B)$ where $d$ is the input tensor, and $(A^T \times y \times A)$ where $y$ is the result of the element-wise multiplication between the transformed input tensor and input kernel. In practice, using the tiled Winograd CNN con-

$$Y = A^T \times ((G \times g \times G^T) \odot (B^T \times d \times B)) \times A$$

where
$\odot$ is element-wise multiplication.
$g$ is the input kernel.
$d$ is the input tensor.
$A^T, G, B^T$ are the matrices representing the Winograd Convolution algorithm.

Figure 6-20: Matrix form algorithm for 2D Winograd Convolution created from nesting the same 1D Winograd Convolution algorithm.

volution algorithm outlined in sections 6.5.2 and 6.7, we will need to transform $(\lceil H/out\_els \rceil \times \lceil W/out\_els \rceil \times C)$ i-tiles using the $(B^T \times d \times B)$ transformation, $(M \times C)$ input kernels using the $(G \times g \times G^T)$ transformation, and $(\lceil H/out\_els \rceil \times \lceil W/out\_els \rceil \times M)$ trans-product-tiles using the $(A^T \times y \times A)$ transformation. These transformations take up a large percentage of Winograd CNN convolution runtime, so optimizing them is important.

The transformations are performed using the *transformation matrices*, $G$, $B^T$, and $A^T$. The values of the transformation matrices are calculated in advance, and constant for any Winograd CNN convolution. This allows us to replace the matrix multiplications involving the transformation matrices with a set of equations. While it is possible to implement the transformations using matrix multiplications and the transformation matrices, replacing the multiplications with a set of optimized equations can lead to much more efficient code. There are a number of reasons for this:

- The transformation matrices typically contain some zero values. Any computations involving these values can be entirely removed.
- The transformation matrices also typically contain some $\pm 1$ values. Multiplications relating to these values can be removed or replaced with a negation.
- There are usually many common sub-expressions between different equations in the created set of equations. These only need to be calculated once and shared between equations to reduce computations.

- The transformation matrix multiplications involve very small matrices. Standard matrix multiplication libraries are usually designed to be efficient on very large inputs, and are often lackluster on small inputs (Zhang and Gennady 2017).

Figure 6-21 shows an example of how the transformation using matrices can be replaced with a much less computational expensive set of equations. Performing the matrix multiplications in figure 6-21 as matrix multiplications would require 16 multiplications and 8 additions. However, using the set of optimized equations only requires 8 multiplications and 4 additions.

Most Winograd CNN convolution implementations replace the transformation matrix multiplications with a set of equations. For example, the ARMCL library implements the transforms for $F(2 \times 2, 7 \times 7)$ using sets of equations (ARM 2019c). However, creating the set of equations can be very laborious to perform by hand. This is especially true for large transformation matrices. For example, when using a $F(4 \times 4, 5 \times 5)$ 2D Winograd Convolution, the result of $(B^T \times d \times B)$ is an $(8 \times 8)$ output matrix, where the equation for each point in the output matrix contains 64 terms (if the equations have not been reduced in any capacity).

$$\left( \begin{array}{|c|c|} \hline -1 & 0 \\ \hline 3 & 2 \\ \hline \end{array} * \begin{array}{|c|c|} \hline A & B \\ \hline C & D \\ \hline \end{array} \right) * \begin{array}{|c|c|} \hline -1 & 3 \\ \hline 0 & 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline V & X \\ \hline Y & Z \\ \hline \end{array}$$

```
V = (A*-1 + C*0)*-1 + (B*-1 + D*0)*0
X = (A*-1 + C*0)*3 + (B*-1 + D*0)*2
Y = (A*3  + C*2)*-1 + (B*3 + D*2)*0
Z = (A*3  + C*2)*3 + (B*3 + D*2)*2
```

```
T0 = (3*A + 2*B)
V = A
X = -T0
Y = -(3*A + 2*C)
Z = 3*T0 + 6*C + 4*D
```

Figure 6-21: Reducing matrix multiplications to a set of expressions.

### 6.9.1 Equation Generation

Given a set of transformation matrices, Winogen can produce three sets of equations that perform the equivalent transformations needed for Winograd convolution (i.e. transforming the input tensor, the input kernel, and the result of the element-wise multiplication). Winogen also simplifies and optimizes the produced equations using a set of simple algorithms.

First, all values stored in the transformation matrices are converted to a rational format datatype. This datatype can represent any rational number accurately, unlike standard IEEE754 floating point datatypes. This makes it much simpler to search for common sub-expressions while optimizing and transforming the set of equations, because it guarantees that values that should be equal will be exactly equal. Irrational numbers are initially stored to 15 significant digits, but the precision of the datatype grows as needed. When the final set of optimized equations are being produced, the rational datatype values are replaced with the nearest IEEE754 binary32 floating-point value.

The set of equations for each transformation are generated separately. To generate the initial set of equations for a transformation, the transformation is performed using matrix multiplication on a input matrix containing placeholder variables. For example, to calculate the initial set of equations for transforming the input kernel, $(G \times fake\_g \times G^T)$ is performed, where $fake\_g$ is a matrix containing variables representing the values that will be stored in the input kernel during runtime. Figure 6-22 shows an example of the matrix multiplications, and a subset of the resulting set of equations for $(G \times fake\_g \times G^T)$, using a $F(2 \times 2, 3 \times 3)$ 2D Winograd convolution algorithm.

Following the creation of the initial set of equations, a list of simple transformations is applied to each equation in the set. The transformations are listed in figure 6-23.

Figure 6-24 shows an example of how the list of transformations in figure 6-23 can simplify an equation. Applying rule (1) from figure 6-23 to equation (i) in figure 6-24 results in equation (ii) in figure 6-24. Applying rule (2) from figure 6-23 to equation (ii) in figure 6-24 results in equation (iii) in figure 6-24.

$$(G \times fake\_g \times G^T) = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ g_{20} & g_{21} & g_{22} \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}^T$$

$$\begin{aligned} trans\_kernel_{00} &= 1(1g_{00} + 0g_{10} + 0g_{20}) + 0(1g_{01} + 0g_{11} + 0g_{21}) \\ &\quad + 0(1g_{02} + 0g_{12} + 0g_{22}) \\ trans\_kernel_{01} &= 0.5(1g_{00} + 0g_{10} + 0g_{20}) + 0.5(1g_{01} + 0g_{11} + 0g_{21}) \\ &\quad + 0.5(1g_{02} + 0g_{12} + 0g_{22}) \\ &\dots \\ trans\_kernel_{33} &= 0(0g_{00} + 0g_{10} + 1g_{20}) + 0(0g_{01} + 0g_{11} + 1g_{21}) \\ &\quad + 1(0g_{02} + 0g_{12} + 1g_{22}) \end{aligned}$$

Figure 6-22: Calculating the initial set of equations for transforming the input kernel for a $F(2 \times 2, 3 \times 3)$ 2D Winograd convolution.

1. Multiplications by $1$ are removed, and multiplications by $-1$ are replaced with negations. Values multiplied by zero are removed, double negations are recursively replaced.
2. All bracketed sub-expressions are expanded so every term is a co-efficient with a single variable.
3. Terms that share the same variable are combined by summing their co-efficients.
4. Terms that share the same co-efficient are combined. The co-efficients only need to match in scale, not sign.

Figure 6-23: Simplification transformations applied to the initial set of equations representing Winograd transformations.

This pattern repeats for rules (3) and (4) from figure 6-23, and equations (iv) and (v) from figure 6-24.

After the list of transformations have been applied, every equation can be represented by a set of terms. Each term is a tuple that contains a co-efficient, and a set of positive or negative variables. The variables are stored as a tuple containing the sign, and the variable. For example, equation (v) in figure 6-24 can be represented as $\{(30, \{(+, x), (-, z)\}), (7, \{(+, y)\})\}$. In it, $30(x - z)$ becomes $(30, \{(+, x), (-, z)\})$ and $7y$ becomes $(7, \{(+, y)\})$. The set representing an equation is be referred to as the *equ-set* for that equation. In tuples representing a term, the set of variables is referred to as a *var-set*. For example, $\{(+,x),(-,z)\}$ is the var-set in the tuple $(30,\{(+,x),(-,z)\})$.

$$\begin{aligned}
i)\quad & p1 = 5(3x+2y)+0(2x+y)-3(-5x+1y)--30(-z)\\
ii)\quad & p1 = 5(3x+2y)-3(-5x+y)-30z\\
iii)\quad & p1 = 15x+10y+15x-3y-30z\\
iv)\quad & p1 = 30x+7y-30z\\
v)\quad & p1 = 30(x-z)+7y
\end{aligned}$$

Figure 6-24: Example of simplifying an equation using the transformations in figure 6-23.

### 6.9.2 Sub-Expression Generation

Once an *equ-set* has been created for each equation in the set of equations, Winogen attempts to find common sub-expressions between the equations. A new set is created that contains all the var-sets from any equ-set. This *var-set set* also contains every var-set with the signs of the variables inverted. For example, set (iii) in figure 6-25 is a var-set set created from all the var-sets present in equ-set-1 and equ-set-2 in figure 6-25. Set (iv) in 6-25 is the same as set (iii), except it also contains every var-set with it's signs inverted. For example, set (iv) contains $\{(-,x),(+,z)\}$, which is the var-set $\{(+,x),(-,z)\}$ from equ-set-1 with it's signs inverted.

$$\begin{aligned}
i)\quad & \textit{equ-set-}1 = \{(30,\{(+,x),(-,z)\}),(7,\{(+,y)\})\}\\
ii)\quad & \textit{equ-set-}2 = \{(5,\{(-,x),(-,z)\}),(2,\{(+,y)\}),(8,\{(+,p),(+,q)\})\}\\
iii)\quad & \textit{var-set set} = \{\{(+,x),(-,z)\},\{(+,y)\},\{(-,x),(-,z)\},\{(+,p),(+,q)\}\}\\
iv)\quad & \textit{var-set set} = \{\{(+,x),(-,z)\},\{(+,y)\},\{(-,x),(-,z)\},\{(+,p),(+,q)\},\\
& \qquad \{(-,x),(+,z)\},\{(-,y)\},\{(+,x),(+,z)\},\{(-,p),(-,q)\}\}
\end{aligned}$$

Figure 6-25: Collecting all the var-sets from two equ-sets into a single set (iii). Set (iv) also contains the var-sets with signs inverted.

We use the var-set set to create a hierarchy of common sub-expressions between the equations. First, we rename the var-set set as the *L0 var-set set*. Next, we create the *L1 var-set set*. The L1 var-set set is the set containing all intersections between two elements in the L0 var-set set. Figure 6-26 shows the construction of the L1 var-set set.

$$L1\ var\text{-}set\ set = \{x \mid y \in L0\ var\text{-}set\ set, z \in L0\ var\text{-}set\ set, y \neq z, |x| > 1, x = y \cap z\}$$

Figure 6-26: Set Builder equation for creating the L1 var-set set.

The elements of the L1 var-set set represent sub-expressions that exist between the equations in the set of equations. Note that, sub-expressions of a single variable are not included in the L1 var-set set, because no performance gain can be gotten from pre-computing them. Sub-expressions can also be found between the sub-expressions represented by the var-sets in the L1 var-set. To find these sub-expressions, we construct the L2 var-set set. It is constructed in the same way as the L1 var-set set, except the L1 var-set set is used as the input set. This is repeated until a $LN$ var-set set is created which has no elements. The L0 to LN var-set sets represent a hierarchy of common sub-expressions, where the $L(N)$ var-set set contains the sub-expressions in the elements of the $L(N-1)$ var-set set. Figure 6-27 shows an example hierarchy with an L0, L1, and L2 var-set set.

$$\begin{aligned}
L0\ var\text{-}set\ set = \{&\{(+,x),(+,y),(+,z),(+,w)\},\ \{(+,y),(+,z),(+,w),(+,p)\},\\
&\{(+,u),(+,v),(+,p)\},\ \{(+,u),(+,v),(+,w),(+,z)\}\}\\
L1\ var\text{-}set\ set = \{&\{(+,y),(+,z),(+,w)\},\ \{(+,u),(+,v)\},\ \{(+,w),(+,z)\}\}\\
L2\ var\text{-}set\ set = \{&\{(+,w),(+,z)\}\}
\end{aligned}$$

Figure 6-27: Example of the common sub-expression hierarchy. The L2 var-set set is created from the L1 var-set set, and the L1 var-set set from the L0 var-set set.

### 6.9.3 Transformation Generation

Once the hierarchy of sub-expressions has been created, the var-sets in the equ-sets and the var-sets in the var-set sets are updated. First, the var-set in every equ-set is compared with every var-set in the L1 var-set set. For every var-set $VS$ in an equ-set, a reference to the largest var-set in the L1 var-set set that is subset of $VS$ is inserted into $VS$, and every element in the subset is removed from $VS$. For example, in figure 6-28, equation (i) contains the var-set $\{(+,x),(+,y),(-,z)\}$. The largest element in the L1 var-set set (equation (ii) in figure 6-28) that is a subset of $\{(+,x),(+,y),(-,z)\}$ is $\{(+,x),(-,z)\}$. Therefore, the elements $(+,x)$ and $(-,z)$ are removed from $\{(+,x),(+,y),(-,z)\}$ and replaced with a reference to the element in the L1 var-set set (i.e. $L1\text{-}element\text{-}0$).

$$\begin{aligned}
i)\quad & equ\text{-}set\text{-}1 = \{(30,\{(+,x),(+,y),(-,z)\}),(7,\{(+,p),(-,q)\}),(1,\{(+,m)\})\}\\
ii)\quad & L1\ var\text{-}set\ set = \{\{(+,x),(-z)\},\{(+,p),(-,q)\},\}\\
iii)\quad & equ\text{-}set\text{-}1 = \{(30,\{(+,L1\text{-}element\text{-}0),(+,y)\}),\\
& \qquad (7,\{(+,L1\text{-}element\text{-}1)\}),(1,\{(+,m)\})\}
\end{aligned}$$

Figure 6-28: Replacing sub-expressions in the var sets of equ-set 1 with references to common sub-expressions in the L1 var-set set.

Note that a sub-set of a var-set can be replaced by a reference to an element in the L1 var-set set if the sub-set and the element share the same variables, but have opposite signs. For example, in figure 6-29, the element $\{(+,w),(-,x),(+,z)\}$ from the L1 var-set set can be inserted into the var-set $\{(-,w),(+,x),(+,y),(-,z)\}$, because it shares the same variables, but with the signs inverted. The only difference is that it is inserted with a negative sign (e.g. $(-,L1\text{-}element\text{-}0)$), rather then a positive sign (like $(+,L1\text{-}element\text{-}0)$ in equation (iii) of figure 6-28).

Next, subsets from the var-sets in the L1 var-set are replaced by references to the L2 var-set sets in a similar manner. This repeats for all levels of the sub-expression hierarchy.

The equ-sets and the sub-expression hierarchy are now used to generate a C++ code implementation of the set of equations that perform the wanted

$$
\begin{aligned}
i)\quad & equ\text{-}set\text{-}1 = \{(30, \{(-, w), (+, x), (+, y), (-, z)\})\} \\
ii)\quad & L1\ var\text{-}set\ set = \{\{(+, x), (-z)\}, \{(+, w), (-, x), (+, z)\}, \} \\
iii)\quad & equ\text{-}set\text{-}1 = \{(30, \{(-, L1\text{-}element\text{-}0), (+, y)\})\}
\end{aligned}
$$

Figure 6-29: Replacing a sub-expression in equ-set 1 with a (negated) reference to an equivalent sub-expression in the L1 var-set set.

transformation. The equ-sets are converted into a simple AST similar to the ASTs covered in section 4.7. The AST is then used to generate C++ in a similar manner to Genvolution. However, before a sub-tree representing an equ-set is generated, the equ-set is checked to see if it refers to any common sub-expressions that must be generated first. This is applied recursively, so that common sub-expressions of larger sub-expressions are also correctly generated. All sub-expressions that have already been generated are tracked so that sub-expressions aren't needlessly generated multiple times. Figure 6-30 shows example generated code without the use of common sub-expressions. Figure 6-31 shows the same example code with sub-expressions. The version without sub-expressions needs 48 additions/substractions in total, while the sub-expression version needs 27 additions/substractions in total.

```
kerTrans[0]  = (ker[0]);
kerTrans[1]  = (1.0/2.0)*(ker[0]+ker[3]+ker[6]);
kerTrans[2]  = (1.0/2.0)*(ker[0]-ker[3]+ker[6]);
kerTrans[3]  = (ker[6]);
kerTrans[4]  = (1.0/2.0)*(ker[0]+ker[1]+ker[2]);
kerTrans[5]  = (1.0/4.0)*(ker[0]+ker[1]+ker[2]+ker[3]
                 +ker[4]+ker[5]+ker[6]+ker[7]+ker[8]);
kerTrans[6]  = (1.0/4.0)*(ker[0]+ker[1]+ker[2]-ker[3]
                 -ker[4]-ker[5]+ker[6]+ker[7]+ker[8]);
kerTrans[7]  = (1.0/2.0)*(ker[6]+ker[7]+ker[8]);
kerTrans[8]  = (1.0/2.0)*(ker[0]-ker[1]+ker[2]);
kerTrans[9]  = (1.0/4.0)*(ker[0]-ker[1]+ker[2]+ker[3]
                  -ker[4]+ker[5]+ker[6]-ker[7]+ker[8]);
kerTrans[10] = (1.0/4.0)*(ker[0]-ker[1]+ker[2]-ker[3]
                  +ker[4]-ker[5]+ker[6]-ker[7]+ker[8]);
kerTrans[11] = (1.0/2.0)*(ker[6]-ker[7]+ker[8]);
kerTrans[12] = (ker[2]);
kerTrans[13] = (1.0/2.0)*(ker[2]+ker[5]+ker[8]);
kerTrans[14] = (1.0/2.0)*(ker[2]-ker[5]+ker[8]);
kerTrans[15] = (ker[8]);

```

Figure 6-30: Pseudo-code implementation of $B^T * b * B$ for F(2x2, 3x3) transforming the kernel to the Winograd domain. The matrix multiplications have been simplified to a set of equations.

```
kerTrans[0] = (ker[0]);
float t0 = ker[0]+ker[6];
kerTrans[1] = (1.0/2.0)*(t0+ker[3]);
kerTrans[2] = (1.0/2.0)*(t0-ker[3]);
kerTrans[3] = (ker[6]);
float t1 = ker[0]+ker[2];
kerTrans[4] = (1.0/2.0)*(t1+ker[1]);
float t2 = ker[3]+ker[5];
float t3 = t2+ker[4];
float t4 = ker[1]+ker[7];
float t5 = ker[6]+ker[8];
float t6 = t5+t1;
float t7 = t4+t6;
kerTrans[5] = (1.0/4.0)*(t3+t7);
kerTrans[6] = (1.0/4.0)*(-t3+t7);
kerTrans[7] = (1.0/2.0)*(t5+ker[7]);
kerTrans[8] = (1.0/2.0)*(t1-ker[1]);
kerTrans[9] = (1.0/4.0)*(-t4-ker[4]+t2+t6);
kerTrans[10] = (1.0/4.0)*(-t4+ker[4]-t2+t6);
kerTrans[11] = (1.0/2.0)*(t5-ker[7]);
kerTrans[12] = (ker[2]);
float t8 = ker[2]+ker[8];
kerTrans[13] = (1.0/2.0)*(t8+ker[5]);
kerTrans[14] = (1.0/2.0)*(t8-ker[5]);
kerTrans[15] = (ker[8]);

```

Figure 6-31: The same implementation as figure 6-30 however common sub-expression code has been inserted.

## 6.10 Creating i-tiles, and Synthesizing i-tile Values

As covered in section 6.5.2, large input tensors are split into a number of overlapping *i-tiles*. Each separate i-tile is used as the input to a different Winograd convolution, and the output of the convolution is part of the output tensor. In section 6.5.2, it was suggested that all the i-tiles are calculated and stored separately in a single step, and then a second step would transform all the i-tiles to *i-trans-tiles*. In practice, i-tiles are not actually constructed. Instead, the *i-trans-tiles* are constructed by transforming data taken directly from the input tensor. Figure 6-32 contains pseudo-code that creates all the necessary i-trans-tiles, using a $F(2 \times 2, 3 \times 3)$ Winograd convolution without creating the i-tiles. The i-tiles are not explicitly created, instead the top left corner position of every i-tile is calculated, and the i-tile data is read data directly from the input tensor (called $img$ in figure 6-32).

```
img[H][W][C]; //the input tensor for the CNN convolution
trans_i_tiles[in_els*in_els][H/out_els][W/out_els][C];
for (hTile = 0; hTile < H/out_els; hTile++) {
  for (wTile = 0; wTile < W/out_els; wTile++) {
    for (c = 0; c < C; c++) {
      //(h,w) is top-left corner of current i-tile
      h = hTile*out_els - K;
      w = wTile*out_els - K;
      //create i-trans-tile from i-tile at (h,w)
      trans_i_tiles[0][h][w][c] =
              img[h][w][c] - img[h][w+2][c]
            - img[h+2][w][c] + img[h+2][w+2][c];
      trans_i_tiles[1][h][w][c] =
              img[h+1][w][c] - img[h+1][w+2][c]
            + img[h+2][w][c] - img[h+2][w+2][c];
      /* ... trans_i_tiles[2][h][w][c] thru
               trans_i_tiles[14][h][w][c] ... */
      trans_i_tiles[15][h][w][c] =
              img[h+1][w+2][c] - img[h+1][w+3][c]
            - img[h+3][w+1][c] + img[h+3][w+3][c];
    }
  }
}

```

Figure 6-32: Calculating trans-i-tiles using values taken directly from the input tensor ($img$ in the pseudocode).

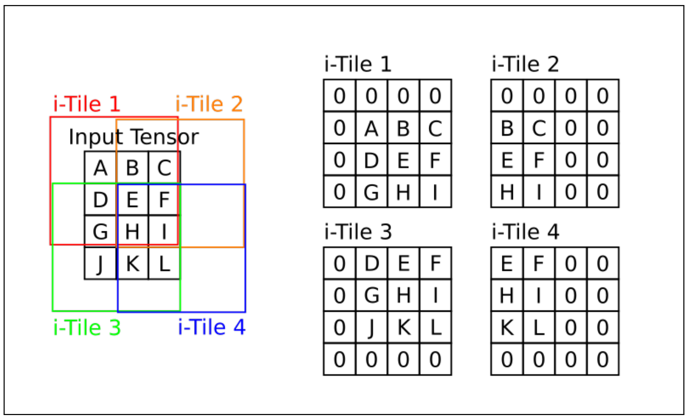


Figure 6-33: A $(4 \times 3)$ input tensor being split into four i-tiles. Each i-tile requires some synthesized values.

However, i-tiles often contain data that must be synthesized from the input tensor. For example, in figure 6-33, the first row and column of *i-tile 1* lie outside the bounds of the input tensor, and must be filled with synthesized values. In figure 6-33, zero-padding is used to synthesize values, so the first row and column of i-tile 1 are filled with zeroes. The code in figure 6-32 can not handle i-tiles that lay partially outside the input tensor, and will produce incorrect i-trans-tiles because of this. Handling synthesized values is difficult, because the position of the values in an i-tile that must be synthesized change for every i-tile. We investigated three methods for handling the synthesized values. For all three methods, we assumed the synthesizing method was zero padding (i.e. all synthesized values are zero).

### 6.10.1 I-Tile Buffer

The first method tested for handling synthesized values was the *i-tile buffer*. The i-tile buffer contained all the values from the i-tile currently being transformed, including synthesized values. Figure 6-34 shows a pseudo-code example of using an i-tile buffer. Lines 5 and 6 open two loops that iterate across (the positions) of all the i-tiles. Lines 13 to 24 fill up the i-tile buffer with values from the current i-tiles. The i-tile buffer stores $C$ i-tiles in figure 6-34. We construct $C$ i-tiles at a time because for every $(h, w)$ position calculated from $hTile$ and $wTile$, there is an i-tile for every input channel. These $C$ i-tiles all have synthesized values in the same positions, so we can construct all of them at once using the same conditional checks. Lines 25 to 38 then create $C$ trans-i-tiles using the data in the i-tile buffer.

```
 1 img[H][W][C]; //the input tensor for the CNN convolution
 2 //buffer for containing C i-tiles
 3 i_tile_buffer[in_els][in_els][C];
 4 trans_i_tiles[in_els*in_els][H/out_els][W/out_els][C];
 5 for (hTile = 0; hTile < H/out_els; hTile++) {
 6   for (wTile = 0; wTile < W/out_els; wTile++) {
 7     //(h,w) is top-left corner of current i-tile
 8     h = hTile*out_els - K;
 9     w = wTile*out_els - K;
10     //fill up the i-tile buffer with C i-tiles
11     //All C i-tiles will have synthesized values
12     //at the same position
13     for (ih = 0; ih < in_els; ih++) {
14       for (iw = 0; iw < in_els; iw++) {
15         //fill synthesized positions with zeroes
16         if (synthesized(h+ih, w+iw)) {
17           memset(i_tile_buffer[ih][iw], C, 0);
18         } else { //if not-synthesized, copy from img
19           memcpy(
20             i_tile_buffer[ih][iw],
21             C, img[h+ih][w+iw]);
22         }
23       }
24     }
25     for (c = 0; c < C; c++) {
26       //create i-trans-tile from i-tile at (h,w)
27       trans_i_tiles[0][hTile][wTile][c] =
28         i_tile_buffer[0][0][c] - i_tile_buffer[0][2][c]
29         - i_tile_buffer[2][0][c] + i_tile_buffer[2][2][c];
30       trans_i_tiles[1][hTile][wTile][c] =
31         i_tile_buffer[1][0][c] - i_tile_buffer[1][2][c]
32         + i_tile_buffer[2][0][c] - i_tile_buffer[2][2][c];
33       /* ... trans_i_tiles[2][hTile][wTile][c] thru
34       trans_i_tiles[14][hTile][wTile][c] ... */
35       trans_i_tiles[15][hTile][wTile][c] =
36         i_tile_buffer[1][2][c] - i_tile_buffer[1][3][c]
37         - i_tile_buffer[3][1][c] + i_tile_buffer[3][3][c];
38     }
39   }
40 }
41
```

Figure 6-34: Creating i-trans-tiles using an i-tile buffer to store the current i-tile being transformed. The buffer includes any required synthesized values.

The main issue with this method is that it is very memory movement intensive. In total, approximately $\lceil H/out\_els \rceil \times \lceil w/out\_els \rceil \times C$ values will be copied into the i-tile buffer, depending on the shape of the i-tiles and the input tensor. The amount of time spent copying data can have a noticeably effect on performance.

### 6.10.2 Indirection Buffer

The second method tested is an optimization of the i-tile buffer method. The i-tile buffer is replaced with a buffer of $(in\_els \times in\_els)$ pointers. Instead of copying data into the i-tile buffer, the indirection buffer points at the data for the current i-tile(s) in the input tensor. If a position in the current i-tile needs to be synthesized, the matching pointer in the indirection buffer points at the zero-vector instead. The zero vector is a zero-filled array of $C$ elements. Figure 6-35 shows a diagram of the indirection buffer that points to the data for $C$ i-tiles, where the right-most column of each i-tile is synthesized. Figure 6-36 shows a pseudo-code example of using the the indirection buffer. The pseudocode is similar to that in figure 6-34. The main differences is the zero vector (line 4), and how only a reference to the data is made, rather then copying the data (line 19 in figure 6-34 and line 17 in figure 6-36). For all tested input sizes, the indirection buffer method was found to be faster then the i-tile buffer method.

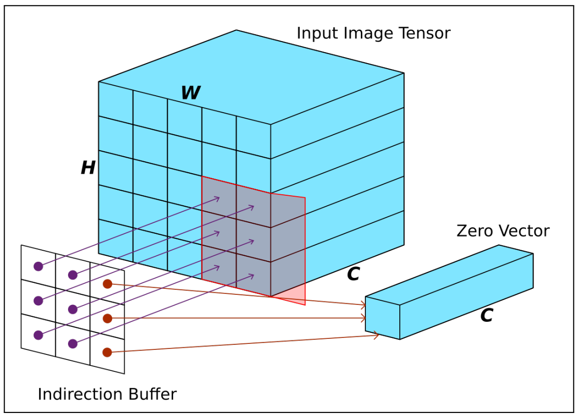


Figure 6-35: Using an indirection buffer to point at the data needed for an i-tile. One column of the i-tile requires (zero-padded) synthesized values.

```
img[H][W][C]; //the input tensor for the CNN convolution
//buffer for pointing at i-tile data in img
float* indirect_bufr[in_els][in_els]; //pointers
zero_vector[C] = {0}; //zero-filled
trans_i_tiles[in_els*in_els][H/out_els][W/out_els][C];
for (hTile = 0; hTile < H/out_els; hTile++) {
  for (wTile = 0; wTile < W/out_els; wTile++) {
    //(h,w) is top-left corner of current i-tile
    h = hTile*out_els - K;
    w = wTile*out_els - K;
    //create indirection buffer for current i-tile
    for (ih = 0; ih < in_els; ih++) {
      for (iw = 0; iw < in_els; iw++) {
        if (synthesized(h+ih, w+iw)) {
          indirect_bufr[ih][iw] = &(zero_vector[0]);
        } else {
          indirect_bufr[ih][iw] = &(img[h+ih][w+iw][0]);
        }
      }
    }
    for (c = 0; c < C; c++) {
      //create i-trans-tile from i-tile at (h,w)
      trans_i_tiles[0][hTile][wTile][c] =
        indirect_bufr[0][0][c] - indirect_bufr[0][2][c]
        - indirect_bufr[2][0][c] + indirect_bufr[2][2][c];
      trans_i_tiles[1][hTile][wTile][c] =
        indirect_bufr[1][0][c] - indirect_bufr[1][2][c]
        + indirect_bufr[2][0][c] - indirect_bufr[2][2][c];
      /* ... trans_i_tiles[2][hTile][wTile][c] thru
      trans_i_tiles[14][hTile][wTile][c] ... */
      trans_i_tiles[15][hTile][wTile][c] =
        indirect_bufr[1][2][c] - indirect_bufr[1][3][c]
        - indirect_bufr[3][1][c] + indirect_bufr[3][3][c];
    }
  }
}
```

Figure 6-36: Creating i-trans-tiles using an i-tile buffer to point at the data that makes up the current i-tile. The buffer also points at any required synthesized values.

