Automatic Generation of Specialized Direct Convolutions for Mobile GPUs

Naums Mogers
University of Edinburgh
Edinburgh, United Kingdom
naums.mogers@ed.ac.uk

Valentin Radu
University of Edinburgh
Edinburgh, United Kingdom
vradu@inf.ed.ac.uk

Lu Li
University of Edinburgh
Edinburgh, United Kingdom
lu.li@ed.ac.uk

Jack Turner
University of Edinburgh
Edinburgh, United Kingdom
jack.turner@ed.ac.uk

Michael O'Boyle
University of Edinburgh
Edinburgh, United Kingdom
mob@inf.ed.ac.uk

Christophe Dubach
University of Edinburgh
Edinburgh, United Kingdom
christophe.dubach@ed.ac.uk

Abstract

Convolutional Neural Networks (CNNs) are a powerful and versatile tool for performing computer vision tasks in both resource constrained settings and server-side applications. Most GPU hardware vendors provide highly tuned libraries for CNNs such as Nvidia's cuDNN or ARM Compute Library. Such libraries are the basis for higher-level, commonly-used, machine-learning frameworks such as PyTorch or Caffe, abstracting them away from vendor-specific implementation details. However, writing optimized parallel code for GPUs is far from trivial. This places a significant burden on hardware-specific library writers which have to continually play catch-up with rapid hardware and network evolution.

To reduce effort and reduce time to market, new approaches are needed based on automatic code generation, rather than manual implementation. This paper describes such an approach for direct convolutions using Lift, a new data-parallel intermediate language and compiler. Lift uses a high-level intermediate language to express algorithms which are then automatically optimized using a system of rewrite-rules. Direct convolution, as opposed to the matrix multiplication approach used commonly by machine-learning frameworks, uses an order of magnitude less memory, which is critical for mobile devices. Using Lift, we show that it is possible to generate automatically code that is ×10 faster than the direct convolution while using ×3.6 less space than the GEMM-based convolution of the very specialized ARM Compute Library on the latest generation of ARM Mali GPU.

CCS Concepts • **Computer systems organization** → *Neural networks*; • **Software and its engineering** → **Source code generation**; *Parallel programming languages*;

Keywords code generation, convolution, mobile GPU, parallelism

ACM Reference Format:
Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael O'Boyle, and Christophe Dubach. 2020. Automatic Generation of Specialized Direct Convolutions for Mobile GPUs. In *General Purpose Processing Using GPU (GPGPU '20), February 23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3366428.3380771

1 Introduction

Convolutional neural networks [8] (CNN) dominate the field of computer vision and image processing. Due to the availability of parallel accelerators such as mobile GPUs, we are able to use CNNs to perform these complex tasks on resource constrained mobile devices. However, modern neural networks are computationally demanding, yielding large memory footprints and slow inference times, which has slowed their adoption in embedded settings.

CNNs typically have several convolution layers and one or more fully connected layers. Most of their execution time is spent in convolutions [15].Convolutions slide several kernels across a multi-channel 2D image (*e.g.*, the first input has typically three channels, RGB). The layers configurations vary significantly across networks and even among layers of the same network. For instance, in VGG architectures [19], the first convolutional layer operates on a 224×224 image with 3 channels while the 7th layer operates on a 112×112 image with 128 channels. The size and shape of convolutional kernels might also vary between networks or layers. This diversity in convolution input shapes represents a significant challenge for high-performance software engineers. In fact, obtaining good performance for rapidly evolving networks, hardware and workloads is a significant engineering challenge for library vendors relying on hand-coded solutions.

Most neural network libraries, such as Caffe [13] for CPU and CuDNN [5] for Nvidia GPU, solve this issue by expressing convolutions as General Matrix Multiplication (GEMM), since heavily optimized implementations are readily available. While this approach leads to high-performance, it significantly increases the required memory footprint, which can be a problem when running on mobile devices. For instance, a GEMM implementation of the 2nd convolutional layer of VGG requires 116 MB of memory for a single image while the direct convolution requires only 13 MB. If a large neural network processes multiple images (*e.g.*, a video stream) at once, the device memory is quickly filled up.

Support for high performance direct convolution is not as common given that it is a specialized operation compared to the more generic GEMM. As a result, vendors typically do not invest as much effort in providing a tuned direct convolution implementation. As an example, the ARM Compute Library implementation of direct convolution only supports a handful of convolution shapes and is actually 10× slower than its GEMM counterpart on the ARM Mali

GPGPU '20, February 23, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7025-7/20/02…$15.00
https://doi.org/10.1145/3366428.3380771

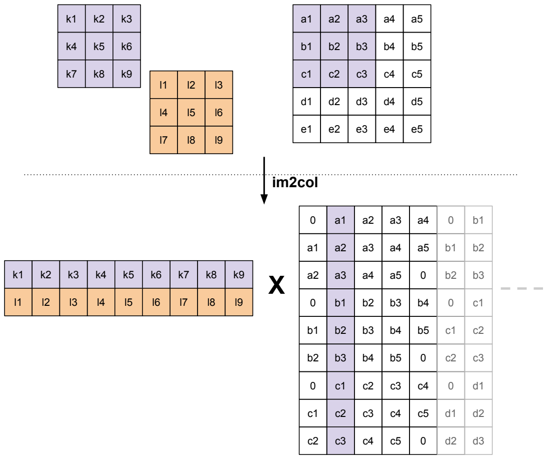

Figure 1. GEMM input preparation: im2col transformation applied to two 3×3 kernels and portion of the larger input matrix generated from the smaller original 5 × 5 input image.

GPU. This calls for an automatic approach that produces highly-specialized high-performance code for direct convolutions.

This paper presents an automatic code generation approach for direct convolution based on LIFT. LIFT expresses algorithms using a high-level data-parallel Intermediate Representation (IR). A system of rewrite rules optimizes LIFT expressions to specialize on the target architecture.

