Automatic Generation of High-Performance Convolution Kernels on ARM CPUs for Deep Learning

Jintao Meng, Chen Zhuang, Peng Chen, Mohamed Wahib, Bertil Schmidt, *Senior Member, IEEE*, Xiao Wang, Haidong Lan, Dou Wu, Minwen Deng, Yanjie Wei, and Shengzhong Feng

Abstract—We present *FastConv*, a template-based code auto-generation open-source library that can automatically generate high-performance deep learning convolution kernels of arbitrary matrices/tensors shapes. FastConv is based on the Winograd algorithm, which is reportedly the highest performing algorithm for the time-consuming layers of convolutional neural networks. ARM CPUs cover a wide range of designs and specifications, from embedded devices to HPC-grade CPUs. The leads to the dilemma of how to consistently optimize Winograd-based convolution solvers for convolution layers of different shapes. FastConv addresses this problem by using templates to auto-generate multiple shapes of tuned kernels variants suitable for skinny tall matrices. As a performance portable library, FastConv transparently searches for the best combination of kernel shapes, cache tiles, scheduling of loop orders, packing strategies, access patterns, and online/offline computations. Auto-tuning is used to search the parameter configuration space for the best performance for a given target architecture and problem size. Results show 1.02x to 1.40x, 1.14x to 2.17x, and 1.22x and 2.48x speedup is achieved over NNPACK, ARM NN, and FeatherCNN on Kunpeng 920. Furthermore, performance portability experiments with various convolution shapes show that FastConv achieves 1.2x to 1.7x speedup and 2x to 22x speedup over NNPACK and ARM NN inference engine using Winograd on Kunpeng 920. CPU performance portability evaluation on VGG–16 show an average speedup over NNPACK of 1.42x, 1.21x, 1.26x, 1.37x, 2.26x, and 11.02x on Kunpeng 920, Snapdragon 835, 855, 888, Apple M1, and AWS Graviton2, respectively.

Index Terms—AI, convolution, deep learning

- *Jintao Meng, Chen Zhuang, Dou Wu, and Yanjie Wei are with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, Guangdong 518055, China. E-mail: {jt.meng, chen.zhuanng, dou.wu, yj.wei}@siat.ac.cn.*
- *Peng Chen and Mohamed Wahib are with the National Institute of Advanced Industrial Science and Technology (AIST), Tokyo 100-8921, Japan, and also with the RIKEN Center for Computational Science (R-CCS), Kobe, Hyogo 650-0047, Japan. E-mail: {chin.hou, mohamed.attia}@aist.go.jp.*
- *Bertil Schmidt is with the Institute of Computer Science, Johannes Gutenberg University Mainz, 55122 Mainz, Germany. E-mail: bertil.schmidt@uni-mainz.de.*
- *Xiao Wang is with Oak Ridge National Laboratory, Oak Ridge, TN 37830 USA. E-mail: xiaowangatpurdue@gmail.com.*
- *Haidong Lan and Minwen Deng are with Tencent AI Lab, Shenzhen, Guangdong 518000, China. E-mail: turbo0628@163.com, danierdeng@tencent.com.*
- *Shengzhong Feng is with National Supercomputer Center in Shenzhen, Shenzhen, Guangdong 510330, China. E-mail: sz.feng@siat.ac.cn.*

Manuscript received 2 September 2021; revised 30 December 2021; accepted 11 January 2022. Date of publication 27 January 2022; date of current version 23 May 2022.

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0204403, in part by Strategic Priority CAS Project under Grant XDB38050100, in part by the National Science Foundation of China under Grant U1813203, in part by Shenzhen Basic Research Fund under Grants RCYX20200714114734l9, KQTD20200820113106007, and JSGG20190220164202211, in part by CAS Key Lab under Grant 2011DP173015, in part by JST, PRESTO under Grant JPMJPR20MA, in part by JSPS KAKENHI under Grant JP21K17750, in part by AIST Emerging Research, Japan under Grant AAZ2029701B, and in part by Artificial Intelligence Initiative at Oak Ridge National Laboratory. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. J. Meng and C. Zhuang contributed equally to this work.

(Corresponding authors: Peng Chen and Mohamed Wahib.)

Recommended for acceptance by A. J. Peña, M. Si and J. Zhai.

Digital Object Identifier no. 10.1109/TPDS.2022.3146257

1 Introduction

Deep learning (DL) inference is becoming a common workload on edge devices, such as smartphones, and in data centers [1]. Thus, real-time, and often on-device, DL inference is becoming increasingly important. Convolution layers are the main computational bottleneck for the inference computation of Convolutional Neural Networks (CNNs). Table 1 shows that convolution layers are on average responsible for 95% of the compute load for a list of widely used CNNs.