### 6.10.3 Loop Unswitching the Transformation

Loop unswitching is a loop optimization technique. The code in figure 6-37 is a for loop that iterates over $N$ elements. Inside the for loop is a conditional statement dependant on the iterator (i.e. dependent on the variable $i$). We can optimize this code snippet by splitting the one single loop into two smaller loops, and removing the conditional statement. The optimized code is shown in figure 6-38. This is an example of loop unswitching. Loop unswitching is where a loop containing a conditional statement is split into multiple smaller loops containing no conditional statements. The smaller loops are either nested inside a new conditional statement, or the differences between the smaller loops encode the conditional statement implicitly. For example, the condition of the if statement in figure 6-37 (line 4) is made into the iteration condition for the first for loop in figure 6-37 (line 3).

```
1   float in[N];
2   float out[N];
3   for (int i = 0; i < N; i++) {
4     if (i < 100) { out[N] = 2 * in[N]; }
5     else { out[N] = in[N]*in[N]; }
6   }
7
```

Figure 6-37: Pseudocode to double the first 100 elements of a list, and square the rest of the elements.

```
1   float in[N];
2   float out[N];
3   for (int i = 0; i < 100; i++) {
4     out[N] = 2 * in[N];
5   }
6   for (int i = 100; i < N; i++) {
7     out[N] = in[N]*in[N];
8   }
9
```

Figure 6-38: Equivalent pseudocode to figure 6-37, however the loop has been unswitched.

Figure 6-39 shows a code snippet for constructing an i-trans-tile using data directly from data from the input tensor. This code snippet is the same as the transformation code shown in figure 6-32. As covered in section 6.10, this code is incorrect as it cannot handle when a value needs to be synthesized. For example, $(h, w)$ will equal $(-1, -1)$ when constructing the first i-trans-tile (as the top-left element of the first i-tile is located at $(-1, -1)$). However, if we are using zero padding to synthesize values, then we know that every synthesized value is equal to zero. Following this, a method to fix the code in figure 6-

39 is to wrap every access to the input tensor in a conditional statement, that either returns the value in the input tensor if the values does not need to be synthesized, or returns zero if it does. Figure 6-40 shows an example of this approach. The $synthesized(int, int)$ function returns true if the position given lies outside the bounds of the input tensor, and it returns false otherwise. The code in figure 6-40 is correct, and will construct all the trans-i-tiles correctly. However, performing a conditional statement before every access to the input tensor has a significant effect on performance.

```
trans_i_tiles[0][h][w][c] =
  + img[h][w][c]
  - img[h][w+2][c]
  - img[h+2][w][c]
  + img[h+2][w+2][c];
trans_i_tiles[1][h][w][c] =
  + img[h+1][w][c]
  - img[h+1][w+2][c]
  + img[h+2][w][c]
  - img[h+2][w+2][c];
/* ... trans_i_tiles[2][h][w][c] thru
        trans_i_tiles[14][h][w][c] ... */
trans_i_tiles[15][h][w][c] =
  + img[h+1][w+2][c]
  - img[h+1][w+3][c]
  - img[h+3][w+1][c]
  + img[h+3][w+3][c];

```

Figure 6-39: Code snippet from figure 6-32. The snippet creates an i-trans-tile using data directly from the input tensor ($img$).

However, we can remove these conditional checks using loop unswitching. Each i-tile will have different synthesized values depending on its position. If we know the height and width (i.e $H$ and $W$) of the input tensor before compilation, we can calculate the position of every i-tile beforehand, and generate a separate code block for each i-tile (or more specifically, a code block for the construction of each i-trans-tile). Each code block would replace any accesses to synthesized values with zeroes. Creating a separate code block for every i-tile would increase the amount of transformation code by approximately $(\lceil H/out\_els \rceil \times \lceil W/out\_els \rceil)$, the number of i-tiles needed. This amount of

```
 trans_i_tiles[0][h][w][c] =
+ (synthesized(h,w) ? 0 : img[h][w][c])
− (synthesized(h,w+2) ? 0 : img[h][w+2][c])
− (synthesized(h+2,w) ? 0 : img[h+2][w][c])
+ (synthesized(h+2,w+2) ? 0 : img[h+2][w+2][c]);
 trans_i_tiles[1][h][w][c] =
+ (synthesized(h+1,w) ? 0 : img[h+1][w][c])
− (synthesized(h+1,w+2) ? 0 : img[h+1][w+2][c])
+ (synthesized(h+2,w) ? 0 : img[h+2][w][c])
− (synthesized(h+2,w+2) ? 0 : img[h+2][w+2][c]);
/* ... trans_i_tiles[2][h][w][c] thru
       trans_i_tiles[14][h][w][c] ... */
 trans_i_tiles[15][h][w][c] =
+ (synthesized(h+1,w+2) ? 0 : img[h+1][w+2][c])
− (synthesized(h+1,w+3) ? 0 : img[h+1][w+3][c])
− (synthesized(h+3,w+1) ? 0 : img[h+3][w+1][c])
+ (synthesized(h+3,w+3) ? 0 : img[h+3][w+3][c]);

```

Figure 6-40: Similar code to figure 6-39, however every access to the input tensor is guarded by a conditional to check if it should return a synthesized value instead.

code growth is unwanted though, as it would make compilation time extremely long. It would also reduce performance by increasing pressure on the instruction cache, and increase the number of compulsory instruction cache misses.

However it is not necessary to generate $(\lceil H/out\_els \rceil \times \lceil W/out\_els \rceil)$ different code blocks, because many i-tiles share the same pattern for which values are synthesized. For example, if the input tensor is large, a large number of i-tiles use no synthesized values. Theses i-tiles can all use the same transformation code block.

Figure 6-41 shows how i-tiles can be grouped by what values in the i-tiles need synthesized. Figure 6-41 shows an $(8 \times 8)$ input tensor for a $F(2 \times 2, 3 \times 3)$ tiled Winograd CNN convolution. Each i-tile is $(4 \times 4)$. The cell marked with an X marks the $(0, 0)$ position in the input tensor. Each coloured cell represents the top-left corner position of one of the necessary i-tiles. The letters in each coloured cell show which i-tiles can be grouped together based on which values must be synthesized. For example, The B-group of i-tiles (at positions $(-1, 2)$ and $(-1, 4)$) need the top row of values synthesized. The E-group of i-tiles need no values synthesized. In total, for the dimensions in figure 6-41, nine separate transformation code blocks would be needed to construct all the i-

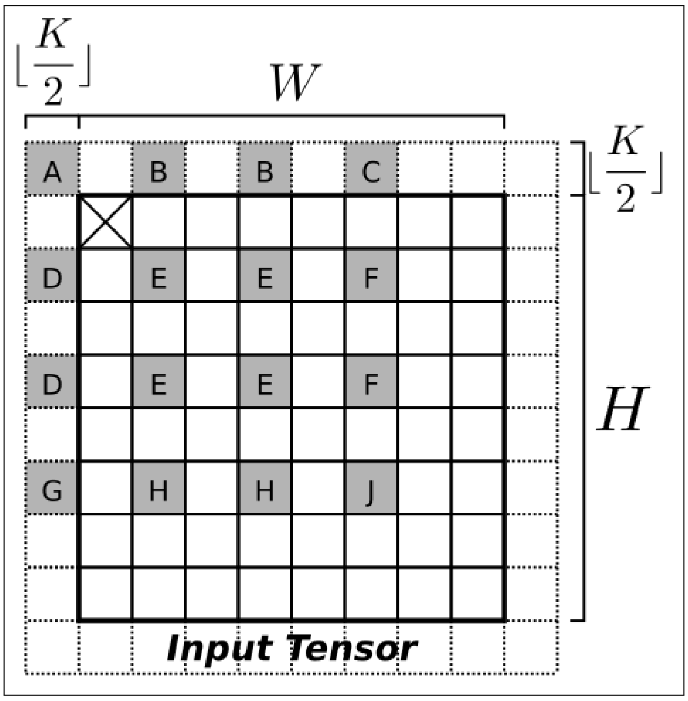


Figure 6-41: Diagram showing how different i-tiles can be grouped based on what values they synthesized. Each grey square marks the position of the top-left corner of an i-tile. The letters denote the groupings.

trans-tiles while handling synthesized values without conditional statements. In general, no matter what size the input tensor is, between 9 and 14 distinct code blocks are required. Winogen was extended to be able to perform the loop unswitching and i-tile grouping technique covered above.

Grouping i-tiles by their synthesized values allows us to perform loop unswitching to remove conditionals, while limiting how much code growth occurs. We found that the loop unswitching method produced the fastest transformation code in the majority cases, with the indirection buffer method being marginally slower. However, even with i-tile grouping, the code growth of the transformation code had a significant effect on compilation time. This is especially true when using aggressive optimization flags and function inlining. The effect on compilation time is so significant that the loop unswitching technique was not considered when producing final evaluation results for Winogen, as it would make building the experiments prohibitively long.

## 6.11 Impact of skipping half-tiles during Winograd

As covered in section 6.5.2, large output tensors must be broken into smaller *o-tiles* to cover the output tensor, each with a matching *i-tile* from the input tensor. Also the i-tiles and o-tiles may lie partially outside the bounds of tensors if the tile's sides are not perfect multiplies of the tensor they cover. For example, if the dimensions of an o-tile are $(3 \times 3)$, but the output image tensor is $(28 \times 28)$, then to cover the entire output tensor we will need 30 o-tiles. This means we will calculate the values for 900 output points, even though the output tensor only has 784 output points. We will discard around 14.7% of the output points calculated. While it is possible to avoid the final $A^T * y * A$ transformation for the unnecessary output points, the entire *trans-i-tile*, and *trans-product-tile* must be calculated even if they are only used to produce a single valid output point.

An alternative to the above approach is to only use Winograd convolution to convolve the output points that would be part of an entirely used o-tile, and use a second method to convolve the remaining points. Winogen was used to investigate this approach of handling partial o-tiles. When generating a convolution, Winogen can either use Winograd convolution for all output points, or use a mix of Winograd convolution and direct convolution. During testing, both versions were generated for all problem sizes to see what effect it had on performance. In the worst case, direct convolution will be used to calculate $(H * (out\_els - 1) + W * (out\_els - 1) - ((out\_els - 1) * (out\_els - 1))$ output points for each kernel used, where $H$ and $W$ are the width and height of the output tensor.

## 6.12 CNN Convolution using Multiple 1D Winograd Convolutions

Winograd CNN convolution has been used to create fast CNN convolution implementations. However, the standard Winograd CNN convolution requires a significant extra memory overhead. Storing the transformed i-tiles requires

$(\lceil H/out\_els \rceil) \times (\lceil w/out\_els \rceil) \times (in\_els \times in\_els) \times C$ floats, storing the transformed kernel requires $(in\_els \times K) \times C \times M$ floats, and storing the $trans\_product$ requires $(\lceil H/out\_els \rceil) \times (\lceil w/out\_els \rceil) \times (in\_els \times in\_els) \times M$ floats. On lower end devices, with less memory and smaller caches, this amount of memory overhead can have a significant impact on performance, or place a limit on the size of the convolutions possible. We propose a new CNN convolution method that uses sums of 1D Winograd convolutions to perform CNN convolution. The new method attempt to leverage the reduction in problem complexity that Winograd convolution causes, while reducing the amount of temporary memory required.

### 6.12.1 2D Convolution as a Sum of 1D Convolutions

2D convolution can be implemented as a sum of multiple 1D convolutions. First, the $(K \times K)$ input kernel is split into $K$ 1D kernel row vectors of length $K$. Each kernel vector is convolved with the input tensor to produce $K$ intermediate outputs. The $K$ intermediate outputs are then summed with an offset to produce the final output tensor. The output tensor contains the result of the wanted 2D convolution. Figure 6-42 shows an example of this.

In figure 6-42, The $(3 \times 3)$ input kernel is split into three kernel row vectors. Each kernel row vector is convolved with the input tensor to produce 3 intermediate outputs. The intermediate outputs are summed to produce the final output tensor. Note that only the *full* output points (i.e. the output points that rely on no synthesized values) are calculated in figure 6-42. The offset of the intermediate outputs is related to the row of the 2D kernel used to produce it. In general, the intermediate output calculated using the $kr$ row from the 2D kernel is vertical offset up by $kr$ when summing intermediate outputs, assuming $kr$ is zero-indexed. For example, in figure 6-42, the intermediate output calculated using $ker\_row\_0$ has an vertical offset of $0$ when summing the intermediate outputs, because $ker\_row\_0$ is the first row of the 2D input kernel. The intermediate output from $ker\_row\_1$ has a vertical offset of $1$, and the intermediate output from $ker\_row\_2$ has a vertical offset of $2$.

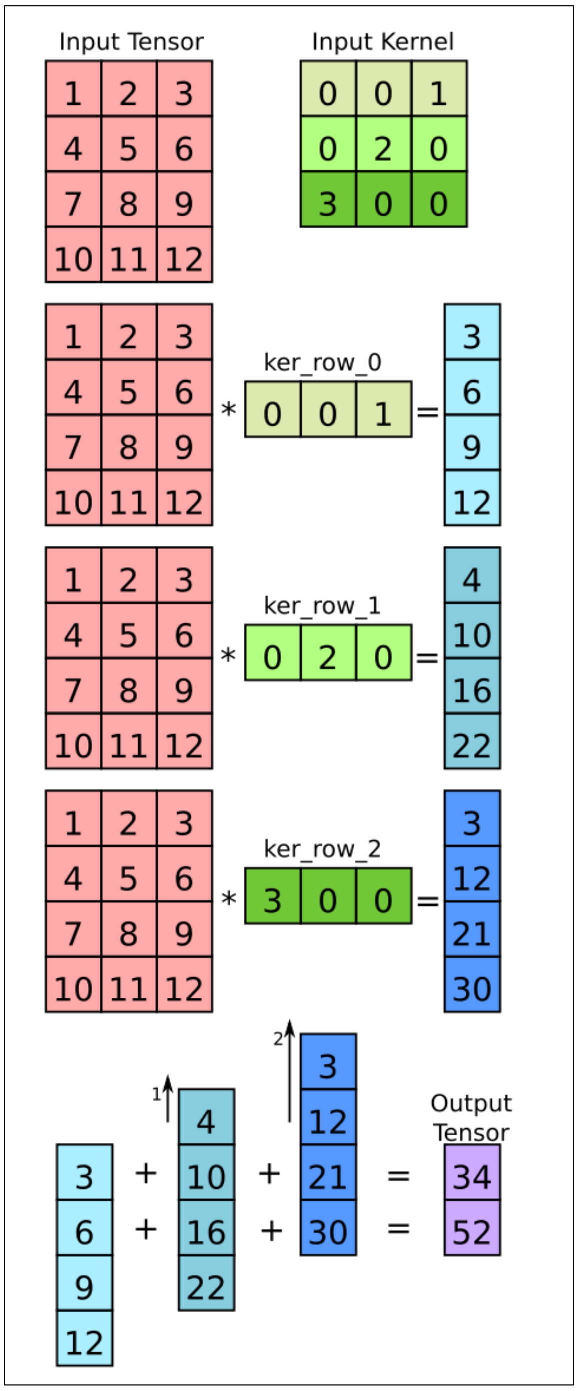


Figure 6-42: Performing a 2D CNN convolution as a sum of 1D convolutions.

### 6.12.2 2D Convolution as a Sum of 1D Winograd Convolutions

2D Winograd convolution can also be performed as a sum of 1D Winograd convolutions. Figure 6-44 shows an example of this, performing a $F(2\times2,3\times3)$ Winograd convolution by summing multiple $F(2,3)$ Winograd convolutions. First, the input tensor and the input kernel are split into rows. Each row is

$$Y = A^T \times ((G \times g) \odot (B^T \times d))$$

where
$\odot$ is element-wise multiplication.
$g$ is the input kernel vector.
$d$ is the input vector.
$A^T, G, B^T$ are the matrices representing the fast filtering algorithm.

Figure 6-43: Matrix form algorithm for performing 1D Winograd Convolution.

then transformed using the appropriate matrix, following the equation shown in figure 6-43. For example, in figure 6-44, the first row of the input kernel (containing {k0, k1, k2}) is transformed into $trans\_ker\_row\_0$ (containing {w0, w1, w2, w3}). Another example is the third row of the input tensor (containing {i8, i9, i10, i11}), which is transformed into $trans\_in\_row\_2$ (containing {c0, c1, c2, c3}).

Each row of the output tensor is calculated using a different sum of 1D Winograd convolutions. In general, the $Nth$ row of the output tensor is the sum of the 1D Winograd convolutions between the $[N{+}0]$ trans input row and the $[0]$ trans kernel row, the $[N{+}1]$ trans input row and the $[1]$ trans kernel row, and the $[N{+}2]$ trans input row and the $[2]$ trans kernel row. For example, to calculate the 0th row of the output tensor in figure 6-44 (containing {o0, o1}), we sum the 1D convolutions between $trans\_in\_row\_0$ with $trans\_ker\_row\_0$, $trans\_in\_row\_1$ with $trans\_ker\_row\_1$, and $trans\_in\_row\_2$ with $trans\_ker\_row\_2$. The result of the summing is stored in $trans\_sum\_row\_0$. As shown in figure 6-44, the correct result is still obtained if the results of the 3 1D convolutions are summed in the Winograd domain. The result of the summing can then be transformed to the spatial domain once, which reduces the number of transforms required. Figure 6-45 shows a pseudo-code implementation of figure 6-44.

### 6.12.3 Input Channels

Performing 2D Winograd convolution as a sum of 1D Winograd convolutions can also be performed on input tensors with multiple input channels. The process for handling multiple input channels is the same as for how it was handled when performing 2D Winograd convolution directly. In other words, separate

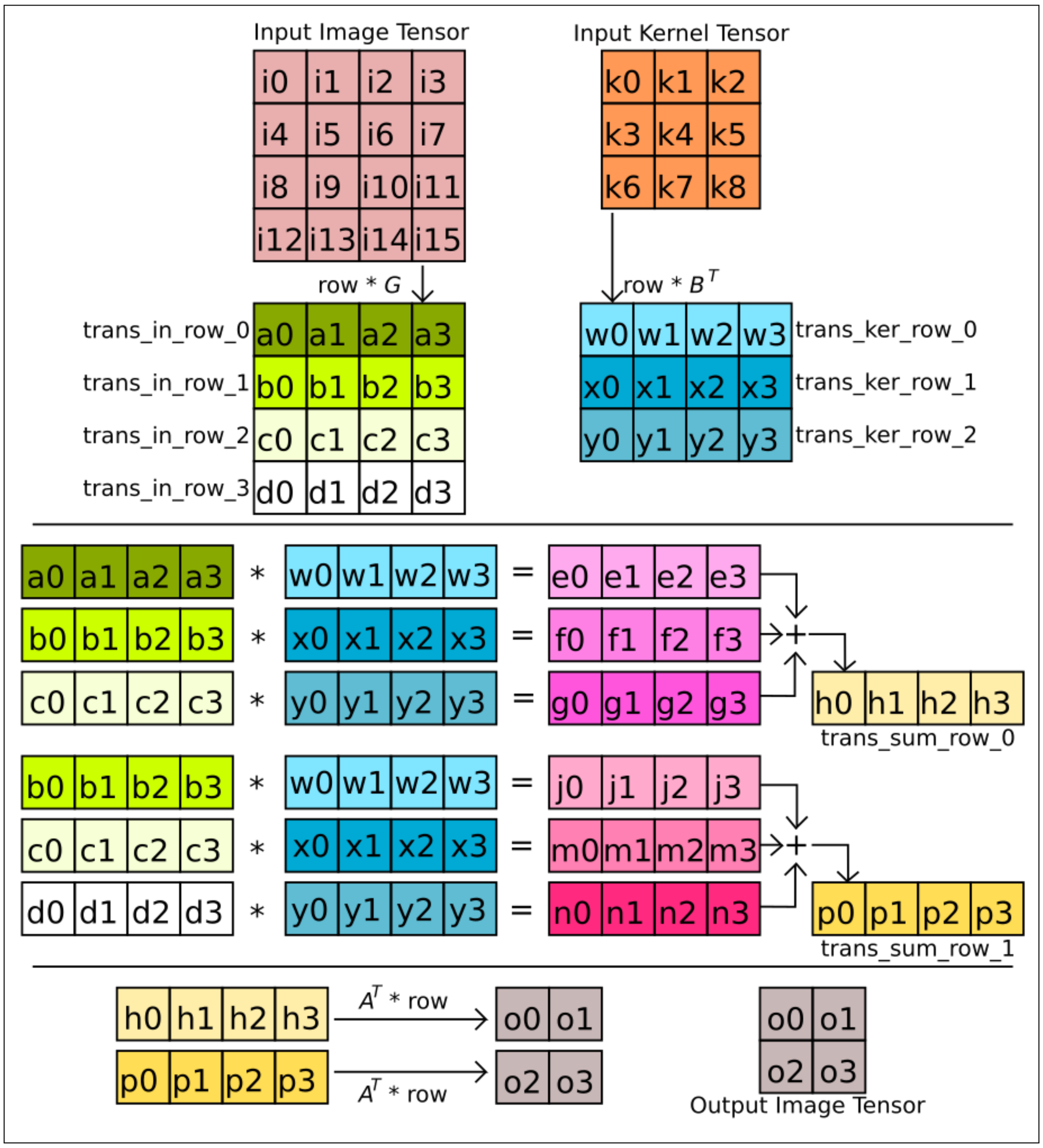

Input Image Tensor
i0 i1 i2 i3
i4 i5 i6 i7
i8 i9 i10 i11
i12 i13 i14 i15
Input Kernel Tensor
k0 k1 k2
k3 k4 k5
k6 k7 k8
row * G
row * $B^T$
trans_in_row_0 a0 a1 a2 a3
trans_in_row_1 b0 b1 b2 b3
trans_in_row_2 c0 c1 c2 c3
trans_in_row_3 d0 d1 d2 d3
w0 w1 w2 w3 trans_ker_row_0
x0 x1 x2 x3 trans_ker_row_1
y0 y1 y2 y3 trans_ker_row_2
a0 a1 a2 a3 * w0 w1 w2 w3 = e0 e1 e2 e3
b0 b1 b2 b3 * x0 x1 x2 x3 = f0 f1 f2 f3
c0 c1 c2 c3 * y0 y1 y2 y3 = g0 g1 g2 g3
h0 h1 h2 h3
trans_sum_row_0
b0 b1 b2 b3 * w0 w1 w2 w3 = j0 j1 j2 j3
c0 c1 c2 c3 * x0 x1 x2 x3 = m0 m1 m2 m3
d0 d1 d2 d3 * y0 y1 y2 y3 = n0 n1 n2 n3
p0 p1 p2 p3
trans_sum_row_1
h0 h1 h2 h3 $A^T$ * row o0 o1
p0 p1 p2 p3 $A^T$ * row o2 o3
o0 o1
o2 o3
Output Image Tensor

Figure 6-44: Performing 2D Winograd convolution by summing 1D Winograd Convolutions

Winograd convolutions are performed for every input channels, which are then all summed together to produce the final result. The different input channel results are again summed while the values are in the Winograd domain. This reduces the number of elements that must be transformed back to the spatial domain. Figure 6-46 shows how the pseudo-code from figure 6-45 can be extended to handle an arbitrary number of input channels.

```
in[4][4];
ker[3][3];
out[2][2]
trans_in_rows[4][4];
trans_ker_rows[3][4];
//transform the rows of the two inputs
for (h=0; h < 4; h++) {
  trans_in_rows[h] = transform(in[h], B^T);
}
for (kh = 0; kh < 3; kh++) {
  trans_ker_rows[kh] = transform(ker[kh], G);
}
//for each output row, calculate the sum of 1D
//convolutions at [N+0], [N+1], and [N+2].
for (oh = 0; oh < 2; oh++) {
  r0[4], r1[4], r2[4], rx[4];
  //assume * performs element-wise multiplication
  //on vectors
  r0 = trans_in_rows[oh+0] * trans_ker_rows[0];
  r1 = trans_in_rows[oh+1] * trans_ker_rows[1];
  r2 = trans_in_rows[oh+2] * trans_ker_rows[2];
  rx = (r0 + r1 + r2);
  out[oh] = transform(rx, A^T);
}

```

Figure 6-45: Pseudocode for performing 2D Winograd convolution using sums of 1D Winograd convolutions.

### 6.12.4 Extending the Height of the Input Tensor

Section 6.12.2 shows how 2D Winograd convolution can be performed by summing multiple 1D Winograd convolutions. In section 6.12.2, the height of the input tensor is a fixed size, however there are actually no restraints on the height of the input tensor (and the output tensor). As we are only applying 1D Winograd convolution on the rows of the input tensor, only the width of the input tensor must be fixed size. For example, if we are using a $F(2,3)$ 1D Winograd convolution, the width of the input tensor must be 4, but the height of the input tensor is not constrained. For example, the input tensor could be a $(7 \times 4)$ tensor, which will produce a $(5 \times 2)$ output tensor. Although, before the output tensor is transformed to the spatial domain, it will be stored in a $(5 \times 4)$ *trans_sum_rows* tensor. Figure 6-47 shows an example transformed $(7 \times 4)$ transformed input tensor, transformed kernel, and trans_sum_rows tensor. Each

```
in[C][4][4];
ker[C][3][3];
out[2][2]
trans_in_rows[C][4][4];
trans_in_kers[C][3][4];
for (c=0; c < C; c++) {
  for (h=0; h < 4; h++) {
    trans_in_rows[c][h] = transform(in[c][h], B^T);
  }
  for (kh = 0; kh < 3; kh++) {
    trans_in_kers[c][kh] = transform(ker[c][kh], G);
  }
}
for (oh = 0; oh < 2; oh++) {
  sum_row[4] = {0,0,0,0};
  //Perform a convolution for each input channels, and
  //sum them up.
  for (c = 0; c < C; c++) {
    r0[4], r1[4], r2[4];
    r0 = trans_in_rows[c][oh+0] * trans_ker_rows[c][0];
    r1 = trans_in_rows[c][oh+1] * trans_ker_rows[c][1];
    r2 = trans_in_rows[c][oh+2] * trans_ker_rows[c][2];
    sum_row += (r0 + r1 + r2);
  }
  out[oh] = transform(sum_row, A^T);
}

```

Figure 6-46: Pseudocode for performing multi-channel 2D Winograd convolution using sums of 1D Winograd convolutions.

row of the output tensor is calculated in the same way as outlined in section 6.12.2. The $Nth$ output tensor row is the sum of the 1D Winograd convolution between the $trans\_in\_rows[N + 0]$ and $trans\_ker\_rows[0]$, $trans\_in\_rows[N + 1]$ and $trans\_ker\_rows[1]$, and $trans\_in\_rows[N + 1]$ and $trans\_ker\_rows[1]$. These three convolutions are summed in the Winograd domain and the result is stored in the $Nth$ row of trans_sum_rows, to be transformed to the spatial domain later. For example, in figure 6-47, $trans\_sum\_rows[3]$ is calculated as $(trans\_in\_rows[3] \odot trans\_ker\_rows[0]$ + $trans\_in\_rows[4] \odot trans\_ker\_rows[1]$ + $trans\_in\_rows[5] \odot trans\_ker\_rows[2])$, where $\odot$ is element-wise multiplication.

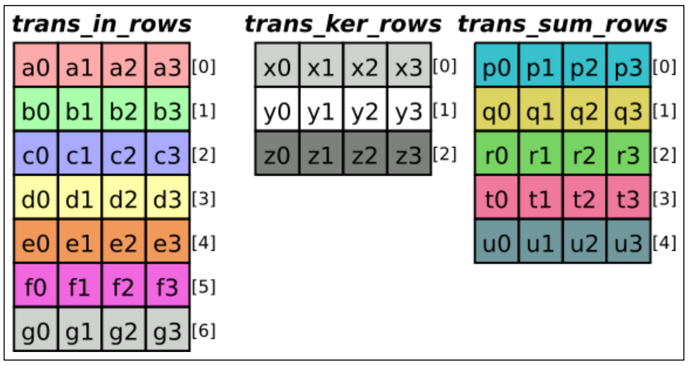


Figure 6-47: Example values for *trans_in_rows*, *trans_ker_rows*, and *trans_sum_rows*.

### 6.12.5 Accumulating using 1D Convolution

It's possible to calculate the values of the trans_sum_rows tensor using 1D convolutions between the columns of trans_in_rows and trans_ker_rows. The $(H \times in\_els)$ trans_in_rows tensor is split into $in\_els$ column vectors of length $H$, and the $(K \times in\_els)$ trans_ker_rows tensor is split into $in\_els$ column vectors of length $K$. A 1D convolution is then performed between matching pairs of column vectors from from trans_in_rows and trans_ker_rows to calculate the matching column in trans_sum_rows. For example, in figure 6-48, the second column vector from trans_in_rows is convolved with the second column vector from trans_ker_rows to calculate values for the second column of trans_sum_rows.

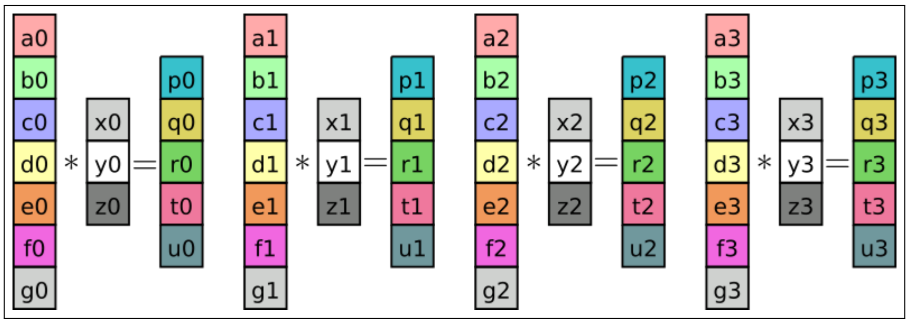


Figure 6-48: Calculating the values in *trans_sum_rows* by performing 1D convolution between the columns of *trans_in_rows*, and *trans_ker_rows*.