More specifically, this paper shows how CNN convolutions are expressed and optimized in LIFT. This is achieved by exploring a parametric space which includes tile sizes, amount of padding, amount of data reusage and the number of sequential operations performed by a thread. A series of constraints is automatically produced to restrict the search to valid combinations of tuning parameters (*e.g.*, input size must be divisible by tile size). Using the latest generation of ARM Mali GPU, we demonstrate that LIFT generates high-performance direct convolution code that is on average ×10 faster than the ARM compute library direct convolution implementation, while using ×3.6 less space than GEMM-based convolution provided by the same library.

To summarize, the main contributions are:

- Show how we leverage LIFT to express the convolutional layers of neural networks;
- Evaluate a large optimization space of 1,000 points with LIFT;
- Produce code automatically for direct convolution that achieves a speedup of ×10 and memory saving of ×3.6 over the ARM's own high performance library on the ARM Mali GPU

2 Motivation

2.1 Convolutional Neural Networks (CNNs)

CNNs are the tool of choice for most computer vision problems. They are composed of stacked layers of convolutions over multi-channel inputs, where each layer produces a feature map per convolution *kernel*. In computer vision, the first image passed to a convolutional neural network has three channels, red, green and blue channels. They get transformed in scale and value based on the learned kernel *weights* at each layer.

For classification tasks, the output tensor flattens each feature map into a vector and passes it to one or more affine transforms. These affine transformations account for very little of the total inference time. For example, in SENet [11], the most recent ImageNet winner, convolution accounts for 99.99% of total floating point operations. Therefore, this paper focuses primarily on the convolution operation.

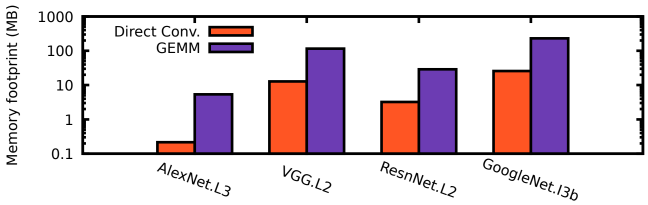

Figure 2. Runtime memory footprint of largest layers in some of the most popular deep neural networks.

2.2 Direct Convolution

Each convolution kernel has a receptive field of spatial size $(kernel_{width} \times kernel_{height})$ in 2D, usually square, $K \times K$, and a depth to match the input number of channels C, across all M kernels. On an input image size $C \times H \times W$ the direct convolution is performed with nested loops:

```
input[C][H][W]; kernels[M][K][K][C]; output[M][H][W];
for h in 1 to H do
  for w in 1 to W do
    for o in 1 to M do
      sum = 0;
      for i in 1 to K do
        for j in 1 to K do
          for c in 1 to C do
            sum += input[c][h+i][w+j]*kernels[o][i][j][c];
      output[o][w][h] = sum;
```

2.3 GEMM

The convolution operation is commonly implemented as matrix multiplication due to the availability of highly optimized GEMM routines available in libraries for both CPU (openBLAS) and GPU (CLBlas, cuDNN). This is achieved through the *image to column (im2col)* transformation, which unrolls each kernel into a row to form a matrix of all kernels, and each patch of image is mapped to a column to form another large matrix with a number of columns equal to the times each kernel should be convoluted over the image for the direct convolution approach. Matrices formed by each image channel are concatenated row-wise. The entire convolution operation is performed by executing one single dot product over these two large matrices using an efficient GEMM routine.

Figure 1 presents the im2col operation, where two 3×3 kernels are convoluted on a single channel 5×5 image. With direct convolution, the image has 25 elements and the two kernels have 9 elements each. To perform GEMM, kernels are unrolled into two rows, and through im2col the input is mapped to the input-patch matrix which is 9× larger than the original image. In total, this simple convolutional layer requires at least 9× more memory for the GEMM method than it would otherwise with the direct convolution.

2.4 Memory footprint

Figure 2 shows the actual run-time memory footprint required by the largest layer in the most popular deep neural networks. GEMM requires consistently more memory than direct convolution

```
def stencil2D(weights   : [[float]_3]_3,
              inputData : [[float]_width]_height)
              : [[float]_width-2]_height-2 = {
  mapWrg(0)(mapLcl(0)(neighborhood -> {
    join(toGlobal(id,
      reduceSeq(toPrivate(id,0.0f),
        (acc, (l,r)) -> {acc + l * r},
        zip(join(neighborhood),
            join(weights))))}),
    slide2D((3,3),(1,1),inputData))}
```

Listing 1. Example of a 2D Stencil

(one order of magnitude) due to increased memory size of the transformed input, which can be a limitation when deploying on mobile and embedded devices.

3 Background: the Lift System

The design goal of Lift is to raise the programming abstraction and enable automatic performance optimizations on massively parallel accelerators, such as GPUs. Lift provides a high level Intermediate Representation (IR) [20], and a compiler that automatically translates the high level IR to low level target code. The Lift IR is functional where operations are side-effect free, enabling composition of Lift primitives naturally. Optimizations choices are encoded using a system of rewrite rules that capture the algorithmic and hardware-specific optimizations.

3.1 Lift Abstractions

Lift includes hardware agnostic algorithmic primitives, and low-level primitives which encodes specific hardware details.

3.1.1 Algorithmic primitives

The main algorithmic primitives supported by Lift and used in this paper are listed below. These algorithmic primitives only express *what* need to be computed, shielding programmers from any hardware-specific details.

$$\begin{aligned}
\textbf{\textit{map}} &: (f : T \to U,\ in : [T]_n) \to [U]_n \\
\textbf{\textit{reduce}} &: (init : U,\ f : (U,T) \to U,\ in : [T]_n) \to [U]_1 \\
\textbf{\textit{zip}} &: (in1 : [T]_n,\ in2 : [U]_n) \to [(T,U)]_n \\
\textbf{\textit{split}} &: (m : \texttt{int},\ in : [T]_n) \to [[T]_m]_{n/m} \\
\textbf{\textit{join}} &: (in : [[T]_m]_n) \to [T]_{m \times n} \\
\textbf{\textit{slide}} &: (size : \texttt{int},\ step : \texttt{int},\ in : [T]_n) \to [[T]_{\text{size}}]_{\frac{n-\text{size}+\text{step}}{\text{step}}} \\
\textbf{\textit{pad}} &: (l : \texttt{int},\ r : \texttt{int},\ value : T,\ in : [T]_n) \to [T]_{l+n+r} \\
\textbf{\textit{transpose}} &: (in : [[T]_n]_m) \to [[T]_m]_n \\
\textbf{\textit{reorder}} &: (f : int \to int, in : [T]_n) \to [T]_n \\
\textbf{\textit{let}} &: (f : T \to U, input : T) \to U
\end{aligned}$$

$[T]_n$ denotes an array T and length n, where n is a symbolic arithmetic expression. (T,U) is a tuple whose elements are of type T and U. $T \to U$ is a function from T to U.

