Efficient Winograd or Cook-Toom Convolution Kernel Implementation on Widely Used Mobile CPUs

Partha Maji
Partha.Maji@cl.cam.ac.uk
University of Cambridge

Andrew Mundy
Andrew.Mundy@arm.com
Arm Research

Ganesh Dasika
Ganesh.Dasika@arm.com
Arm Research

Jesse Beu
Jesse.Beu@arm.com
Arm Research

Matthew Mattina
Matthew.Mattina@arm.com
Arm Research

Robert Mullins
Robert.Mullins@cl.cam.ac.uk
University of Cambridge

Abstract

The Winograd or Cook-Toom class of algorithms help to reduce the overall compute complexity of many modern deep convolutional neural networks (CNNs). Although there has been a lot of research done on model and algorithmic optimization of CNN, little attention has been paid to the efficient implementation of these algorithms on embedded CPUs, which usually have very limited memory and low power budget. This paper aims to fill this gap and focuses on the efficient implementation of Winograd or Cook-Toom based convolution on modern Arm Cortex-A CPUs, widely used in mobile devices today. Specifically, we demonstrate a reduction in inference latency by using a set of optimization strategies that improve the utilization of computational resources, and by effectively leveraging the ARMv8-A NEON SIMD instruction set. We evaluated our proposed *region-wise multi-channel* implementations on Arm Cortex-A73 platform using several representative CNNs. The results show significant performance improvements in full network, up to 60%, over existing *im2row/im2col* based optimization techniques.

Keywords CNN, Winograd, Cook-Toom, Embedded CPU

1 Introduction

The agility of cloud computing is great - but it simply isn't sufficient. In the near future there will be more demand for AI at the edge than in the cloud. As people need to interact with their digitally-assisted technologies (e.g. personal assistants, wearables, autonomous cars, healthcare, and other smart IoT devices) in real-time, waiting on a datacenter many miles away isn't going to work. Not only the latency matters, but often these edge devices are not within the range of the cloud needing them to operate autonomously for the most part. Even when these devices are connected to the cloud, moving high-volume of data to the centralized datacenter is not scalable, due to communication cost that impacts performance and energy consumption [9]. Since the latency and security risk of relying on the cloud are intolerable, we need a significant portion of computation closer to the edge to permit secure, autonomous, and real-time decision making. This poses an enormous challenge in terms of implementing emerging AI workloads on resource constrained low power embedded systems. When it comes to image and video the performance of many modern embedded applications is enhanced by application of neural networks, and more specifically by convolutional neural network (CNN). Although there has been a lot of research done on algorithmic optimization of CNN [9], such as the Winograd, the Cook-Toom, and the Strassen, little attention has been paid to the efficient implementation of these algorithms on widely available energy efficient embedded CPUs. This paper aims to fill this gap and investigates if emerging, compute-heavy deep CNNs can be implemented efficiently using such fast arithmetic scheme on widely used resource constrained mobile class CPUs. Specifically, we target Cortex-A class processors as Arm-based SoCs are ubiquitous in today's mobile computing [5].

We introduce a novel *region-wise multi-channel* scheme using GEMM (General Matrix Multiplication) for energy efficient implementation of Winograd or Cook-Toom based convolution on resource-constrained mobile CPUs. We show that our scheme performs better than classical *im2row/col* techniques. Unlike existing implementations which are limited to 2D convolutions only, we apply variations of the base algorithms to both the 2D ($N\times N$) and 1D layers ($1\times N$, $N\times 1$), where N is the height/width of the filter. We demonstrate the efficiency of our scheme by implementing a number of widely used state-of-the-art deep CNNs on the energy-efficient Arm Cortex-A73 processor [4].

Our results show that by effectively using Armv8-A NEON SIMD instructions and appropriate choice of variations of Winograd or Cook-Toom based convolution an average $2-3\times$ and a peak $4\times$ per layer speedup on top of aggressively optimized solutions using the classical *im2row/col* technique is achievable. As an example, our multithreaded implementation of SqueezeNet on Arm Cortex-A73 can achieve an average inference rate of 47 frames/sec – sufficient for many real-time embedded applications [1]. Our scheme can be readily deployed to other widely used ARMv8-A cores.