Three algorithms, namely direct convolution [7], GEMM-based [8], and Winograd [9], [10], are commonly used in production libraries to compute the operations of convolution layers. Among them, Winograd has recently attracted the majority of use and research attention (e.g., [9], [11], [11], [12], [13], [14], [15]) since it can perform unstrided convolution with the least amount of arithmetic operations [16]. More specifically, in comparison to the two other algorithms, the Winograd algorithm can reduce the number of arithmetic operations by up to a factor of 5.04x [9]. Consequently, Winograd convolution has became widely used and is supported by modern DL libraries such as ARM® Compute Library [17], NNPACK [18], Nvidia® cuDNN [19], and Intel® oneDNN [20]. However, although Winograd can offer significant speedups over other convolution algorithms [14], it remains a challenge to efficiently implement Winograd convolution on a large variety of ARM devices with different specifications. For instance, NNPACK when used as a backend in PyTorch delivers only $6\% \sim 35\%$ of the single core peak performance on convolution layers of VGG–16 (depending on the utilized ARM processor). This instability in performance is also, overall, far below the

TABLE 1
Computational Footprint of Various Layer Types Measured in Terms of MFlops for Six CNN Architectures

Network	Convolution layer		FC	Pool	Others
	Wino	General			
VGG-16 [2]	29,271	0	236	6	13
GoogLeNet [3]	1,836	1,180	2	12	166
ResNet-50 [4]	3,528	3,827	4	2	407
MobileNet–V1 [5]	0	1,088	0	0	73
Inception–V3 [6]	4,684	6,209	2	25	27
Inception–V4 [4]	7,459	15,911	2	45	46

Here Wino, General, FC, Pool, and Other denotes Winograd convolution, general convolution, fully-connected, pooling, and other types of layers, respectively.

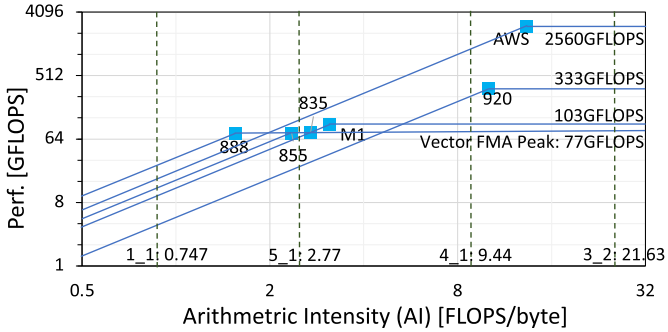


Fig. 1. Roofline analysis [25], [26] of the machine balance of six mainstream Arm CPUs used in mobile phones, desktops, and data centers. 835, 855, and 888 are short name for Snapdragon 835, 855, and 888 respectively. M1, 920 and AWS denote Apple M1, Kunpeng 920, AWS Graviton 2. The input shape of convolution also affects the Arithmetic intensity (AI) of Winograd algorithm. We plot the AI of four typical layers from VGG-16 with green dotted lines to demonstrate this variation.

desired efficiency. Furthermore, using multiple cores for convolution layers of VGG–16 results only in a speedup between $2\times$ and $4\times$ when scaling the number of cores from 1 to 64 (i.e. 3% to 6% parallel efficiency).

The diversity in the design of ARM-based processors presents a challenge for performance portable and effective optimizations. ARM CPUs have been widely used in mobile phones, embedded devices, consumer PCs, data center servers, and supercomputers. Thus, current ARM architectures feature significantly different configurations with respect to compute units, caches, and memory hierarchies. Clock frequencies can vary from 100MHz to 3GHz and available memory bandwidth can range from 10GB/s to 1.6TB/s. For the cache hierarchy, [21], there is a complex and diverse configuration at each level of cache (as shown in Table 4). Moreover, cache sizes can vary between cores at the same level on one ARM CPU (know as ARM Big.Little [22], and DynamIQ [23]). Additionally, FMA units, ROB sizes, pipelines, cache placement policies, type of scheduler, and interconnections are all redesign-able for SoC vendors [24]. As a consequence, there are currently thousands of ARM SoCs with different configurations available in the market. To demonstrate their diversity, we compare six processors in Fig. 1: their AI (Arithmetic Intensity) [25], [26] ranges between 1.55 and 14 Flops/Bytes. This variability in AI leads to different bounds for the same code running on different ARM CPUs, which poses a challenge to performance-portable optimization. More specifically, the AI of convolution operations is determined by its input shapes with a corresponding value ranging anything between 0.747 to 21.63 Flops/Bytes (in Table 5). Thus, both hardware diversity and varying computation pattern are the two challenges for Winograd optimization in DL that motivates the work in this paper. It is important to note that hand-coding optimizations for different ARM CPUs lead to convoluted codebases with heavy code branching and unsustainable technical debt. Previous work has so far mainly focused on fixed AI with constant architecture specification on CPUs or GPUs [7], [15]. This is the first work that considers the portability of Winograd optimizations w.r.t. both changing computation patterns and hardware diversity.