The map, reduce, zip are self-explanatory from their type. split creates an extra dimension in an array while join flattens a 2D array into a 1D one. slide moves a window of a fixed size across the input by a given step and produces an array of neighborhoods. pad pads an array with a fixed value, l times to the left and r times to the right. transpose simply transposes a 2D matrix; reorder permutes elements in the input array using an indexing function. An example indexing function is striddenIndex(s), which orders elements with a stride s thus mapping an element i to position $i/n + s * (i\%n)$. Finally, the let primitive bounds an input value to a scope which is used in Lift to express reusage as we will see later. The type of let is similar to that of a function application.

The Lift compiler mostly handles 1D primitives. Higher-level abstractions for multi-dimensional arrays [10] can be built by reusing 1D primitives. For instance, we can define map, slide and pad that operates on 2D arrays as follows:

```
map2D(f,input) = map(x -> map(f,x),input)
```

```
slide2D((size1,size2),(step1,step2),input) =
  map(transpose,slide(size1,step1,map(row ->
    slide(size2,step2,row), input)))
```

```
pad2D(l,r,t,b,value,input) =
  transpose(
    map(col ->
      pad(t,b,value,col),transpose(map(row ->
        pad(l,r,value,row),input))))
```

3.1.2 Hardware-specific primitives

In order to support the generation of code for parallel accelerators, Lift introduces low-level primitives that are tightly coupled with the hardware-specific programming model. We review brifely the main OpenCL primitives that are used in this paper to target a mobile GPU.

Map & reduce Lift exposes variation of the map primitive corresponding to the OpeCL programming model: mapWrg and mapLcl. These assign computation to the workgroups and local thread, respectively. These primitives take an additional parameter specifying the dimension in which to map the computation in the thread iteration space. Sequential versions of the reduction and map primitives also exist in the form of mapSeq and reduceSeq.

Vectorization Lift provides asVector and asScalar which cast scalar arrays to vector types (*e.g.*, float4) and vice versa. vectorize is provided to vectorize any scalar operator.

Address Spaces Finally, Lift expresses OpenCL address spaces using toGlobal and toPrivate, which force the enclosed function to write its results into either address spaces. Private memory usually corresponds to registers while global refers to off-chip GPU RAM accessible by all threads.

3.2 Example Stencil Program

This section reviews how Lift expresses stencil computations [10], which forms the basis for convolutions. Listing 1 shows an example

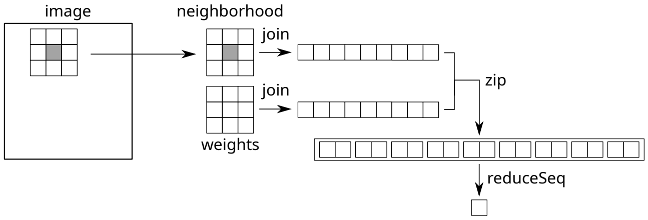

Figure 3. Visualization of the 2D Stencil Example in Listing 1

```
1  kernel stencil2D(global float * weights,
2                   global float * inData,
3                   global float * outData,
4                   int width, int height) {
5   for (int wrg_i = get_group_id(0); wrg_i<height;
6        wrg_i += get_num_groups(0)) // mapWrg
7     for (int lcl_i = get_local_id(0); lcl_i<height;
8          lcl_i += get_local_size(0)) { // mapLcl
9
10     private float acc = 0.0f;     // toPrivate(id)
11     for(int i=0; i<9; i++)        // reduceSeq
12       acc = acc + weights[i] *
13             inData[wrg_i*width + lcl_i +
14                    (i%3-1) + (i/3*width-width)]
15     out[wrg_i*width+lcl_i] = acc;}} //toGlobal(id)
```

Listing 2. Code generated with Lift for the 2D Stencil example from listing 1.

code using Lift primitives to express a stencil computation. The function stencil2D takes a 3 × 3 weight array and a 2D image array.

The body consists of two parts: the data layout arrangement and the core computation. For data layout, it first creates a 3 × 3 sliding window using slide2D in line 10 which results in a 2D neighborhood. Then, two maps are used in line 4 to schedule the work to each local thread in each workgroup running on the GPU hardware.

Each thread performs the core computation part on a neighborhood, in lines 8 to 6. This process is visualized in fig. 3. First, two joins are used to flatten the two 2D arrays: the weight array and the sliding window (*i.e.*, neighborhood), into simpler 1D arrays. Then, the two 1D arrays are zipped into a single array of tuples, and reduced sequentially to a single scalar value which is the convolution output for each single image pixel position.

3.3 Code Generation Example

Lift produces parallel OpenCL code by walking the program IR tree and emits code for each primitive. The exception to this process are the primitives that are changing the data layout, such as `join`, `split`, `zip`, `pad` or `slide`. In these cases, the compiler builds an internal representation called a view [20], which captures the effects that these primitives have on data layout. Then, when the data is accessed by other primitives, the compiler uses the information stored in the view to produce the right accesses to memory.

Listing 2 shows the code produced by the Lift compiler (with minor cosmetic changes such as naming and indentation) for the example in listing 1. First, a for loop for distributing the work among workgroup in the dimension 0 is generated on line 5 corresponding to the `mapWrg`. Then, a second loop for distributing the work among local threads is generated on line 7 corresponding to the `mapLcl`. The reduction accumulator is allocated in private memory and initialized on line 10. The reduction for loop follows, which accumulates the results of multiplying an element of the weight together with the corresponding element of the input data. Note that the array accesses are automatically generated using the information in the view built from the `slide` and `zip` primitives.