HPCA.EMC2, Feb 17, 2019, Washington D.C., USA

2 Strategies for Efficient Multichannel Winograd or Cook-Toom Kernel Implementation on Armv8-A Cores

The Winograd or Cook-Toom class of algorithms [7, 8] help to reduce the overall compute complexity of convolution by reducing the number of required multiplication. Implementations of these algorithms are well suited to CNNs consisting of small filters and low power embedded systems as the resources and power budget are very limited. Using the Winograd or Cook-Toom based convolution, a typical layer of a convolutional neural network (CNN) can be expressed in the following matrix equation

$$f = Z^T(\sum_{c=0}^{C}[(WwW^T)_c \odot (X^T xX)_c])Z \quad (1)$$

where W and X are the transform matrices for the weight and the input sequence w, x, respectively. Z is the inverse transformation matrix, and $\odot$ is the elementwise (Hadamard) product.

First, we note that the equation shown above applied a (w×w) filter to only a small input region of size (x×x) to produce an output region of size (z×z) (a.k.a. $F(z{\times}z, w{\times}w, x{\times}x)$). To perform larger convolutions we must, therefore, break the input tensor into multiple regions of size (x×x). The output tensor must also be divided into an equivalent number of regions, each of which is computed as the elementwise multiplication and accumulation of the corresponding input regions (representing C input channels) with their respective weight tiles. This algorithm is illustrated in Listing 1.

```
// For each output channel
for (unsigned int m = 0; m < M; m++)
  // For each output region
  for (unsigned int r = 0; r < R; r++)
    // Summation across the input channels
    for (unsigned int c = 0; c < C; c++)
      output_region[m, r] += HadamardProduct(
        input_region[c, r], weight_region[m,c]);
```

Listing 1. Sample Winograd convolution algorithm

We break our region wise multi-channel algorithm into three steps:

1. **Input Transform** Progresses over regions of the input tensor, transforms them into the Winograd domain and *scatters* the results into the 'A' matrices for the GEMMs.
2. **GEMM** *Multiplies* the 'A' matrices generated in the *input transform* with 'B' (matrices generated when the weights were transformed into the Winograd domain) to form the 'C' matrices.
3. **Output Transform** Repeatedly *gathers* regions of values from the 'C' matrices, transforms them back into the spatial domain and writes the results into the output tensor.

2.1 Data Layout and SIMD Computation

There are a variety of ways in which 4D tensors can be arranged in memory. Two common options are called *NCHW* and *NHWC* – where N stands for the number of batches (or concurrent inferences), C for the number of channels, and H and W stand for height and width, respectively. In *NCHW* each plane of the tensor is stored contiguously in memory – i.e., pixel (n, c, i, j) is followed by $(n, c, i, j + 1)$ – whereas in *NHWC* all of the channels of a given pixel are stored contiguously (i.e., value (n, i, j, c) is followed by $(n, i, j, c+1)$). When writing vectorized (SIMD) code, tensor ordering is crucial to achieving performance.

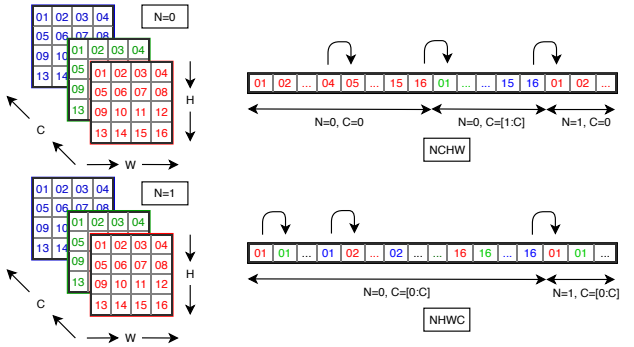

Figure 1. NCHW vs NHWC Layout

In the Armv8-A architecture, there are thirty-two 128-bit SIMD registers. Each SIMD register can, therefore, store four 32-bit single-precision values. Hence, under *NCHW* a single SIMD register will store, after a 128-bit load, a row of four pixels, whereas under *NHWC* the same register would store four channels of data for a single pixel. We can see the effect of these orderings through the example of implementing the input transform for $F(2{\times}2, 3{\times}3, 4{\times}4)$.