Using 1D convolution to calculate trans_in_rows can also be used when the input tensors have multiple input channels. The input tensor is now a $(H \times in\_els \times C)$ tensor, and the trans_ker_rows tensor is a $(K \times in\_els \times C)$ 3D tensor. Figure 6-49 shows the values from figure 6-47 with an input channels dimension added.

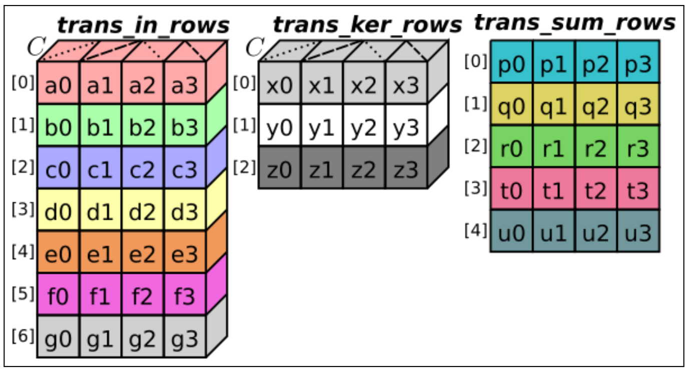


Figure 6-49: Example values for multi-channel $trans\_in\_rows$, $trans\_ker\_rows$, and $trans\_sum\_rows$.

Now to calculate the values in trans_sum_rows, the 3D trans_in_rows tensor is split into $in\_els$ 2D matrices with shape $(H \times C)$, and trans_ker_rows is split into $in\_els$ 2D matrices with shape $(K \times C)$. A 1D convolution is performed between the matching matrices to calculate each column of trans_sum_rows. Figure 6-50 shows an example of this, where trans_in_rows and trans_ker_rows have been split into 4 2D matrices. Four 1D convolutions are then performed to calculate the values in the four columns of $trans\_sum\_rows$.

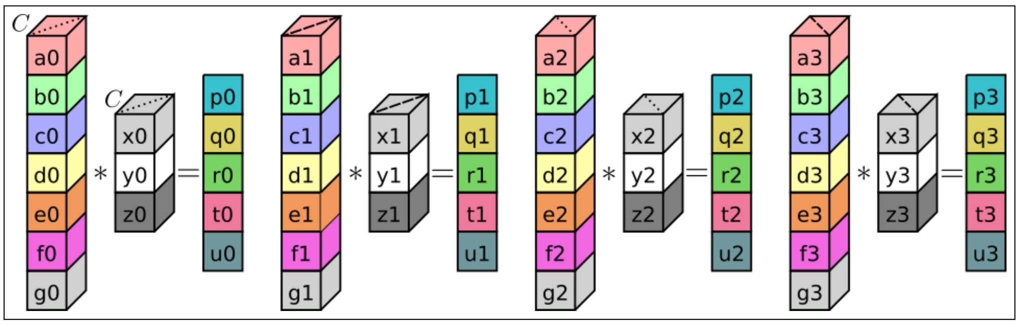

C
a0 b0 c0 d0 e0 f0 g0 * x0 y0 z0 = p0 q0 r0 t0 u0
a1 b1 c1 d1 e1 f1 g1 * x1 y1 z1 = p1 q1 r1 t1 u1
a2 b2 c2 d2 e2 f2 g2 * x2 y2 z2 = p2 q2 r2 t2 u2
a3 b3 c3 d3 e3 f3 g3 * x3 y3 z3 = p3 q3 r3 t3 u3

Figure 6-50: Calculating the values in $trans\_sum\_rows$ by performing multi-channel 1D convolution between the columns of $trans\_in\_rows$, and $trans\_ker\_rows$.

### 6.12.6 Tiling the Input and Output Tensor

Up to now, our sum of 1D Winograd method can only handle input tensors with a width equal to the size expected by the 1D Winograd convolution algorithm being used. For example, if we are using a $F(2,3)$ Winograd convolution, the width of the input tensor must be 4 (and the width of the output tensor will be 2). However, by tiling the rows of the input and output tensor in a similar manner to section 6.5.2, we can handle input tensors with any width.

First, the rows of the output tensor are split into non-overlapping 1D tiles. The length of the tiles is equal to the output given by the 1D Winograd convolution being used. For example, if a $F(2,3)$ Winograd convolution is being used, each row of the output tensor is split into tiles of length 2, as $F(2,3)$ produces 2 output points.

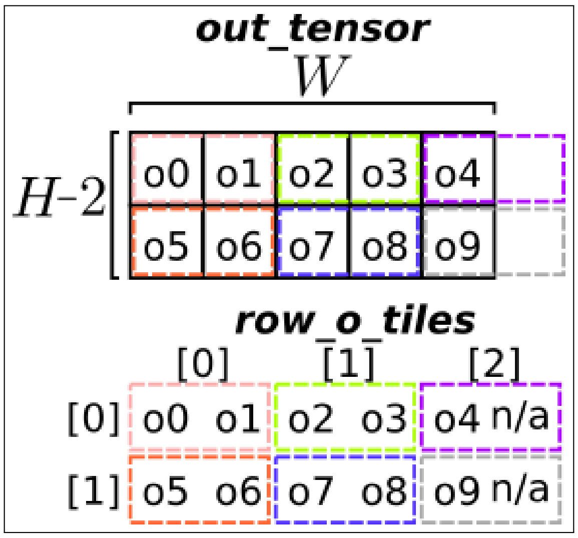


Figure 6-51: Splitting each row of a $(2 \times 5)$ output tensor into 3 o-tiles.

Figure 6-51 shows an $(2 \times 5)$ output tensor, where each row has been split into 3 tiles of length 2. The last tile of each row partially lays outside the bounds of the row. The points outside the bounds of the row will be discarded if they are calculated during convolution.

The rows of the input tensors are also split into 1D tiles. The tiles are overlapping, and are the size of the input expected by the 1D Winograd convolution being used. For example, a $F(2,3)$ Winograd convolution expects 4 input points, so the input tensor tiles would be have a length of four. The first tile on each row starts at position $(-\lfloor K/2 \rfloor)$, where $K$ is length of the kernel expected by the 1D Winograd convolution being used. The next tile is placed $out\_els$ steps ahead of the previous tile, where $out\_els$ is the number of output points produced by the 1D Winograd convolution being used. For example, if a $F(2,3)$ Winograd convolution was being used, the tiles for each row of the input tensor would be at positions $\{-1, 1, 3, 5, ...\}$, and they would all be of length 4.

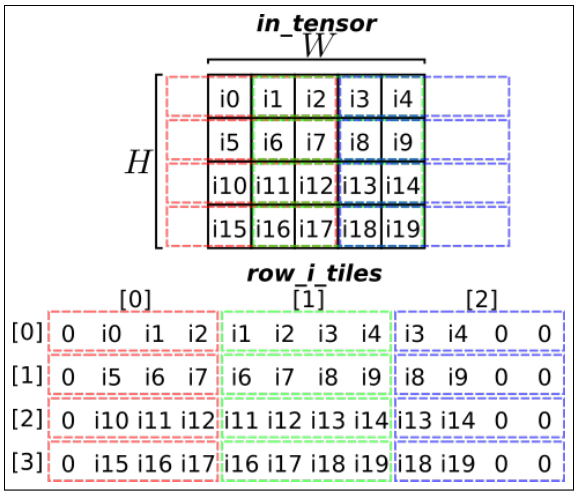

in_tensor
W
H
i0 i1 i2 i3 i4
i5 i6 i7 i8 i9
i10 i11 i12 i13 i14
i15 i16 i17 i18 i19
row_i_tiles
[0] [1] [2]
[0] 0 i0 i1 i2 i1 i2 i3 i4 i3 i4 0 0
[1] 0 i5 i6 i7 i6 i7 i8 i9 i8 i9 0 0
[2] 0 i10 i11 i12 i11 i12 i13 i14 i13 i14 0 0
[3] 0 i15 i16 i17 i16 i17 i18 i19 i18 i19 0 0

Figure 6-52: Splitting each row of a $(4 \times 5)$ input tensor into 3 overlapping i-tiles.

Figure 6-52 shows each row of a $(4 \times 5)$ input tensor being split into 3 overlapping tiles, each of length 4. As some tiles lie outside the bounds of the input tensor, they have values that must be synthesized. For example, the first value of $row\_i\_tiles[0][0]$ in figure 6-52 must be synthesized.

Each tile of an input tensor row (now referred to as a *row i-tile*) is transformed separately to create the transformed row i-tiles. The transformed row i-tiles are stored in *trans_row_i_tiles*. Figure 6-53 shows the trans_row_i_tiles data structure created from the row_i_tiles data structure in figure 6-52. The input kernel is not tiled, and is transformed in the same way as covered in section 6.12.2 to create the *trans_ker_rows* data structure.

Calculating the values of the tiles that make up the rows of the output tensor (now referred to as *row o-tiles*) is similar to the process for calculating the output points in section 6.12.2. To calculate the values in $row\_o\_tiles[N][M]$, we sum the results of the 1D Winograd convolutions between $row\_i\_tiles[N + 0][M]$ with $trans\_ker\_rows[0]$, $row\_i\_tiles[N + 1][M]$ with $trans\_ker\_rows[1]$, and $row\_i\_tiles[N+2][M]$ with $trans\_ker\_rows[2]$. For example, to calculate the

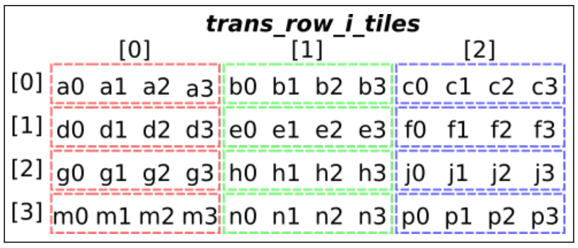

trans_row_i_tiles
[0] [1] [2]
[0] a0 a1 a2 a3 b0 b1 b2 b3 c0 c1 c2 c3
[1] d0 d1 d2 d3 e0 e1 e2 e3 f0 f1 f2 f3
[2] g0 g1 g2 g3 h0 h1 h2 h3 j0 j1 j2 j3
[3] m0 m1 m2 m3 n0 n1 n2 n3 p0 p1 p2 p3

Figure 6-53: Example values for the transformed row i-tile, created from the example values in figure 6-52.

values in $row\_o\_tiles[1][2]$ from figure 6-51, we sum the results of the 1D Winograd convolution between $row\_i\_tiles[1][2]$ with $trans\_ker\_rows[0]$, $row\_i\_tiles[2][2]$ with $trans\_ker\_rows[1]$, and $row\_i\_tiles[3][2]$ with $trans\_ker\_rows[2]$.

Section 6.12.5 covered how all the values in the *trans_sum_rows* data structure (the data structure that contains all the output tensor values in the Winograd domain) can be calculated using *in_els* 1D convolutions between the columns of the input tensor and the columns of the input kernel. This technique can still be used when the input and output tensor are tiled. The *trans_ker_rows* is split into *in_els* columns like in section 6.12.5. The values in the transformed row i-tiles are split into *in_els* matrices of shape $(H \times \lceil W/out\_els \rceil)$. $\lceil W/out\_els \rceil$ is the number of tiles need to cover each row of the input (or output) tensor. The first matrix is filled with the first value from every transformed row i-tile of the input tensor, the second matrix is filled with the second value from every transformed row i-tile, and so on. Figure 6-54 shows an example of four matrices, constructed using the values from figure 6-53. A convolution is then performed between each matrix and a column from trans_ker_rows. This creates *in_els* 2D matrices of shape $((H - (2 \times \lfloor K/2 \rfloor)) \times \lceil W/out\_els \rceil)$. Figure 6-54 shows four convolutions performed to produce the output matrices using the values from figure 6-53.

The values spread across the *in_els* output matrices are the values of the *row_trans_sum_tiles* data structure. *row_trans_sum_tiles* contains $(H-(2\times\lfloor K/2 \rfloor))\times \lceil W/out\_els \rceil)$ tiles of length *in_els*. Each tile in *row_trans_sum_tiles* can be transformed by $A^T$ (one of the transformation matrices) to calculate the values in a corresponding o-tile from the output tensor. Figure 6-54 shows how the values in the output matrices are reordered to create *row_trans_sum_tiles*. The tile at

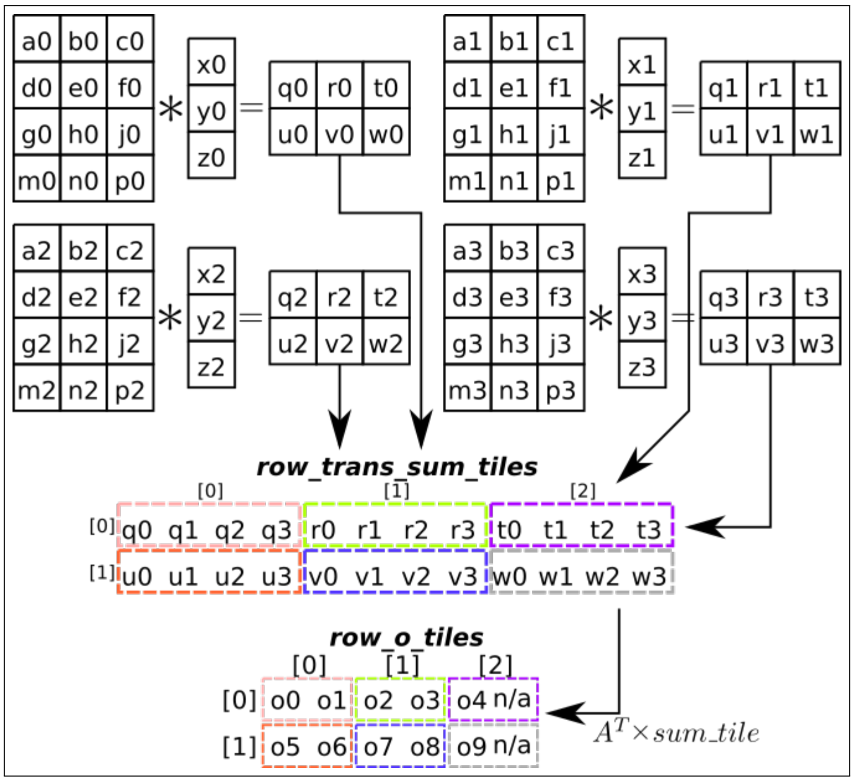


Figure 6-54: Performing 2D convolution using smaller convolutions between values from $trans\_row\_i\_tiles$ and the columns of $trans\_ker\_rows$.

position $row\_trans\_sum\_tiles[N][M]$ is used calculate the o-tile $row\_o\_tiles[N][M]$ that makes up the output tensor.

### 6.12.7 Temporary Memory and Genvolution

In comparison to a normal Winograd CNN convolution that uses a 2D Winograd convolution with matrix multiplication to perform the element-wise multiplication, the CNN convolution outlined above requires significantly less temporary memory. The extra memory for the transformed i-tiles is reduced from $(\lceil H/out\_els\rceil)\times(\lceil w/out\_els\rceil)\times(in\_els\times in\_els)\times C$ floats to $H\times(\lceil w/out\_els\rceil)\times(in\_els)\times C$ floats. This is a reduction of $((out\_els+K-1)/out\_els)$ times. The memory required for the transformed kernels is reduced from $(in\_els\times in\_els)\times C\times M$ floats to $(in\_els\times K)\times C\times M$ floats, which is a reduction of $((out\_els+K-1)/K)$ times. The data structure containing the output values in the Winograd domain is reduced from $(\lceil H/out\_els\rceil)\times(\lceil w/out\_els\rceil)\times(in\_els\times in\_els)\times M$ floats to $H\times(\lceil w/out\_els\rceil)\times(in\_els)\times M$ floats. This is also a reduction of $((out\_els + K - 1)/out\_els)$ times. The reduction in the temporary memory required is because the input tensor and input kernel are only transformed in one dimension, rather then two. Table 6.1 shows the reduction in temporary memory between standard 2D Winograd CNN convolution and our Sum of 1D Winograd Convolution, for six common Winograd input sizes.

| Winograd | Trans Kernel Reduction | Trans I-Tiles Reduction | Winograd Output Reduction |
|---|---|---|---|
| $F(2,3)$ | $1^1/_3$ | 2 | 2 |
| $F(3,3)$ | $1^2/_3$ | $1^2/_3$ | $1^2/_3$ |
| $F(4,3)$ | 2 | $1^1/_3$ | $1^1/_3$ |
| $F(2,5)$ | $1^1/_6$ | 3 | 3 |
| $F(3,5)$ | $1^2/_7$ | $2^1/_7$ | $2^1/_7$ |
| $F(4,5)$ | $1^3/_8$ | 2 | 2 |

Table 6.1: Memory Size Reduction of transformed tensors using 1D Winograd vs 2D Winograd

Standard CNN convolution requires $(K \times K) \times (H \times W)$ multiplications to perform a single channel CNN convolution and produce $(H \times W)$ output points. Winograd CNN Convolution using 2D Winograd convolution and tiling requires $((\lceil H/out\_els \rceil) \times (\lceil W/out\_els \rceil) \times (in\_els \times in\_els))$ multiplications to produce $(H \times W)$ output points. This reduces the number of multiplications needed by roughly $((out\_els^2 K^2)/(out\_els + K - 1)^2)$ times, depending on the effect of the two ceiling operations. Our sum of 1D Winograd convolutions requires $(H \times (\lceil W/out\_els \rceil) \times K \times in\_els)$ multiplications. This reduces the number of multiplications needed by roughly $((out\_els \times K)/(out\_els + K - 1))$ times, depending on the effect of the two ceiling operations. This reduction is $((out\_els \times K)/(out\_els + K - 1))$ times smaller than the reduction in multiplications allowed by the 2D Winograd CNN convolution method. It is hoped that for low end devices, the decrease in memory usage will outweigh the theoretically better computational load. Table 6.2 show the reduction in the number of multiplication required using the standard 2D winograd CNN convolution, and our sum of 1D Winograd method, assuming we are using a $F(2,3)$ Winograd convolution.

| Method | H/W=13 | H/W=14 | H/W=27 | H/W=28 | H/W=56 |
|---|---|---|---|---|---|
| CNN Convolution | 1521 muls | 1764 muls | 6561 muls | 7056 muls | 28224 muls |
| 2D Winograd CNN | 1.94 × | 2.25 × | 2.092 × | 2.25 × | 2.25 × |
| Sum of 1D CNN | 1.393 × | 1.5 × | 1.446 × | 1.5 × | 1.5 × |

| Method | H/W=96 | H/W=127 | H/W=192 | H/W=227 |
|---|---|---|---|---|
| CNN Convolution | 82944 muls | 145161 muls | 331776 muls | 463761 muls |
| 2D Winograd CNN | 2.25 × | 2.214 × | 2.25 × | 2.23 × |
| Sum of 1D CNN | 1.5 × | 1.488 × | 1.5 × | 1.493 × |

Table 6.2: Reduction in number of multiplications required for convolution when using 2D Winograd CNN convolution, and when using our sum of 1D Winograd CNN convolution.

As covered in section 6.7, 2D Winograd CNN convolution uses matrix multiplication to perform element-wise multiplication and input channel accumulation. Matrix multiplication is a heavily researched topic and the matrix multiplications in the convolution can be performed by a highly optimized matrix multiplication library on the majority of processors. As covered in section 6.12.5, our sum of 1D Winograd CNN convolution uses a smaller convolution with a 1D kernel to perform the necessary element-wise multiplication and input channel accumulation. This means we cannot use the optimized matrix multiplication libraries available. Instead, Genvolution is used to generate fast convolution implementations that perform the necessary convolutions.

## 6.13 Evaluation Of Results

To evaluate the effectiveness of Winogen, Winogen was used to generate Winograd Convolution implementations for a set of CNN convolution input sizes for two target machines (ARM Target 1, and ARM Target 2). The CNN convolution input sizes were taken from five commonly used CNN networks: AlexNet (Krizhevsky, Sutskever, and Hinton 2012), Inception V4 (Singh and Markovitch 2017), MobileNet V2 (Howard et al. 2017), ResNET-152 (He et al. 2016), and VGG ILSRVC (Bengio and LeCun 2015). A unique implementation using standard Winograd convolution was generated for each input size on each target machine (labelled as 'Winogen' in section 6.15). A unique implementation using our novel Winograd method (covered in section 6.12) was also generated for each input size on each target machine (labelled as 'Winogen-1D' in section 6.15). The average execution time and cache miss rate for each generated implementation was collected on each target. The execution time and cache miss rates are the mean average of 20 runs. The ARMCL Library (see section 3.1.4) implementation of Winograd convolution was used as the baseline method to compare performance against. The ARMCL Winograd convolution implementation is labelled as 'ARMCL-Winograd' in section 6.15.

### 6.13.1 ARM Target 1

Winogen outperformed ARMCL-Winograd for all input sizes. Winogen-1D outperformed ARMCL-Winograd for 14 of 29 input sizes. Winogen performed best overall in 17 of the 29 input sizes. Winogen-1D performed best in 12 of the 29 input sizes.

Figures 6-55, 6-56, and 6-57 suggest that there is a relationship between the relative performance of Winogen-1D, and the ratio between $H \times W$ and $C \times M$. When the height and width of the input tensor (i.e. $H$ and $W$) is much larger than the number of input channels and input kernels (i.e. $C$ and $M$), Winogen-1D is significantly faster than ARMCL-Winograd. For example, in the first two input sizes of figure 6-56(a) and 6-57, $H$ and $W$ are much larger than $C$ and $M$, and Winogen-1D performs very well relative to ARMCL-Winograd. The opposite is also true. In the last two input sizes of figure 6-56(a) and 6-57, $H$ and $W$ are much smaller than $C$ and $M$, and Winogen-1D performs very poorly. Figures 6-61, 6-62, and 6-63 also suggest that the ratio between $H \times W$ and $C \times M$ has an effect on the L1 cache miss rate of Winogen-1D. Figures 6-67, 6-68, and 6-69 also suggest that this relationship holds for L2 cache miss rates too. The higher cache miss rates may explain the worse relative performance when $C$ and $M$ are larger than $H$ and $W$.

### 6.13.2 ARM Target 2

Winogen outperformed ARMCL-Winograd for 8 of 29 input sizes. Winogen-1D outperformed ARMCL-Winograd for 2 of 29 input sizes.

Winogen performed best when compared to ARMCL-Winograd in the first input size of figure 6-58(a) (where $H$=27, $W$=27, $C$=96, $M$=256, $K$=5). Figure 6-64 shows that Winogen has a lower L1 cache miss rate for this input size, which may explain the difference in performance. This input size is also the only tested input size with a $(5 \times 5)$ input kernel. Like Winogen-1D on ARM Target 1, both Winogen and Winogen-1D on ARM Target 2 show a relationship between performance and the ratio between $H \times W$ and $M \times C$. This can be most clearly seen in figures 6-59(a) and 6-60, where the relative performance of both Winogen and Winogen-1D degrade as the ratio between $H \times W$ and $M \times C$ inverts. However, the relationship between the ratio of $H \times W$ and $M \times C$, and higher cache miss rates is not as distinct in the cache miss rate graphs for ARM Target 2 as it was in cache miss rate graphs for ARM Target 1.

## 6.14 Conclusions

Our two main goals relating to Winograd convolution were: 1) investigate a number of Winograd convolution optimizations and produce a program generator that would produce optimized Winograd convolution implementations, and 2) investigate the performance of a novel CNN convolution algorithm that used a sum of 1D Winograd convolutions to reduce the temporary memory overhead needed by standard Winograd implementations.

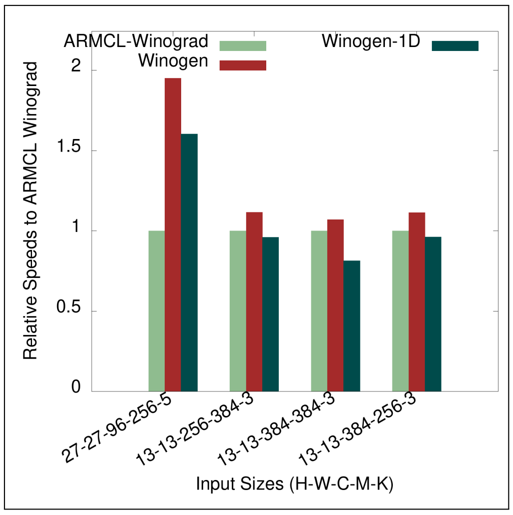
We believe we were successful in achieving our first goal. With very minimal user input, Winogen was able to generate optimized Winograd convolution implementations for a number of different input sizes that matched or outperformed vendor library (ARMCL) Winograd convolution implementations. Winogen was extremely successful on ARM Target 1. We believe this shows that automatic code generation should definitely be considered when optimizing single threaded code for smaller ARM devices.

We believe we were also successful in achieving our second goal. Our novel algorithm performed better than our standard Winograd implementation and the vendor library Winograd implementation for 12 of 29 input sizes on ARM Target 1. For these input sizes, our novel algorithm required significantly less memory. Our novel algorithm was not as successful on ARM Target 2, not being the fastest for any input size. However, we do not consider this as significant, because our novel algorithm was designed with cache and memory constrained devices such as ARM Target 1 in mind. We believe we have shown that our novel algorithm should be considered when implementing Winograd convolution on memory constrained devices. The increased performance and reduced temporary memory overhead, even if only for certain CNN convolutions in a CNN network, could have a significant impact if the network is to be run many times over it's lifetime.

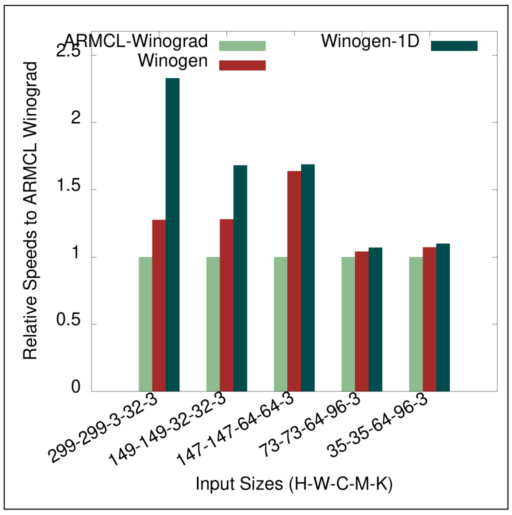
We have now concluded our discuss of Winogen and Winograd convolution. We now return to Genvolution, and investigate alternative uses. Starting with using Genvolution to generate matrix multiplication implementations.

## 6.15 Results

### Execution Time

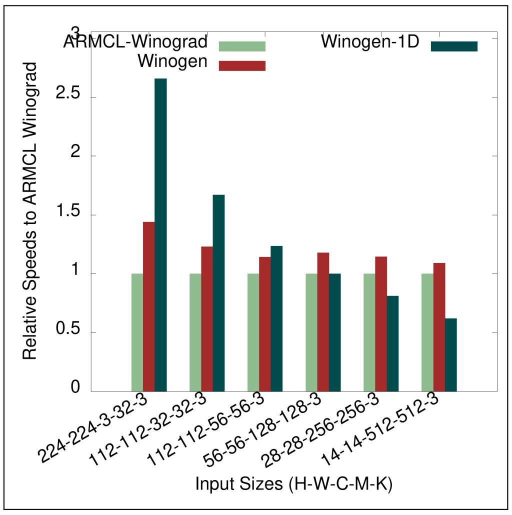


(a) AlexNet convolutions.

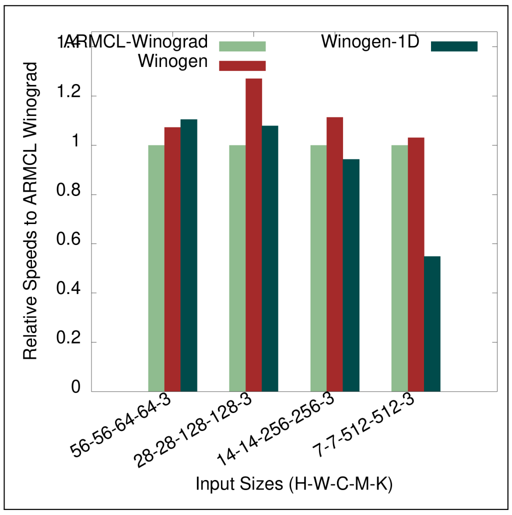


(b) Inception V4 convolutions.