3.4 Optimization through Rewrites

Lift uses rewrite rules to encode optimization choices. This section briefly discusses two examples of such rewrites.

3.4.1 Tiling

Tiling improves locality and enables work distribution to independent groups of threads. When tiling the input data of convolutions, care must be taken to ensure that the tiles overlap. To achieve this, tiling of convolutions is achieved by simply reusing the `slide2D`. This optimization is encoded using the following rewrite rule:

```
f(slide2D((size1,size2),(step1,step2),input))
                        ⟺
map2D(f,
  slide2D((ts1,ts2),(ts1-step1,ts2-step2),
    slide2D((size1,size2),(step1,step2),input))
```

This rewrite matches a function f applied to the results of a slide2D. The function f could be performing a convolution as in the example from listing 1. In order to perform the tiling optimization, this rewrite replaces the matched expression by two level of nested `slide2D` and a `map2D` applied to f. The first `slide2D` at the bottom is the original one producing a 2D array of neighborhoods. The second one on top is the actual tiling of size $ts1 \times ts2$ which is performed by sliding the tile in 2D. The step is equal to the desired tile size minus the original step. This results in a 2D array of overlapping tiles containing 2D neighborhoods. The function f is finally mapped in 2D over each tile.

3.4.2 Vectorization

Vectorization is another example of an important optimization that highly benefits GPUs such as the ARM Mali GPU by using vector loads and stores and the built-in dot operator.

The following rewrite expresses this optimization:

```
                  map(f,input)
                       ⟺
asScalar(map(vectorize(f),(asVector(input))))
```

When a function f is mapped, it is possible to vectorize the function with the Lift `vectorize` primitive. The `asVector` cast the input scalar array into vector type while `asScalar` does the reverse.

4 Direct Convolution in Lift

We now describe how a convolutional layer is expressed in Lift and introduces the low-level optimizations applied.

4.1 High-level Lift Expression

Listing 3 shows the Lift expression of a convolution layer with three inputs. `kernelsWeights` contains the weights of all the kernels across the width, height and input channels. `kernelsBiases` are the biases, one per kernel. `inputData` contains the layer's input which is a 3D array (width × height × input channels). `padSize` is a tuple of four values that specifies how much padding is required in each direction by the layer specification. `kernelStride` specifies by how much each kernel is displaced across the input (the step). The output data is a set of feature maps represented as a 3D array with the outer dimension corresponding to the number of kernels.

This Lift program in listing 3 consists of three steps. First, data is padded with zeros as per the configuration of the layer. Then, we slide in 2D across the padded input along the two spatial dimensions (`inputWidth` and `inputHeight`) producing the sliding windows. Finally, convolution is performed using a combination of Lift primitives. First, we map over each sliding window using

```
1  def convLayer(kernelsWeights : [[[[float]_inputChannels]_kernelWidth]_kernelHeight]_numKernel, kernelsBiases :
       [float]_numKernel,
2                inputData      : [[[float]_inputChannels]_inputWidth]_inputHeight,
3                padSize        : (int,int,int,int),          kernelStride   : (int,int))
4                : [[[float]_outWidth]_outHeight]_numKernel = {
5    val paddedInput = pad2D(padSize, value = 0 ,inputData)
6    val slidingWindows = slide2D(kernelHeight, kernelWidth, kernelStride._1, kernelStride._2, paddedInput)
7     map2D(slidingWindow ->
8        map((singleKernelWeights, singleKernelBias) ->
9           reduce(init = singleKernelBias, f = (acc, (x, w)) -> {acc + x * w},
10             zip(join(join(slidingWindow)), join(join(singleKernelWeights)))),
11          zip(kernelsWeights, kernelsBiases)),
12       slidingWindows)}
```

Listing 3. High-level Lift expression of convolutional layer

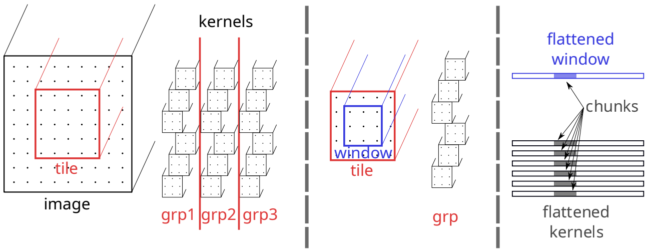

Figure 4. Visualization of the low-level Lift expression.

map2D on line 7. Then, kernelsWeights and kernelsBiases are zipped together on line 11 and mapped over on line 8. On line 9, we finally reduce over the flattened and zipped slidingWindow and singleKernelWeights. The zipping of the slidingWindow and singleKernelWeight ensures that the reduction operates on pair of corresponding elements from both arrays. The reduction operator multiplies the corresponding elements and adds to the accumulator which is initialized with a singleKernelBias.

4.2 Low-Level Lift Expression

As shown in Listing 3, convolution is expressed as a set of reductions of sliding windows. However, in popular deep CNNs such as VGG, ResNet and GoogleNet, most convolutional layers are wide to such extent that the whole input does not fit in the cache (*e.g.*, L2). We address this issue by tiling the input and splitting reduction in two steps. The first GPU kernel tiles the input, splits each sliding window of each tile into chunks and reduces each chunk to a single value. This resulting vector of values per sliding window is reduced to one final value in the second GPU kernel.

To ensure that the tiles fit perfectly with the input sizes, extra padding might be required on the input using another GPU kernel before processing the data. Conversely, an extra GPU kernel might be required at the end to crop back the output. We discuss all four stages below.

4.2.1 Padding

The padding expression has a dual purpose. First, it pads the input with zeros along all four edges as per the neural network architecture. Secondly, it zero-pads the input across the right and bottom edges so that the resulting array can be perfectly tiled. The amount of padding ρ is determined automatically by a constraint solver and is explained later.

Dimension	Size
Workgroup dim. 1	Number of tiles in the input
Workgroup dim. 0	Number of kernel groups
Thread dim. 1	Number of sliding window groups in a tile
Thread dim. 0	Number of sliding window partitions

Table 1. OpenCL dimension sizes defined in terms of tuning parameters

4.2.2 Partial Convolution

Figure 4 presents an overview of the partial convolution algorithm. Acquiring input image and a set of convolutional kernels, we split the image into tiles and kernels – in kernel groups. Each combination of a tile and kernel group is processed by a single work group. Then, a window of the spatial size kernelWidth x kernelHeight is slided across the tile. This results in a set of sliding windows, which at this point are just virtual views into data.