2.1.1 Input Transform For $F(2{\times}2, 3{\times}3, 4{\times}4)$

The characteristic equation for this transform is:

$$X^\top xX = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} x \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \quad (2)$$

Under *NCHW* ordering we would use four registers to store x, a 4×4 region of the input tensor. The transform matrices could be hard-coded as a series of row-transformations, such that computing $X^\top x$ could be expressed as:

```
XTx[0] = vsubq_f32(x[0], x[2]); // x_1i - x_3i
XTx[1] = vaddq_f32(x[1], x[2]); // x_2i + x_3i
XTx[2] = vsubq_f32(x[2], x[1]); // x_3i - x_2i
XTx[3] = vsubq_f32(x[1], x[3]); // x_2i - x_4i
```

By transposing the result, this code sequence can be repeated such that we compute $\left(X^\top \left((X^\top x)^\top\right)\right)^\top = X^\top x X$. Once this is completed we have 16 values (four registers containing four values each) which must be scattered, as described before, to 16 separate matrices.

In contrast, under *NHWC* ordering, we would use sixteen SIMD registers to represent four channels of a 4×4 region of the input tensor. The transformation can be hardcoded, but in this case we operate on four channels of data simultaneously, as in Listing 2. Once the transformation is complete we are left with sixteen registers, each containing four channels worth of data. These registers can be scattered directly into the the input matrices for the GEMMs.

```
// Compute X^T x and U = (X^T x) X
for (int j = 0; j < 4; j++) {
// For each column in X^T x
   XTx[0][j] = vsubq_f32(x[0][j], x[2][j]);
   XTx[1][j] = vaddq_f32(x[1][j], x[2][j]);
   XTx[2][j] = vsubq_f32(x[2][j], x[1][j]);
   XTx[3][j] = vsubq_f32(x[1][j], x[3][j]);
   }
for (int i = 0; i < 4; i++) {
// For each row in U
   U[i][0] = vsubq_f32(XTx[i][0], XTx[i][2]);
   U[i][1] = vaddq_f32(XTx[i][1], XTx[i][2]);
   U[i][2] = vsubq_f32(XTx[i][2], XTx[i][1]);
   U[i][3] = vsubq_f32(XTx[i][1], XTx[i][3]);
   }
```

Listing 2. Input Transforms.

2.1.2 Choice of NHWC over NCHW

For the specific instance of $F(2\times2, 3\times3, 4\times4)$ there are merits to both approaches. However, when we consider using either different data widths (such as half-precision floating point) or different version of Winograd or Cook-Toom algorithms we begin to see advantages to the *NHWC* ordering.

For example, although we can use four SIMD registers to represent 16 values *in single-precision floating point* in *NCHW* – four values to a register – this breaks down when we move to half-precision and each register can contain eight values, whereas the *NHWC* code could be simply modified to work on eight channels of data simultaneously.

Likewise, were we to implement the input transform for $F(4\times4, 3\times3, 6\times6)$, which requires use of 6×6 input regions, we could, in *NHWC* ordering use 36 values (and the stack) to represent each input region. However, in *NCHW*, we would need to use one-and-a-half registers to represent each row of six values. For these reasons, we prefer the use of *NHWC* ordered data.