The concrete challenges of developing a transparent and performance portable Winograd convolution library for ARM CPUs to use in DL inference include: a) diversity of target architectures (in terms of memory hierarchies and compute capabilities), and b) the skinny tall and long rectangular matrices/tensors generated by CNNs [27] for which existing BLAS libraries are not optimized for. Hand-tuning libraries for each target specification is a futile task.

This paper addresses these challenges by making the following contributions:

1) The Winograd algorithm consists of three stages: transforming the input to the Winograd domain, computing multiple tensor multiplication operations to perform convolution in the Winograd domain, and finally transforming the results from the Winograd domain. Since the repeated tensor multiplication operations are the bottleneck of Winograd, we developed a highly tuned code auto-generator based on C ++ templates (named TensorGEMM), it generates code optimized for computing tensor multiplications of arbitrary shapes (especially for skinny tall and long rectangular tensors). The auto-generated kernels also minimize the data movement in the reshaping phase and are optimized for efficient register and cache blocking for the considered target ARM CPU.
2) We designed a transparent library (*FastConv*) for Winograd convolutions on ARM CPUs. The library internally generates the highest performing code variant for the considered target CPU. The code variants, optimized for different targets, cover a wide range, and combinations, of optimizations: tuning the data layout for unit-strided access patterns, loop reordering, packing strategies for data blocks to interleave indexing and packing the layout back to enable TensorGEMM tuning, register blocking, and cache blocking. We use an empirical auto-tuning strategy to search all parameter configurations for the best performance for a given hardware specification and convolution problem size.
3) To demonstrate the effectiveness of FastConv, we use a a variety of ARM processors ranging from embedded/mobile to server grade CPUs and compare the performance to two state-of-the-art libraries for inference: ARM NN inference engine [28] and NNPACK [18]. Our portability test with various convolution shapes shows that FastConv achieves 1.2x to 1.7x speedup and 2x to 22x speedup over NNPACK and ARM NN inference engine using Winograd on Kunpeng 920 with all 8 cores. Device portability evaluations on the *VGG–16* model show an average speedup over NNPACK of 1.42x, 1.21x, 1.26x, 1.37x,

2.26x, and 11.02x on Kunpeng 920, Snapdragon 835, 855, 888, Apple M1, and AWS Graviton2, respectively.
4) FastConv is open source and publicly available at https://github.com/Mengjintao/FastConv.

The rest of this paper is organized as follows: In Section 2, we present the background and related work. Section 3 elaborates on our implementation for FastConv library. Section 4 shows the evaluated result. Finally, Section 5 concludes.

2 Background

This section first introduces the convolution operator and then elaborates on related work.

2.1 Convolution

A convolutional layer maps an input tensor D in the order of [batch, input_channel, height, width] (or "NCHW") and a filter tensor G in the order [output_channel, input_channel, height, width] (or "KCRS"), to an output tensor S of shape [batch, output_channel, height, width] (or "NKEF"). Images are processed individually during inference when data-parallel batch processing is infeasible. Consequently, we set $N = 1$ for better readability, without sacrificing generality; the analysis holds when adjusting for $N > 1$ to add an extra dimension of coarse-grained data parallelism. The convolution layer computes the output tensor S by accumulating the input tensor along the input channels dimension C to reduce $K \times C$ finite-impulse-responses to exactly K output channels:

$$S_{k,x,y} = \sum_{c=0}^{C-1} \sum_{u=0}^{R-1} \sum_{v=0}^{S-1} D_{c,x+u,y+v} \cdot G_{k,c,u,v} \tag{1}$$

Where $0 \le k < K$, $0 \le c < C$, $0 \le x < H - R + 1$, $0 \le y < W - S + 1$.

