mGEMM: Low-latency Convolution with Minimal Memory Overhead Optimized for Mobile Devices

Jongseok Park
Seoul National University
cakeng@snu.ac.kr

Kyungmin Bin
Seoul National University
kmbin@snu.ac.kr

Kyunghan Lee
Seoul National University
kyunghanlee@snu.ac.kr

ABSTRACT

The convolution layer is the key building block in many neural network designs. Most high-performance implementations of the convolution operation rely on GEMM (General Matrix Multiplication) to achieve high computational throughput with a large workload size. However, in mobile environments, the user experience priority puts focus on low-latency inferences over a single or limited batch size. This signifies two major problems of current GEMM-based solutions: 1) GEMM-based solutions require mapping the convolution operation to GEMM, causing overheads in both computation and memory, 2) GEMM-based solutions lose large opportunities of data reuse while mapping, leading to under-utilization of the given hardware. Through an in-depth analysis of current GEMM-based solutions, we identify the root cause of these problems, and we propose mGEMM, a convolution solution that overcomes the aforementioned problems, without changes in accuracy. mGEMM expands the structure of GEMM in such a way that it can accommodate the convolution operation without any overhead, while the existing algorithms suffer from inefficiencies in converting the convolution operation to a static GEMM algorithm. Our extensive evaluations done over various neural networks and test devices show that mGEMM outperforms the existing solutions in the aspects of latency, memory overhead, and energy consumption. In running a real-world application, YoloV3-Tiny object detection, mGEMM achieves up to 1.29× and 1.58× speedup in total latency and convolution latency compared to the state-of-the-art, resulting in 15.5% reduction in energy consumption while using only near-minimum heap memory.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Convolutional Neural Networks; Parallel Computing Algorithms

ACM Reference Format:
Jongseok Park, Kyungmin Bin, and Kyunghan Lee. 2022. mGEMM: Low-latency Convolution with Minimal Memory Overhead Optimized for Mobile Devices . In *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), June 25–July 1, 2022, Portland, OR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3498361.3538940

MobiSys '22, June 25–July 1, 2022, Portland, OR, USA

ACM ISBN 978-1-4503-9185-6/22/06…$15.00
https://doi.org/10.1145/3498361.3538940

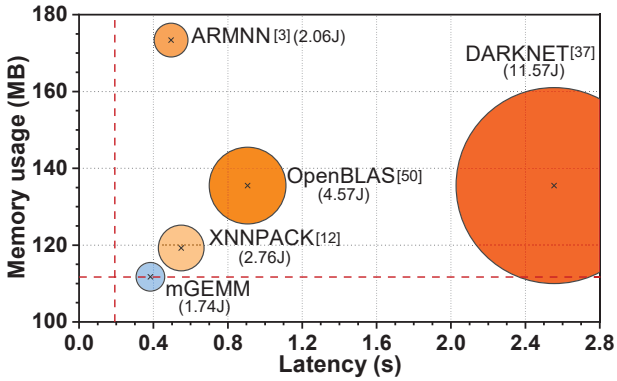


Figure 1: Latency, energy, and max heap memory usage of different GEMM-based convolution solutions while running YoloV3-Tiny[38] object detection network on Raspberry Pi 4. The diameter of each bubble is proportional to the energy used. Red dotted lines denote the ideal figures. mGEMM outperforms all others.

1 INTRODUCTION

Deep neural networks (DNNs), often represented by convolutional neural networks (CNNs), have become the foundation of various learning-based solutions to computer vision problems as well as general inference-oriented problems, such as image classification[40], object detection[38], video analytic[22], and time-series prediction[51]. These networks have seen vast improvements since the introduction of AlexNet[24], and have reached a stage where CNN-based solutions are commercially deployed, assisting the end consumers using mobile platforms such as smartphones, autonomous vehicles, or industrial IoT devices.

Mobile devices are often based on ARM CPUs, which are more power-efficient but are with less hardware capabilities compared to high-performance processors. As such, fast and efficient implementation of the convolution algorithm is becoming more critical to providing smooth DNN-based services to the end-users. Out of possible implementations, GEMM (General Matrix Multiplication) based solutions are known as the most widely adopted technique. These solutions map the convolution tensors into a matrix through transformations such as im2col (image to column)[5], effectively translating the convolution operations into matrix multiplication operations. This can be largely accelerated by BLAS (Basic Linear Algebra Subprograms) libraries optimized for the given processors[4, 21, 33, 50].

Given a large enough matrix size, these highly optimized GEMM routines can easily saturate the underlying hardware, achieving its maximal computational throughput. To provide such a large matrix,

GEMM-based convolutions often process their workloads in batches, mapping multiple images into a single large matrix. This lowers the average processing time of each workload, but also delays the completion time of a single workload, leading to a longer per-image latency.

To this end, we raise the question of whether the current GEMM-based convolution design is the most efficient solution for mobile given the characteristics of mobile platforms. To the best of our knowledge, most of the existing GEMM-based solutions such as im2col+GEMM[5] or implicit GEMM[34, 46] have focused on achieving high throughput with abundant computation power to deal with large workloads such as neural network training and batched inferences, over high-performance processors. This design goal of conventional GEMM-based solutions shows a stark contrast to what is expected in mobile environments, where the user experience priority is on the low-latency inference over a single image or data with limited batch size. Our observations on this gap bring up two challenges in designing a new mobile-optimized GEMM technique as follows.

Minimizing the preprocessing overhead: Mapping the convolutions into matrix multiplications requires preprocessing of the tensors, using methods such as im2col. These mapping steps require duplication and reordering of data, which incur computation time and memory overheads. In high-performance systems with larger memory bandwidth and capacity, these overheads have been considered trivial compared to the main computation loops, especially when processing larger workloads. But in mobile platforms, these overheads suddenly become significant, especially when making latency-critical inferences. We later show that these overheads occupy up to 29% of total computation time when executing mobile inferences.

Maximizing the data reuse rate: As shown in Section 2, the convolution operation is an operation that inherently contains a large amount of data reuse between its computation loops. On the other hand, matrix multiplications experience much less data reuse given the same number of computations. As such, conventional GEMM-based solutions are inevitably losing large opportunities for data reuse during the mapping process. This is not a serious problem when training or inferencing on large workloads, as one could easily increase the data reuse by intentionally increasing the batch size, to the point where the operation is computation-bound rather than memory-bound. However, in mobile platforms that often carry the batch size of one for immediate inference service to users, this strategy is not applicable. Combined with the limited memory bandwidth, the inefficiency in memory reuse makes conventional GEMM-based solutions to be significantly memory-bound on mobile platforms.

To tackle these critical challenges, we propose *mGEMM*, a new GEMM-based convolution solution with focuses on low-latency (i.e., low per-input latency) and minimal memory overhead, tailored for mobile platforms. We base our design on GEMM, enabling us to utilize the existing optimization techniques and extensive hardware supports that GEMM algorithms use. The key difference is that mGEMM also leverages our observations on the root cause of inefficiencies in existing solutions: the filter spatial dimensions of the convolution operation being unrepresentable in matrix multiplication. With these factors taken into consideration, we propose to directly accommodate the convolution dimensions with adaptive expansions applied to GEMM kernels rather than forcing them to be translated into static GEMM kernels. This approach greatly reduces the memory footprint and memory accesses required for the convolution computation, while leaving the accuracy of the result unaffected. As the mobile memory system is restricted in performance and often shared by the hardware on the SoC, reducing the memory traffic and footprint of the computation provides substantial benefit to the whole system, regardless of the hardware being used.

We confirm this with tests on an off-the-shelf mobile platform; we find that our implementation of mGEMM greatly improves the memory-bound problems, resulting in efficiencies in various performance metrics of mobile platforms, more specifically in the ARMv8 architecture. Although our tests are conducted on CPU, we believe that the benefits of mGEMM are applicable to other mobile hardware such as GPUs and NPUs as well. These hardware often have a higher computational ceiling than CPUs, and algorithmic reduction in memory traffic from mGEMM would allow these hardware to reach a higher computation speed, with the same memory bandwidth.

The efficacy of using mGEMM as an instance is depicted in Figure 1 in comparison with state-of-the-art mobile neural network libraries such as ARM's ARMNN [3] and Google's XNNPACK [12]. The figure plots the total network latency, maximum heap memory usage, and the resulting energy consumption while executing YoloV3-Tiny [38] on Raspberry Pi 4's CPU with mGEMM and other GEMM-based solutions. The dotted red lines depict the ideal figures of total latency and memory, obtained by skipping all convolution computations while running YoloV3-Tiny. Compared to the second fastest method from ARMNN [3], mGEMM achieves 1.29× speedup in total latency, and 1.58× speedup in convolution latency, resulting in a 15.5% reduction in total energy consumption. mGEMM achieves this outstanding performance with only 0.018MB more heap space usage than the ideal usage of 111.717MB. Detailed results are provided in Section 5.

The main contributions of this paper are as follows:

- In-depth analysis of the convolution operation and the existing GEMM-based solutions, revealing the problems of the existing solutions in mobile platforms.
- Proposal of a new convolution algorithm (mGEMM), which overcomes the limitations of previous GEMM-based methods, while still leveraging the existing optimizations and hardware supports that have been developed intensively for GEMM algorithms.
- Proof of concept for mGEMM on its performance gains over the current state-of-the-art GEMM-based solutions, with a real-world implementation tested on various off-the-shelf mobile platforms.