2.1.3 Efficient Tensor Ordering for ARMv8-A Cores

The convolution of a tensor consisting of C input layers and R regions with a M deep set of filters can be expressed as x^2 GEMMs of the form $[R\times C]\times[C\times M]$ or $[M\times C]\times[C\times R]$, and that, of these, we preferred the former as shown in Figure 2. This selection follows directly from our choice of *NHWC* tensor ordering. Specifically, we note that, under *NHWC*, each SIMD register contains multiple channels of data and that these values must be written into matrices of shape $R\times C$ or $C\times R$. Assuming row-major ordered matrices we note that, in the latter case, we could use multi-element structured stores (e.g., ST4 (single structure), [2]) to combine and store values from different registers. Alternatively, an unstructured store (STR [2]) could be used to write out a whole register into successive columns of an $R\times C$ matrix. Since we found unstructured stores to have a higher throughput than their structured counterparts we choose to use the first form.

2.2 Using GEMM to Compute Hadamard Products

By inspecting the basic convolution algorithm illustrated in Listing 1 we observe, firstly, that the fundamental operation is an element-wise multiply-accumulate (element-wise addition of Hadamard products). Secondly, we note that there are two axes in which data is reused - (1) Weight tile (m, c) is used across all input regions in layer c, and, (2) Input region (c, i, j) contributes to all M output regions at (i, j). These observations suggest that one way of implementing a complete convolution is to leverage the GEMM (General Matrix Matrix Multiplication) algorithm since there exist a wide range of good GEMM implementations (e.g., [6]) capable of exploiting the SIMD instructions of the Armv8-A architecture. Figure 2 shows an example for a 3×6×6 tensor being convolved with four filters. An array of 16 GEMMs of size $[R\times C]\times[C\times M]$ is constructed, with the input tensor being represented by the first set of matrices and the weights by the latter.

3 Evaluation and Results

We chose five widely used CNNs of different sizes and complexities to validate our implementation, namely, VGG19, VGG16, GoogleNet, Inception-v3, and SqueezeNet [9]. We benchmarked our implementation on the Huawei HiKey 960 development platform using IEEE 754 fp32 standard.

3.1 Results – per-layer speedup

We implemented five different variants of the fast algorithm and bench-marked them on individual layers of all the selected models. In each case we measured the number of cycles taken to perform all three stages of our algorithm (Input transform, GEMMs and Output transform) on the 'big'-cluster which consists of four Cortex-A73 core. As a baseline against which to compare we also benchmarked the GEMM calls which would result from application of the classical *im2row* technique to the same layers. Table 2 presents the speedup achieved by our region-wise multi-channel Winograd scheme over the GEMM.