Each sliding window is flattened across two spatial dimensions and input channels, and split into chunks. Each chunk is processed sequentially by a single thread. Each thread can process chunks from more than one sliding window. Each kernel is split into chunks accordingly; kernels are flattened across three dimensions and split. Each sliding window chunk is coupled with corresponding chunks in each of the kernels in the group. A thread processes each pairing of the input chunk with the kernels in a kernel group.

Processing each input-kernel chunk pair involves multiplying input values and corresponding weights, and summing the resulting vector. Thus, each sliding window is reduced to a vector of values, corresponding to each chunk in the sliding window. This is partial reduction; another expression further reduces the vector to each value resulting in a full convolution of each sliding window to a single output value.

Listing 4 shows our Lift algorithm. First, the input is tiled using Slide2D and the 2D array of tiles is flattened (line 5). The tile size is controlled by the parameter θ and the stride is calculated to minimize the amount of tile overlap:

$$tilingStride = \theta - (kernelWidthHeight - kernelStride)$$

We express convolution within each tile by nesting a second slide2D on line 6. This new five-dimensional view of the input data is further transformed using the inner expression on line 8. The 3D sliding window and convolutional kernels are represented as flat vectors; this simpler data layout enables coalescing of data

```
1  def partialConv(kernelsWeights : $[[[[float]_{inputChannels}]_{kernelWidth}]_{kernelHeight}]_{numKernels}$,
2                 paddedInput    : $[[[float]_{inputChannels}]_{paddedInputWidth}]_{paddedInputHeight}$,
3                 kernelStride   : (int, int))
4                 : $[[[[[[float]]_{windowSize/\omega}]_{\sigma}]_{nWindowsInTile/\sigma}]_{\kappa}]_{numKernels/\kappa}]_{nTilesInInput}$ = {
5    val tiledInput4D = join(slide2D($\theta$, $tilingStride$, paddedInput))
6    val tiledSlidedInput5D = map(join(slide2D(($kernelHeight$, $kernelWidth$), $kernelStride$)), tiledInput4D)
7    val $windowSize = inputChannels * kernelWidth * kernelHeight$
8    def coalesceChunkVectorizeWindow(window : $[[[float]_{inputChannels}]_{kernelWidth}]_{kernelHeight}]$)
9                                           : $[[float_{v}]_{\omega}]_{windowSize/\omega}$ = {
10     val flatWindow1D = join(join(window))
11     val flatCoalescedWindow1D = reorder(striddenIndex($windowSize/\omega$), flatWindow1D)
12     val flatCoalescedChunkedWindow1D = split($\omega$, flatCoalescedWindow1D)
13     asVector($v$, flatCoalescedChunkedWindow1D) }
14   val tiledSlidedCoalescedChunkedVectorizedInput4D = map(tile4D -> split($\sigma$, map(window3D ->
15     coalesceChunkVectorizeWindow(window3D), tile4D)), tiledSlidedInput5D)
16   val groupedCoalescedChunkedVectorizedKernelsWeights4D = split($\kappa$, map(singleKernelWeights ->
17     coalesceChunkVectorizeWindow(singleKernelWeights), kernelsWeights))
18   mapWrg(1, inputTile3D ->
19     mapWrg(0, kernelsGroupWeights3D -> transpose(
20       mapLcl(1, inputWindows2D -> transpose(
21         mapLcl(0, (inputWindowsChunk1D, kernelsGroupChunk2D) ->
22           mapSeq(singleKernelReducedChunk -> toGlobal(singleKernelReducedChunk),
23             join(
24               reduceSeq(
25                 init = mapSeq(toPrivate(id(Value(0, $[float]_{\kappa}$)))),
26                 f = (acc, (inputsValue, kernelsGroupValue1D)) ->
27                   let(inputsValuePrivate ->
28                     mapSeq((accValue, singleKernelValue) ->
29                       mapSeq((inputValuePrivate) ->
30                         accValue + vectorize($v$, dot(inputValuePrivate, singleKernelValue)),
31                       inputsValuePrivate,
32                       zip(acc, kernelsGroupValue1D),
33                     mapSeq(toPrivate(vectorize($v$, id(inputValue))))),
34                 zip(transpose(inputWindowsChunk1D), transpose(kernelsGroupChunk2D))))),
35           zip(inputWindows2D, transpose(kernelsGroupWeights3D)))),
36         inputTile3D)),
37       groupedCoalescedChunkedVectorizedKernelsWeights4D),
38     tiledSlidedCoalescedChunkedVectorizedInput4D)
```

Listing 4. Low-level Lift expression example of partial convolution

accesses using `reorder`, an important GPU optimization that improves locality. The elements are virtually reordered with the stride of $windowSize/\omega$, where ω refers to the size of the partial window processed by one thread. The resulting stride is the number of threads processing the same window, ensuring each thread access consecutive elements. The window is vectorised with vector length v which is important for the Mali GPU. Finally, windows are split in groups; each thread will process chunks from the whole group of sliding windows.

Lines 18-21 express mapping of parallel computations onto OpenCL threads; for the sizes of the respective work group dimensions, see 1. In dimension 1, each work group processes one input tile; in dimension 0, each work group is assigned one group of convolutional kernels. The grouping of kernels is expressed on line 16; the size of a kernel group is controlled by the parameter κ.

In local dimension 1, threads are assigned an input window group. In local dimension 0, threads are assigned a chunk of each input window in a window group and a set of corresponding chunks of a group of kernels. By reading the input window chunk only once and reusing it for κ kernels within the same thread, we reduce the number of reads by a factor of κ; by reading the kernels once and reusing them for σ sliding windows within the thread, we further reduce the number of reads by a factor of σ. By iterating across the fastest changing dimension 0 in the innermost loop, we ensure that the quad threads access consecutive window chunks; thanks to the prior coalescing now stored in the view, quad threads access consecutive locations in memory further reducing the number of reads by a factor of four.