When using a non-unit stride, the sums over x and y are incremented with step size $stride > 1$. The naïve evaluation of Equation (1) results in $\Theta\big((K \times C) \cdot (H \times W) \cdot (R \times S)\big)$ operations. When $R \times S$ is 3×3, general convolution can be viewed as Winograd convolutions. The 2-dimensional Winograd formula can be written as:

$$S = A^T\big(\big[GgG^T\big] \odot \big[B^T dB\big]\big)A = A^T\big(U \odot V\big)A \tag{2}$$

With Equation 2, the actual computation of a Winograd convolution, illustrated in Fig. 2 using $F(2 \times 2, 3 \times 3)$ as an example, can be partitioned into four stages:

(I) Filter transformation: $U = GgG^T$
(II) Input transformation: $V = B^T dB$
(III) Tensor Multiplication: $M = U \odot V$
(IV) Output transformation: $S = A^T MA$

Here B, G, A are constant matrices with fixed values defined in [9], g is a $R \times S$ matrix embedding the filter entries, and d is a 4×4 (for $F(2 \times 2, 3 \times 3)$ schema) or 8×8 (for $F(6 \times 6, 3 \times 3)$ schema) sliding window tile extracted from the input images. $F(2 \times 2, 3 \times 3)$ requires $4 \times 4 = 16$ multiplications, whereas the standard algorithm requires $2 \times 2 \times 3 \times 3 = 36$. Thus the number of arithmetic operations are reduced by a factor of 2.25x with $F(2 \times 2, 3 \times 3)$ or similarly 5.04x with $F(6 \times 6, 3 \times 3)$, in comparison to general convolution in Equation 1 [9].

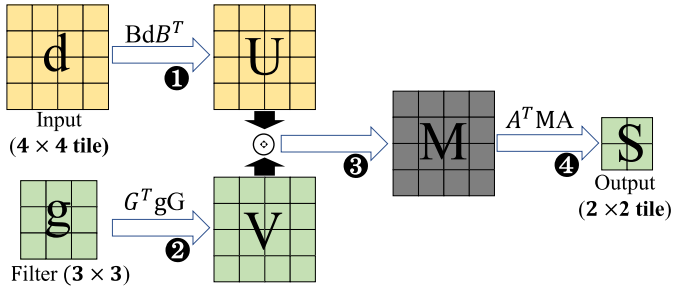


Fig. 2. An example of 2D convolution by Winograd algorithm $F(2 \times 2, 3 \times 3)$. The Winograd algorithm consists of a pipeline of filter transformation, input transformation, tensor multiplication, and output transformation.

2.2 Arm and Arm Neon Intrinsics

We briefly introduce the Arm and Arm Neon intrinsics (more details on Arm Neon can be found in the Arm Neon user guide [29]). Arm is a family of reduced instruction set computing (RISC) architectures for computer processors. Arm Ltd. develops the architecture and licenses it to other companies, who in turn design their products that implement one of those architectures, including systems-on-chips (SoC) and systems-on-modules (SoM) used in both mobile devices and servers. Arm Neon is an advanced single instruction multiple data (SIMD) architecture extension included in all Armv8 devices [30], [31], it supports 128-bit vectors, and can execute 128 bits or 4×32-bit floating-point operations at a time.

2.3 Related Work

Convolution algorithms have been researched widely in the past years. *Direct convolution*[7], [32] and *GEMM-based convolution* [8], [33] are two major algorithms used in the calculation of Equation (1). Direct convolution is implemented by Intel® oneDNN for X86 CPU[7] and NCNN for Arm CPU [32]. oneDNN achieves $60\% \sim 80\%$ of theoretical peak performance with offline data layout pre-packing, whereas NCNN avoids that offline routine and keeps the original tensor layout in favor of its framework flexibility, but at the cost of the lower percentage of peak performance (30%). GEMM-based convolution [8], [33] rearranges the input images of shape "NCHW" into "N · CRS · EF" in a step known as GEMM-based convolution, and then invokes N times a GEMM routine to calculate the output image of "NKEF". The open source implementation of GEMM-based convolution for Arm architecture is provided by NNPACK [18] and used by PyTorch [34].

Winograd convolution [9] is implemented by oneDNN, cuDNN, and NNPACK using batched GEMM [35], [36], [37], [38]. NNPACK [18], Arm NN inference engine [28], and FeatherCNN [10] are three public available libraries with Winograd implementations optimized for Arm CPUs. NNPACK and Arm NN inference engine follows Lavin *et al.* [9] approaches using Winograd $F(6 \times 6, 3 \times 3)$ and $F(4 \times 4, 3 \times 3)$ repectively, while FeatherCNN adopts a novel TensorGEMM reformulated Winograd algorithm of both $F(6 \times 6, 3 \times 3)$ and $F(2 \times 2, 3 \times 3)$.