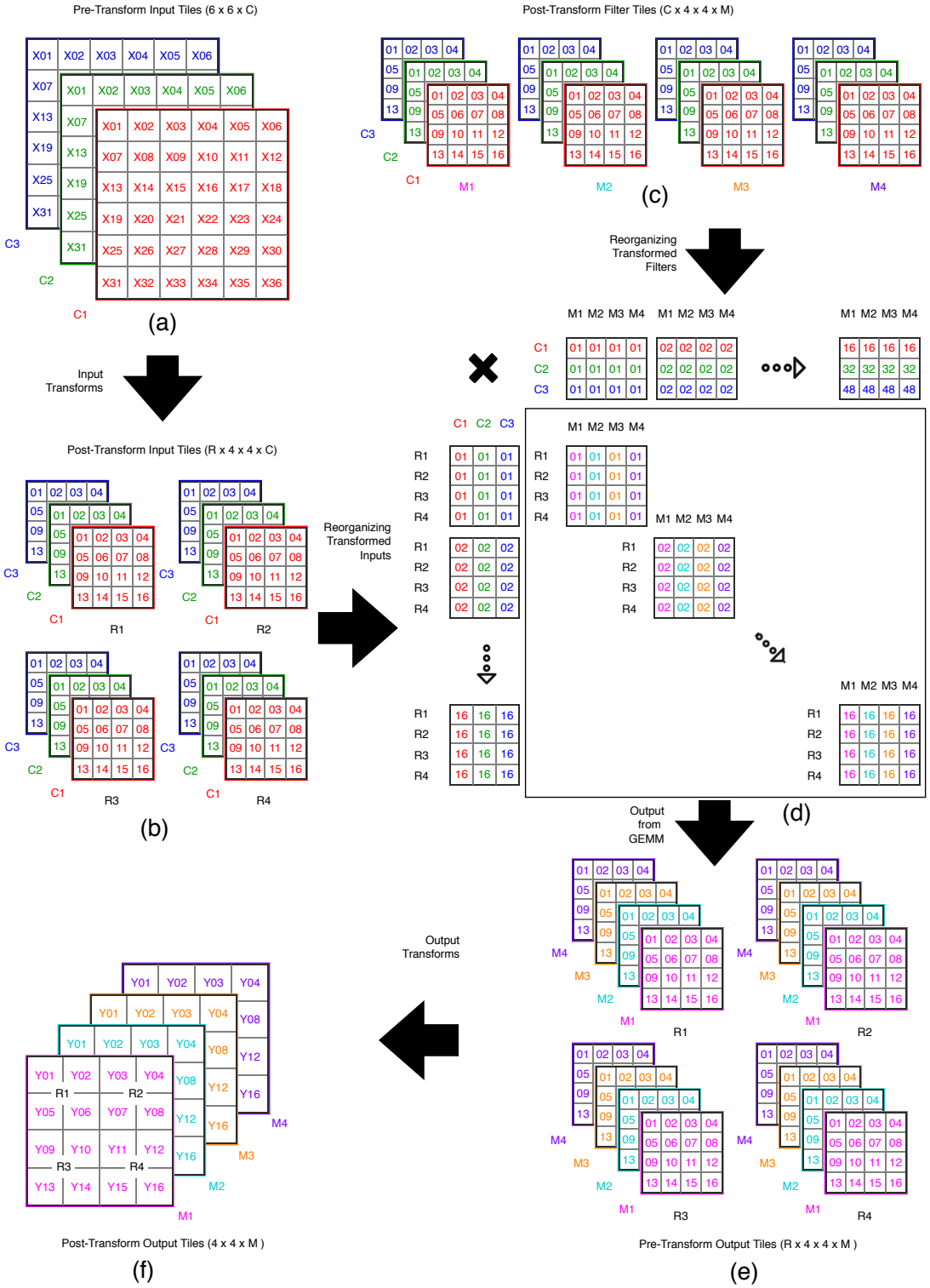

Figure 2. Detailed Data-Flow Diagram in Region-wise Multi-channel Winograd or Cook-Toom based Scheme – (a) Pre-transform Input Channels, (b) Transformed Input Channel Regions, (c) Transformed Filters, (d) GEMM Kernels, (e) Output of GEMM in the Residue Domain, (f) Final Output Channels after applying Inverse Transforms

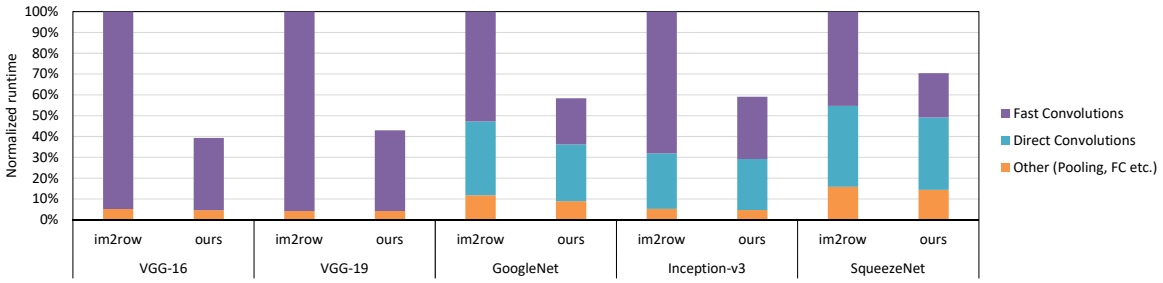

Figure 3. Speed-up achieved in the Winograd or Cook-Toom suitable layers as a **fraction** of the entire model (batch size = 1)

Table 1. Summary of **mean absolute runtime** of the **whole-network** in milliseconds (msec) for batch size of 1