Figure 6-55: Execution time on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 6-56: Execution time on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

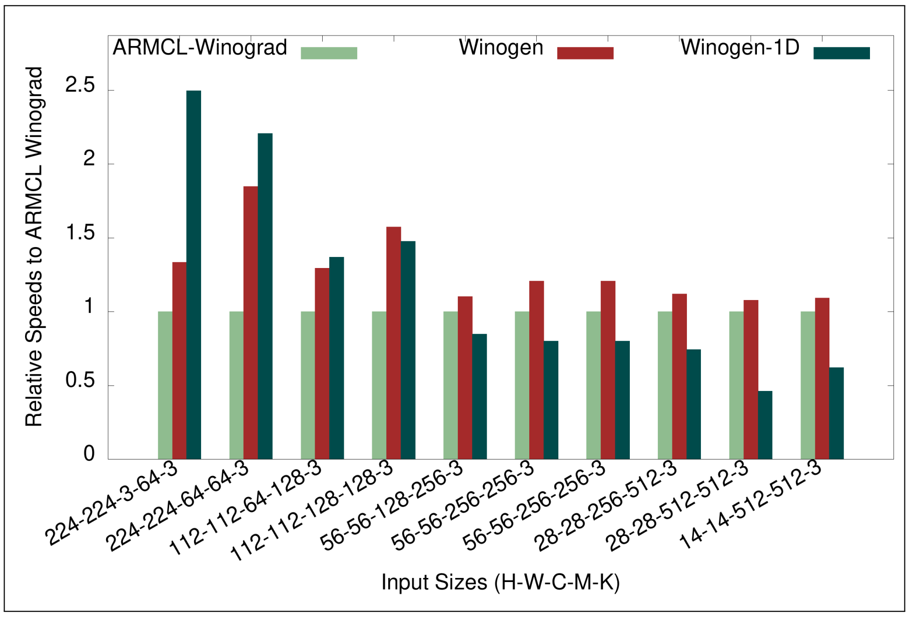
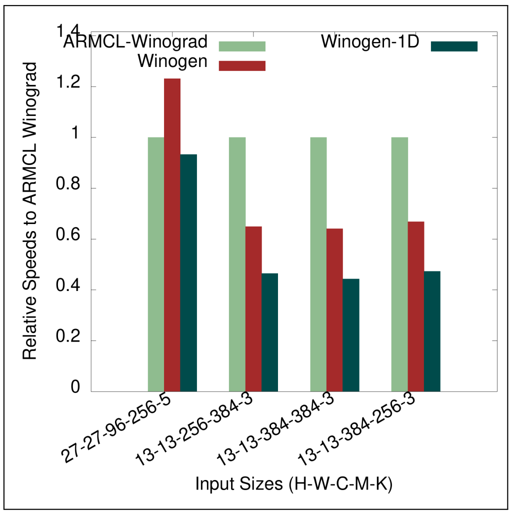
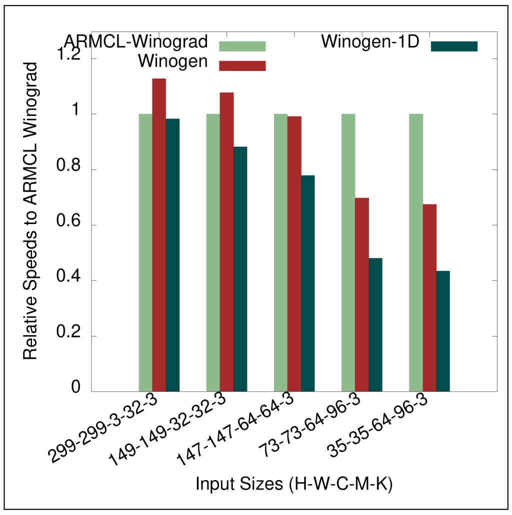


Figure 6-57: Execution Time of Winogen implementations on VGG ILSRVC Convolutions on ARM Target 1.

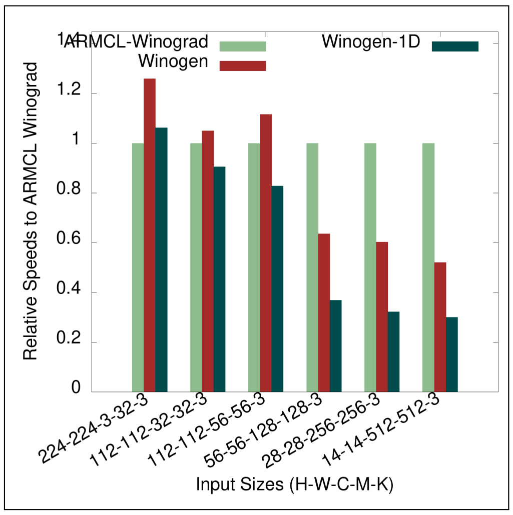


(a) AlexNet convolutions.

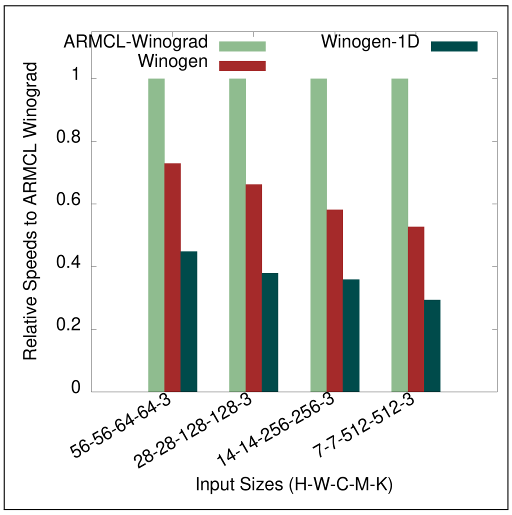


(b) Inception V4 convolutions.

Figure 6-58: Execution time on AlexNet and Inception V4 convolutions on ARM Target 2.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 6-59: Execution time on MobileNet V2 and ResNET-152 convolutions on ARM Target 2.

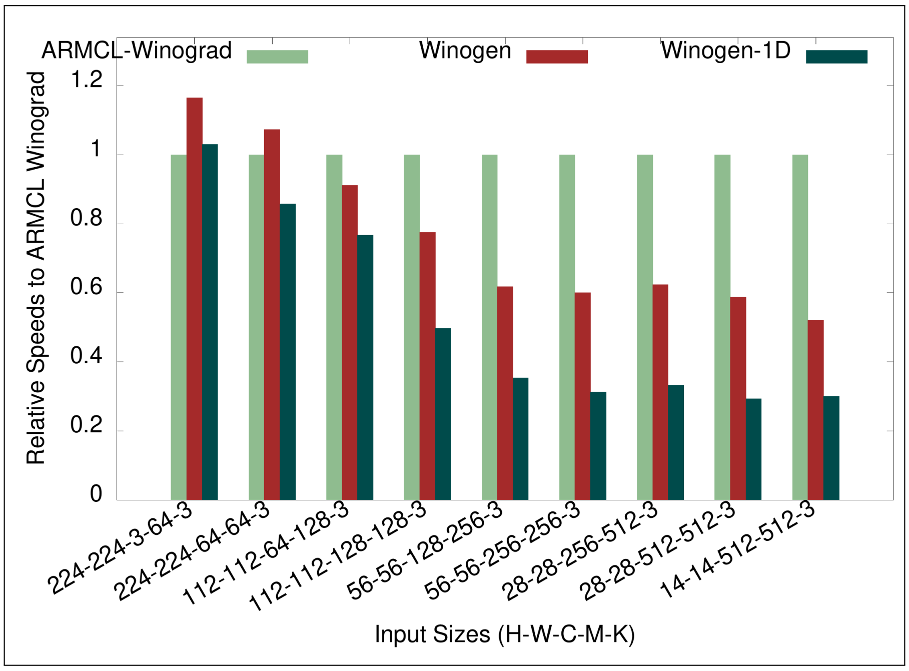
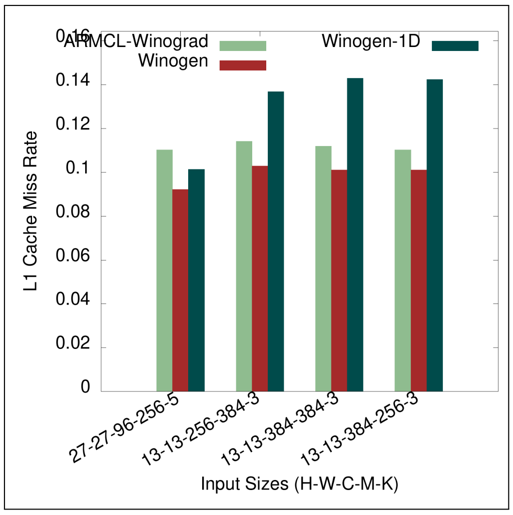
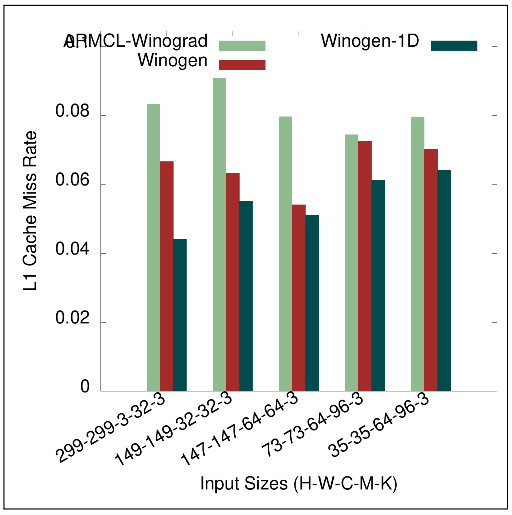


Figure 6-60: Execution Time of Winogen implementations on VGG ILSRVC Convolutions on ARM Target 2.

## L1 Cache Miss Rate

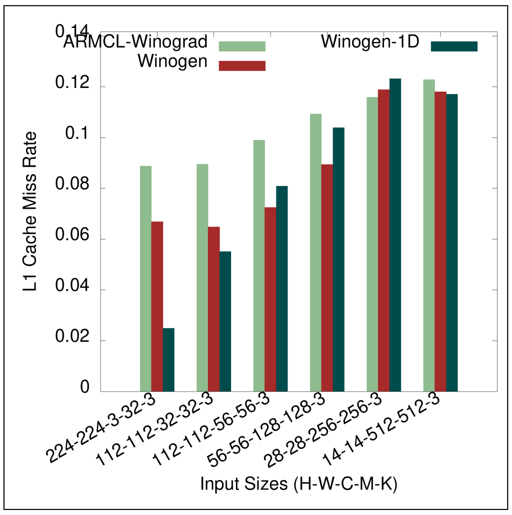


(a) AlexNet convolutions.

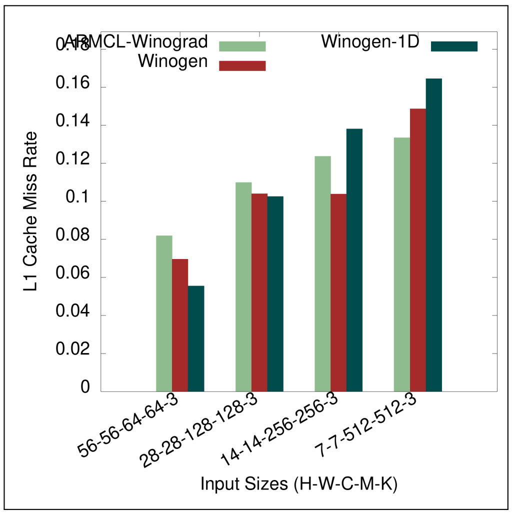


(b) Inception V4 convolutions.

Figure 6-61: L1 cache miss rate on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet convolutions.



(b) ResNET-152 convolutions.

Figure 6-62: L1 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

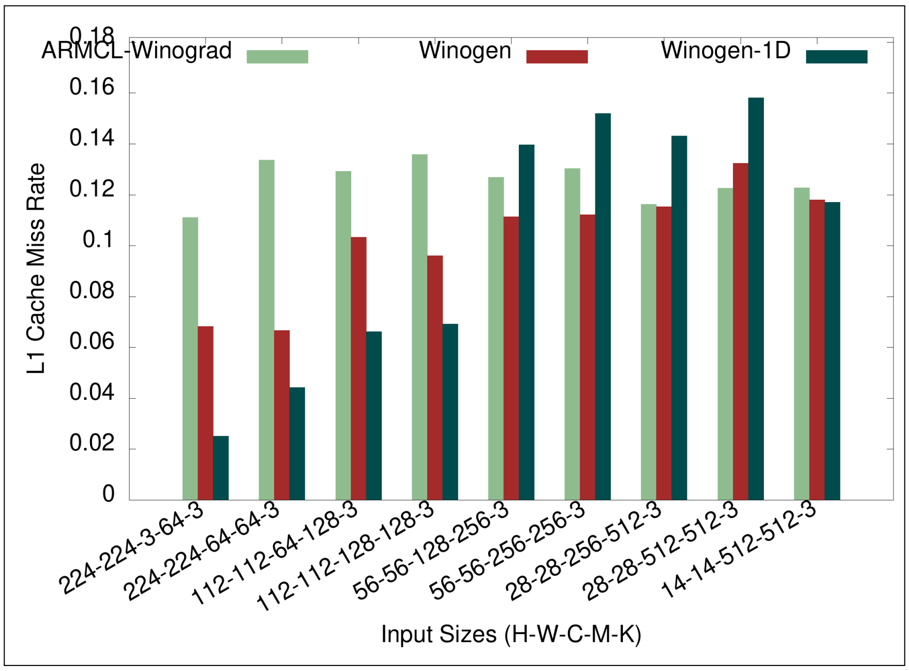
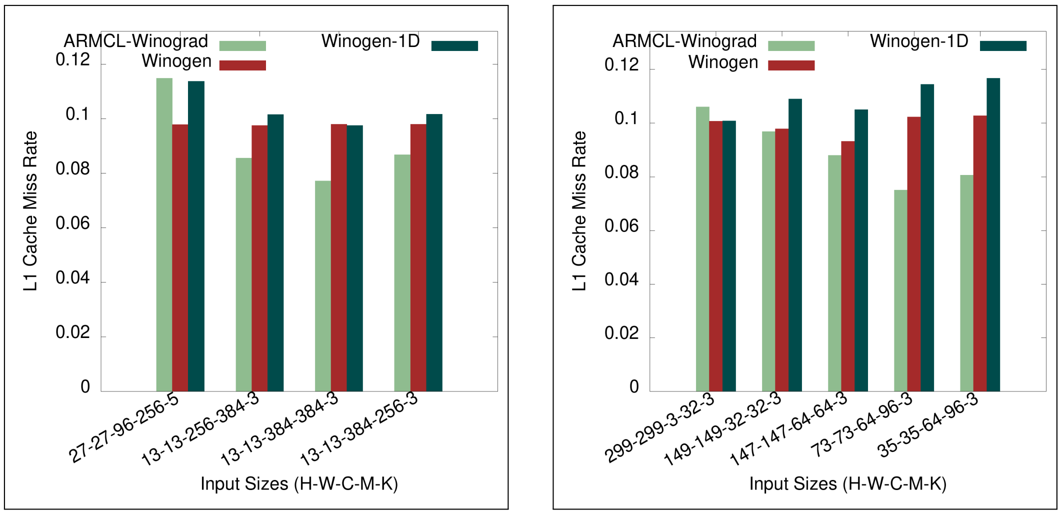
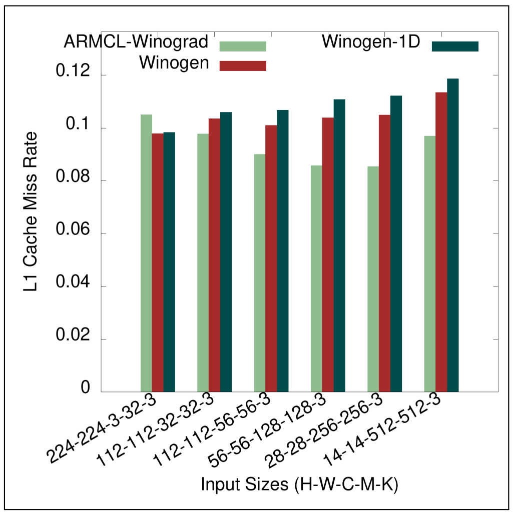


Figure 6-63: L1 Cache Miss Rate of Winogen implementations on VGG ILSRVC Convolutions on ARM Target 1.

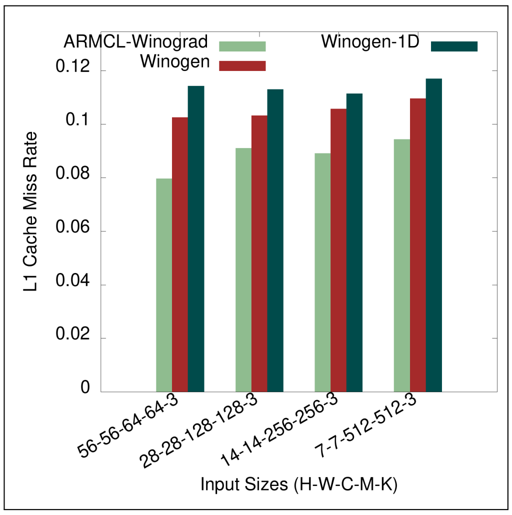


(a) AlexNet convolutions. (b) Inception V4 convolutions.

Figure 6-64: L1 cache miss rate on AlexNet and Inception V4 convolutions on ARM Target 2.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 6-65: L1 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 2.

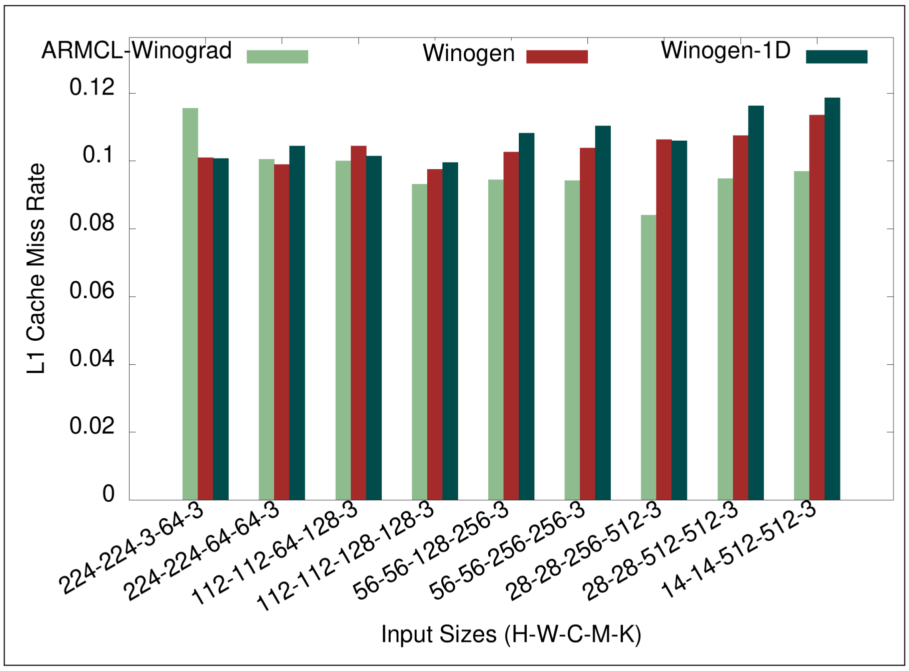
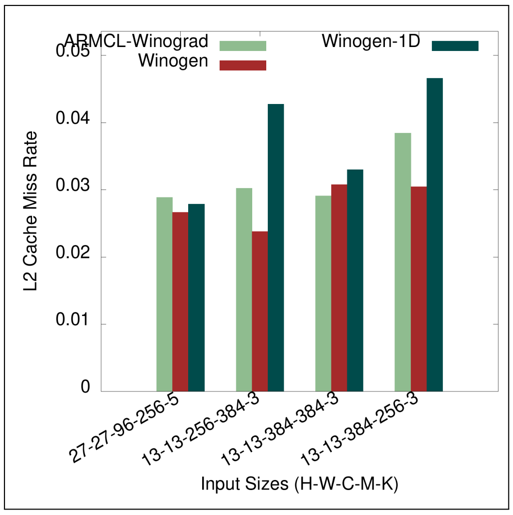
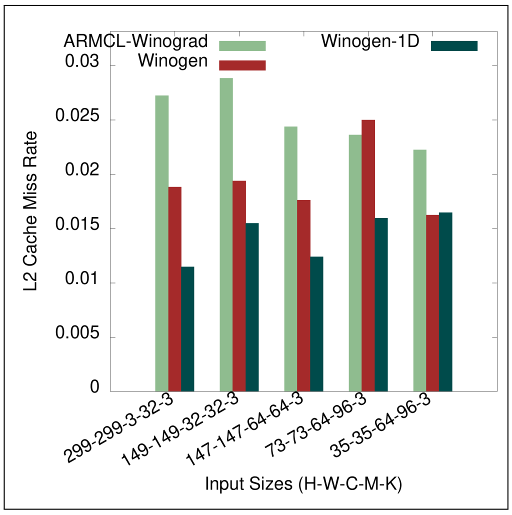


Figure 6-66: L1 Cache Miss Rate of Winogen implementations on VGG ILSRVC Convolutions on ARM Target 2.

## L2 Cache Miss Rate

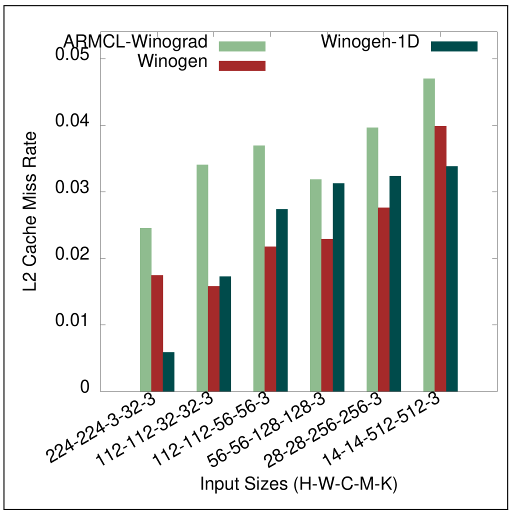


(a) AlexNet convolutions.

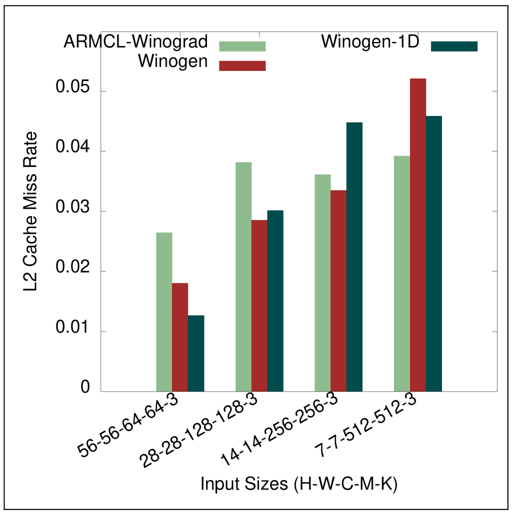


(b) Inception V4 convolutions.

Figure 6-67: L2 cache miss rate on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 6-68: L2 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

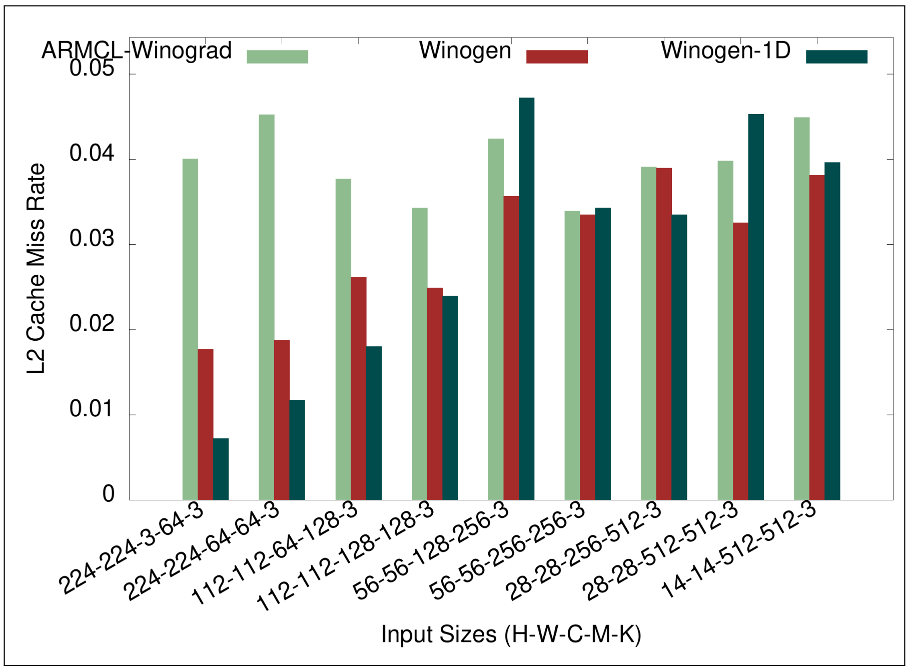
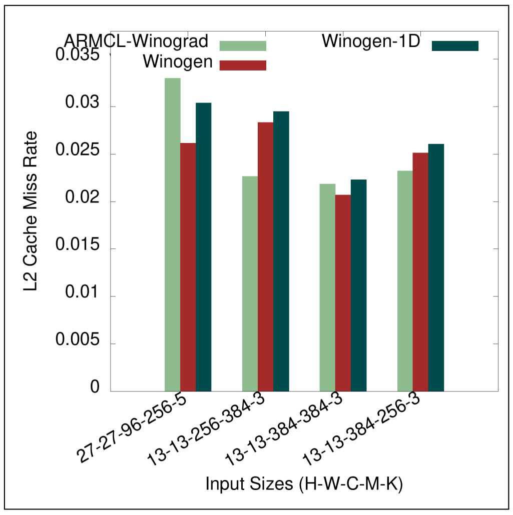
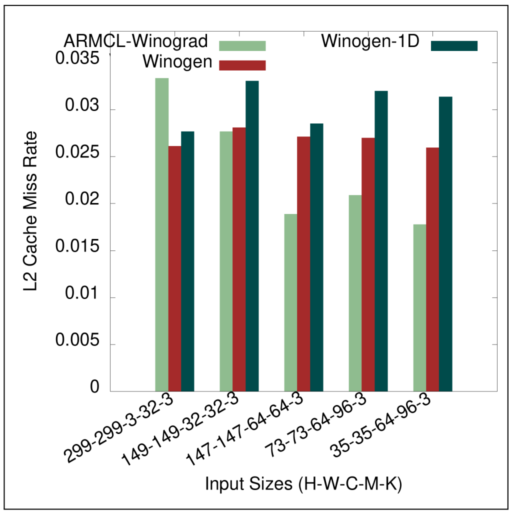


Figure 6-69: L2 Cache Miss Rate of Winogen implementations on VGG ILSRVC Convolutions on ARM Target 1.

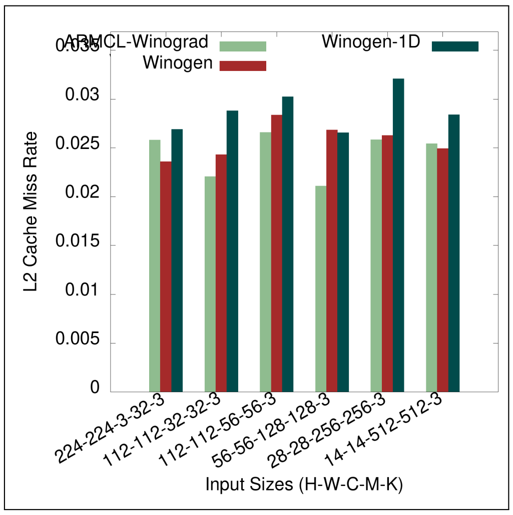


(a) AlexNet convolutions.

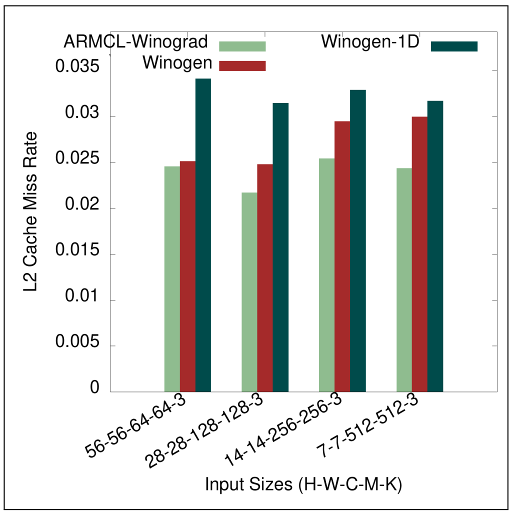


(b) Inception V4 convolutions.

Figure 6-70: L2 cache miss rate on AlexNet and Inception V4 convolutions on ARM Target 2.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 6-71: L2 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 2.

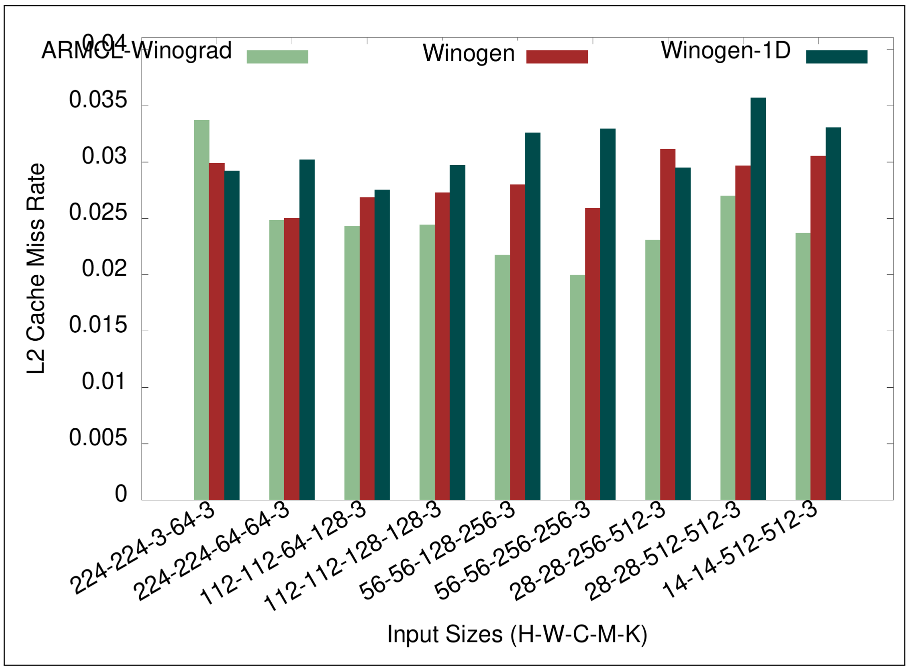


Figure 6-72: L2 Cache Miss Rate of Winogen implementations on VGG ILSRVC Convolutions on ARM Target 2.