The reduction of the partial window across several kernels is expressed on line 24: the accumulator is initialised to a vector of κ zeros on line 25 and the input to `reduceSeq` on line 34 is an array of tuples of partial window elements and corresponding elements from kernel weights.

The `let` primitive on line 27 ensures that the input values are fetched into the private memory once on line 33 and are reused across iterations of the sequential loop on line 28.

4.2.3 Summing partial results

The third expression completes the convolution by reducing the partial weighted sums of each window. Each work group processes a single tile for a single kernel group; each thread reduces one or more sliding windows in one output channel.

4.2.4 Cropping

The final expression reverses the effect of the extra padding performed in the first expression. It crops the output using `pad2D` with negative values for padding sizes. The amount of horizontal and vertical cropping is calculated as:

$$cropSize = \frac{\rho}{kernelStride}$$

The *cropSize* is guaranteed to be whole by the `slide` constraint discussed later in section 5.2.

5 Space Exploration

When exploring the search space of possible implementation, we leverage rich algorithmic information captured by the Lift IR. Type safety and provable correctness of rewrite rules allow to automatically explore structural code transformations that would otherwise require costly static analysis.

Lift supports symbolic parameter values into the types. Parameter tuning consists of finding valid combinations of tuning values, replacing them at the type level and generating a specialized implementation. This leads to GPU kernels that are specialized for the given input parameters and tuning values.

5.1 Tuning parameters and rewrite rules

Table 2 shows the tuning parameters.

Input tiling Splitting the input optimizes cache locality by ensuring that adjacent threads process the same neighborhood. The tile size is explored in the range from kernel size to double the padded input size.

Padding Changing the input size solves the problem of finding an efficient tile size that both splits the input evenly and can be evenly split by the convolutional kernels. Though time might be wasted on processing dummy data, we can achieve better data alignment and cache locality.

Kernel and sliding window grouping Processing multiple kernels and sliding windows per thread results in data re-usage: input data is fetched once into private memory and is reused during output channel computation; same for the weight coefficients. The benefit of increased re-usage is a tradeoff since large values of κ increase register pressure.

Sliding window chunking Each sliding window and kernels are flattened and split into chunks, processed sequentially within threads. Smaller values for chunk size result in more parallel operations. Varying the amount of sequential work allows to explore work group sizes which influences register consumption and maximum occupancy of the compute cores.

Vectorization The parameter space includes scalar and vector operations, which is achieved by automatically rewriting the expression to use `asVector` and `vectorise` functions. Vectorizing data loads reduces memory access times and allows Lift to use the `dot` builtin function tor compute dot-product in optimized hardware units.

Coalescing Data accesses can be coalesced through rewriting so that adjacent threads access consecutive scalar or vector values in memory. Coalescing results in batch reads and cache line reusage.

Symbol	Parameter
θ	Input tile size
ρ	Optimizational padding size
κ	Number of kernels per workgroup
σ	Number of sliding windows per thread
ω	Sequentially processed input elements
υ	Vector size

Table 2. Convolution expression tuning parameters

Unrolling During rewriting, Lift optionally unrolls the innermost reduction of the partial convolution. The compiler also removes the loops over work group or work item indices where the corresponding dimensions sizes are the same as the number of elements being processed.

5.2 Constraint inference

The expressiveness of Lift and the complex space produced by rewriting results in a high number of dependent and independent parameters which is hard to manual analysis. To address the problem of parameter validation, we used automatic constraint inference based on the information encoded in the IR and the type system. By traversing the AST, we collect variables from types and parameters, and infer continuous and discrete constraints on the parameter values. A constraint is expressed as a record specifying the condition that must hold true and the list of parameters the condition is imposed upon. We present examples of the constraints that are automatically derived from a Lift expression.

Algorithmic Algorithmic constraints are inferred based on the type of an IR primitive and the values of its parameters. Satisfying such constraints is required for producing semantically correct results. For the `split` primitive, the inferred constraint is as follows:

$$\boldsymbol{split} : (m : \texttt{int},\ in : [T]_n) \Rightarrow n \,\%\, m = 0$$

This constraint ensures that the `split` input is divisible evenly into chunks of m elements. The compiler traverses the arithmethic expression of the condition $n\%m = 0$ and collects all the parameters; they are marked as co-dependent.

`asVector` imposes a similar constraint to that of `split`:

$$\boldsymbol{asVector} : (m : \texttt{int},\ in : [T]_n) \Rightarrow n \,\%\, m = 0$$

`slide` comes in two conceptual flavours based on the constraints it imposes on the variables. The `slideStrict` requires that the sliding window covers perfectly the input:

$$\boldsymbol{slideStrict} : (sze : \texttt{int},\ step : \texttt{int},\ in : [T]_n)$$
$$\Rightarrow \frac{(n - sze)}{step} + 1 = 0$$

`slideStrict` must be used for tiling, when the semantic correctness of the expression must be preserved for all parameter values. For kernel sliding, we use the normal `slide` since sliding is allowed to produce partial results; a notable example is the first layer of AlexNet [14].

Hardware The specifications of the target hardware impose the constraints on the maximum amount of threads in a single dimension, work group size, total memory allocated and maximum single

Layer	Input	Conv	ARM Direct	ARM GEMM	Lift
0	3x224x224	64x3x3	38.61	2.98	9.09
2	64x224x224	64x3x3	852.03	80.14	77.08
5	64x112x112	128x3x3	426.22	37.94	40.65
7	128x112x112	128x3x3	906.66	88.09	69.60
10	128x56x56	256x3x3	452.48	23.73	58.90
12 14	256x56x56	256x3x3	975.69	60.45	84.75
17	256x28x28	512x3x3	546.63	22.30	46.07
19 21	512x28x28	512x3x3	1201.93	58.78	94.83
24 26 28	512x14x14	512x3x3	311.04	17.13	19.8

Table 3. All unique convolutional layer configurations of VGG-16 and the runtime [ms] evaluated for the ARM Compute Library (Direct and GEMM) and the Lift-generated code for the HiKey 970 (Kirin 970 processor)

buffer size. These constraints can be inferred by calculating the minimum resources necessary to compute an expression and matching them against respective OpenCL driver information.