	VGG-16		GoogleNet		Inception-v3		SqueezeNet	
	Full Network	Fast Layers	Full Network	Fast Layers	Full Network	Fast Layers	Full Network	Fast Layers
Using Im2Row Scheme	1929.43	1829.10	173.13	91.42	750.37	510.92	29.72	13.47
Using Our Scheme	758.05	670.79	101.04	38.38	443.40	224.42	20.91	6.29
Speedup (msec)	1171.38	1158.31	72.09	53.04	306.98	286.51	8.81	7.18
Speedup (%)	60.71%	63.33%	41.64%	58.02%	40.91%	56.08%	29.64%	53.28%

Table 2. per-layer speedup comparison: im2row vs ours

Model	Per-layer Speedup		
	Layer-type	Average Speedup	Peak Speedup
VGG-16	3 × 3	2.7×	3.5×
VGG-19	3 × 3	2.8×	3.5×
GoogleNet	3 × 3	2.6×	4.1×
GoogleNet	5 × 5	2.3×	3.2×
Inception-v3	1 × 7	2.0×	2.1×
Inception-v3	7 × 1	2.0×	2.1×
Inception-v3	3 × 3	3.1×	3.8×
Inception-v3	5 × 5	2.7×	2.8×
SqueezeNet	3 × 3	2.2×	2.6×

3.2 Results – whole-network speedup

To measure the effectiveness of Winograd or Cook-Toom based acceleration for end-to-end CNN, we also benchmarked the runtime of entire models. In these cases we used the Arm Compute Library [3] to evaluate single-batch (batch size of 1) inferences of these networks on multi-threaded (4×) Cortex-A73. Two sets of benchmarks were run: in one, layers suitable for the Winograd-based acceleration use our scheme, and the rest use baseline *im2row* scheme; in the other all layers use *im2row*. Figure 3 and Table 1 shows the normalized and the absolute runtime of the five CNNs (whole-network), respectively.

4 Conclusions

Winograd or Cook-Toom based acceleration on Arm's Cortex A CPUs can dramatically reduce the compute time and energy cost of individual convolution layers – by up to 4×. However, these speedup numbers are lower than the theoretical values. Partially, this is due to the challenges involved in implementing the algorithm in a real system but largely it is because the theoretical speed-up of this class of algorithm disregards the cost of transforming to and from the alternative domain of computation. This gap between the theoretical and achieved speed-ups can be somewhat overcome by amortizing the transform costs over those of the GEMMs. As the number of output channels increases, the speed-up will asymptotically approach the maximum achievable.

References

[1] G. Ananthanarayanan, P. Bahl, P. BodÃŋk, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha. 2017. Real-Time Video Analytics: The Killer App for Edge Computing. *Computer* 50, 10 (2017), 58–67. https://doi.org/10.1109/MC.2017.3641638

[2] Arm-Ltd. 2017. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile.

[3] Arm-Ltd. 2017. Compute Library Arm Developer. https://developer.arm.com/technologies/compute-library. (Accessed on 03/28/2018).

[4] Mike Demler. 2016. The Linley Group - Cortex-A73 Improves Mobile Efficiency. https://www.linleygroup.com/newsletters/newsletter_detail.php?num=5536. (Accessed on 09/19/2018).

[5] Anthony Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. Springer-Verlag, 243–258. http://dx.doi.org/10.1007/978-3-642-14052-5_18

[6] Gianluca Frison, Dimitris Kouzoupis, Andrea Zanelli, and Moritz Diehl. 2017. BLASFEO: Basic linear algebra subroutines for embedded optimization. *CoRR* abs/1704.02457 (2017). arXiv:1704.02457 http://arxiv.org/abs/1704.02457

[7] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). arXiv:1509.09308 http://arxiv.org/abs/1509.09308

[8] Partha Maji and Robert Mullins. 2018. On the Reduction of Computational Complexity of Deep Convolutional Neural Networks. *Entropy* 20, 4 (2018). https://doi.org/10.3390/e20040305

[9] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *CoRR* abs/1703.09039 (2017). arXiv:1703.09039 http://arxiv.org/abs/1703.09039