# Chapter 7

# Optimizing GEMM for Low Power ARM Devices

## 7.1 Chapter Motivation

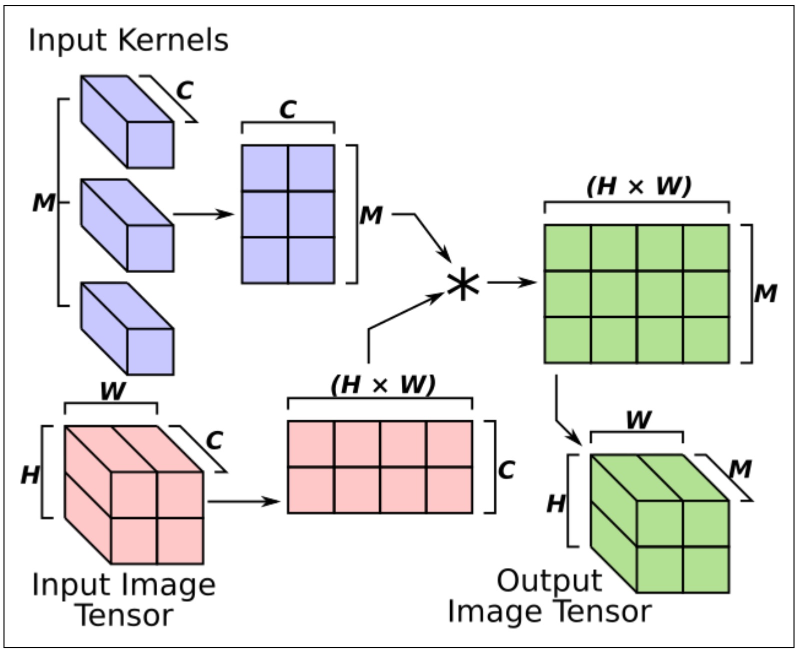


Figure 7-1: Performing a $(1 \times 1)$ kernel convolution as a matrix multiplication

Figure 7-1 shows how a CNN convolution using input kernels with a kernel height and width of 1 (i.e. the input kernels are stored in a tensor $(M \times 1 \times 1 \times C)$) can be performed using a matrix multiplication. The input image tensor

$(H \times W \times C)$ is converted into a 2D matrix $((H \times W) \times C)$, and the input kernels $(M \times 1 \times 1 \times C)$ are converted into a 2D matrix $(C \times M)$. Performing a matrix multiplication between the two 2D matrices will perform the necessary dot products between the input channels from the input image tensor and the input kernels. The result of the matrix multiplication is a 2D matrix $((H \times W) \times M)$, which can be converted into the output image tensor $(H \times W \times M)$. The reverse is also true, a matrix multiplication can be implemented using CNN convolution code. This means we can apply our previous work on optimizing CNN convolution to also optimize matrix multiplication.

We focus on improving matrix multiplication solely on low power ARM devices, because there has already been an extremely large amount of research put into optimizing matrix multiplication on Intel architecture processors. Genvolution was used to automatically optimize and tune matrix multiplication implementations. The same microkernels covered in chapter 5 were used for generating matrix multiplication implementations. Minor alteration were required that are covered in sections 7.2.1 and 7.3.

## 7.2 Data Prefetching

Data prefetching is an optimization technique for reducing processor stalls caused by slow memory accesses (Callahan, Kennedy, and Porterfield 1991). Often in computer programs, memory locations are accessed in a consistent manner, where there is a constant stride between all the locations being accessed. For example, when iterating across an array, memory addresses are accessed sequentially. Data prefetching is where some mechanism will attempt to predict what memory addresses will be accessed next, based on previous accesses or other information about the program. The data in the predicted addresses is pre-emptively moved into a faster cache. If the prediction is correct, the executing program will soon request the predicted data, and it will be able to access it faster because the data is waiting in a fast cache. Data prefetching can have very positive effects on performance. However, in the worst case data

```
1   //float in1[N];
2   //float in2[N];
3   float sum = 0;
4   for (signed i = 0; i < N; i++) {
5     sum += in1[i] * in2[i];
6     //__builtin_prefetch(const void* addr_to_prefetch)
7     __builtin_prefetch(&(in1[i+128]));
8   }
9
```

Figure 7-2: Using the GCC macro *__builtin_prefetch* to preform software prefetching.

prefetching can hurt performance (Cao et al. 1995). If the predicted data is not used, it will take up space in the faster caches, and can cause important data to be evicted from the cache. Data prefetching also uses memory bandwidth, which may be needed by other memory activities.

### 7.2.1 Software Prefetching in Genvolution

Prefetching can be performed by hardware, or in software. In software, this usually takes the form of special instructions that prefetch the address given as a parameter. Line 7 of figure 7-2 shows a GCC macro for adding prefetching to C and C++ code. In the example, data at index $(i + 128)$ from the *in1* array is prefetched because we know that it will be used used in a later loop iteration. Software prefetching is either added by the programmer, or by a compiler optimization (Mowry, Lam, and Gupta 1992).

Large Out-of-Order processors (such as the major Intel processors) have pieces of hardware that perform data prefetching (Chen and Baer 1995) by tracking previous memory accesses at runtime. However, many smaller processors, such as low power ARM devices, do not have (or have limited) hardware prefetchers. While adding software prefetching can be beneficial on any architecture, it usually has a greater positive effect on devices without (or with limited) hardware prefetchers (Lee, Kim, and Vuduc 2012).

Software prefetching tuning was added to Genvolution. A new AST manipulator (see section 4.8.4 for more on AST manipulators) was added. The new manipulator walks the AST and inserts *PrefetchNodes* into the AST. The *PrefetchNodes* are inserted after memory accesses. The manipulator uses a predicate function to filter what type of memory accesses (e.g. only on memory reads, only on accesses to the input) have prefetching attached to them. The parameter search space (see section 4.5) is expanded so that Genvolution will attempt to find the optimal number of bytes to jump ahead when prefetching data.

## 7.3 Other Genvolution Modifications

Other changes were also made to Genvolution. Generation of code for the $Y$, and $X$ loops of the input kernel were removed as they are not required. Code that is used to perform zero-padding is also removed, because the input kernels are guaranteed to have a height and width of one.

## 7.4 Evaluation Of Results

To evaluate the effectiveness of Genvolution at generating matrix multiplication implementations, Genvolution was used to generate matrix multiplication implementations for a set of CNN convolution input sizes for two target machines (ARM Target 1, and ARM Target 2). The CNN convolution input sizes were taken from four commonly used CNN networks: AlexNet (Krizhevsky, Sutskever, and Hinton 2012), Inception V4 (Singh and Markovitch 2017), ResNET-152 (He et al. 2016), and SqueezeNet (Iandola et al. 2016). A unique implementation was generated for each input size on each target machine. The average execution time and cache miss rate for each generated implementation was collected on each target machine. The execution time and cache miss rates are the mean average of 20 runs. The ARMCL library (see section 3.1.4) was used as the baseline library for evaluation. The ARMCL matrix multiplication implementation is referred to as ARMCL-GEMM in sections 7.4 and 7.5.

### 7.4.1 ARM Target 1

Genvolution-generated code outperforms ARMCL-GEMM for 7 of the 42 tested input sizes.

Genvolution-generated code performs best relatively in the second input of figure 7.5 (where $H$=56, $W$=56, $C$=64, $M$=16, $K$=1). Figure 7.11 shows that Genvolution-generated code and ARMCL-GEMM has similar L1 cache miss rates for this input size. However, figure 7.17 shows that Genvolution-generated code has a significantly lower L2 cache miss rate. The lower L2 cache miss rate may explain the difference in performance. Genvolution-generated code performs worst relatively in the 15th input of figure 7.5 (where $H$=14, $W$=14, $C$=512, $M$=64, $K$=1). Figures 7-11 and 7-17 show that Genvolution-generated code has higher L1 and L2 cache miss rates for this input size. The higher cache miss rates may explain the poor performance for this input size.

Figure 7-5 and 7-3(b) both suggest that there may be a relationship between the relative performance of Genvolution-generated code and the ratio between $H \times W$ and $C \times M$. As $H \times W$ shrinks and $C \times M$ grows the relative performance of Genvolution-generated code drops. This is most obvious between the second and ninth input size of figure 7-5. The best and worst performing input sizes for Genvolution-generated code also fit into this pattern. One possible explanation for this is that the baseline ARMCL-GEMM is better optimized for input matrices that have many values to sum up. The length of the rows of the first input matrix for the matrix multiplication is $C$. Therefore, as $C$ grows, more values must be multiplied and summed to produce a value in the output matrix. It is possible ARMCL-GEMM is optimized for these sorts of input matrices, and is not well optimized for long thin input matrices (for example, for the input size where Genvolution-generated code performed best, the two input matrices had sizes $16 \times 64$ and $3136 \times 64$).

However, it should be noted that the relationship between the relative performance of Genvolution-generated code and the ratio between $H \times W$ and $C \times M$ is not present in figures 7-3(a) or 7-4.

### 7.4.2 ARM Target 2

Genvolution generated code is outperformed by ARMCL-GEMM for all 42 tested input sizes. There is one input size where Genvolution generated code has a lower L1 cache miss rate, and four input sizes where Genvolution generated code has a lower L2 cache miss rate. There is no input size where Genvolution generated code has both lower L1 and L2 cache miss rates when compared to ARMCL-GEMM.

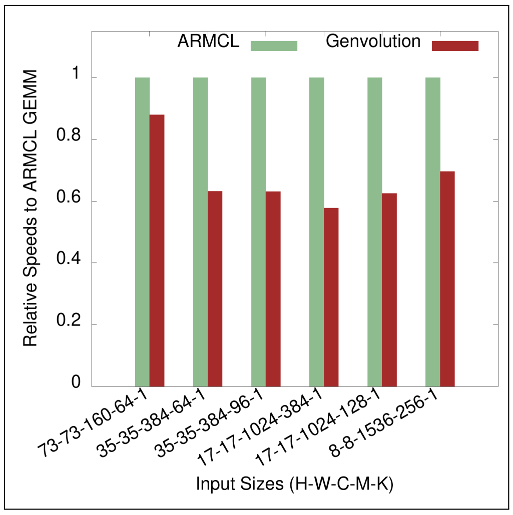
## 7.5 Conclusions

Our goal for this chapter was to see if Genvolution could be leveraged to also optimize matrix multiplication implementations. We found Genvolution had limited success in producing optimized matrix multiplication implementations automatically. However, only limited success is still important. We believe this shows that automatic code generation and optimization can create performance improvements, even in well established areas like matrix multiplications, and further research may yield stronger results.

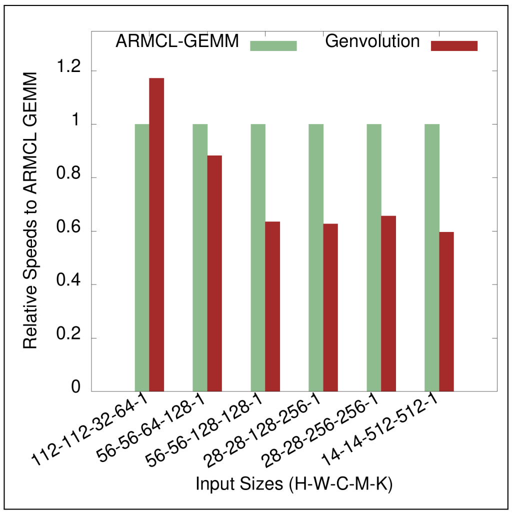
We now move from one alterative Genvolution use to another. The following chapter discusses the Flyte datatype, a quantized floating point format, and how Genvolution can be used to generate Flyte-based CNN convolution implementations to reduce CNN convolution energy usage.

## 7.6 Results

### 7.6.1 Execution Time



(a) Inception V4 matrix multiplications.



(b) AlexNet matrix multiplications.

Figure 7-3: Execution Time on Inception V4 and AlexNet matrix multiplications on ARM Target 1.

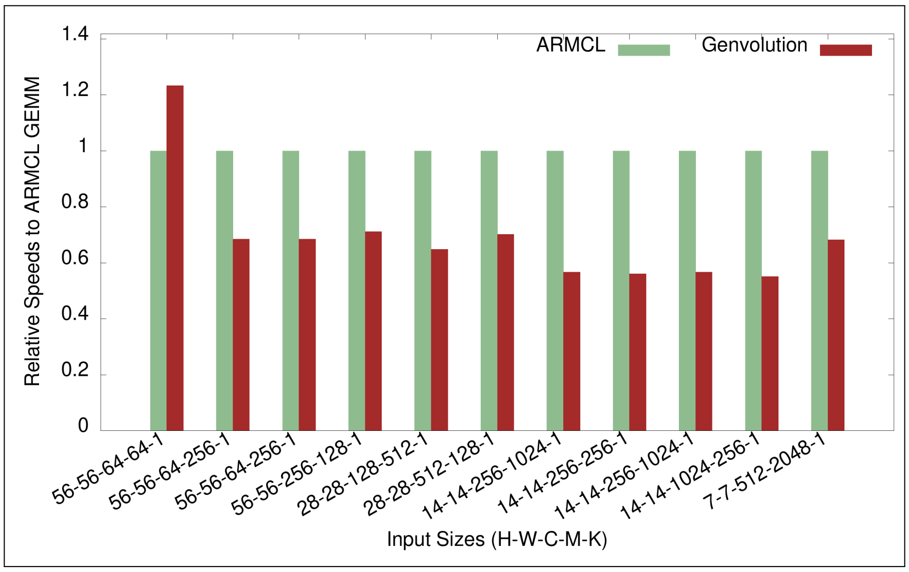


Figure 7-4: Execution Time on ResNET-152 matrix multiplications on ARM Target 1.

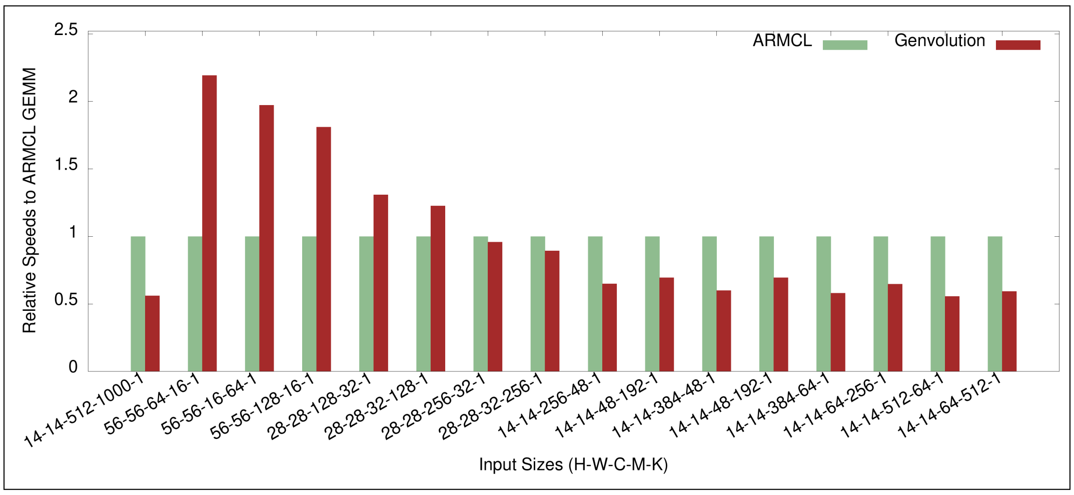
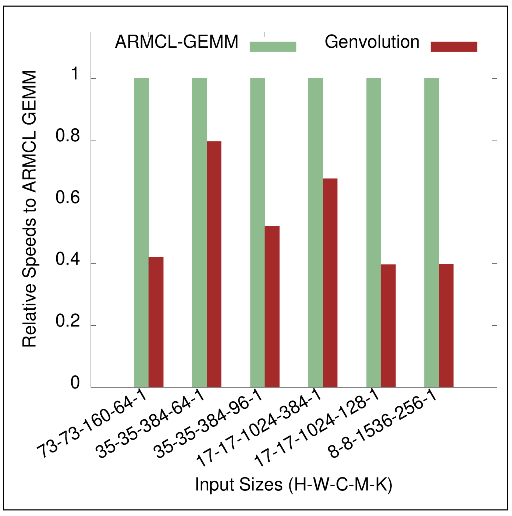
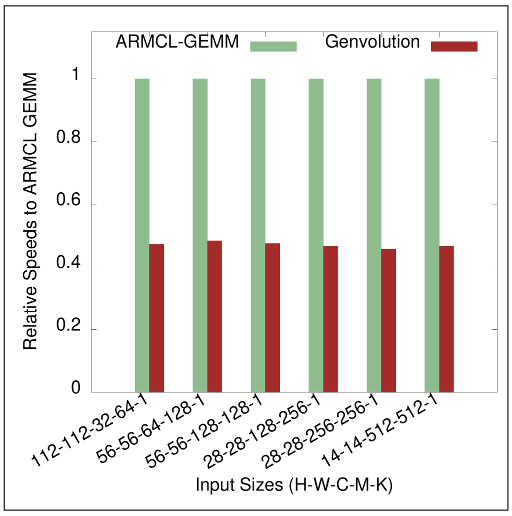


Figure 7-5: Execution Time on SqueezeNet matrix multiplications on ARM Target 1.



(a) Inception V4 matrix multiplications.



(b) AlexNet matrix multiplications.

Figure 7-6: Execution Time on Inception V4 and AlexNet matrix multiplications on ARM Target 2.

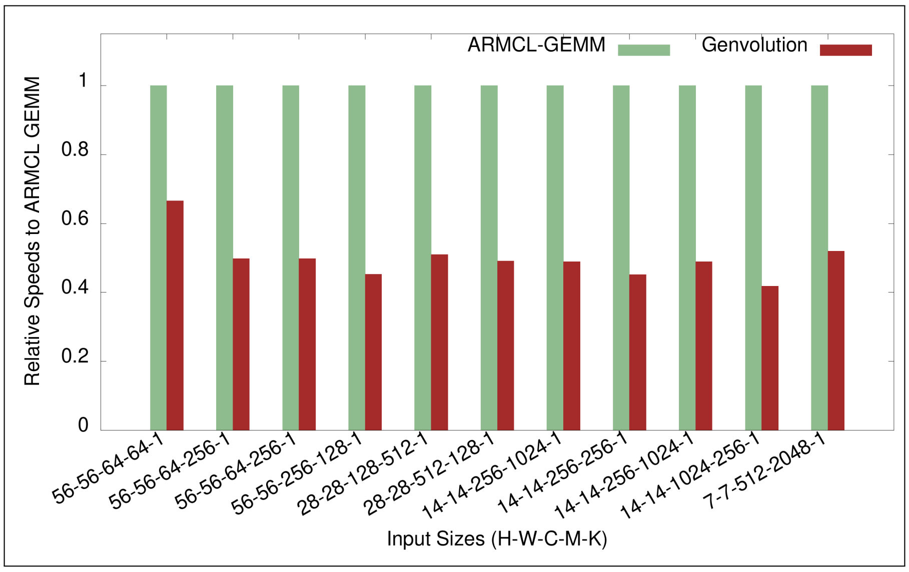


Figure 7-7: Execution Time of Genvolution implementations on ResNET 152 matrix multiplications on ARM Target 2.

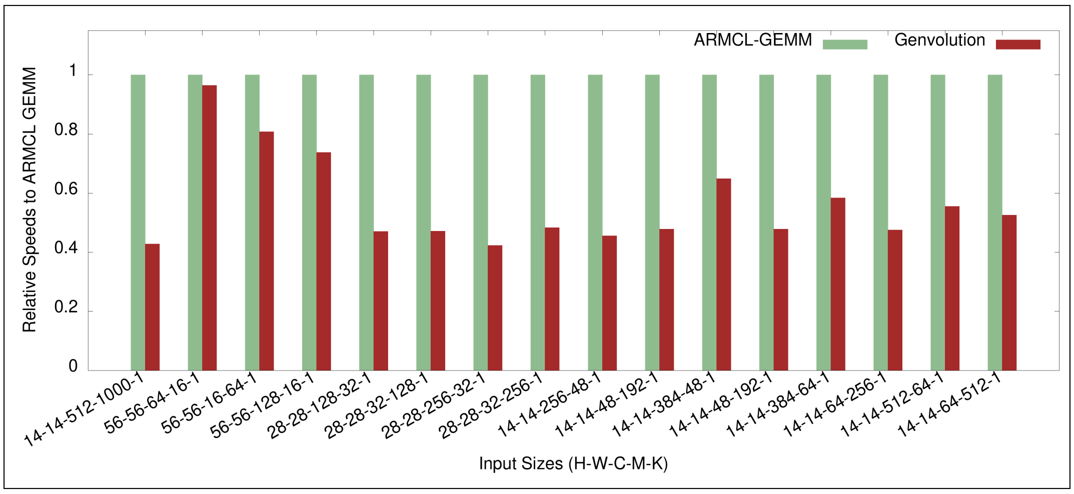
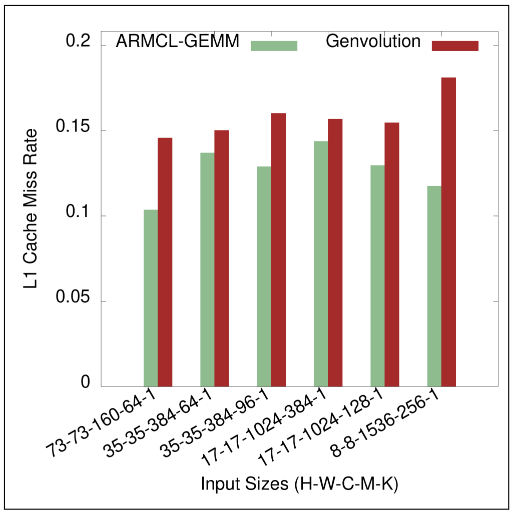
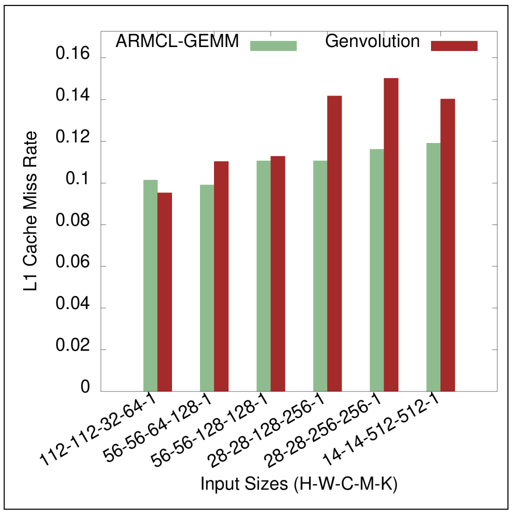


Figure 7-8: Execution Time on SqueezeNet matrix multiplications on ARM Target 2.

## L1 Cache Miss Rate



(a) Inception V4 matrix multiplications.



(b) AlexNet matrix multiplications.

Figure 7-9: L1 cache miss rates on Inception V4 and AlexNet matrix multiplications on ARM Target 1.

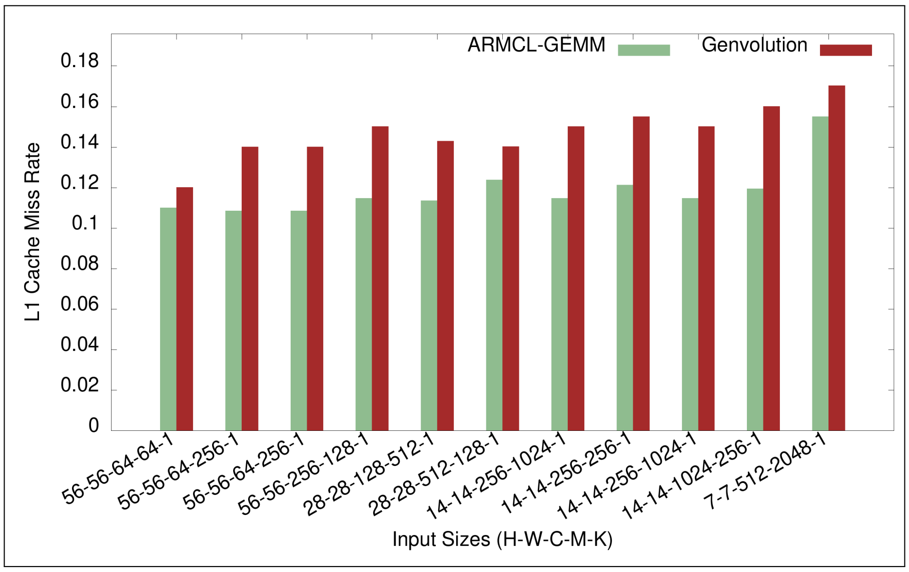


Figure 7-10: L1 cache miss rate on ResNET 152 matrix multiplications on ARM Target 1.

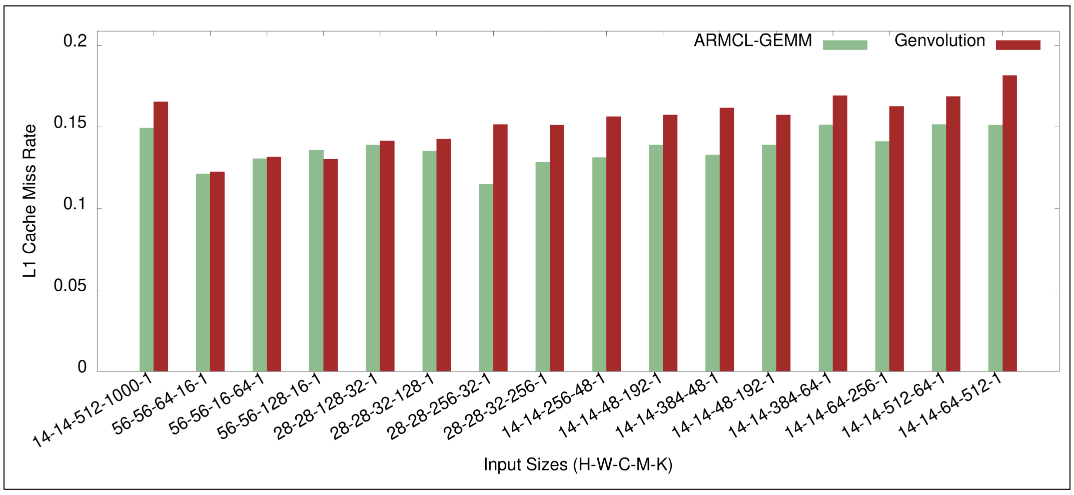
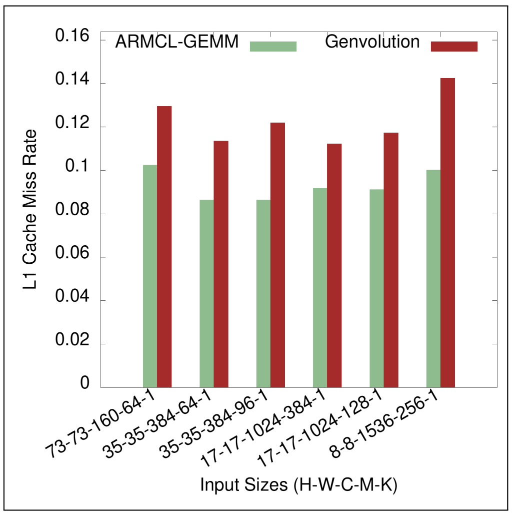
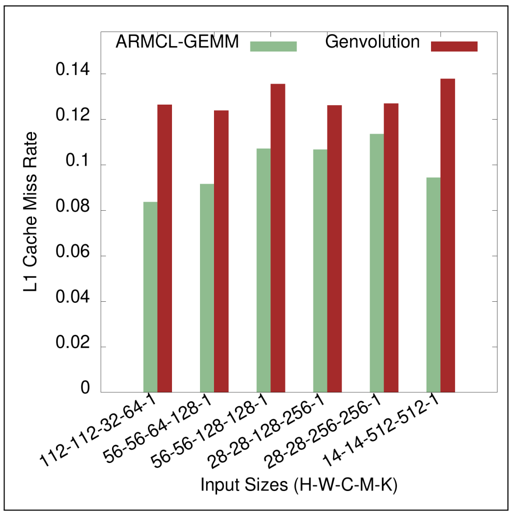


Figure 7-11: L1 cache miss rate on SqueezeNet matrix multiplications on ARM Target 1.



(a) Inception V4 matrix multiplications.



(b) AlexNet matrix multiplications.

Figure 7-12: L1 cache miss rates on Inception V4 and AlexNet matrix multiplications on ARM Target 2.

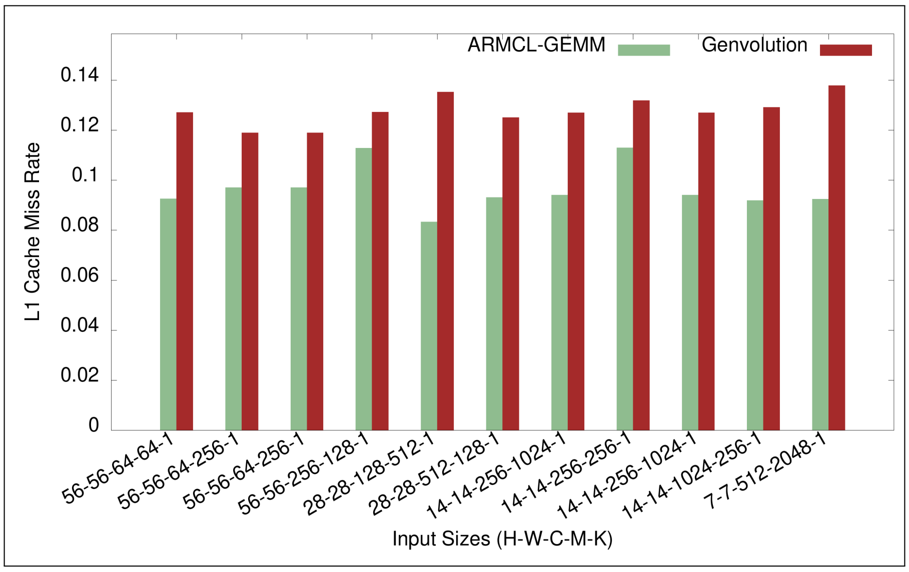


Figure 7-13: L1 cache miss rate on ResNET 152 matrix multiplications on ARM Target 2.