2 BACKGROUND

Before we delve into mGEMM, we provide the background on why GEMM-based methods are being used as the dominant solution for convolution.

2.1 Why GEMM?

There are several different approaches to accelerating the computation of the convolution layer in neural networks. Mathematical transformations such as Fast Fourier Transform (FFT) [28] or

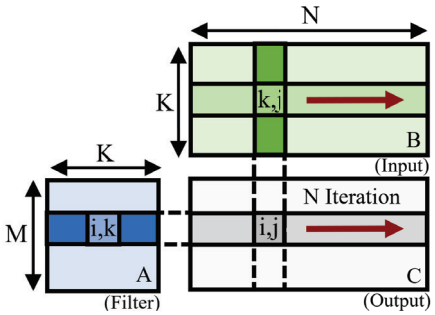


Figure 2: Visualization of the matrix multiplication (GEMM) operation. We can observe that iterating over dimension N (red arrow) requires access to new input ($B_{k,j}$) and output ($C_{i,j}$) elements, but allows reuse of the filter ($A_{i,k}$) element.

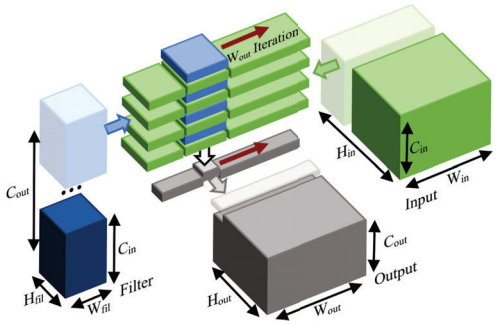


Figure 3: Visualization of the 2D convolution operation. We can observe that iterating over dimension W_{out} (Red arrow) requires access of new input and output elements, but allows the reuse of the filter elements. Batch dimension (B) is omitted for brevity.

Algorithm 1 Naïve Convolution Algorithm

Input: Input tensor I, Filter tensor W
Output: Output tensor O, Stride s

for $i = 0, \ldots, B-1$ **do**
  for $j = 0, \ldots, C_{out}-1$ **do**
    for $k = 0, \ldots, H_{out}-1$ **do**
      for $l = 0, \ldots, W_{out}-1$ **do**
        for $m = 0, \ldots, C_{in}-1$ **do**
          for $n = 0, \ldots, H_{fil}-1$ **do**
            for $o = 0, \ldots, W_{fil}-1$ **do**
              $h_{in} = k + n \times s$
              $w_{in} = l + o \times s$
              $O_{i,j,k,l}$ += $I_{i,m,h_{in},w_{in}} \times W_{j,m,n,o}$

Winograd Transformation[25] accelerates the convolution operation by reducing its computational complexity. Unfortunately, such transformation-based approaches lack flexibility[20, 35], and the transformation processes cause overhead in both computation and memory[25, 32]. Given these limitations and overheads, approaches that reduce the complexity are becoming less attractive[52], especially in mobile platforms with small workload sizes.

A more popular approach is through matrix multiplication (GEMM). In this approach, convolution tensors and dimensions are mapped into matrices, which are then piped into a high-performance BLAS library[1, 5]. Modifications to the convolution operation can be easily incorporated into the mapping process, as various mapping patterns. The main benefit of this approach is that one could effortlessly leverage the high computational throughput provided by the GEMM kernels, by simply piping the transformed tensors to the BLAS library of the given platform.

2.2 The GEMM operation

GEMM operations are expressed as a multiply-accumulate (MAC) computation $C_{i,j} = A_{i,k} \times B_{k,j}$ in nested for loops over 3 computation dimensions, M, N, and K. i, j, and k are iterators over dimension M, N, and K respectively, and $A_{i,k}$, $B_{k,j}$, and $C_{i,j}$ are elements from matrix $A_{M\times K}$, $B_{K\times N}$, and $C_{M\times N}$, respectively. This operation can be understood as a set of independent inner product computations between two sets of vectors, which provides a large opportunity for both parallelization and data reuse.

For example, from Figure 2, we can observe that each output element $C_{i,j}$ is computed as an inner product of column vector $B_{:,j}$ and row vector $A_{i,:}$. As no output elements are required for the computation of another, multiple output elements can be computed in parallel. If we parallelize the computation of output elements over the dimension N, row vector $A_{i,:}$ are shared among the computations, allowing data reuse between parallel computation units. Data reuse exists within a single computation unit as well, by reusing output element $C_{i,k}$ over the inner product dimension K.

One could easily observe that there exist abundant amounts of parallelization and data reuse opportunities over different operand matrices and computation dimensions. Existing GEMM computation researches[8, 13, 14, 27, 41] and libraries[21, 33, 50] exploit these opportunities by carefully partitioning and scheduling the operation in a way that facilitates the most efficient use of memory hierarchy and vector processing capabilities of the underlying hardware.

2.3 Mapping convolution into GEMM

For the convolution operation to take benefit of the highly optimized BLAS libraries, we must first map the convolution tensors to matrices and the convolution computations into matrix multiplications. Details are as follows.

Notations: We will refer the convolution tensors as *Input*, *Filter*, *Output*, instead of *Feature Map* or *Kernel*. Input and output tensor dimensions are denoted as B, C_{in}, H_{in}, W_{in} and B, C_{out}, H_{out}, W_{out} respectively, corresponding to Batch, Channel, Height, and Width dimension of corresponding tensor. Similarly, filter tensor dimensions are denoted as C_{out}, C_{in}, H_{fil} and W_{fil}. The convolution operation can be expressed as a MAC computation in nested for loops over 7 computation dimensions, with elements from input, filter, and output tensor as operands, as shown in Algorithm 1.

Difficulties: Mapping tensors to matrices are easy, as both operations operate over two input and one output operands. We map the output tensor to matrix C, the input tensor to matrix B, and the filter tensor to matrix A. Mapping of the input and filter tensor is arbitrary and can be exchanged. As the total number of computations must remain the same after mapping to a matrix, the product of all convolution dimension sizes must equal that of the GEMM dimensions. This means that during our mapping process the convolution dimensions must be flattened and distributed among the GEMM dimensions. However, this is trickier than mapping operands. We cannot map an arbitrary convolution dimension to a matrix dimension, as the *relationship between dimensions and operands* must be respected.

Operand	Convolution		GEMM	
	Access	Reuse	Access	Reuse
Output	$C_{out}, B, H_{out}, W_{out}$	C_{in}, H_{fil}, W_{fil}	M, N	K
Input	$C_{in}, B, H_{out}, W_{out}$	$C_{out}, H_{fil}, W_{fil}$	K, N	M
Filter	$C_{out}, C_{in}, H_{fil}, W_{fil}$	B, H_{out}, W_{out}	M, K	N

Table 1: Relationship between tensors/matrices and computational dimensions. Note that H_{fil} and W_{fil} are Reuse dimensions of two operands, while GEMM dimensions M, N, K are Reuse dimensions of only one operand.

If not, the mapped dimension will iterate over the wrong operand, resulting in an incorrect computation. To achieve correct results, we must identify what relationships the 7 convolution dimensions and the 3 GEMM dimensions have with the operands, and map the dimensions with the same relationship.

Relationship between dimensions and operands: We first classify the relationship between a dimension and an operand as either a Reuse or an Access. Access dimensions are the dimensions that are involved in determining the element of the given operand in the MAC operation. Conversely, Reuse dimensions are the dimensions that are not involved.

For example, in Figure 2, we can observe that iterating over dimension K allows the reuse of the same output element $C_{i,j}$, while requiring access to new input and filter elements. Therefore dimension K would be a Reuse dimension of the output matrix, while being an Access dimension of input and filter matrices. In the same way, in Figure 3, we can observe that iterating over dimension W_{out} allows the reuse of the same filter elements, while requiring access to new input and output elements. Dimension W_{out} would be a Reuse dimension of filter tensor, while being an Access dimension of input and output tensors.

In both GEMM and convolution, we can observe that the input, filter, and output element used by the MAC computation is determined by a set of iterators corresponding to the computation dimensions. Using this as a reference, we can easily determine whether a dimension is involved in determining the element of a given operand.

Table 1 shows the Access and the Reuse dimensions of each tensor and matrix. Most of the Access and the Reuse relationships can be identified by referring to the computation algorithms, except for relationships regarding the input tensor. For the following analysis, we put the Reuse dimension of the input tensor as C_{out}, H_{fil} and W_{fil}, leaving the Access dimension to be B, C_{in}, H_{out}, W_{out}. This is based on the general cases of convolution where input elements are visited by all spatial elements of the filter. We analyze and build the general structure of our algorithm based on this observation, and incorporate the accurate relationship of the input in the detailed implementation steps in Section 4.

Mapping computation dimensions: From the table we can easily identify that dimensions B, H_{out}, and W_{out} shares the same relationship with GEMM dimension N, being a Access dimension of Output and Input operand, and Reuse of Filter operand. This allows direct mapping of dimension B, H_{out}, W_{out} to dimension N, as the elements and computations governed by the listed dimensions will follow the same pattern when mapped to N. In the same way, C_{out} can be mapped to M, and C_{in} can be mapped to dimension K. However, it is apparent that the spatial filter dimensions, H_{fil} and W_{fil}, **cannot** be mapped to any of the GEMM dimensions, as they are Reuse dimensions of two operands, while GEMM dimensions M, N, and K are Reuse dimensions of only one operand each.