5.3 Constraint solver

To explore the space of valid parameter value combinations for a given layer configuration, we designed the following search strategy. Firstly, we use the rewrite rule system to produce a parametric candidate expression; this expression is traversed for parameter and constraint inference. Next, we sort the parameters in the order in which they need to be explored – for example, if the parameter A depends on the parameter B, B needs to be evaluated first. To find this ordering, we represent the collection of constraints as a directed acyclic graph and sort it topologically. The resulting partial sorting order is finalized by imposing a random order on the unsorted groups of parameters. The derived parameter order is used to incrementally generate random combination of parameter values that satisfy all the constraints.

6 Experimental Methodology

Code generation The Lift compiler is used to generate the code that runs on the GPU. We use an extended version of the Lift compiler to also generate OpenCL host code that sets up the device, compiles the GPU code, sends/retrieves the data and executes the GPU code. For each layer configuration, we generated 1000 randomly chosen implementations that satisfy all the constraints.

As a baseline to evaluate the performance of our generated code, we use the ARM Compute Library (v19.02) with the Graph API, implementing the same layers and running these on the GPU by indicating *cl* as the target from the API. All the ARM compute library results are produced using ARM's built-in auto-tuner.

Benchmarks To evaluate the code generated, we use all nine unique layer configurations of the VGG-16 model [19]. This network is well-studied performance in literature and has higher resource requirements than others such as ResNet and GoogleNet [3]. Table 3 presents the layer configurations.

All results are validated by using a fixed random input and comparing the output with that of PyTorch.

Platform In this paper, we target the ARM Mali-G72 (12 cores) mobile GPU using the HiSilicon Kirin 970 SoC running Debian GNU/Linux 9.8. The highest frequency (767MHz) was used.

GPU execution time For our own results, we measure GPU execution time using the `cl_event` associated with the kernel launches. For the ARM compute library, GPU execution time is measured by intercepting all OpenCL calls using our own profiler, which is an OpenCL wrapper library. The library automatically grabs the `cl_event` associated with each OpenCL kernel launch or creates one on the fly if required. This is done in a fully transparent way and does not influence the application being profiled. This allows us to reuse the exact same methodology for measuring execution time for the Lift generated GPU code and the ARM compute library. The numbers reported are the sum of all the GPU kernels involved in the operations of a convolutional layer, including the time to pad the input and crop the outputs.

7 Evaluation

This section explores the performance of the automatically generated direct convolution in Lift. A comparison is given against the best hand-written library for the ARM Mali GPU: the ARM Compute Library.

7.1 Comparison with ARM Compute Library

Table 3 shows the execution times of the Lift-generated OpenCL kernels and the ARM Compute Library direct convolution and GEMM implementation. Both these versions have been auto-tuned using the tools provided by the ARM Compute Library. As evident from the results, the Lift-generated code is always faster than the ARM Compute Library direct convolution and more space-efficient than its GEMM method. Furthermore, in some cases it is actually on par or better than the highly tuned GEMM implementation.

Figure 5 shows the performance of the Lift generated code expressed as throughput – amount of useful outputs generated per second – compared to that of direct and GEMM-based convolution from the ARM Compute Library. For every layer, Lift is faster than the ARM Compute library direct convolution and is ×10 faster on average. While Lift kernels achieve only ×0.7 the throughput of the GEMM-based implementation, the memory consumption is ×3.6 less and is close to that of the vanilla direct convolution. This demonstrates that our approach based on automatic code generation outperforms a human expert.

7.2 Multi-objective optimization

Depending on application, priorities in neural network inference optimization might shift. In a resource-bound system such as a mobile GPU that is shared among multiple tasks, low memory footprint is required; for time-critical tasks, throughput or latency are to be prioritized. Figure 6 demonstrates how search space exploration allows for multi-objective optimization to cater for various budgets: advancing the Pareto frontier results in a set of implementation candidates to choose from statically or at runtime for specific time and space requirements. In the case of VGG layer 2, the compiler might prioritize space efficiency by using 25 MBytes to compute results in 100 ms; when the memory budget is bigger, the compiler can prefer the 77 ms kernel that uses 31 Mbytes of space.

Populating a sizeable Pareto set is made possible thanks to the exploration of the tuning parameter search space, performed in a safe way thanks to constraint inference. Compared to libraries that depend on sets of handwritten kernels, a compiler can adapt to finer differences in the workload and target hardware.

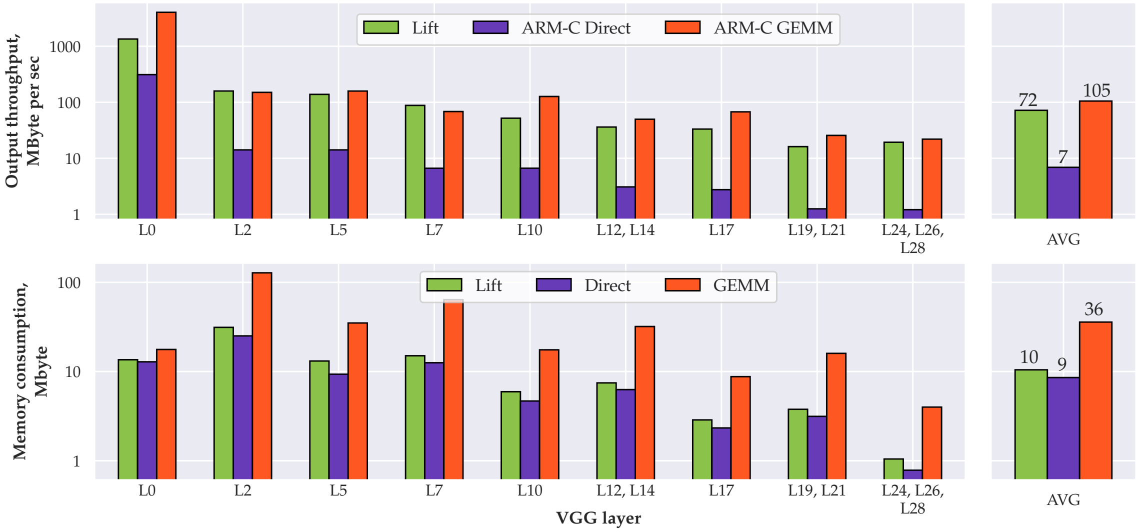

Figure 5. Throughput and memory consumption comparison of Lift-generated kernels versus the direct and GEMM-based convolution methods on VGG-16