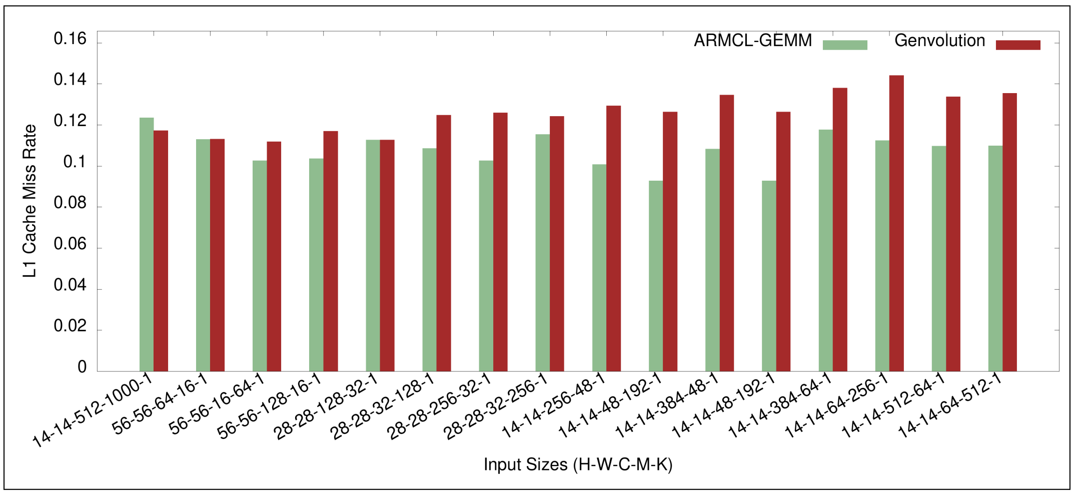
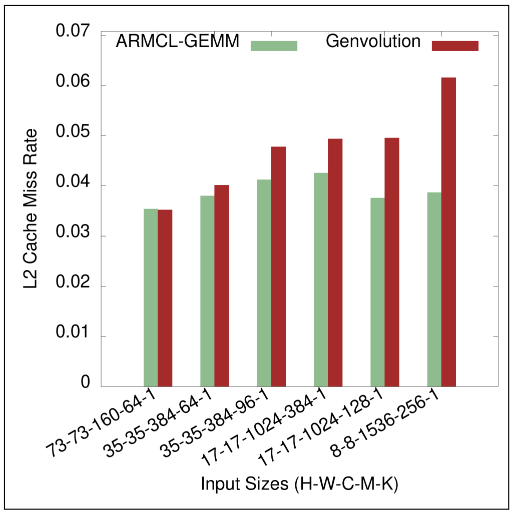
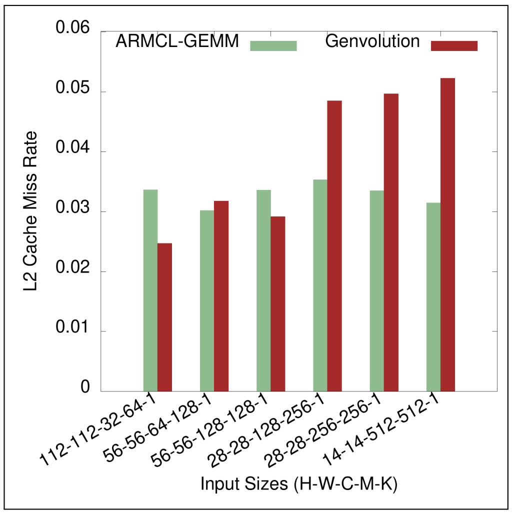


Figure 7-14: L1 cache miss rate on SqueezeNet matrix multiplications on ARM Target 2.

## L2 Cache Miss Rate



(a) Inception V4 matrix multiplications.



(b) AlexNet matrix multiplications.

Figure 7-15: L2 cache miss rates on Inception V4 and AlexNet matrix multiplications on ARM Target 1.

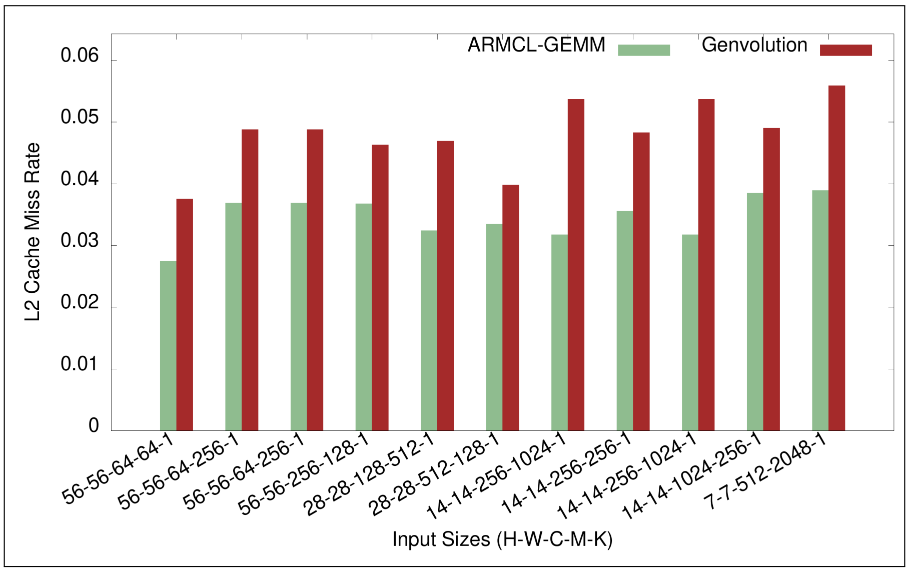


Figure 7-16: L2 cache miss rate on ResNET 152 matrix multiplications on ARM Target 1.

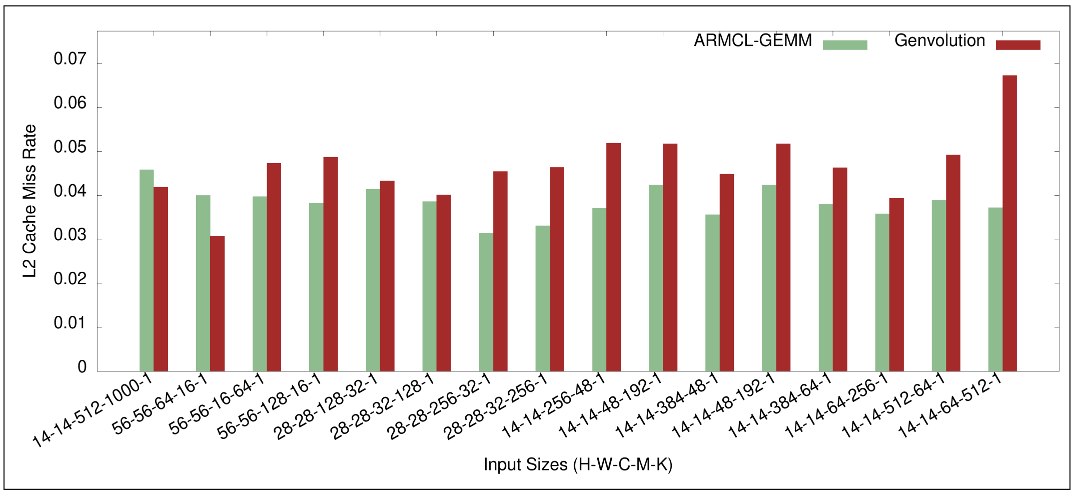
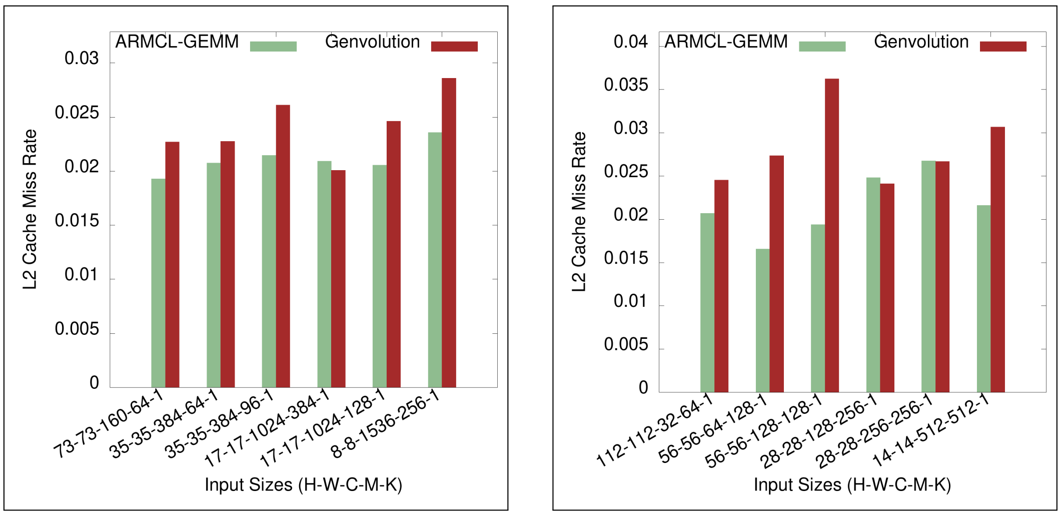


Figure 7-17: L2 cache miss rate on SqueezeNet matrix multiplications on ARM Target 1.



(a) Inception V4 matrix multiplications. (b) AlexNet matrix multiplications.

Figure 7-18: L2 cache miss rates on Inception V4 and AlexNet matrix multiplications on ARM Target 2.

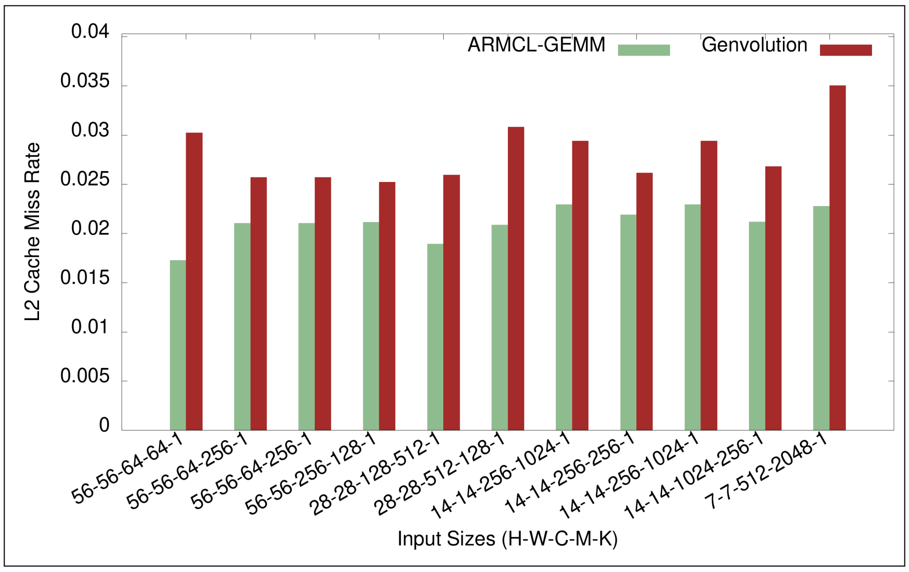


Figure 7-19: L2 cache miss rate on ResNET 152 matrix multiplications on ARM Target 2.

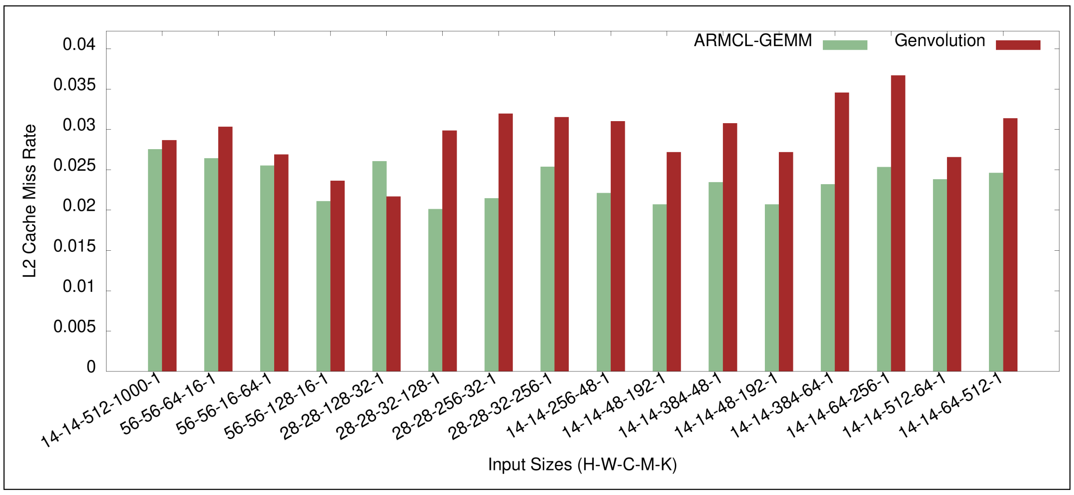


Figure 7-20: L2 cache miss rate on SqueezeNet matrix multiplications on ARM Target 2.

# Chapter 8

# Improving Energy Usage with Flyte Quantization

## 8.1 Reducing Energy Consumption With Quantization

Many CNNs use the standard floating point types (IEEE-754 single precision 32-bit *binary32* and IEEE-754 double precision 64-bit *binary64*) as the base numerical unit for computation. They are often chosen due to the fact that most general purpose processors are optimized for, and usually have hardware for computing binary32 and binary64 values. However, CNN models can often perform inference with much lower precision datatypes while still retaining accuracy. For example, Zhu et al. found that they could quantize kernel values from 32 bits to 2 bits, but only incur a 3% loss in top-1 accuracy on an AlexNet CNN model classifying the ImageNet dataset (Zhu et al. 2017).

The memory footprint of a CNN model has significant impact on the energy needed for model inference. Dally et al. found that on a typical embedded processor, 28% of the processor energy is used on supplying data, while only 6% was used on arithmetic (Dally et al. 2008). This means by quantizing the input and output tensors of a CNN convolution, the energy needed to perform the convolution is reduced, because the memory footprint of the convolution is reduced and less energy is be needed supplying data to the processor. Also, by reducing the size of tensors, a larger percentage of the tensors can be stored in the on-chip caches. Hu et al. and Han et al. both found that accessing on-chip SRAM required at most 10% of the energy needed to access data stored on off-chip DRAM using simulated 45nm processors (Hu et al. 2011) (Han et al. 2016). This means that quantizing the CNN values can lead to less energy consumption, because there will be less off-chip memory accesses.

Quantizing the data tensors to fit better in the on-chip caches will also improve data access performance, because more values can be stored in the caches at once. Arithmetic involving quantized values can also be faster as less bits must be calculated (or more values can be calculated in parallel), however this is only the case if hardware for the quantized values is available.

## 8.2 Flyte Overview

While quantizing CNN tensor values to lower precision can have multiple advantages as covered in section 8.1, if values are quantized to a datatype that does not have custom hardware support on the processors being used, then execution time performance can drop dramatically. If no hardware support is available, arithmetic using the quantized datatypes must be implemented in software, where each operation will be much slower than an equivalent hardware implementation. Quantized datatypes using software-based arithmetic will nearly always be slower than larger datatypes with hardware support. This difference in performance is a deterrent to using quantized datatypes for CNN convolution.

The flyte datatypes (**FL**oating-point multi-b**YTE**) represent a set of IEEE-754 related non-standard floating point multi-byte types. flyte are a quantized *storage datatypes* (i.e. datatypes that values are stored in, but calculations aren't performed with), that can be converted to and from binary32 and binary64 datatypes very quickly. They were proposed by Anderson et al. as a quantized datatype to reduce energy consumption by reducing a problem's memory footprint, while still leveraging the performance of pre-existing binary32 and binary64 hardware (Anderson and Gregg 2016). Anderson et al. found that the binary32 values in common BLAS routines (e.g. matrix multiplication) could be quantized to half-precision with only minimal performance loss using flyte datatypes on Intel Haswell processors (Anderson and Gregg 2016).

### 8.2.1 IEEE-754 2008 Standard Overview

The IEEE-754 2008 standard outlines a number of binary data structures for representing finite precision real numbers. The most important of which are binary32 and binary64, because they are the datatypes that have the most support in existing general purpose processors. Floating-point numbers use an exponential number format with the form: $2^{exponent} * manitissa$. For example, $21.536$ is represented as $2^4 * 1.346$. Floating-point numbers are encoded using three unsigned integer values: a sign value $sign$, an exponent value $expo$, and a mantissa value $mant$. Figure 8-1 shows the formula used for calculating the value stored in a binary32 or binary64 value when $expo$ is non-zero and $expo$ is not the maximum value it can represent (Zuras et al. 2008).

$$val = (-1)^{sign} * (1 + (\sum_{i=1}^{MANT} (mant_i * 2^{-i})) * 2^{expo-bias})$$

where
$MANT$ = number of bits used for the mantissa value
$mant_i$ = the value of the $i^{th}$ bit of $mant$
$bias = (-1 * 2^{EXPO-1}) + 1$
$EXPO$ = number of bits used to store $expo$
$val$ = the value represented by the floating-point value

Figure 8-1: IEEE-754 formula for calculating the value $val$ stored in a floating-point datatype.

The sign value $sign$ is a single bit which indicates whether the value is positive or negative. The exponent value $expo$ is used to calculate a power-of-two value which is then multiplied with the value $frac$, where $frac$ is a fixed-point fractional value between $[1, 2)$ calculated using the mantissa value $mant$. To calculate $frac$, we first assign every bit in $mant$ a numeric value equal to $2^{i-1}$ where $i$ is the position of the bit in $mant$ (with the lowest bit having a position of $i$ = 1). We then sum all represented numeric value of the bits in $mant$ which are set to one. Finally we add a one to this value to calculate $frac$.

In theory, there are multiple valid representations for the same value in the format used by binary32/64. For example, $4.0$ can be represented as $2^2 * 1.0$, or $2^1 * 2.0$. To avoid difficulties that could arise from having multiple representations for same values, all binary32/64 values are normalized. They are normalized so that the mantissa value represented by $mant$ must be in the range $[1, 2)$. As all values $mant$ can represent are in the form $1.0 + fractional$, only the $fractional$ is stored in $mant$, and the extra $1.0$ is implicitly stored. This is why an extra $1$ is added when calculating $frac$.

When $expo$ and $mant$ are set to zero, equation 8-1 is not applied, instead the value is treated as the value $\pm 0.0$, where $sign$ is used to control the sign. If only $expo$ is zero, then the value is treated as being *sub-normal*. Subnormal values are used to represent extremely small values that can't be represented using equation 8-1. Equation 8-2 is used to calculate sub-normal values. The main difference is that the implicit $1.0$ is removed from $mant$, and $mant$ now represents values in the range $[0, 1)$.

$$sval = (-1)^{sign} * (( \sum_{i=1}^{MANT} (mant_i * 2^{-i})) * 2^{-bias})$$

where
$MANT$ = number of bits used for the mantissa value
$mant_i$ = the value of the $i^{th}$ bit of $mant$
$bias = (-1 * 2^{EXPO-1}) + 1$
$EXPO$ = number of bits used to store $expo$
$sval$ = the sub-normal value represented by the floating-point value

Figure 8-2: IEEE-754 formula for calculating the sub-normal value $sval$ stored in a floating-point datatype.

When $expo$ and $mant$ are both set to their maximum possible value, the value is treated as being $\pm\infty$, where $sign$ controls the sign of the infinity. If only $expo$ is set to it's maximum possible value, then the value is assumed to store a $NaN$ value. $NaN$ values represent the results of invalid operations (such as $0/0$), or if some other error occurred during calculation. An error code and a signal to throw an exception can be stored in $mant$ when a value is $NaN$ (Zuras et al. 2008).

### 8.2.2 Accuracy Loss From Truncating Mantissa

When calculating the value of normalized binary32/binary64 values, the lowest bits of $mant$ have only a small effect on the final value. As we move down the bits of $mant$, the values they represent become quadratically smaller, as does their effect on the represented value. Removing these bits causes only a minor loss in accuracy. For example, the maximum percentage error from truncating the lowest byte from the $mant$ of a normalized binary32 value is $0.00152\%$.

If the binary32/binary64 value is in a special state, truncating the lower bits from the $mant$ can have greater effects. Removing bits from the $mant$ of a subnormal binary32/64 value can have an extremely large effect on accuracy. Removing $mant$ bits from a subnormal number can change a non-zero number to zero, which has a relative error of $1$. Removing $mant$ bits from $NaN$ values can change the error code encoded in $mant$. The highest bit of $mant$ represents if a $NaN$ value should throw an exception, therefore this information won't be lost unless the entire $mant$ is removed. The value zero is preserved correctly.

### 8.2.3 Flyte Datatype Layout

The flyte datatypes are a reduced precision *storage format*. The flyte datatypes share the same binary layouts as IEEE-754 binary32 or binary64, just with one or more of the lowest bytes removed. The layouts for the three smallest flyte datatypes are shown in figure 8-3. For example, flyte-24 has the same layout as binary32 except that the lowest byte (which stores the lowest bits of the mantissa $mant$) has been removed. The removal of the eight bits of mantissa has an effect on the accuracy of flyte-24 when compared to binary32, however as covered in section 8.2.2, the effect is small unless the binary32 value represents a sub-normal value.

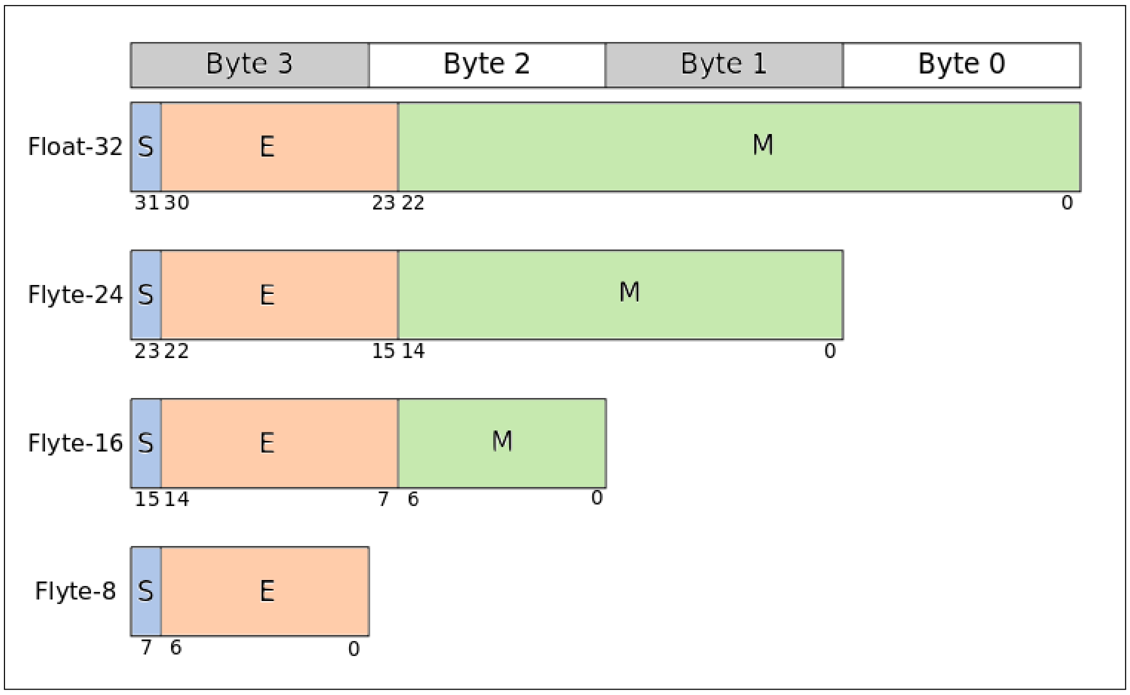


Figure 8-3: Binary layout of IEEE 754 binary32 and related Flyte datatypes.

To convert a binary32/64 value to a flyte value, the least significant bytes of the binary32/64 value are removed until the value is the correct length. Figure 8-4 shows a binary32 value (part (i) of the figure) containing the value 21.35 being transformed to a flyte-16 (part (ii)) and a flyte-8 (part(iii)) value by truncating two, and three bytes respectively. To transform a flyte datatype to a binary32/64 datatype, zero padded bytes are appended until the value is the correct length in bytes. Figure 8-4 shows a flyte-16 value (part (ii) in the figure) being transformed to a binary32 value (part (iv)) by appending two bytes. The figure also shows a flyte-8 (part (iii)) being transformed to a binary32 value (part (v)) by appending three bytes. We can see in figure 8-4 that transforming values to and from the flyte datatypes can introduce accuracy problems. Part (i) of figure 8-4 stores the value 21.35, but when transformed to and from a flyte-16 datatype (part (iv)) the value is now 21.25, a 4.7% relative error. This is because the data in the lower bytes is lost when a binary32/64 value is truncated to fit in a flyte datatype.

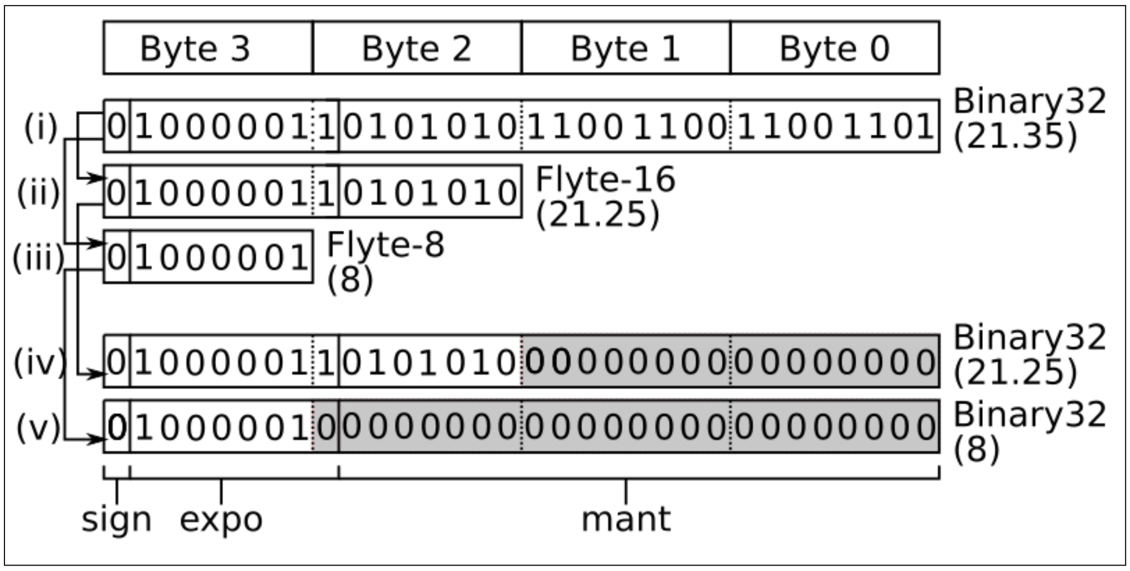


Figure 8-4: A binary32 (i) value transformed to flyte-16 and flyte-8 (parts (ii) and (iii)). the flyte values transformed to binary32 (parts (iv) and (v)).

Computations are not performed using flyte values. The flyte datatypes only work as a storage format. When a calculation is needed, flyte values are expanded to a *computation format* datatype. While in memory, values are stored in a flyte datatype. However, when a value needs to moved into a register, the end of the flyte value is padded with zero-padded bytes to expand it to a binary32/64 value. When a binary32/64 value is being stored from a register to memory, the lowest bytes of the datatype are truncated to convert it back to a flyte datatype. This allows us to use the datatypes with hardware support for calculations, and the quantized datatypes in memory.

### 8.2.4 Reducing the Transformation Overhead

The flyte datatypes are designed to minimise the cost of transforming to and from flyte datatypes and binary32/64 datatypes. However, using flyte values as a storage type still introduces an extra overhead whenever a value is being loaded or stored into memory. Anderson et al. found that this overhead has a noticeable impact on performance if the transformation is not optimized. Anderson et al. propose a method of reducing this overhead cost by designing a SIMD library for transforming multiple values to and from a flyte datatype for Intel SIMD architectures. They found that this optimization allowed flyte based solutions to match binary32 solutions for a set of BLAS operations, especially those with a large memory footprint (Anderson and Gregg 2016).

We extend their work by writing a flyte transformation SIMD library for the ARMv7 NEON and AArch64 NEON architectures. We also investigate the possibility of using conventional data blocking strategies to transform flyte values in memory to float values in chunks, with the aim of reducing the transformation overhead by reducing how many flyte-float transformations are necessary while not increasing the memory usage of the programs significantly.

## 8.3 Flyte Librarys for ARMv7 NEON and AArch64 NEON

We created two SIMD libraries for transforming flyte values. One library for the ARMv7 NEON instruction set architecture (ISA), called *NEON7-flyte*, and a library for the AArch64 NEON ISA, called *NEON64-flyte*. Both libraries are very similar, because the ARMv7 NEON ISA and the AArch64 NEON ISA are very similar. The AArch64 NEON ISA contains every ARMv7 NEON instruction, with 16 more 128-bit vector registers, more instructions that use binary64 lanes, and extending some instructions to work on 128-bit registers, rather than just 64-bit registers (ARM 2019a).

Both flyte libraries implement two basic operations. The first is loading a fixed number of flyte values from memory, transforming them to binary32/64 values, and putting them into the lanes of a vector register. The second is transforming the binary32/64 values in a vector register to flyte values, and storing the flyte values into memory. The number of values that can be stored or loaded depends on the size of the flyte datatypes (see subsection 8.3.3 for an example), and the ISA being used (see subsection 8.3.2 for an example). The two flyte libraries are written using the official ARM NEON intrinsic libraries

```
//uint8x8_t = 128-bit register split into 8 8-bit
lanes
uint8x8_t in_table = { 1,2,3,4, 5,6,7,8 };
//will be used as lookup: lanes indexed right-to-left
uint8x8_t in_lookup = { 0,2,2,1, 8,5,8,3 };
uint8x8_t out = vtbl1_u8(in_table, in_lookup);
//out == {8,6,6,7, 1,3,1,5}

```

Figure 8-5: Example table look-up operation.

to access NEON operations on the two ISA. Most intrinsic functions map to a single NEON instruction.