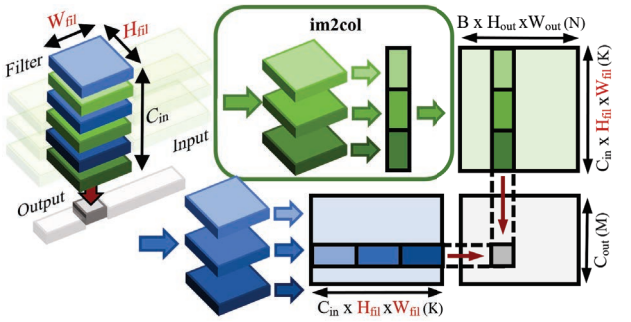


Figure 4: Visualization of im2col + GEMM. Computation of a single output element becomes an inner product of input column vector and filter row vector, and the computation of all output elements becomes matrix multiplication.

2.4 Existing Implementations

In order to solve the problem of spatial filter dimensions and successfully compute the convolution operation using matrix multiplication, various approaches and implementations have been proposed over the years.

Im2col (Image to Column): One of the most widely used approaches is im2col [5]. Im2col flattens all input elements required for calculation of a single output element into a column vector of size $C_{in} \times H_{fil} \times W_{fil}$. With the corresponding flattened filter row vector, the computation of a single output element is now an inner product of the input column and filter row vector. When this is repeated for all $C_{out} \times B \times H_{out} \times W_{out}$ output elements, the resulting set of inner product computations take a form of matrix multiplication, as depicted in Figure 4.

Unfortunately, as each input element is reused over $H_{fil} \times W_{fil}$ neighboring output element computations, there exists $H_{fil} \times W_{fil}$ duplicate entries of the same input element among the column vectors of the input matrix. This forces the input matrix size to be $H_{fil} \times W_{fil}$ times bigger than the original.

Im2col can be understood as expressing the Reuse of the input elements over H_{fil} and W_{fil} dimensions as a separate, duplicated entry in the memory, modifying the relation of H_{fil}, W_{fil} dimensions and the input from Reuse to Access. This allows mapping of H_{fil} and W_{fil} to matrix dimension K, as now they are Access dimensions of both input and filter, while being Reuse dimension of only the output, just like the K matrix dimension. However, duplicating the input entries not only increases the footprint of the input matrix to be $H_{fil} \times W_{fil}$ times that of the original tensor, but also removes the input data reuse opportunity that these dimensions provide, stressing both memory footprint and bandwidth.

Im2col Optimization: MEC[6], Indirect Convolution[9], and Implicit GEMM[34, 46], are all based on im2col, and they propose various techniques to reduce the impact of memory duplication and reordering of im2col. MEC aims to remove overlapping portions of the im2col matrix in the H_{out} dimension, while Indirect Convolution uses pointers to the elements instead of duplicating the

elements. Implicit GEMM tries to hide the overhead by only creating small portions of im2col input matrix usually in the size of high bandwidth, private memories, and interleaving the process with matrix packing. Implicit GEMM is the convolution method NVIDIA's cuDNN[34] deploys, while Indirect GEMM is the method Google's XNNPACK[12] library deploys. All of these methods reduce the amount of overhead and throughput loss to some extent. However, no methods remove these problems completely, and the low data reuse rate of GEMM remains to be a serious problem in mobile environments.

Kn2col (Kernel to Column): Similarly to im2col, Kn2col [1] modifies the relationship H_{fil} and W_{fil} have with the output tensor, moving them from the `Reuse` to `Access` of the output tensor. This allows mapping of H_{fil} and W_{fil} to matrix dimension M. However, during this process, kn2col also increases the output matrix footprint to be $H_{fil} \times W_{fil}$ times that of the original tensor, and removes the data reuse opportunity of the output data.

Direct convolutions: Direct convolution approaches[11, 52] claim to compute convolutions directly, instead of piping the computation into an external matrix multiplication routine. However, a close examination of their computation algorithms reveals that their computations are also based on matrix multiplications[11, 52]. In the essence, what these approaches have done is to partition the convolution computation over H_{fil} and W_{fil} dimensions. This allows each partition to be composed of only C_{out}, C_{in}, B, H_{out}, and W_{out} dimensions, which can be directly mapped into GEMM.

This partitioning approach removes the duplication and memory overhead that im2col and kn2col suffer from modifying H_{fil} and W_{fil} dimensions, but the problem of being unable to utilize the additional reuse that H_{fil} and W_{fil} dimensions provide remains.

2.5 Problems in GEMM-based solutions

From our analysis, we have seen that the convolution dimensions of H_{fil} and W_{fil} cannot be directly mapped to any of the matrix dimensions. To express these dimensions in GEMM, existing solutions deploy various workarounds and modifications, which cause overheads or throughput losses, as described in the previous section. This is inevitable given that although matrix multiplication does provide abundant data reuse opportunities, it is still less than what the convolution operation provides.

As explained in Section 1, these overheads and data reuse losses are less of a problem on high-performance systems, given large batches of workloads. However, in mobile systems, where both the capabilities of the memory system and the workload size is limited, the problems in GEMM-based solutions are non-negligible.

3 mGEMM

3.1 Motivation

The fundamental problem in mapping the convolution operation to GEMM is that the computation dimensions of matrix multiplication are too limited to express all dimensions of the convolution operation; A much bigger operation is being squeezed into a smaller operation, and some dimensions do not even fit.

However, instead of trying to squeeze convolution dimensions into the static GEMM computations, what if we extend the GEMM computation themselves, by adding more computation dimensions into the algorithm with properties of H_{fil} and W_{fil} dimensions?

Based on this idea, we study how current GEMM kernels are designed and expand on their structure. We add an additional computational dimension to represent the convolution dimensions, while utilizing the optimization strategies and hardware support the GEMM libraries use. Through this, we see that it becomes possible to accommodate the convolution operation without any need for preprocessing or modifications to the data, setting us free from the overheads and throughput losses the existing solutions have suffered from.

Also, our modification does not change the computation result of the convolution. Mathematically, our method of computation is equivalent to the existing GEMM-based convolution method, performing the same set of computations on the same data. While the order of the computations is different, commutative and associative properties ensure that the results remain unchanged.

3.2 Design Goals

Our design goal is to create a high-performance convolution solution based on the structure and optimizations of the current GEMM-based solutions, but with an additional dimension with properties of the convolution filter spatial dimension. To achieve this, we take the structure and techniques of exiting high-performance GEMM algorithms[13, 14, 27, 41], and restructure them to suit our design.

The structure of GEMM routines is composed of two parts: the inner computation kernels and the loops outside the inner kernel. We also design our solution in two parts accordingly. Below, we explain the goals of our designs in each part of our algorithm's structure.

Inner computation kernels: Our goal on the inner computation kernel is to provide a block of highly optimized computation that can be repeated to complete our algorithm. To achieve this, we focus on a technique called *Tiling*. Tiling is an approach where we create a small tile of the target computation, convolution in this case, with a set of fixed size computation dimensions, or blocking parameters. We search for the blocking parameter values that allow for maximal utilization of the hardware and minimize memory accesses, under the constraints of the given architecture and algorithm. With these values found, we can create a high-performance code optimal for the given blocking parameter values, by leveraging various hardware features and optimization used by the GEMM kernels, such as vectorization or fused-multiply-accumulate (FMA). This becomes our inner kernel, which we can tile to fill our target convolution operation.

Loops outside the inner kernel: Our goal on the loops outside of the inner kernel is to partition the computation so the inner kernels can be invoked in the most efficient manner. The computation loops in this level must facilitate optimal memory movement in different hierarchies of memory, and allow for higher-level parallelism, such as multi-threading. To achieve this, we apply *loop ordering* techniques on this level. We carefully analyze the computation and decide on which data should be placed on which level of the memory hierarchy, to minimize the memory movement between the levels and to encourage parallelism and data reuse between computation units. Based on this decision, we partition and order each computation dimension loops accordingly.

Memory layout of the tensors is also critical for high-performance computations as it allows us to minimize the stride between memory

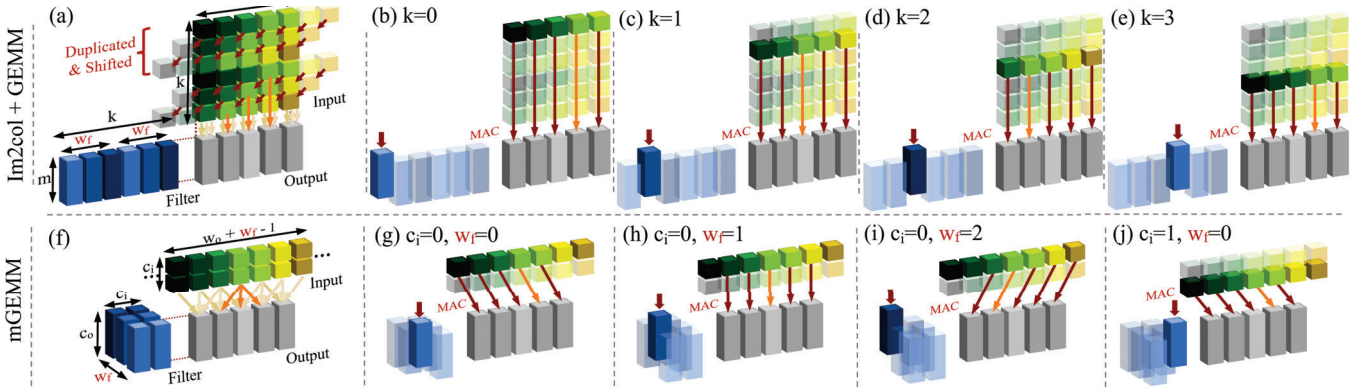


Figure 5: (a) and (f) illustrate the operation of im2col+GEMM and mGEMM, respectively. (b) to (e), and (g) to (j) show four equivalent iterations of GEMM inner kernel operation and mGEMM inner kernel operation, respectively. In (b) to (e) and (g) to (j) non-transparent blocks represent the data loaded in the register, and arrows from input to output depict the MAC computations carried out from the loaded data. In im2col+GEMM the w_f dimension is flattened with c_i into the K matrix dimension, causing the Reuse of input elements from the w_f dimension to be expressed as duplication & shift of the input elements. In mGEMM, however, w_f is expressed as a separate dimension, allowing actual reuse of the input elements, and providing more computation per data loaded.

accesses, achieving greater access locality and ultimately lowering the memory latency. Based on the computation loop order of our algorithm, we search for the memory layout that can minimize the stride and better support the hardware traits such as SIMD vector lane sizes or cache line sizes.