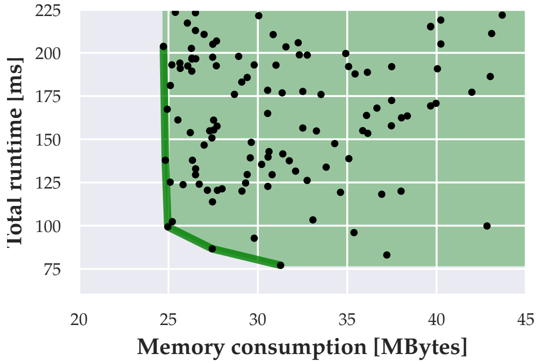

Figure 6. The Pareto frontier of time and space efficiency of the search space explored for layer 2 of VGG-16

Parameter	Value
Input tile size	5 × 5
Number of kernels per workgroup	4
Number of windows per thread	3
Sequentially processed input elements	144
Optimizational padding size	11
Vector size	4
Unrolling	No
Coalescing	Yes

Table 4. Best parameters found for layer 7 of VGG-16

7.3 Analysis of the Best Point

We now analyze one of the best points that we found using the 7th layer of VGG as an exmaple. Table 4 shows the best tuning parameters found together with the thread local sizes for the GPU kernel responsible for performing a partial convolution. These parameters show that a workgroup processes a tile which can fit 9 sliding windows. 4 out of 128 kernels are processed by a work group, enabling reusage of the input data multiple times, without adding too much register pressure; 3 out of 9 sliding windows are processed by each thread, enabling reusage of the weight data. The amount of padding is also quite minimal, which avoids unnecessary work. We also see that this point is vectorized which is good for memory loads on the Mali-G72 architecture.

8 Related Work

Several deep learning frameworks have recently been developed. Most of these frameworks rely on high-level graph-based representations of neural networks [1, 3, 13, 18] to allow for automatic differentiation. Such graphs are too high-level to be mapped optimally to specific hardware, so frameworks rely on hand-written code provided by hardware vendors, as found in Intel's MKL-DNN, Nvidia's TensorRT and ARM's Compute Library.

To address this, multi-level graph representations such as MXNet, XLA and TVM [3, 4, 17] have also been proposed, allowing subgraph and dataflow optimization to be made device-specific. TensorComprehensions [24] make use of the polyhedral compilation model to perform operation optimization and scheduling, but so far only target CUDA-capable GPUs. Depending on the target hardware, MXNet either provides handwritten layer implementations which lack portability or using BLAS libraries such as Atlas, MKL, CuBLAS and OpenBLAS. These libraries are also constrained in how much they can adapt to the target hardware relying just on tuning and handwritten code selection. Another code generator with auto-tuning is Latte [21], which has shown good performance for CPU code, although not evaluated on mobile devices. Their performance is achieved by generating code with cross-layer fusion, which is problematic for modeling exact layer conditions. On mobile platforms, MXNet only supports CPU-based libraries.

Other works have recently explored efficient implementations of direct convolution [2, 9, 25] but are limited in the scope of their available target platforms. In particular, [9, 25] are reliant on the availability of SIMD instructions and are specific to CPUs. Tsai et al. [22] rely on efficient implementation of OpenCL kernels to reduce memory requirement of GEMM by avoiding replication of input patches, however this is not fast enough for mobile devices.

There have also been several developments at the algorithmic level allowing for fast approximations to convolution [16, 23], or computationally cheaper substitutions [6, 7, 12]. In this work we have not considered such approximate methods, but leave them for future exploration.

9 Conclusions

Most machine-learning frameworks rely on GEMM to implement convolutions due to the availability of high-performance implementations on most parallel devices. The downside is that GEMM requires an order of magnitude more memory than direct convolution, which can restrict the application of neural networks for memory limited embedded devices. Direct convolution is an attractive alternative, however, hardware-vendor provided implementations are often an order of magnitude slower than their GEMM counterpart.

This paper has shown how we automatically generate high performance direct convolution with Lift for the ARM Mali GPU. This approach leads to a ×10 speedup and ×3.6 memory saving over the tuned ARM Compute Library implementations.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2017. Low-memory gemm-based convolution algorithms for deep neural networks. *arXiv preprint arXiv:1709.03395* (2017).

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv* arXiv:1410.0759 (2014).

[6] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 1251–1258.

[7] Elliot J Crowley, Gavin Gray, and Amos J Storkey. 2018. Moonshine: Distilling with cheap convolutions. In *Advances in Neural Information Processing Systems*. 2888–2898.

[8] Kunihiko Fukushima. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics* 36, 4 (1980), 193–202.

[9] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 830–841.

[10] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[11] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7132–7141.

[12] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. 2018. Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2752–2761.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[15] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Not all ops are created equal! *arXiv preprint arXiv:1801.04326* (2018).

[16] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[17] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).

[18] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[19] K. Simonyan and A. Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[20] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 74–85. https://doi.org/10.1109/CGO.2017.7863730

[21] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: a language, compiler, and runtime for elegant and efficient deep neural networks. *ACM SIGPLAN Notices* 51, 6 (2016), 209–223.

[22] Yaohung Tsai, Piotr Luszczek, Jakub Kurzak, and Jack Dongarra. 2016. Performance-Portable Autotuning of OpenCL Kernels for Convolutional Layers of Deep Neural Networks. In *Workshop on Machine Learning in HPC Environments*.

[23] Michael Tschannen, Aran Khanna, and Anima Anandkumar. 2017. StrassenNets: Deep learning with a multiplication budget. *Proceedings of the 35th International Conference on Machine Learning* (2017).

[24] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[25] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. 2018. High performance zero-memory overhead direct convolutions. *Proceedings of the 35th International Conference on Machine Learning* (2018).