### 8.3.1 Table Lookup Intrinsics

Transforming between flyte datatypes and binary32/64 datatypes involves removing lower bytes, and inserting zero bytes. To perform this operation on multiple values at once, we use the NEON table lookup operations available in ARMv7 NEON and AArch64 NEON. The table lookup operations take two input vector registers. The first input vector register is treated as a table, where each lane of the vector is a table entry. Each lane of the second input vector register is a lookup key into the table. The operation uses the table vector, and the lookup vector to fill the lanes of a third register with values from the table vector. Figure 8-5 shows a code snippet using one of the available table lookup operations. *vtbl1_u8* takes in two parameters: a 128-bit register containing eight 8-bit lanes, which is treated as a table with eight byte-size entries, and a second eight lane 128-bit register, which is used to lookup entries in the created table. The lanes of the first input register are indexed right-to-left in the table, e.g. in figure 8-5, *in_table[0]* would contain 8, *in_table[1]* would contain 7, and *in_table[7]* would contain 1. *vtbl1_u8* fills a 128-bit register as an eight lane 8-bit vector using values from the table vector, and the lookup entries from the second input vector. The most important table lookup intrinsics are covered in table 8-6.

The table lookup operations also have a special action which is very useful for transforming flyte values to binary32/64 values. If an entry in the lookup

| Intrinsic | Table Vector | Lookup Vector | Output Vector | ARMv7 | AArch64 |
|---|---|---|---|---|---|
| vtbl1_u8 | uint8x8_t | uint8x8_t | uint8x8_t | Yes | Yes |
| vtbl2_u8 | uint8x8x2_t | uint8x8_t | uint8x8_t | Yes | Yes |
| vqtbl1q_u8 | uint8x16_t | uint8x16_t | uint8x16_t | No | Yes |

Figure 8-6: All byte-level table look-up intrinsics available.

vector attempts to access a non-valid entry in the table vector (e.g. the 9th lane of an eight lane vector), the value zero is returned for that table lookup. Subsections 8.3.2 and 8.3.3 cover how the table lookup operations are used in our NEON flyte libraries.

### 8.3.2 Loading Values

Both NEON flyte libraries allow multiple flyte values to be loaded from memory, transformed to binary32/64 values in parallel, and then stored in the lanes of a vector register. Figure 8-7 shows the *NEON7-flyte* function for loading two flyte-24 values from memory, and storing them in two 32-bit lanes of a 64-bit vector register.

The function in figure 8-7 assumes that the flyte values are stored in a packed format in memory. The first step is to load the bytes of wanted flyte values from memory into a vector register with 8-bit lanes. To do this, we use the NEON load operation with the smallest datatype that will load the necessary bytes. For example, in figure 8-7, we want to load 2 flyte-24 values, so we want to load six bytes from memory. As there is no operation to load 6 bytes, we use the intrinsic for loading eight contiguous bytes into a 64-bit vector register instead. The extra two bytes will be discarded when the flyte values are being transformed. Line 11 of figure 8-7 shows eight contiguous bytes from memory being loaded into *source*, a 64-bit vector register containing eight 8-bit lanes. The first six lanes of *source* contains the data for two flyte-24 values.

After performing the load operation, we will have a vector register containing the flyte values we need. However, the data will not be correctly aligned in the vector register. Part (ii) of figure 8-8 shows an example of this. If the vector register in part (ii) was treated as a two-lane 32-bit vector, then the first byte

```
 1 static inline float32x2_t neon7_load2_f24(f24 * m) {
 2   //table lookup setup
 3   constexpr uint8_t WZ = -1; //WZ => WRITE_ZERO
 4   #ifdef ARM_FLYTE_CONFIG_LITTLE_ENDIANESS
 5     uint8x8_t lookup = { 0,1,2,WZ, 3,4,5,WZ};
 6   #endif
 7   #ifdef ARM_FLYTE_CONFIG_BIG_ENDIANESS
 8     uint8x8_t lookup = { WZ,0,1,2, WZ,3,4,5};
 9   #endif
10   //expand f24 to 32 bit
11   uint8x8_t source=vld1_u8(reinterpret_cast<uint8_t*>(m));
12   uint8x8_t result=vtbl1_u8(source, lookup);
13   return reinterpret_cast<float32x2_t>(result);
14 }
15
```

Figure 8-7: *NEON7-flyte* function to load 2 flyte-24 values into 2 binary32 lanes.

of *flyte-24-B* would be in the wrong lane, and the other two bytes of *flyte-24-B* would be misaligned in the second lane. We need to insert bytes containing zero into the vector register first, before it can be treated as a two-lane 32-bit vector. To insert the bytes containing zero into the vector register, we use the table lookup operations covered in subsection 8.3.1. As mentioned in subsection 8.3.1, if the lookup vector attempts to lookup an invalid lane in the table vector, the lookup for that lane returns zero. This means we can use a table lookup operation to copy the bytes containing flyte data into a new vector register, while also inserting bytes containing zeroes as necessary. Line 12 of figure 8-7 shows a table lookup to transform two flyte-24 values. Parts (ii) and part (iii) of figure 8-8 shows the input and output of line 12 of figure 8-7.

As we are using the table lookup instructions to effectively build binary32/64 values at the byte-wise level, our NEON flyte libraries need to take into account the endianness of the architecture being used. Both ARMv7 and AArch64 can support big endian and little endian memory layouts. Depending on the endianness being used, the table lookups being used to transform flyte values to binary32/64 values must be changed. Lines 5 and 8 of figure 8-7 show different lookup vectors needed for little endianness and big endianness.

To transform the flyte values to binary32/64 values, we need to manipulate them on a byte-wise level. To do this, we use the byte-wise table lookup op-

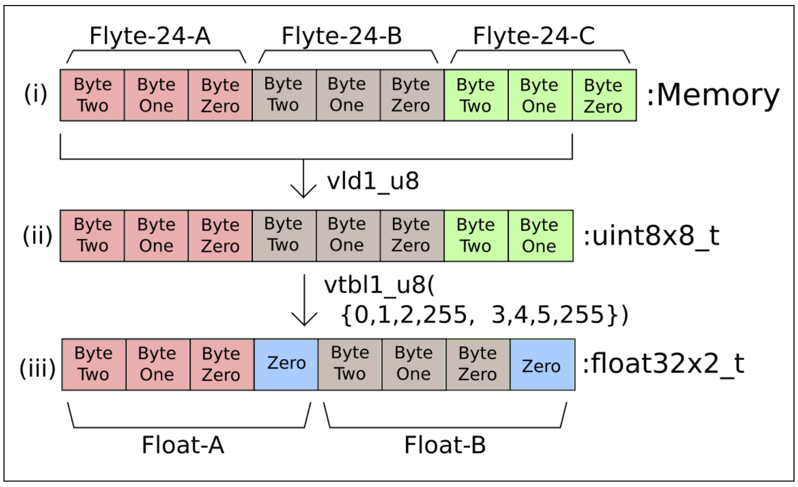


Figure 8-8: Simplified example of reading two flyte-24 values as two float-32.

erations available in NEON instruction sets. However, table 8-6 shows that all the byte-wise table lookup intrinsics available on ARMv7 NEON can only fill the lanes of a 64-bit vector register (given as a 8-bit eight lane vector register). This means we can only load two flyte values in parallel in *NEON7-flyte*, because the table lookup operations can only be used to produce two binary32 values in parallel. Also, *NEON7-flyte* can not be used to load binary64 values in parallel.

However, the AArch64 ISA includes the *vqtbl1q_u8* intrinsic, which is a byte-level table lookup intrinsic that can fill the lanes of a 128-bit vector register (given as 8-bit 16 lane vector register). This means *NEON64-flyte* can be used to load four binary32 values, or two binary64 values in parallel. Figure 8-9 shows the *NEON64-flyte* equivalent of figure 8-7. The function in figure 8-9 uses *vqtbl1q_u8* to transform 4 flyte-24 values to binary32 values in parallel.

### 8.3.3 Storing Values

Our NEON flyte libraries allow multiple binary32/64 values (stored in the lanes of a vector register) to be stored as flyte values in a packed data format. Figure 8-10 shows the *NEON7-flyte* function for storing two binary32 values (stored in a 64-bit vector register) as two flyte-24 values in memory.

The first step of the storing operation is using a byte-level table lookup operation to remove the unwanted bytes from the input vector of binary32/64

```
static inline float32x4_t neon64_load4_f24(f24 * m) {
  //table lookup setup
  constexpr uint8_t WZ = -1; //WZ => WRITE_ZERO
  #ifdef ARM_FLYTE_CONFIG_LITTLE_ENDIANESS
    uint8x16_t lookup = { 0,1,2,WZ, 3,4,5,WZ,
                          6,7,8,WZ, 9,10,11,WZ, };
  #endif
  #ifdef ARM_FLYTE_CONFIG_BIG_ENDIANESS
    uint8x16_t lookup = { WZ,0,1,2, WZ,3,4,5,
                          WZ,6,7,8, WZ,9,10,11 };
  #endif
  //expand f24 to 32 bit
  uint8x16_t source = vld1q_u8(reinterpret_cast<uint8_t*>(
   m));
  uint8x16_t result = vqtbl1q_u8(source, lookup);
  return reinterpret_cast<float32x4_t>(result);
}
```

Figure 8-9: *NEON64-flyte* function to load 4 flyte-24 values into 4 binary32 lanes.

values. The output of the table lookup operation will be a vector register containing the bytes of the flyte values packed together. If not all lanes of the output vector register are needed, the unneeded lanes are filled with zeroes. Line 12 of of figure 8-10 shows the code to transform two binary32 values to two flyte-24 values, and packing the six remaining bytes into an eight-lane vector register. Parts (i) and (ii) of figure 8-11 show the input and output vectors of line 12 of figure 8-10.

In part (ii) of figure 8-11, we can see that while the first six lanes of the vector register contain valid data, the last two lanes contain junk data. This will occur whenever the flyte values we wish to store do no fully fill a vector register. When we use a NEON SIMD store operation to write out the vector register containing flyte data, the junk data lanes will also be written out, and will overwrite some data in memory. This can be a problem if the data being overwritten is still needed. For example, if we have an array of packed flyte-24 values, and we write out the vector register in part (ii) of 8-11, two bytes of following flyte-24 value in memory will be overwritten. There are two ways to solve this problem. the first is to use multiple smaller store operations to only write out the valid parts of the vector register. For example, the vector register

```
static inline void neon7_store2_f24(float32x2_t r, f24
* m){
//table lookup setup
constexpr uint8_t WZ = -1; //WZ => WRITE_ZERO
#ifdef ARM_FLYTE_CONFIG_LITTLE_ENDIANESS
uint8x8_t lookup = { 0,1,2,4, 5,6,WZ,WZ };
#endif
#ifdef ARM_FLYTE_CONFIG_BIG_ENDIANESS
uint8x8_t lookup = { 1,2,3,5, 6,7,WZ,WZ };
#endif
//pack float values to f24
uint8x8_t source = reinterpret_cast<uint8x8_t>(r);
uint8x8_t result = vtbl1_u8(source, lookup);
//write out f24 while preserving surrounding values
#ifdef ARM_FLYTE_PRESERVE_MODE
uint8x8_t bitmask = { 0,0,0,0, 0x0,0x0,0xFF,0xFF };
uint8x8_t dst= vld1_u8(reinterpret_cast<uint8_t*>(m));
uint8x8_t mix= vorr_u8(result, vand_u8(dst, bitmask));
vst1_u8(reinterpret_cast<uint8_t*>(m), mix);
#else
//write out and corrupt following data
vst1_f32(reinterpret_cast<float32x2_t*>(m),
reinterpret_cast<float32x2_t>(result));
#endif
}

```

Figure 8-10: *NEON7-flyte* function to store 2 binary32 lanes as 2 flyte-24 values.

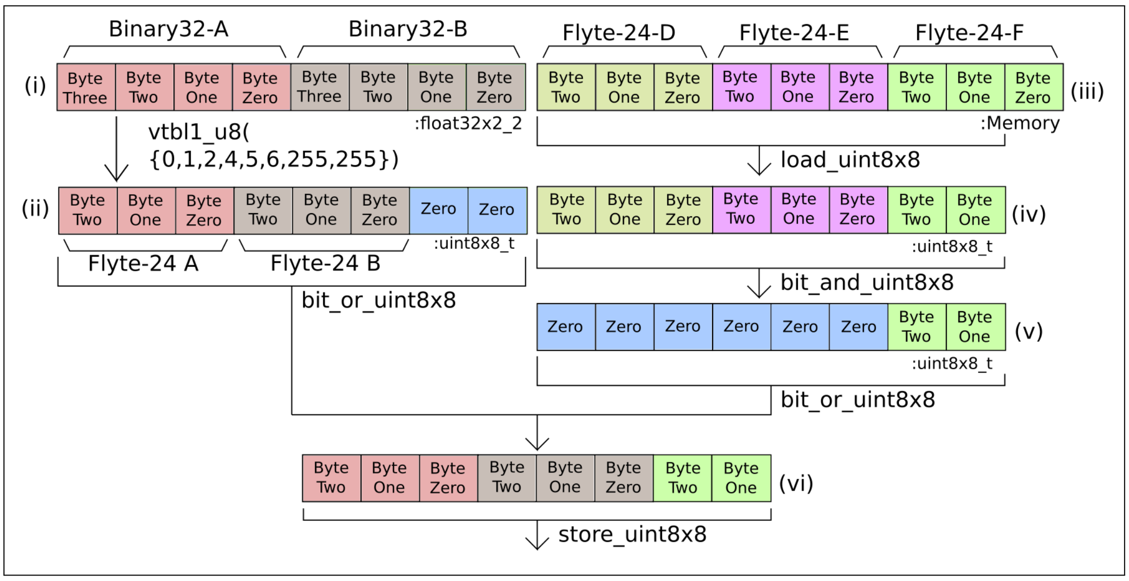


Figure 8-11: Storing two binary32 values as two flyte-24 while preserving data that would be overwritten.

in part (ii) of 8-11 can be treated as a 16-bit 4-lane vector register, and then use the *vst1q_u16* intrinsic (which stores a single lane from a 16-bit 4-lane vector register) three times to write out the valid 6 bytes of flyte-24 data only, and skip the junk data. The second approach is shown in parts (iii) through (vi) of figure 8-11. This approach involves first loading the data at location we will write the data to into a vector register (part (iv) of figure 8-11), masking out the data that should be overwritten (part(v)), then performing a bitwise OR operation with the vector register we want to store, replacing the junk data with the data already in memory (part (vi)).

However, if the data being incorrectly overwritten is also overwritten (e.g. initializing an empty array with values), no special code is needed, and the vector register can be written out normally.

As covered in subsection 8.3.2 and shown in table 8-6, the largest vector register that can be filled by a byte-level table lookup intrinsic on ARMv7 NEON is 64-bit. This means we can only transform two binary32 values to flyte-24 values in parallel, as we can only fit at most two flyte-24 values in parallel. However, we can fit four flyte-16, or eight flyte-8 values in a 64-bit vector register. Also the *vtbl2_u8* intrinsic (the second entry in table 8-6) is a byte-level table lookup function that can take an 128-bit vector register as input (represented as a uint8x8x2_t vector register matrix). A 128-bit vector register can hold four binary32 values. This means we can store four binary32 values as either flyte-16 or flyte-8 values with our *NEON7-flyte* library, but not as flyte-24 values. Figure 8-12 shows the *NEON7-flyte* function for storing four binary32 values (stored in a 128-bit vector register) as four flyte-8 values. AArch64 has access to *vqtbl1q_u8* (the third entry in table 8-6), which can output into 128-bit register, meaning we can transform four binary32 values to four flyte-24 values in parallel.

### 8.3.4 Data Alignment

SIMD vector architectures can make a distinction between aligned and unaligned memory accesses. For example, the Intel AVX architecture has separate

```
static inline void neon7_store4_f8 (float32x4_t r, f8 * m)
    {
  //table lookup setup
  constexpr uint8_t WZ = -1; //WZ => WRITE_ZERO
  #ifdef ARM_FLYTE_CONFIG_LITTLE_ENDIANESS
    uint8x8_t lookup  = { 0,2,4,6, WZ,WZ,WZ,WZ };
  #endif
  #ifdef ARM_FLYTE_CONFIG_BIG_ENDIANESS
    uint8x8_t lookup  = { 9,11,13,15, WZ,WZ,WZ,WZ };
  #endif
  //pack float values to f8
  uint8x8x2_t source = reinterpret_cast<uint8x8x2_t>(r);
  uint8x8_t result = vtbl2_u8(source, lookup);
  //write out f8 while preserving surrounding values
  vst1_lane_u32(reinterpret_cast<uint32_t*>(m),
      reinterpret_cast<uint32x2_t>(result), 0);
}

```

Figure 8-12: *NEON7-flyte* function to store 4 binary32 lanes as 4 flyte-8 values.

loading intrinsics for loading a 256-bit vector from an address that is aligned on a 32-byte boundary, and from an address that is not aligned. Using an aligned AVX load intrinsic on an unaligned memory address can cause an exception. However, an aligned intrinsic can have better performance.

When using SIMD operations with flyte datatypes, we often need to load a SIMD vector from an unaligned address. For example, the *neon7_load2_f24* function from figure 8-7 loads two flyte-24 values from memory using a 64-bit NEON vector load. If we iterated across a packed array of flyte-24 values using *neon7_load2_f24* we would increment the address we were reading from by six bytes (2 * 3 bytes per flyte-24) each iteration. As we increment the loading address by six bytes, we will quickly try to access an unaligned address (addresses for 64-bit NEON vectors are eight byte aligned). This creates an optimization problem, because aligned memory accesses can be faster, but wouldn't work with the above example.

One solution to this problem is to load flyte values in blocks that match the alignment of the vector type being used. Figure 8-13 shows an example of this. In figure 8-13, four flyte-24 values are loaded using three 32-bit vector loads (storing the results in four-lane 32-bit vector registers). By loading four flyte-24 values on every loop iteration, we would move the load address by 12 bytes (4 * 3 bytes per flyte-24) every iteration. Assuming the 32-bit vector load expects data to be aligned to four bytes, we will always perform aligned accesses with this approach. The issue is that some flyte data is spread across registers. Reordering the data between the registers can be expensive.

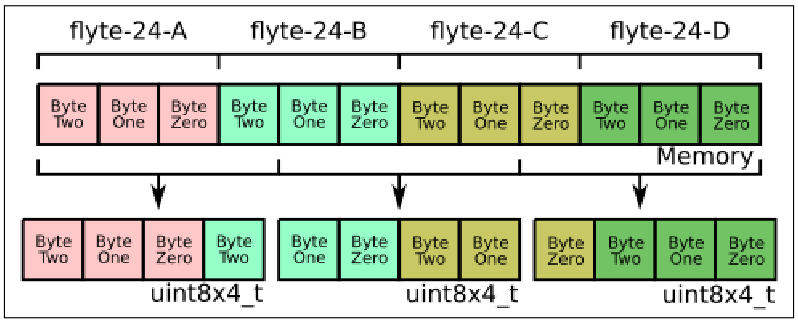


Figure 8-13: Loading four flyte-24 values using 3 32-bit vector register so that all memory accesses are aligned.

The above method of blocking loads was not implemented in our NEON flyte libraries. We found that on our test ARM boards, the cost of performing all memory operations as unaligned operations executed faster then having to reorder data between registers. Also, managing aligned and unaligned data accesses is less important on NEON architectures than on other SIMD vector architectures (like Intel AVX). Many ARM processor have near equal performance when accessing aligned and unaligned memory. For example, the performance of all memory accesses on the ARM Cortex-57 are equal, unless the access breaks a 64-byte cache line boundary.

## 8.4 Data Blocking Flyte Transformation

Anderson et al. showed that vectorizing the results of flyte values improves the performance when compared to scalar transformation code (Anderson and Gregg 2016). However, there is still the constant overhead that all values must be transformed between flyte and binary32/64 form with every use. A possible solution to this is to use a blocking strategy where small blocks of values are transformed at a time. The values inside each block would be transformed to a binary32/64 datatype, and would be used as many times as possible before a new block was transformed. This is the same concept as data blocking, except instead of blocking values so we can reuse data in fast data caches, we block values to reuse data that has been transformed to a binary32/64 datatype.

The incentives and disincentives for flyte blocking are the same as data blocking. Using large blocks of data means more opportunities to reuse transformed data. However, larger blocks may not fit entirely in the fast data caches. If part of a block ends up in the slower sections of the memory hierarchy, we lose the advantages in performance and energy consumption from using flyte datatypes.

## 8.5 Evaluation Of Results

To evaluate the effectiveness of flyte datatypes for reducing energy usage, Genvolution was used to generate direct convolution implementations for a set of CNN convolution input sizes for three different flyte datatypes: Flyte-24, Flyte-16, and Flyte-8. The CNN convolution input sizes were taken from five commonly used CNN networks: AlexNet (Krizhevsky, Sutskever, and Hinton 2012), Inception V4 (Singh and Markovitch 2017), MobileNet V2 (Howard et al. 2017), ResNET-152 (*2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016* 2016), and VGG ILSRVC (Bengio and LeCun 2015). A unique implementation was generated for each input size for each flyte datatype. The average execution time and cache miss rate for each generated implementation was collected on each target. The execution time and cache miss rates are the mean average of 20 runs. Genvolution was also used to generate direct convolution implementations that used binary32 floats only for each input size. The generated binary32 implementations were used as the baseline for evaluation (labelled as 'Float' in section 8.7).

Energy measurements were collected using current and voltage sensors present in ARM Target 1. The energy consumption of a given implementation is treated as the sum of the DRAM energy consumption and the energy consumption of the CPU that the implementation ran on.

Execution time is given as a relative speed-up against the baseline method in the graphs in section 8.7. This means that higher is better for execution time graphs. Energy usage is given as relative energy efficiency against the baseline method in the graphs in section 8.7. This means that higher is better for energy usage graphs. Cache miss rates are given as the average measured rate. A rate of 0.0 denotes no cache misses, and a miss rate of 1.0 denotes that all cache accesses were cache misses. This means that lower rates are better in cache miss rate graphs.

Flyte-8 had the best energy efficiency for 18 of the 29 input sizes. It also had the lowest execution time for 17 of the 18 input sizes where it was most energy efficient. Flyte-16 was more energy efficient than the baseline for 12 of the 29 input sizes. Flyte-24 was more energy efficient for 10 of the 29 input sizes. Flyte-8 had the lowest L1 cache miss rate for 23 of the 29 input sizes, and had the lowest L2 cache miss rate for 25 of the 29 input sizes.

We would suspect that the lower cache miss rates of Flyte-8 were due to the fact that Flyte-8 needed the least total memory to store the input and output data (only needing a byte per value). The lower memory size meant that more of the total problem could be stored in the L1 and L2 caches. Less memory needed to be loaded and stored in total for Flyte-8 implementations as well, which is not shown in the cache miss rate values. As covered in section 8.1, memory operations take up a large amount of total energy consumption, so we would expect to see Flyte-8 implementations using less energy (as happened for 18 of 29 input sizes), because Flyte-8 implementations need the fewest memory operations.

The largest energy efficiency improvement was obtained by Flyte-8 for the last input size of figure 8-16 (where $H$=14, $W$=14, $C$=512, $M$=512, $K$=3). This was also the input size with the largest execution time improvement, also by the Flyte-8 implementation. The worst relative energy efficiency was obtained by Flyte-24 for the last input size of figure 8-14(b) (where $H$=35, $W$=35, $C$=64,

$M$=96, $K$=3). The execution time of Flyte-24 on this input size was also much slower than the baseline method. Figure 8-20(b) and 8-23(b) show that the L1 and L2 cache miss rates of the baseline method and Flyte-24 were similar. Flyte-24 implementations reduce the total memory footprint by the least amount when compared to binary32 implementations (as they only reduce the required memory by 25%). It's possible that the reduction in memory operations was not large enough for the last last input size of figure 8-14(b) to cancel out the cost of transforming between Flyte-24 and binary32 for computations.

## 8.6 Conclusion

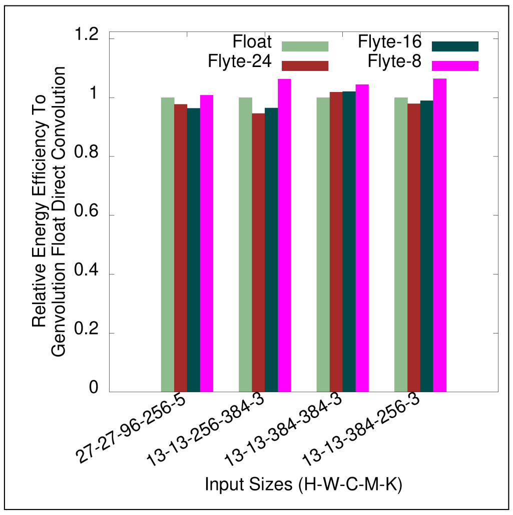
Our goal for this chapter was to reduce the energy requirement of CNN convolution without effecting execution time using Quantized Flyte datatypes. We believe we were successful in achieving our goal. Our quantized implementations lowered the energy usage for performing a CNN convolution for 18 of 29 input sizes. This was done without effecting performance in the 18 input sizes, and in 17 of the input sizes execution time was also lowered. Our Flyte-8 implementation had the best relative energy efficient for the last input size of figure 8-16 (where $H$=14, $W$=14, $C$=512, $M$=512, $K$=3) with 1.206$\times$ relative energy efficiency (i.e. 82.92% of the energy required). The average relative energy efficiency for the 18 input sizes was 1.093$\times$. However, we had hoped to see a larger average drop in energy consumption, and that we would be more energy efficient for every input size, not just some of them.

Despite this, we still think our work was significant. Reducing the energy usage of a CNN network by even 1% has a very significant impact if that network is to be run many times on an energy-constrained embedded device. We also believe our work suggests that investigating the usage of quantized datatypes and automatic code generation to reduce energy usage may have positive results.

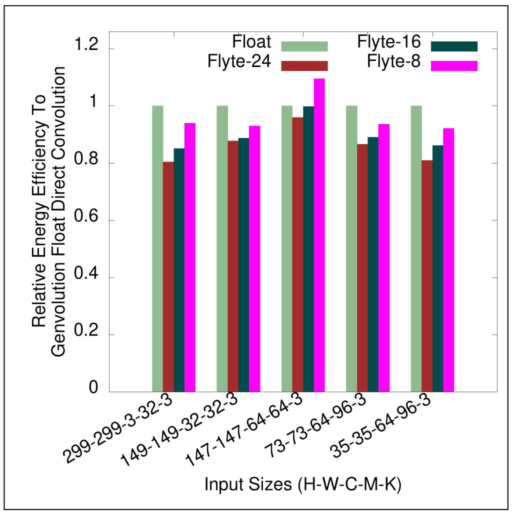
We have now completed discussion of the our research material. the following and final chapter evaluates the research discussed in it's entirety, and topics of possible future research.

## 8.7 Results

### Relative Energy Efficiency

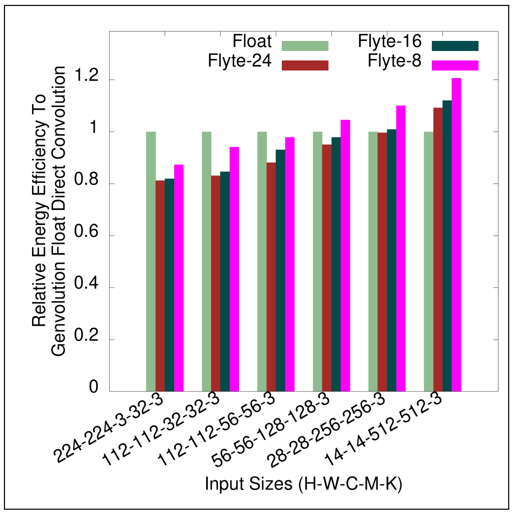


(a) AlexNet convolutions.

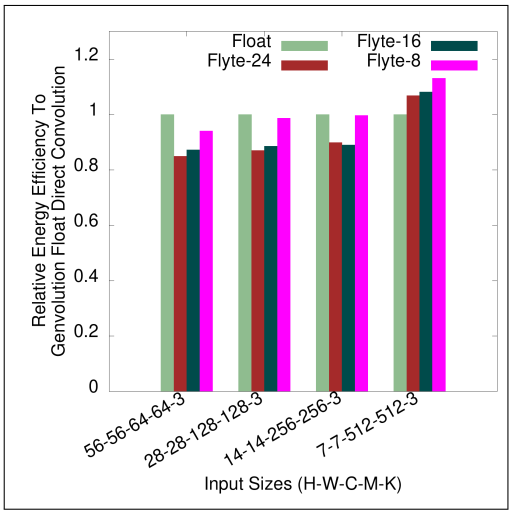


(b) Inception V4 convolutions.