Note that we only focus on the algorithmic structure of our design in this section. Implementation details such as selecting the value of blocking parameters and generating the code of the inner kernel are covered in Section 4 as it is tightly coupled with the underlying hardware.

3.3 Designing the Inner Computation Kernel

3.3.1 Tiling to maximize data reuse. Tiling partitions the computation operands into smaller blocks, namely tiles, that fit in the size of desired memory level. This allows the block of computation to be processed without any further memory access to the lower level, maximizing data reuse in the given memory hierarchy. We determine the size of our tiles using blocking parameters b, c_o, h_o, w_o, c_i, h_f, and w_f, each defining a partition of the original convolution dimensions.

For the given blocking parameters, a small tensor with size $b \times c_o \times h_o \times w_o$, $b \times c_i \times h_o \times w_o$, and $c_o \times c_i \times h_f \times w_f$, each being a subset of output, input, and filter tensor, are loaded into registers. We will call these subset tensors subtensors. From the loaded subtensors, total of $b \times c_o \times h_o \times w_o \times c_i \times h_f \times w_f$ MAC computations can be performed. If we divide the number of MACs with the size of each loaded subtensors, we can see that each element of the output subtensor is being reused in $c_i \times h_f \times w_f$ MACs for the output, $c_o \times h_f \times w_f$ MACs for the input, and $b \times h_o \times w_o$ MACs for the filter, just like in Table 1.

Without any reuse, the number of memory operations per MAC is 4: 2 for output, 1 for input, and 1 for filter. However, when we incorporate the reuse that we gained from tiling, this is reduced down to $\frac{2}{c_i \times h_f \times w_f} + \frac{1}{c_o \times h_f \times w_f} + \frac{1}{b \times h_o \times w_o}$. As the total number of MAC computations is fixed for a given convolution layer, our interest in the inner kernel now goes to minimizing this memory operation per MAC by selecting the optimal combination of blocking parameters.

3.3.2 Tiling parameter simplification. In Section 2, we observe that dimensions B, H_{out}, and W_{out} share the same dimensional traits, and can be flattened and mapped into a single dimension. As such, we set $b = 1$, $h_o = 1$, and focus on w_o in our kernel. The reason we choose w_o is because W_{out} is the fastest (innermost) dimension in the memory order between the three dimensions. If $b \neq 1$ or $h_o \neq 1$, there would exist large memory stride while loading elements over these dimensions, inducing cache misses.

We also focus on utilizing the W_{fil} dimension, and not H_{fil}, in our kernel. Due to the limitations of the current ARMv8 architecture and its vector extension NEON, there are not enough registers to efficiently express both W_{fil} and H_{fil} in the inner kernel. We explain more about this issue in Section 4.

After simplification, parameters c_o, w_o, c_i, and w_f remain. The memory operation per MAC can now be expressed as $(\frac{2}{c_i \times w_f} + \frac{1}{c_o \times w_f} + \frac{1}{w_o})$, and we have to minimize this memory operation per MAC, to maximize data reuse. For this, we search for the optimal parameter values in Section 4, and compare our improved memory operation per MAC with that of the original GEMM kernel.

In Section 2, we also observe that dimensions C_{out}, W_{out}, and C_{in} can be directly mapped to matrix dimensions M, N, and K. As a result, the design of our inner kernel with parameters c_o, w_o, c_i, and w_f can be interpreted as a matrix multiplication kernel with dimension W_{fil} added, to represent the Access and Reuse relationships that are not present in M, N, and K. This is why we name our method an expansion of the GEMM kernel; In cases where $w_f = 1$ it is identical to the GEMM kernel, but with $w_f > 1$ it becomes equivalent to performing multiple GEMM computations while iterating over w_f, which becomes a 1D convolution computation kernel on the loaded subtensors.

Algorithm 2 mGEMM Algorithm[1]

Input: Input tensor I, Filter tensor W
Output: Output tensor O, Stride s
1: **for** $i = 0, \ldots, B-1$ **do**
2: **for** $j_1 = 0, \ldots, C_{out} \div (cc_o \times c_o) - 1$ **do**
3: **for** $k = 0, \ldots, H_{out} - 1$ **do**
4: **for** $l_1 = 0, \ldots, W_{out} \div w_o - 1$ **do**
5: **for** $n = 0, \ldots, H_{fil} - 1$ **do**
6: **for** $j_2 = 0, \ldots, cc_o - 1$ **do**
7: **for** $m = 0, \ldots, C_{in} - 1$ **do**
8: **for** $o = 0, \ldots, W_{fil} - 1$ **do**
9: **for** $l_2 = 0, \ldots, w_o - 1$ **do**
10: $l = l_1 \times w_o + l_2$
11: **for** $j_3 = 0, \ldots, c_o - 1$ **do**
12: $j = j_1 \times cc_o \times c_o + j_2 \times c_o + j_3$
13: $h_{in} = k + n \times s$
14: $w_{in} = l + o \times s$
15: $O_{i,j,k,l}$ += $I_{i,m,h_{in},w_{in}} \times W_{j,m,n,o}$

3.4 Designing the Outer Loops

3.4.1 Loop Ordering. The inner kernel loads all subtensors corresponding to the blocking parameters and executes all available MAC computations using SIMD FMA instructions. After the execution, we have to make a choice on which dimension to iterate over.

The obvious choice is W_{fil}. W_{fil} is the Reuse dimension of *both* the input and output subtensor, allowing maximum data reuse from the previous inner kernel execution. After iterating over W_{fil}, we choose to iterate over C_{in} and reuse the output elements again, as they require 2 memory operations per MAC unlike input or filter elements.

We can set $w_f = 1$ in the inner kernel, as iterating over W_{fil} gives the full W_{fil} reuses to both input and output subtensor loaded in the registers. Similarly, we can set $c_i = 1$ in the inner kernel, as iterating over C_{in} gives the full C_{in} reuses to output subtensor. As shown Figure 5 (g) to (j), this allows us to reuse the output subtensors over W_{fil} and C_{in}, and input subtensors over W_{fil}, without allocating registers to represent them, reducing the use of registers and allowing for larger tiling values for c_o and w_o.

While $c_i = 1$ is optimal, use of vectorized load instructions requires c_i to be a multiple of SIMD vector lane size, 4. Again, we explain more about this issue Section 4.

For the outermost loops, we select W_{out}, H_{out}, and B dimensions. These dimensions are the Access dimension of the output tensor and therefore have no dependency among indexes. They can be partitioned and distributed among different threads for multi-threaded execution. Also, as these dimensions are Reuse dimensions of only the filter tensor, we naturally block filter elements on the shared, last level cache, so that different threads can share the data.

As input elements are accessed more often than output elements, and as input subtensors often cause large stride when iterating over C_{in} in the inner kernel, we block input elements in the L1 cache. This puts the dimensions C_{out} and H_{fil} in the middle of the loops, as they are the Reuse dimension of the input tensor.

Unfortunately, due to the limited cache sizes of mobile processors, filter tensors often do not fit in the last level cache in layers with large channel sizes. To compensate for this, we divide C_{out} dimension again using a parameter cc_o, so that the blocked filter size is smaller than that of the last level cache. The remaining C_{out} loops are placed as the outermost loop before B, to be distributed among different threads and to allow reuse of the filter data in the last level cache as much as possible.

The final algorithm is shown in Algorithm 2. Note that lines 8 to 13 are unrolled and executed using SIMD vector instructions.

[1] Inner kernels with smaller (less optimal) parameter sizes are also provided to process the remaining computations if the convolution dimensions are not exactly divisible with the most optimal sizes.

3.5 Memory Layout

Minimizing the stride between memory accesses increases the locality of the access, increasing utilization and hit rates on the cache. It also increases prefetcher accuracy, ultimately minimizing the latency of each access. To achieve this, we modify the memory layout of the tensors.

Input and output tensors are modified from the conventional N-C-H-W or N-H-W-C to N-(C/c_i)-H-W-c_i, where c_i is the input tensor channel blocking parameter, constant over all layers and kernel implementations. Although the NHWC order does work with the proposed Algorithm 2, we decided to split the channel dimension into small, SIMD vector lane size (c_i) chunks, similar to [26, 52]. This is due to the channel dimensions often having sizes in powers of 2, which causes critical stride between elements with neighboring width index. By splitting the channel dimension, these elements gain locality, while allowing vectorized memory access in the channel dimension. There is no need for any memory reordering during execution, unlike previous solutions. As all input and output tensors share the same layout, the output can be directly used as an input for the next layer.

Filter tensor is modified from the conventional K-C-R-S to (K/c_o)-R-C-S-c_o, to assure unit stride while iterating over the H_{fil} - C_{in} - W_{fil} - c_o loops shown in line 5 to 11 of Algorithm 2. This is especially important considering that filter values are replaced most often in our algorithm. To keep the constant stream of filter values into the registers, sequential access of the filter values is crucial, as it reduces the latency of each access.

3.6 Pointwise and Depthwise Convolution

Many modern CNNs designed for mobile such as MobileNet[19, 39] or MnasNet[43] deploy pointwise and depthwise convolutions. Pointwise convolution can be understood as convolution with 1×1 filter spatial dimensions, and depthwise convolution can be understood as a set of independent single-channel 2D convolutions.

Pointwise convolutions are directly mapped to the GEMM kernel if H_{fil} and W_{fil} are both 1. Our mGEMM kernel can also be reduced to a GEMM kernel when $W_{fil} = 1$. This means that in pointwise convolutions, algorithmic differences between mGEMM and existing GEMM are minor. Therefore, we do not discuss improvements on the pointwise kernel, as it is better suited for GEMM optimizations.

However, in depthwise convolution, our idea of utilizing the Reuse characteristics of H_{fil} and W_{fil} dimensions have a huge impact. In depthwise convolutions, computations in different channel iterations are completely independent of each other and require the replacement of all output, input, and filter elements. This means that C_{in} and C_{out} dimensions no longer provide Reuse for the output and input elements, unlike the traditional convolution. Therefore, the only Reuse opportunities of output and input elements that remain are H_{fil} and W_{fil} dimensions.

To better utilize this property, unlike the conventional convolution kernel where we choose to reuse output and input subtensors, we restructure the algorithm to reuse filter elements in the inner kernel and loop our computations accordingly. The reason for this is that the number of filter elements required for a complete depthwise computation of a single output channel is only $H_{fil} \times W_{fil}$, which is often small enough to fit entirely in the registers. We load all $H_{fil} \times W_{fil}$ filter elements needed for c_o channels, and Reuse the filter elements over all of $B \times H_{out} \times W_{out}$ computation. Input and output elements are streamed from the memory, as they only need to be used once with the filter element in the register, and not with the other filter elements. This greatly increases the data reuse, achieving a much lower latency compared to that from existing solutions, as can be seen in Section 5.

4 DETAILED IMPLEMENTATIONS

To create a truly high-performance implementation of the algorithm, there are two tasks that must be done to complete our implementation: 1) finding the parameter values that are optimal for the given hardware, and 2) creating a code that fully utilizes the features and the capabilities of the underlying hardware.

4.1 Understanding ARMv8-A and NEON

We focus on ARMv8-A[2] architecture, the de facto standard on mobile platforms, for our implementation, but this can be done with any hardware. The main feature of ARMv8 most relevant to the performance of our algorithm is its vector processing capabilities, namely the NEON SIMD extension. NEON provides 32 128-bit vector registers which can be used in 8, 32, or 64-bit mode. As NEON vectorized load instruction requires that data must be loaded from sequential memory space, this means that we can hold up to 32 vectors with 4 sequential elements, or 128 elements, when using 32-bit float for our data. NEON also provides FMA, which is an optimized hardware implementation of the MAC operation. To fully saturate the computational pipeline of ARMv8-A, it is required to provide enough number of independent FMA instructions. The exact number of instructions required differs by the mircroarchitecture, but more than 8 instructions are generally considered enough. This means that we need at least 8 vector registers for the output elements, or $c_o \times w_o \geq 32$. This also provides enough FMA instructions for load and store instruction to be interleaved in-between, allowing memory operations and computations to be executed in parallel, exploiting the instruction-level parallelism (ILP) capabilities of the ARMv8-A architecture.

4.2 Kernel Parameter Selection

As mentioned in Section 3, our goal is to find the combination of blocking parameters c_i, c_o, w_o, and w_f that minimizes the memory operation per MAC, $(\frac{2}{c_i \times w_f} + \frac{1}{c_o \times w_f} + \frac{1}{w_o})$. There also are architectural limitations:

- **Available registers:** $c_o \times w_o + c_i \times w_o + c_o \times w_f \leq 128$ [2]
- **Saturating the pipeline:** $c_o \times w_o \geq 32$ [3]
- **Vectorized load granularity:** c_o and c_i must be a multiple of the SIMD vector lane size 4, being the innermost dimension on memory.

In Section 3, our optimization allowed us to set c_i to 1 while utilizing the whole C_{in} for output register data reuse. Unfortunately, limitations of vectorized load instruction of NEON forces c_i to be a multiple of 4, consuming *four* times more registers for input elements than what is needed.

ARM's next-generation vector extension, the Scalable Vector Extension (SVE)[42], provides *gather* type vector load instructions that completely solve this issue. We believe that with the additional register space freed by using SVE, we will be able to include H_{fil} dimension in our kernels, creating a more complete and much more efficient version of the mGEMM kernel than the current one with only W_{fil} dimension added. Note that, SVE is currently a non-standard feature in ARMv8-A, but is expected to be widely available in the next generation ARM architecture, ARMv9.

We searched the parameter space and found $c_o = 8$, $c_i = 4$, $w_o = 8$, and $w_f = 1$ achieve the best results, with a memory operation per MAC value of $(\frac{2}{C_{in} \times W_{fil}} + \frac{1}{8 \times W_{fil}} + \frac{1}{8})$, giving about 0.169 loads/stores per MAC when $W_{fil} = 3$ and $C_{in} = 256$. If we translate this into flops/byte, we get $\frac{2}{0.169 \times 4} = 2.96$, on the register level.

This value is much higher than what matrix multiplication kernels can achieve. If we look at single-precision GEMM kernel with the minimum memory access per MAC value available on ARMv8, the blocking parameters are $m = 8$, $n = 12$, and $k = 1$, which the memory access per MAC comes to $(\frac{2}{K} + \frac{1}{8} + \frac{1}{12})$, roughly 0.211 when $K = 3 \times 256 = 768$. if we translate this into flops per byte, we get $\frac{2}{0.211 \times 4} = 2.37$, on the register level. This means that mGEMM processes about 25% more computations per byte loaded, which is roughly in line with the level of performance increases observed in our experimental evaluations.

4.3 Automated inner kernel generation

Until now we have conveniently treated the spatial dimensions of the input tensor to be equal to that of the output tensor. In the actual implementation of the kernel, the spatial position of the correct input element must be determined by a complicated relationship of various parameters.

Fortunately, while the computations to determine the position is complicated, the position is fixed once the required parameters are set. Using this characteristic, we fix the w_f, c_o, w_o, stride, dilation, and padding for each inner kernel so the position of the correct input element can be embedded in the inner kernel. To easily generate inner kernels for various layer configurations, we write a Python script that takes the parameter values and outputs a correct AArch64 (ARMv8-A 64bit ISA) assembly inner kernel code, with hardware optimizations such as FMA, vectorized memory access, and prefetch instructions interleaved in an optimal sequence. We use this script to generate all of our inner kernels except for the depthwise kernels for which we use a separate modified version of the original script.

The script works by first calculating the input spatial positions that each output spatial position requires, using the parameters w_f, w_o, padding, stride, and dilation. The script detects overlapping input spatial positions and maps one 128-bit vector register for each unique input position. The script then maps the appropriate number of

[2] Maximum number of 32-bit float data that can be stored in the registers
[3] Minimum number of output elements required to saturate the FMA pipeline

	Raspberry Pi 4	Odroid N2+	Pixel 4
SoC	BCM2711	S922X	Snapdragon 855
Core Config.	A72 Quad	A73 Quad + A53 Dual	A76 Quad + A55 Quad
Memory	4GB LPDDR4	4GB DDR4	6GB LPDDR4X
OS	Raspberry Pi OS (64bit)	Ubuntu 18.04	Android 11

Table 2: Specifications of the test devices.

vector registers for the output values, typically $w_o \times 2$ with $c_o = 8$. The script utilizes the remaining vectors for the filter access, which at least two registers should remain for proper interleaving of computation(FMA) and memory access instructions. If the 32 vector registers provided by the ARMv8 architecture cannot satisfy the above allocations, the script concludes that the given parameters are too big for the hardware, and fails. The script can be run inside a nested loop covering the parameter space to find the most optimal parameter combination for the given hardware.

If there are enough registers, the script first creates a register map that holds info on which register is holding which data of the subtensors. It then starts printing ARMv8 assembly instructions following the lines 8 to 13 of Algorithm 2, with the referenced register of each instruction determined by the register map. FMA instructions referencing the same output data are placed as far away as possible, and data on a register is replaced with a new load instruction as soon as all FMA instructions using that data are placed. FMA instructions that are not dependent on the new load instruction are placed after, to hide the memory access latency. L1 data prefetch instructions are placed in-between the FMA instructions, and away from memory access instructions. They are with an address bias of +256 from the nearest loaded address to allow just-in-time access of data and uniform interleaving of computation and data access instructions.

We used this script to search and generate the kernel with the most optimal parameter combinations, which are the values explained in Section 4.2. We also generated kernels with smaller parameter sizes than that of Section 4.2, to fill the gaps when the kernel with the optimal combination is too big for the required computation tile.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

To conduct a thorough and comprehensive evaluation of our algorithm, we design our experiments focusing on three different aspects.