Figure 8-14: Relative energy efficiency on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 8-15: Relative energy efficiency on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

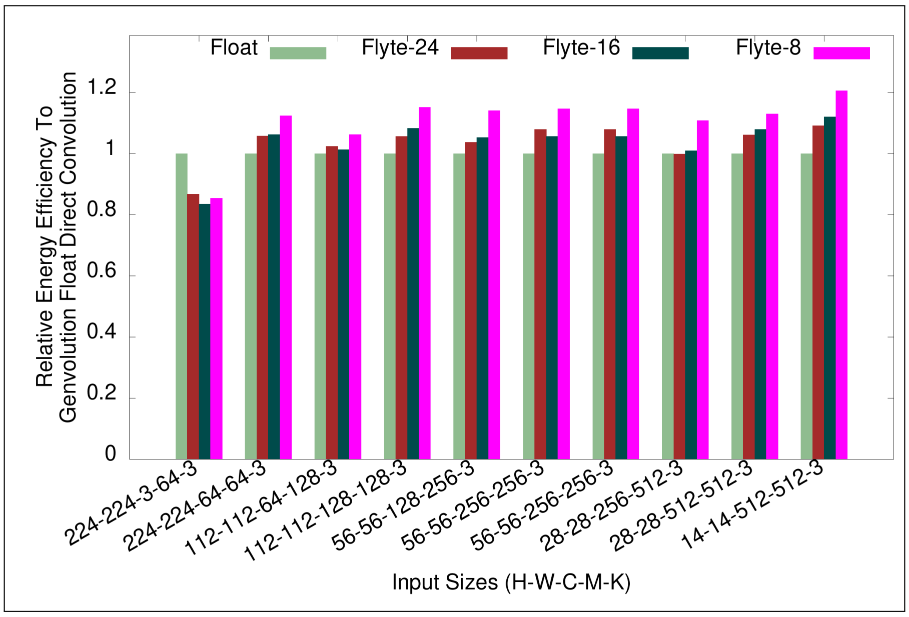
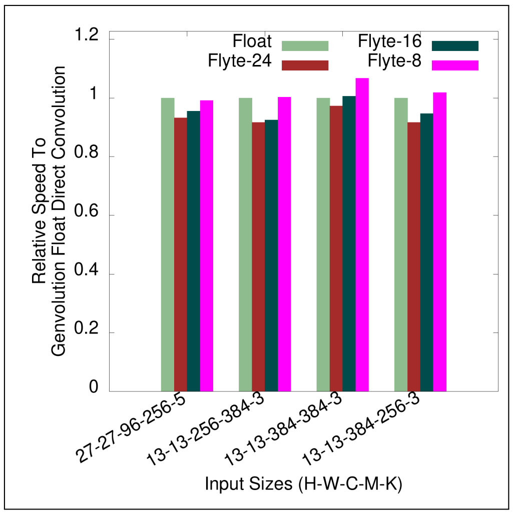
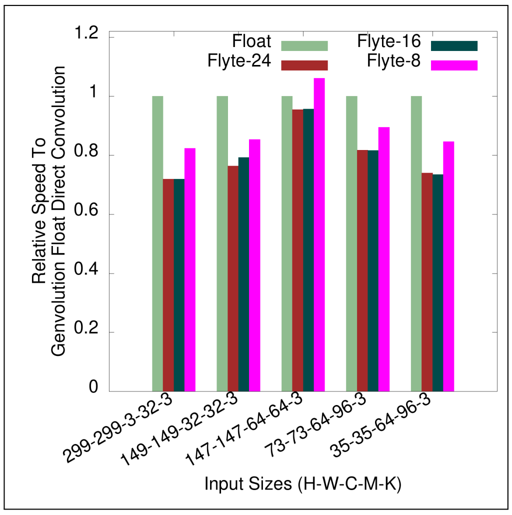


Figure 8-16: Relative energy efficiency of Flyte implementations on VGG ILSRVC Convolutions on ARM Target 1.

## Execution Time

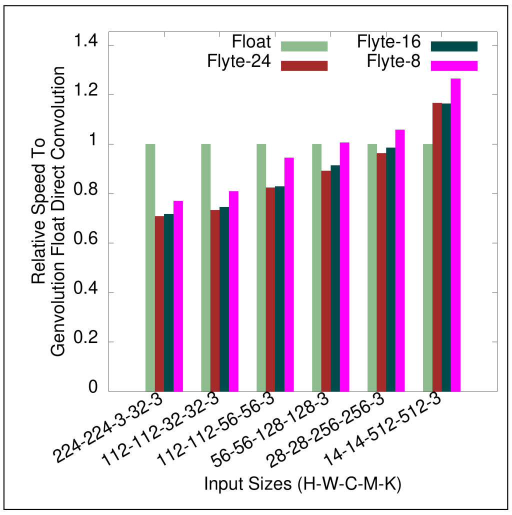


(a) AlexNet convolutions.

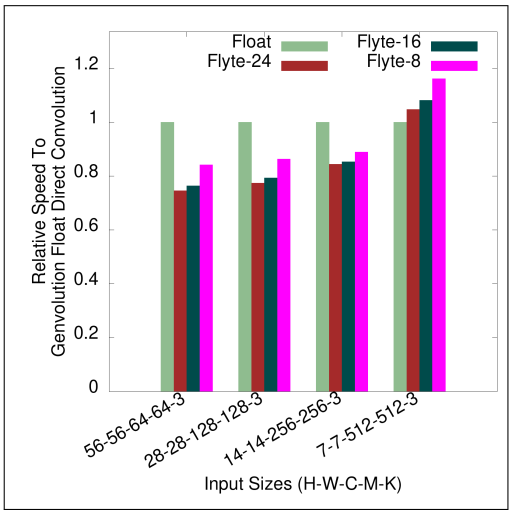


(b) Inception V4 convolutions.

Figure 8-17: Execution times on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 8-18: Execution times on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

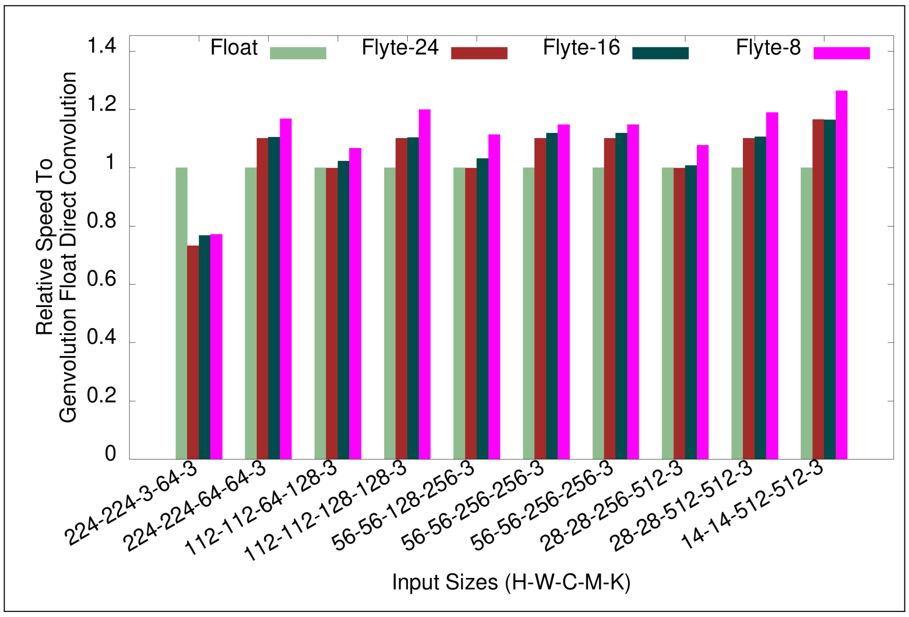
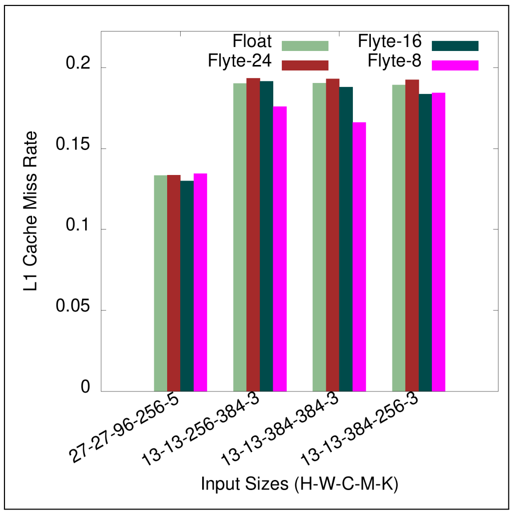
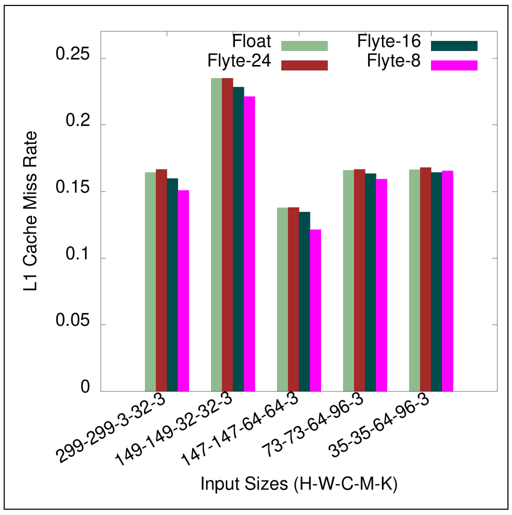


Figure 8-19: Execution Time of Flyte implementations on VGG ILSRVC Convolutions on ARM Target 1.

## L1 Cache Miss Rate

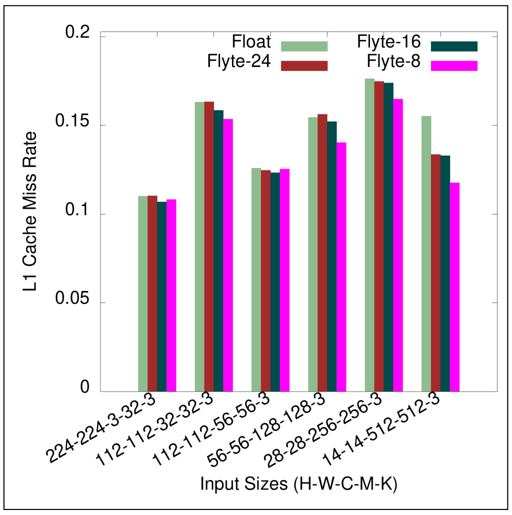


(a) AlexNet convolutions.

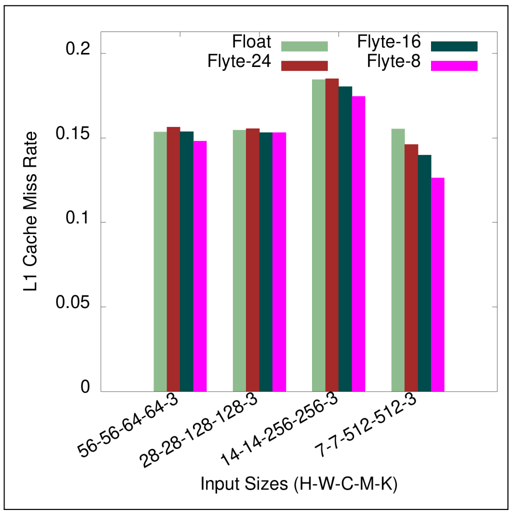


(b) Inception V4 convolutions.

Figure 8-20: L1 cache miss rate on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet V2 convolutions.



(b) ResNET-152 convolutions.

Figure 8-21: L1 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

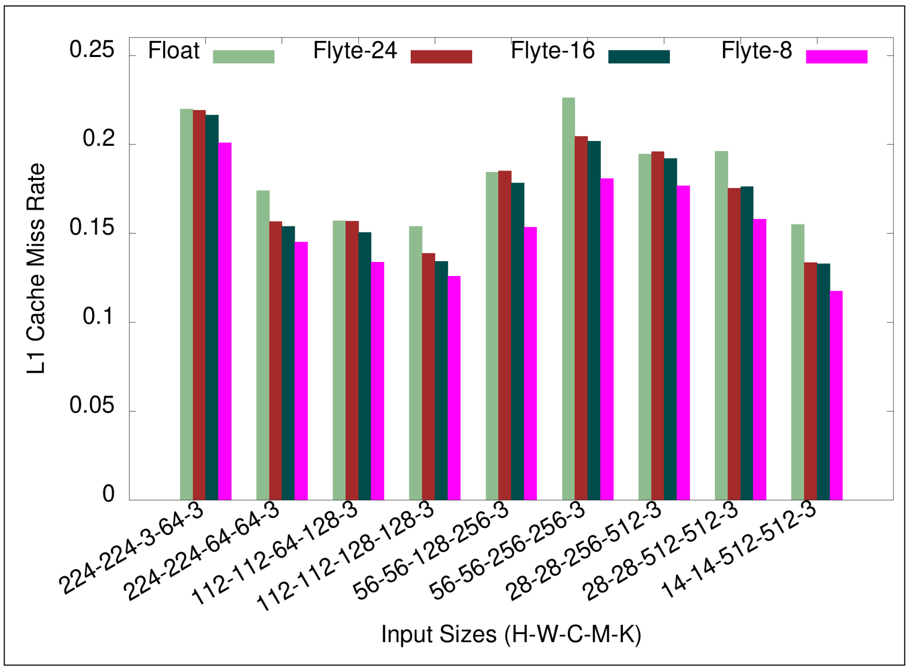
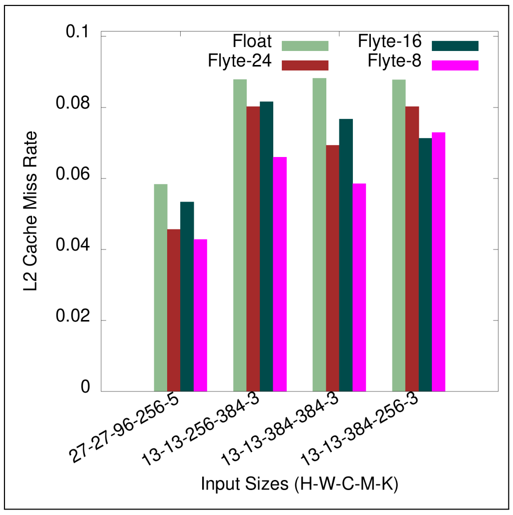
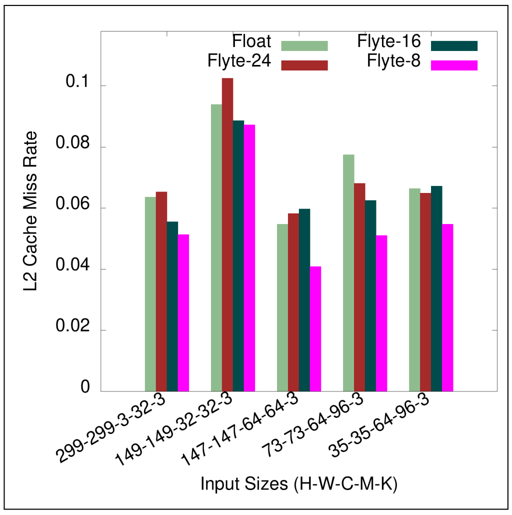


Figure 8-22: L1 cache miss rate of Flyte implementations on VGG ILSRVC Convolutions on ARM Target 1.

## L2 Cache Miss Rate

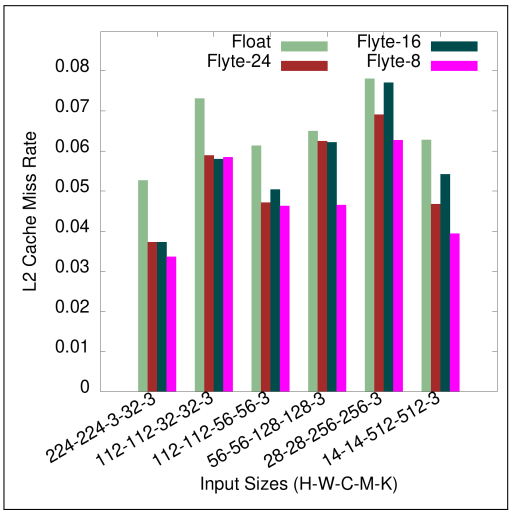


(a) AlexNet convolutions.

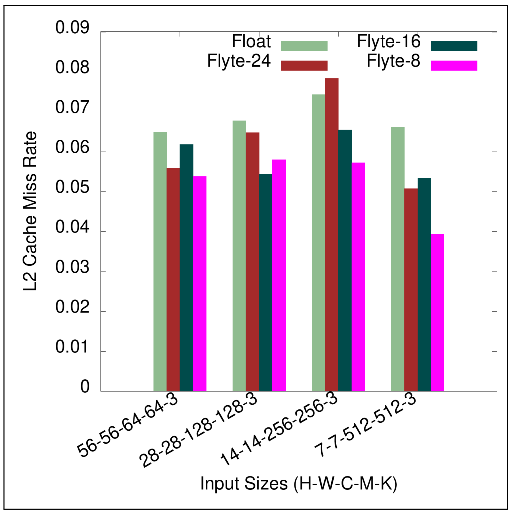


(b) Inception V4 convolutions.

Figure 8-23: L2 cache miss rate on AlexNet and Inception V4 convolutions on ARM Target 1.



(a) MobileNet convolutions.



(b) ResNET-152 convolutions.

Figure 8-24: L2 cache miss rate on MobileNet V2 and ResNET-152 convolutions on ARM Target 1.

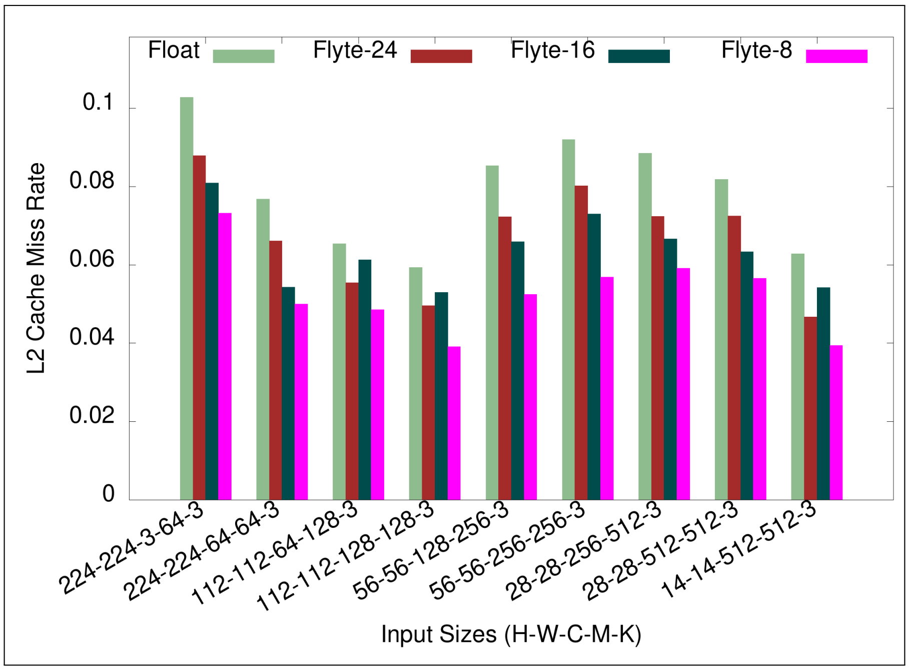


Figure 8-25: L2 cache miss rate of Flyte implementations on VGG ILSRVC Convolutions on ARM Target 1.

# Chapter 9

# Conclusion

## 9.1 Revisiting Goals

The main goal of this thesis was to investigate automatic code generation and optimization to reduce the execution time, memory allocation size, and energy usage of CNN convolution, with an emphasis on reducing resource usage on low-power ARM devices.

We believe we were successful in achieving this goal. Genvolution was successfully used to automatically generate direct convolution implementations that matched the execution time of more memory intensive Im2Col convolution implementations for 10 of 29 input sizes, and 7 of 29 input sizes on ARM Targets 1, and 2 respectively (section 4.12). Winogen was also successfully used to automatically generate Winograd convolution implementations that outperformed vendor library Winograd convolution implementations for all input sizes on ARM Targe 1 (section 6.15). To a lesser extent, Genvolution was also successfully used to generate efficient matrix multiplication algorithms (section 7.6).

We were also successful in reducing CNN convolution resource usage through other means. Our novel Winograd CNN convolution implementation (section 6.12) was able to outperform standard Winograd convolution implementations for 17 of 29 input sizes while using less temporary memory overhead. We were also able to reduce the energy usage of direct CNN convolution on ARM devices by storing the tensors in a quantized Flyte datatype (section 8.7).

We believe we have shown that automatic code generation can be used to improve the performance of CNN convolution. While we only improve resource usage for some input sizes in most cases, our results are still significant. On mass produced embedded devices, small improvements to software can lead to important gains when reproduced at scale. While extra programmer work is required to empirically test if vendor library code or our automatically generated code performs better for each CNN convolution of a given CNN network, this effort is warranted if the network is going to run many times.

We believe our work also suggests that it may be fruitful to investigate if automatic program generation can be used to improvement the performance of code in other computer science problems. We believe this is especially true in domains where there is a large amount of static information available at program generation time.

## 9.2 Future Work

### 9.2.1 Multi-Threaded Performance

As our research focused on resource constrained devices, such as embedded ARM devices, we restricted research to single-threaded performance only. We would be interested in extending the functionality of Genvolution and Winogen to include generating and optimizing multi-threaded CNN convolution implementations. Either by generating multi-threaded code directly (e.g. generating POSIX pthread code), or generating code that made use of pre-existing multi-threading frameworks, such as inserting OpenMP multi-threading pragmas.

### 9.2.2 Automatic Generation of CNN Convolution Variants

Our research focused entirely on "standard" CNN convolution, where we use a dense input tensor and a dense input kernel to produce every output point in an output tensor. However, there are many CNN convolution variants, such as strided CNN convolution, dilated CNN convolution, and depthwise CNN convolution. All these variants need separate implementations and have different memory access and performance characteristics. The number of CNN convolution variants is also increasing as more CNN research is performed. Hand-optimizing all CNN convolution variants is a very large task. Therefore, automating the generation and optimization of these variants could be very beneficial to save on programmer time.

### 9.2.3 Automatic Generation of GPU CNN Convolution

We had originally considered investigating using Genvolution to also generate GPU CNN convolution implementations. In particular, using Genvolution to generate OpenCL GPU kernel functions, because OpenCL kernels are written in subset of C++14 which Genvolution already generates. However, due to time constraints, we chose not to pursue this topic of research.

## 9.3 Final Thoughts

Computer architectures continue to become more complex. New hardware mechanisms (e.g. L0 data caches, larger SIMD registers, hardware accelerators) are continually being introduced that make producing optimal code more complex. It is becoming more difficult for human programmers to reason about optimization problems, and to correctly optimize code using all the available resources on a machine. We believe automatic and/or machine-assisted optimization will become more common in the future as computer architecture complexity grows.

# Bibliography

*2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016* (2016). IEEE Computer Society.

Abtahi, Tahmid et al. (2018). "Accelerating Convolutional Neural Network With FFT on Embedded Hardware". In: *IEEE Trans. VLSI Syst.* 26.9, pp. 1737–1749.

Anderson, Andrew (2019). *TriNNity Library*. `https://bitbucket.org/STG-TCD/trinnity/src/master/`.

Anderson, Andrew and David Gregg (2016). "Vectorization of Multibyte Floating Point Data Formats". In: *CoRR* abs/1601.07789. arXiv: `1601.07789`.

ARM (2018). *Arm A64 Instruction Set Architecture: Armv8, for Armv8-A architecture profile: SIMD and Floating-point Instructions*.

— (2019a). "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile". In: *Arm Architecture Reference Manual*.

— (2019b). *ARM Compute Library*. `https://github.com/ARM-software/ComputeLibrary`.

— (2019c). *ARMCL 19.05 Doxygen: winograd_input_transform.cl File Reference*. ARM.

Athanasaki, Evangelia, Nectarios Koziris, and Panayotis Tsanakas (2005). "A tile size selection analysis for blocked array layouts". In: *9th Annual Workshop on Interaction between Compilers and Computer Architectures, INTERACT-9 2005, San Francisco, California, USA, February 13, 2005*. IEEE Computer Society, pp. 70–80.

Barabasz, Barbara and David Gregg (2019). "Winograd Convolution for DNNs: Beyond linear polinomials". In: *CoRR* abs/1905.05233. arXiv: `1905.05233`.

Bengio, Yoshua and Yann LeCun, eds. (2015). *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Blahut, Richard E. (2010). “Fast algorithms for short convolutions”. In: *Fast Algorithms for Signal Processing*. Cambridge University Press, 145âĂŞ193.

Böhm, Christian, Martin Perdacher, and Claudia Plant (2016). “Cache-oblivious loops based on a novel space-filling curve”. In: *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*. Ed. by James Joshi et al. IEEE Computer Society, pp. 17–26.

Callahan, David, Ken Kennedy, and Allan Porterfield (1991). “Software Prefetching”. In: *ASPLOS-IV Proceedings - Forth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, USA, April 8-11, 1991*. Ed. by David A. Patterson. ACM Press, pp. 40–52.

Cao, Pei et al. (1995). “A Study of Integrated Prefetching and Caching Strategies”. In: *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, Ottawa, Canada, May 15-19, 1995*. Ed. by Blaine D. Gaither. ACM, pp. 188–197.

Chen, Tien-Fu and Jean-Loup Baer (1995). “Effective Hardware Based Data Prefetching for High-Performance Processors”. In: *IEEE Trans. Computers* 44.5, pp. 609–623.

Dally, William et al. (2008). “Efficient Embedded Computing”. In: *IEEE Computer* 7.

“High Performance Computing for Computational Science - VECPAR 2012, 10th International Conference, Kobe, Japan, July 17-20, 2012, Revised Selected Papers” (2013). In: ed. by Michel J. Daydé, Osni Marques, and Kengo Nakajima. Vol. 7851. Lecture Notes in Computer Science. Springer, p. 395.

Facebook (2018). *Pytorch QNNPack Library*. Facebook.

Flynn, Michael J. (1972). “Some Computer Organizations and Their Effectiveness”. In: *IEEE Trans. Computers* 21.9, pp. 948–960.

Fog, Agner (2018). *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical University of Denmark.

Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge (2016). "Image Style Transfer Using Convolutional Neural Networks". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Han, Song et al. (2016). "Deep compression and EIE: Efficient inference engine on compressed deep neural network". In: *2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, August 21-23, 2016*. IEEE, pp. 1–6.

He, Kaiming et al. (2016). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, pp. 770–778.

Hennessy, John L. and David A. Patterson (2012). "Computer Architecture - A Quantitative Approach, 5th Edition". In: Morgan Kaufmann, pp. 282–288.

Howard, Andrew G. et al. (2017). "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861. arXiv: `1704.04861`.

Hu, Jingtong et al. (2011). "Towards energy efficient hybrid on-chip Scratch Pad Memory with non-volatile memory". In: *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*. IEEE, pp. 746–751.

Iandola, Forrest N. et al. (2016). "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size". In: *CoRR* abs/1602.07360. arXiv: `1602.07360`.

Intel (2016). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel.

— (2018). *MKLDNN Library*. `https://github.com/intel/mkl-dnn`.

— (2019). *Intel Intrinsics Guide*. Intel.

Jacovi, Alon, Oren Sar Shalom, and Yoav Goldberg (2018). "Understanding Convolutional Neural Networks for Text Classification". In: *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2018, Brussels, Belgium, November 1, 2018*. Ed. by Tal Linzen, Grzegorz Chrupala, and Afra Alishahi. Association for Computational Linguistics, pp. 56–65.

Khan, S. et al. (2018). "A Guide to Convolutional Neural Networks for Computer Vision". In: Synthesis Lectures on Computer Vision. Morgan & Claypool Publishers, pp. 49–50.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., pp. 1097–1105.

Lam, Monica S., Edward E. Rothberg, and Michael E. Wolf (1991). "The Cache Performance and Optimizations of Blocked Algorithms". In: *ASPLOS-IV Proceedings - Forth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, USA, April 8-11, 1991*. Ed. by David A. Patterson. ACM Press, pp. 63–74.

Lavin, Andrew and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, pp. 4013–4021.

LeCun, Yann et al. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551.

Lee, Jaekyu, Hyesoon Kim, and Richard W. Vuduc (2012). "When Prefetching Works, When It Doesn't, and Why". In: *TACO* 9.1, 2:1–2:29.

Maji, Partha et al. (2019). "Efficient Winograd or Cook-Toom Convolution Kernel Implementation on Widely Used Mobile CPUs". In: *CoRR* abs/1903.01521. arXiv: `1903.01521`.

Mowry, Todd C., Monica S. Lam, and Anoop Gupta (1992). "Design and Evaluation of a Compiler Algorithm for Prefetching". In: *ASPLOS-V Proceedings - Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, USA, October 12-15, 1992*. Ed. by Barry Flahive and Richard L. Wexelblat. ACM Press, pp. 62–73.

Müller, Matthias S. et al., eds. (2010). *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Springer.

Park, Neungsoo, Bo Hong, and Viktor K. Prasanna (2003). “Tiling, Block Data Layout, and Memory Hierarchy Performance”. In: *IEEE Trans. Parallel Distrib. Syst.* 14.7, pp. 640–654.

Patterson, David A., ed. (1991). *ASPLOS-IV Proceedings - Forth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, USA, April 8-11, 1991*. ACM Press.

Rullgard, Mans (2014). *Cortex-A7 Instruction Cycle Timings*.

Sewak, M., M.R. Karim, and P. Pujari (2018). “Practical Convolutional Neural Networks: Implement advanced deep learning models using Python”. In: Packt Publishing, pp. 53–54.

Singh, Satinder P. and Shaul Markovitch, eds. (2017). *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press.

Team, GCC (2019). “GCC 4.7 Release Series Changes, New Features, and Fixes”. In:

Vasudevan, Aravind, Andrew Anderson, and David Gregg (2017). “Parallel Multi Channel convolution using General Matrix Multiplication”. In: *28th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2017, Seattle, WA, USA, July 10-12, 2017*. IEEE Computer Society, pp. 19–24.

Winograd, S. (1980). *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics.

Zee, Field G. Van and Robert A. van de Geijn (2015). “BLIS: A Framework for Rapidly Instantiating BLAS Functionality”. In: *ACM Trans. Math. Softw.* 41.3, 14:1–14:33.

Zhang, Zhang and F Gennady (2017). “Improve Intel MKL Performance for Small Problems: The Use of MKL_DIRECT_CALL”. In:

Zhu, Chenzhuo et al. (2017). “Trained Ternary Quantization”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Zuras, Dan et al. (2008). “IEEE Standard for Floating-point Arithmetic”. In: *IEEE Std 754-2008*.