Test Devices: There exist a huge range of mobile devices, each varying in many different aspects. Our test devices are selected to cover a wide range of mobile devices, starting from a cheap single-board computer to a flagship smartphone. We use off-the-shelf devices in their stock settings, in order to ensure that the results show the true benefits that existing devices would gain from using our algorithm.

Software: There are various convolution solutions available in ARM. We include both the widely used solutions and the competitive solutions in our tests for comparison. Results in time, memory, and energy are given, not only to show the excellence of our solution, but also to speculate on multi-dimensional aspects of mobile computing.

Benchmarks: We test convolution layers of three different CNNs, namely VGG-16 [40], ResNet-50 [17], and MobileNet V2 [39]. We also include results on an end-to-end application task, using YoloV3-Tiny [38], an object detection network. Each of the CNNs we select represents a classical, a modern, and a mobile-focused CNN design. We do not limit our tests on the mobile-focused CNNs for the sake of generality, as many networks are designed without a particular focus on mobile and are still deployed on mobile.

5.1.1 Test Devices. We run our experiment on three ARM-based platforms, with differing levels of hardware performance, each representing low-cost, mid-tier, and higher-end platforms, respectively. Specifications of each device are in Table 2.

5.1.2 Software. We test our algorithm against various existing GEMM-based convolutions, starting from the baseline im2col + GEMM, two high-performance DNN libraries, and a direct convolution method we write ourselves in C++, based on [52].

im2col + OpenBLAS: We implement the im2col routine in C++, based on the original im2col code from Caffe[23]. We apply multi-threading to im2col using OpenMP[7], and piped the result to a high-performance GEMM routine provided by OpenBLAS[50] library. This approach of combining OpenBLAS with an im2col routine is used as a reference solution in many recent studies [1, 20, 52, 54]. It provides a low-effort and good-performing implementation of a GEMM-based convolution.

ARMNN: ARMNN[3] is ARM's own high-performance DNN library, designed to provide various neural network routines for higher-level NN frameworks. As the backend of ARMNN, we have used ARM's Compute Library[4]. Compute Library is ARM's collection of high-performance computation kernels optimized for ARM CPU and GPU designs.

XNNPACK: Another high-performance neural network library we use is Google's XNNPACK[12]. XNNPACK provides much faster performance compared to the default backend of TensorFlow Lite, Ruy, with hand-coded kernels optimized to the microarchitecture level[10]. While TensorFlow Lite and Ruy are more popular, we choose XNNPACK for a more competitive comparison.

Our comparison targets all include various computation kernels highly optimized for a wide range of hardware, algorithms, and formats. To provide an even ground on the comparisons, we compile these libraries to only use kernels targeted for CPU, GEMM-based solutions, and in single-precision floating-point. Optimization techniques that alter the accuracy of the tests, such as pruning or quantization, are turned off, but the ones that do not, such as hardware accelerations (e.g., NEON SIMD acceleration, multi-threading) or GEMM optimizations (e.g., tiling, array packing), are left untouched.

All tests are run with the batch size of one while utilizing all available cores, to represent typical mobile scenarios. Memory layout of NHWC was used for the im2col + OpenBLAS, XNNPACK, and ARMNN tests, to achieve the best possible performance.

As mentioned in Section 3, as all computation methods are mathematically equivalent, the results should remain unchanged, except for the minor differences from floating point representations. In all of our test cases, we confirm that the result and the accuracy of the network remain the same for all computation methods.

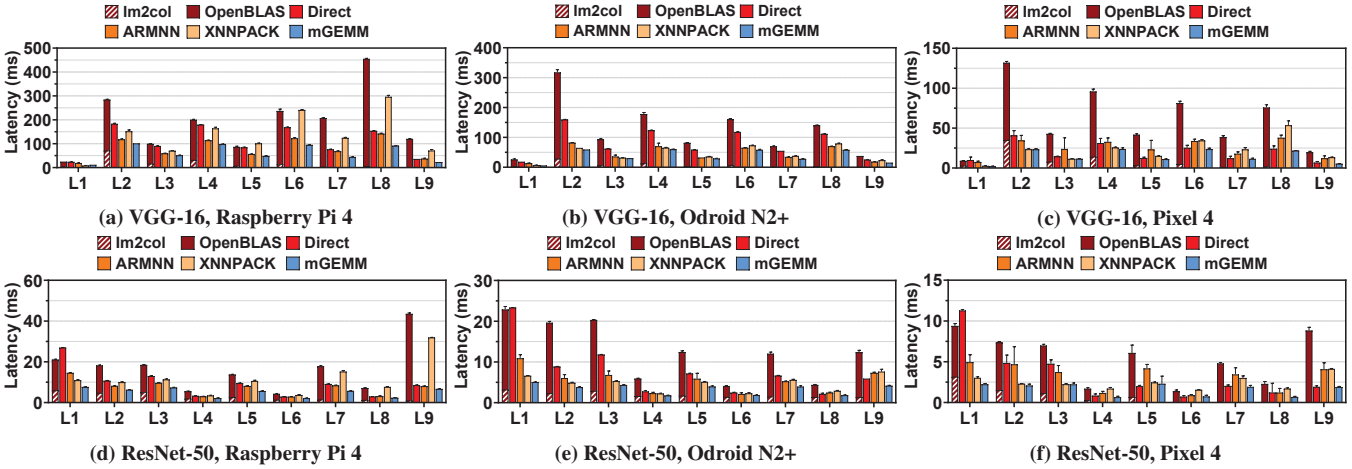


Figure 6: Average layer latency of different non-pointwise convolution layers of VGG-16[40] and ResNet-50[17]. All results are wall-clock time, averaged from 1000 runs, with batch size of one.

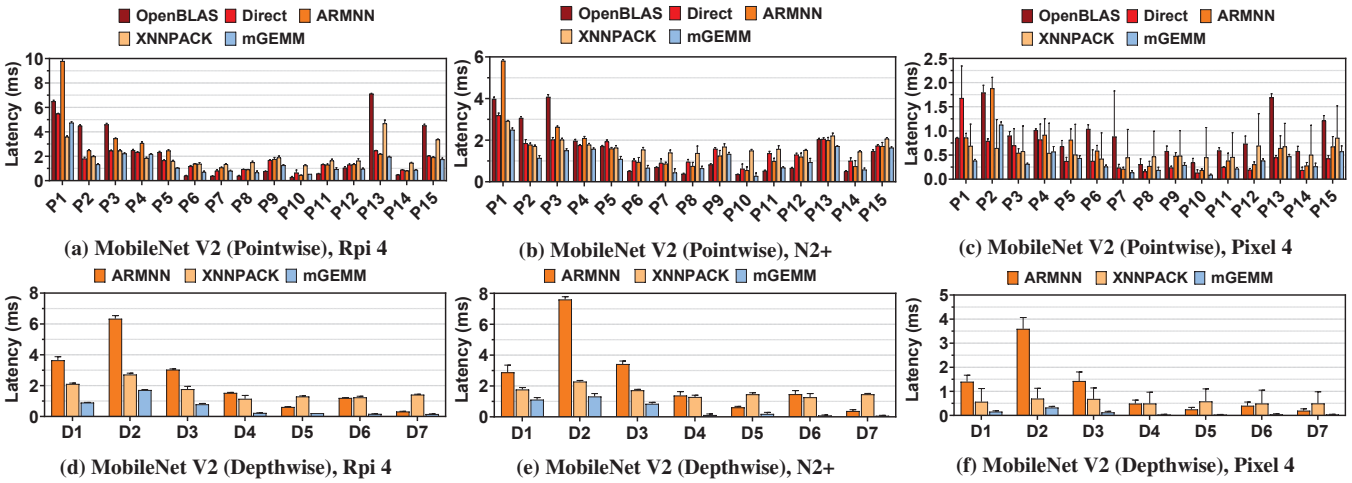


Figure 7: Average layer latency of different pointwise layers and depthwise layers of MobileNet V2(1.4×)[39]. All results are wall-clock time, averaged from 1000 runs, with a batch size of one.

5.2 Performance Evaluation

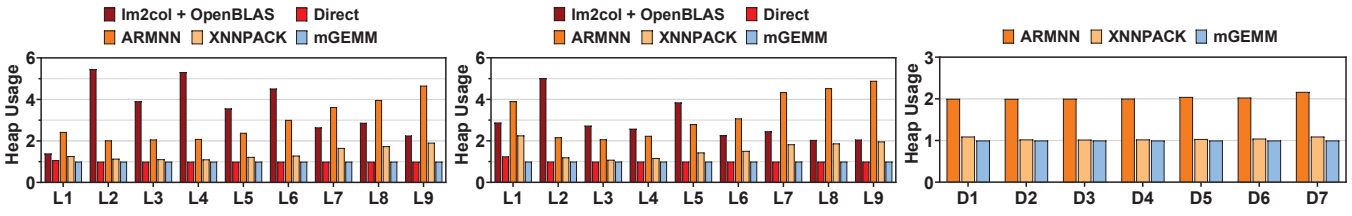
5.2.1 Classical Convolutions. Figure 6 shows the layer latency of our mGEMM convolution kernel compared to existing GEMM-based solutions on traditional non-pointwise convolution layers of VGG-16[40] and ResNet-50[17]. Our method outperforms the existing solutions in almost all layers, on all test devices. We achieve average speedup of 1.66× compared to XNNPACK, 1.47× compared to ARMNN, and 2.81× compared to im2col + OpneBLAS. Performances of existing solutions tend to vary drastically across networks, layers, and devices, resulting in no clear winner. Only mGEMM shows steady performance across the board, outperforming others.

There are cases where mGEMM showed slower performance, such as L1 of Figure 6 (a), in which it still remains competitive within 10% of the fastest one. We believe that the size of the input tensor and the number of computations is too small for the advantage of our solution to be significant on these layers. As such, the differences in latency from these layers are inherently insignificant compared to the large speedups we have on other layers.

5.2.2 Depthwise and Pointwise Convolutions. Figure 7 shows the layer latency of our mGEMM convolution kernel compared to existing GEMM-based solutions on pointwise and depthwise layers from MobileNet V2(1.4×)[4][39]. As im2col + GEMM[5] and direct convolution method[52] does not specify approaches to compute depthwise convolution, we have not included them for the depthwise convolution tests.

On the depthwise layers, our method has shown dominant performance. Our kernel achieves up to 16× speedup in some layers, and

[4]MobileNet V2 with 224 × 224 input size



(a) VGG-16 (b) ResNet-50 (c) MobileNet V2 (Depthwise)

Figure 8: Normalized max heap usage while processing different convolution layers and depthwise layers of VGG-16[40], ResNet-50[17], and MobileNet V2(1.4×)[39]. Measured with Valgrind-Massif[31] heap profiler.

Convolution Backend	Raspberry Pi 4			Odroid N2+			Pixel 4			Max Heap Usage (MB)
	Latency (s)	Power (W)	Energy (J)	Latency (s)	Power (W)	Energy (J)	Latency (s)	Power (W)	Energy (J)	
None	0.1926	3.23	0.62	0.0979	3.65	0.36	0.0686	1.29	0.09	111.717
Darknet Original [37]	2.5543	4.53	11.57	1.2666	5.71	7.23	0.7309	2.54	1.86	135.494
OpenBLAS [50]	0.9059	5.04	4.57	0.6075	5.66	3.44	0.3064	1.96	0.6	135.483
ARMNN [3]	0.4951	4.17	2.06	0.3187	4.9	1.56	0.3674	0.77	0.28	173.366
XNNPACK [12]	0.5488	5.03	2.76	0.2698	6.5	1.75	0.161	2.2	0.35	119.252
mGEMM	0.3838	4.53	1.74	0.2184	5.34	1.17	0.1317	1.8	0.24	111.735

Table 3: Latency, average power, energy consumption, max heap usage while running the YoloV3-Tiny[38] object detection network on Darknet[37] neural network framework, with various convolution backends. "None" means the case where no convolution backend is selected, effectively skipping all convolution computation layers.

maintained at least 1.41× speedup against XNNPACK, and 1.76× speedup against ARMNN, across all layers in all devices.

As stated in Section 3, on pointwise convolution, the differences between an mGEMM kernel and a conventional GEMM kernel are minor. Despite this, mGEMM performs faster, or at least competitive with the existing solutions in these layers.

5.2.3 Memory Usage per Layer. Figure 8 shows the normalized max heap memory usage of non pointwise layers of VGG-16[40], ResNet-50[17], and depthwise layers of MobileNet V2(1.4×)[39]. As Valgrind-Massif [31] operates by recording the heap allocations of the test program, its output is device-independent. We normalized the max heap usage of each solution to the theoretical minimum memory footprint required to store the input, output, and filter tensors of each layer. We have not included the pointwise layers as they can be directly calculated without any additional memory footprint.

Our mGEMM kernel and the direct convolution method[52] do not require any additional memory footprint, being a zero memory overhead solution. On the other hand, im2col + OpenBLAS and ARMNN require 1.4× to 5.5× to the minimum memory footprint, while the more optimized XNNPACK requires 1.1× to 2×.

5.2.4 End to End Application Latency. To measure the benefits of our convolution algorithm in a real-world application such as object detection, we compare the latency, memory usage, and energy consumption of several convolution solutions while running an object detection network end-to-end. We used the popular YoloV3-Tiny[38] as our object detection network, on the Darknet[37] neural network framework. We modify the Darknet framework so that the convolution operation of the network can be piped to a library we choose. All non-convolution operations are processed using the Darknet framework, in order to limit the differences in performance to that of the convolution layer only. We used Monsoon Power Monitor[30] to measure the energy consumption of the device while running the network.

Table 3 shows the YoloV3-Tiny object detection performance of five different convolution backends running on our test devices. mGEMM achieves 22% to 29% speedup in latency compared to the second-fastest solution of each device, which also translates into a reduction of energy consumption by a similar ratio, while using only 0.018MB more memory than the theoretical minimum, measured by skipping all convolution layers.

Interestingly, while mGEMM outperforms other methods in almost all parameters, it is consistently outperformed by ARMNN in power usage. However, the energy usage of ARMNN being consistently higher than that of mGEMM due to longer computation latency indicates that the difference in average power is not because mGEMM is less power-efficient than ARMNN, but because mGEMM aims for lower latency and therefore a higher device utilization. mGEMM could choose to lower its computation density for a better average power (i.e. thermal characteristics), but as our focus is on single-shot, low latency convolutions for mobile, we optimize for the lowest latency, memory, and energy.

6 DISCUSSION

Hardware acceleration: Many mobile platforms include hardware that have higher computational throughput than that of CPUs, such as GPUs or NPUs. As mGEMM is an algorithmic improvement over GEMM, mGEMM is also able to benefit from the high performance of such accelerators. However, there are several issues to be further considered. As these processors often require overheads on initialization and data movement, the overall latency of the computation may be larger than that of the CPU[16]. Moreover, to utilize the high throughput of these processors, large batch size is necessary

to saturate the hardware and amortize the overheads[16], which are often unavailable in mobile situations.

Theoretically, mGEMM is likely to perform more efficiently on accelerators than traditional GEMM-based algorithms, especially on mobile. As hardware on mobile SoCs often share the same memory bus, the available memory bandwidth for the accelerators would be similar to that of CPU, despite their higher computational throughput. This means that they require more reuse of the loaded data to properly utilize their high computational throughput[48]. Therefore, we believe that the removal of memory overheads and the higher arithmetic intensity from mGEMM would improve the performances of the mobile accelerators, compared to the GEMM-based algorithms.

JIT Compilation & Auto-Tuning: GEMM kernels are often only defined by the M, N dimension blocking parameters, but our kernels depend on additional variables such as stride, dilation, and padding, on top of c_o and w_o. This increases the number of inner kernels that need to be provided, compared to GEMM. We suggest the use JIT (Just-In-Time) compilation and automatic performance tuning, or auto-tuning of the required kernels as a solution. JIT compilation and auto-tuning are often combined on GEMM solutions[18, 45, 49] to dynamically create kernels optimized for the given device. As kernels required for the given model are always pre-determined, a high-performance kernel can be easily JIT compiled before deployment, and its cost be amortized over multiple inferences.

Quatizations: Quantization to lower precision formats is common on mobile to reduce computation complexity and increase throughput. Quantization allows a near proportional decrease of total computation and memory access operations, by operating on a larger number of smaller data at once[48]. This change can be understood as having larger registers, and therefore larger blocking parameters in Section 4. This benefits mGEMM more than GEMM. Unlike GEMM, where the blocking parameters are used linearly when calculating memory operation per MAC, mGEMM multiplies the parameters for its calculation. Therefore, larger blocking sizes allow for a larger decrease in memory operation per MAC for mGEMM.

Large-scale impact of mGEMM: While mGEMM was able to achieve noticeable improvements in latency, memory, and energy in YoloV3-tiny object detection task, the impact of these improvements in the large-scale system, and whether the impacts would still be relevant with the advancement of mobile hardware, needs further validation. Studies such as nn-Meter[53] provide us with hints to these questions; Data from nn-Meter shows that a large portion (over 80%) of both mobile CPU and GPU model latency consists of the convolution layer (including depthwise convolutions), in a wide range of networks. As such, the improvements of the convolution layer that mGEMM brings are likely to benefit the more sophisticated services based on large-scale models. Also, as the usage and complexity of CNN-based solutions on mobile would scale with the advancement of mobile hardware, the algorithmic improvement of the convolution operation that mGEMM brings would continuously benefit these CNN-based services, even with hardware improvements.

7 RELATED WORKS

Our work has mainly focused on optimizing the computation algorithm for convolutions. Optimizing network structures or optimizations specific to the mobile hardware are also valid approaches to accelerating neural networks for mobile. We briefly introduce related works covering these topics as well as algorithmic optimizations focusing on different aspects.

Network structures: MobileNet V1[19] uses depthwise separable convolution to greatly reduce the number of total computations and network size. MobileNet V2[39] uses the *Inverted bottleneck layer* to expand the number of channels before using depthwise convolution, which provides better accuracy with a lower number of computations and network size. Inverted Bottleneck layer has become the main building block of many modern efficient CNN designs, such as MnasNet[43], and EfficientNet[44].

Mobile hardware optimizations: Wang et al. solve the throughput losses on ARM CPUs caused by big.LITTLE heterogeneous cluster configuration[47]. They implement a pipeline structure that splits the convolution layers across different clusters to minimize communication across clusters. Zhang et al. optimize the pointwise (GEMM) and depthwise convolution focusing on the ARM Cortex-A CPUs [54]. With optimizations specific to ARM architectures, it achieves higher performance compared to TFLite linked to OpenBLAS and Ruy. Mogers et al. provide a data-parallel intermediate language and compiler for mobile GPUs, which can generate direct convolution codes optimized for the target GPU architecture[29].

Algorithmic optimizations: Gural et al. perform extreme optimizations on the memory usage of the convolution algorithm and succeed MNIST-10 classification task with only 2KB of memory[15]. While impressive, this technique sacrifices computation time a lot for the reduction of memory.

Unlike most works that focus on a single layer, there also exist works such as [26, 36] that focus on inter-layer optimizations by constructing and optimizing graph-based representations of the neural networks, which can add orthogonal merit to our work.

8 CONCLUSION

Limited memory capabilities and low-latency requirement of mobile CNN inferences render the current GEMM-based convolution solutions less optimal for mobile. Memory overhead and the low data reuse rate of GEMM are the two main problems current solutions face in mobile. We analyzed the process of mapping convolution to GEMM and identified the root cause of these problems. Based on this analysis, we create a computation algorithm, mGEMM, that is able to compute the convolution operation in a more efficient manner, eliminating the aforementioned problems. Our implementation of mGEMM is shown to outperform the existing high-performance neural network libraries in both latency and memory, in various test platforms and networks.

ACKNOWLEDGMENTS

This research was supported in part by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Numbers SRFC-TD2003-01, SRFC-TB1803-03 and the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2021-0-02048) supervised by the IITP(Institute of Information & Communications Technology Planning & Evaluation). Kyunghan Lee is the corresponding author.

REFERENCES

[1] Anderson, A., Vasudevan, A., Keane, C., and Gregg, D. Low-memory gemm-based convolution algorithms for deep neural networks. arXiv preprint arXiv:1709.03395 (2017).

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

[3] ARM. Armnn : Inference engine for cpus, gpus and npus. https://github.com/ARM-software/armnn, 2021.

[4] ARM. Compute library: Collection of software functions for the arm family of cpus and gpus. https://github.com/ARM-software/ComputeLibrary, 2021.

[5] Chellapilla, K., Puri, S., and Simard, P. High performance convolutional neural networks for document processing. In Tenth International Workshop on Frontiers in Handwriting Recognition (2006), Suvisoft.

[6] Cho, M., and Brand, D. Mec: Memory-efficient convolution for deep neural network. In Proceedings of ICML (2017), pp. 815–824.

[7] Dagum, L., and Menon, R. Openmp: an industry standard api for shared-memory programming. IEEE Computational Science and Engineering 5, 1 (1998), 46–55.

[8] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software (TOMS) 16, 1 (1990), 1–17.

[9] Dukhan, M. The indirect convolution algorithm. arXiv preprint arXiv:1907.02129 (2019).

[10] Dukhan, M. Accelerating tensorflow lite with xnnpack integration. https://blog.tensorflow.org/2020/07/accelerating-tensorflow-lite-xnnpack-integration.html, 2020.

[11] Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D., Henry, G., Pabst, H., and Heinecke, A. Anatomy of high-performance deep learning convolutions on simd architectures. In Proceedings of IEEE SC'18 (2018), pp. 830–841.

[12] Google. Xnnpack: Highly optimized library of floating-point neural network inference operators for arm, webassembly, and x86 platforms. https://github.com/google/XNNPACK, 2021.

[13] Goto, K., and Geijn, R. A. v. d. Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software (TOMS) 34, 3 (2008), 1–25.

[14] Gunnels, J. A., Henry, G. M., and Van De Geijn, R. A. A family of high-performance matrix multiplication algorithms. In Proceedings of the International Conference on Computational Science (ICCS) (2001), Springer, pp. 51–60.

[15] Gural, A., and Murmann, B. Memory-optimal direct convolutions for maximizing classification accuracy in embedded applications. In Proceedings of ICML (2019), pp. 2515–2524.

[16] Hanhirova, J., Kämäräinen, T., Seppälä, S., Siekkinen, M., Hirvisalo, V., and Ylä-Jääski, A. Latency and throughput characterization of convolutional neural networks for mobile computer vision. In Proceedings of the ACM Multimedia Systems Conference (MMSys) (2018), pp. 204–215.

[17] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In Proceedings of IEEE CVPR (2016), pp. 770–778.

[18] Heinecke, A., Henry, G., Hutchinson, M., and Pabst, H. Libxsmm: Accelerating small matrix multiplications by runtime code generation. In Proceedings of IEEE SC'16 (2016), pp. 981–991.

[19] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017).

[20] Huang, X., Wang, Q., Lu, S., Hao, R., Mei, S., and Liu, J. Evaluating fft-based algorithms for strided convolutions on armv8 architectures. Performance Evaluation 152 (2021), 102248.

[21] Intel. Intel oneapi math kernel library (mkl): Library of optimized math routines for science, engineering, and financial applications. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl, 2021.

[22] Jain, S. D., Xiong, B., and Grauman, K. Fusionseg: Learning to combine motion and appearance for fully automatic segmentation of generic objects in videos. In Proceedings of IEEE CVPR (2017), pp. 2117–2126.

[23] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014).

[24] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems 25 (2012), 1097–1105.

[25] Lavin, A., and Gray, S. Fast algorithms for convolutional neural networks. In Proceedings of IEEE CVPR (2016), pp. 4013–4021.

[26] Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing cnn model inference on cpus. In Proceedings of the USENIX Annual Technical Conference (USENIX ATC) (2019), pp. 1025–1040.

[27] Low, T. M., Igual, F. D., Smith, T. M., and Quintana-Orti, E. S. Analytical modeling is enough for high-performance blis. ACM Transactions on Mathematical Software (TOMS) 43, 2 (2016), 1–18.

[28] Mathieu, M., Henaff, M., and LeCun, Y. Fast training of convolutional networks through ffts. arXiv preprint arXiv:1312.5851 (2013).

[29] Mogers, N., Radu, V., Li, L., Turner, J., O'Boyle, M., and Dubach, C. Automatic generation of specialized direct convolutions for mobile gpus. In Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU) (2020), pp. 41–50.

[30] Monsoon-solutions. High Voltage Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/, 2019.

[31] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices 42, 6 (2007), 89–100.

[32] Nvidia. Comparison of convolution methods for GPUs. http://ska-sdp.org/sites/default/files/attachments/nvidia-sdp-directconvolution.pdf, 2020.

[33] Nvidia. cublas: Industry standard blas apis highly optimized for nvidia gpus. https://developer.nvidia.com/cublas, 2021.

[34] Nvidia. Optimizing Convolutional Layers. https://docs.nvidia.com/deeplearning/performance/pdf/Optimizing-Convolutional-Layers-User-Guide.pdf, 2021.

[35] Pan, J., and Chen, D. Accelerate non-unit stride convolutions with winograd algorithms. In Proceedings of the IEEE Asia and South Pacific Design Automation Conference (ASP-DAC) (2021), pp. 358–364.

[36] Pisarchyk, Y., and Lee, J. Efficient memory management for deep neural net inference. arXiv preprint arXiv:2001.03288 (2020).

[37] Redmon, J. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[38] Redmon, J., and Farhadi, A. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767 (2018).

[39] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of IEEE CVPR (2018), pp. 4510–4520.

[40] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014).

[41] Smith, T. M., Van De Geijn, R., Smelyanskiy, M., Hammond, J. R., and Van Zee, F. G. Anatomy of high-performance many-threaded matrix multiplication. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2014), pp. 1049–1059.

[42] Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., et al. The arm scalable vector extension. IEEE Micro 37, 2 (2017), 26–39.

[43] Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In Proceedings of IEEE CVPR (2019), pp. 2820–2828.

[44] Tan, M., and Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In Proceedings of ICML (2019), pp. 6105–6114.

[45] Tanner, D. E. Tensile: Auto-tuning gemm gpu assembly for all problem sizes. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (2018), pp. 1066–1075.

[46] Wang, Q., Mei, S., Liu, J., and Gong, C. Parallel convolution algorithm using implicit matrix multiplication on multi-core cpus. In Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN) (2019), pp. 1–7.

[47] Wang, S., Ananthanarayanan, G., Zeng, Y., Goel, N., Pathania, A., and Mitra, T. High-throughput cnn inference on embedded arm big.little multicore processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 39, 10 (2019), 2254–2267.

[48] Wang, S., Pathania, A., and Mitra, T. Neural network inference on mobile socs. IEEE Design & Test 37, 5 (2020), 50–57.

[49] Whaley, R. C., Petitet, A., and Dongarra, J. J. Automated empirical optimizations of software and the atlas project. Parallel computing 27, 1-2 (2001), 3–35.

[50] Xianyi, Z. Openblas: An optimized blas library. https://www.openblas.net/, 2021.

[51] Xie, H., Zhang, L., and Lim, C. P. Evolving cnn-lstm models for time series prediction using enhanced grey wolf optimizer. IEEE Access 8 (2020), 161519–161541.

[52] Zhang, J., Franchetti, F., and Low, T. M. High performance zero-memory overhead direct convolutions. In Proceedings of ICML (2018), pp. 5776–5785.

[53] Zhang, L. L., Han, S., Wei, J., Zheng, N., Cao, T., Yang, Y., and Liu, Y. Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (2021), pp. 81–93.

[54] Zhang, P., Lo, E., and Lu, B. High performance depthwise and pointwise convolutions on mobile devices. In Proceedings of the AAAI Conference on Artificial Intelligence (2020), vol. 34, pp. 6795–6802.