Code Automation is a useful technique for performance optimization. The just-in-time compilation (JIT) and automatic code generation are becoming increasingly used in the development of next-generation high-performance convolution

kernels used in back-end libraries [39], [40], [41]. An effective JIT approach is used by LIBXSMM [39] to map assembly instructions to opcodes in order to avoid invoking the compiler. LIBXSMM can handle problem dimensions that are normally not available and targets high-performance execution of small GEMMs with $M \times N \times K < 80^3$ on Intel x86. The shape and number of LIBXSMM's kernels are determined at compilation time and are thus not suitable for the diversity of abnormal matrix shapes generated by DL models.

TVM [40] is an end-to-end compilation and optimization stack for the deployment of DL workloads. TVM is designed to deal with a large number of hardware configurations and problem shapes generated by DL. However, TVM underperforms on fine-grained kernels for specific hardware targets [42], [43], [44], [45]. TVM's fine-grained kernels are comprised of three parts: one is generated by the compilers [46], [47], the second is generated by Halide [48], and the third part comes from other libraries, e.g., LIBXSMM [39] and OpenBLAS [49]. Without manual expert tuning and highly efficient auto-generation of the fine-grained kernels and fine-grained scheduling, TVM's high performance cannot be achieved.

2.4 Novelty

FeatherCNN [10] optimized a CNN inference framework on Arm CPUs, with an emphasis on providing thirteen types of CNN layers, e.g., convolution, pooling. The GEMM-based convolution and Winograd algorithms were manually optimized for accelerating the convolution operations in FeatherCNN. It is worth mentioning that FeatherCNN is used in production by Tencent's <<Honor of Kings>> game [50] as the inference engine. FeatherCNN didn't employ an automated approach for the skinny tall matrices in GEMM operations, it followed the same approach as Arm NN inference engine [28] and NNPACK [18]: hand-tuned implementations. Additionally, FeatherCNN is not performance portable to a wide range of Arm CPUs. More specifically, the manual optimization of FeatherCNN makes it incapable of pushing the performance limits for convolution computations on variants of Arm architectures having different specifications.

To address the performance portability and transparency issues with FeatherCNN (and also NNPACK [18] and Arm NN inference engine [28]), in this work we propose a code auto-generation framework built on C++ templates for portable and transparent high-performance DL inference. We auto-generate convolution kernels using a configurable Winograd algorithm to reduce the memory traffic and improve the data locality, e.g., cache/register blocking, for a specific Arm target. The automated convoluted kernels consistently outperform state-of-the-art libraries (e.g., Arm NN inference engine [28] and NNPACK [18]) on a wide range of Arm CPUs. The results are shown in Section 4.

3 FastConv: A Library for Auto-Generating Winograd Convolution Kernels

In this section, our four-fold optimization for Winograd convolutions is illustrated in Fig. 3. First, in Section 3.1 we propose the formulation of the improved Winograd algorithm that we use with our C++ templates auto-generator (TensorGEMM) to avoid the interleaved data packing overhead [9]. Second, in Section 3.2 we elaborate on how the four steps of tile transformation, parameterized cache blocking, register blocking, and loop reordering are applied in our Winograd formulation. Third, Section 3.3 discusses the computational intensity analysis, inner-kernel shape selection, and template-based auto-generation of a series of highly efficient fine-grained kernels. Fourth, in Section 3.4 we discuss an auto-tuning scheme that provides the composability of cache-aware blocking sizes and dozens of kernels with different shapes to deliver the highest performance by searching the parameter space for optimal configurations. Finally, we briefly discuss our library's user interface and implementation.

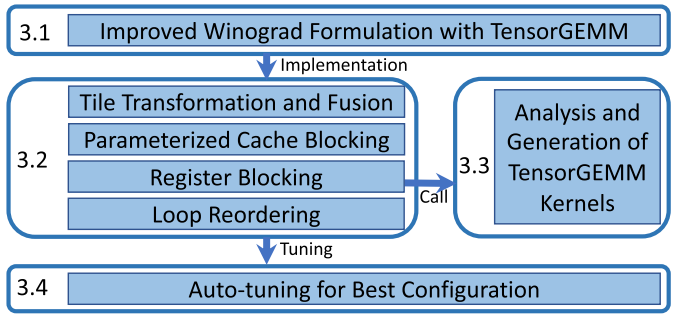


Fig. 3. A step-by-step flow chart of our Winograd optimizations in FastConv.

3.1 Improved Winograd Formulation With TensorGEMM

As shown in Fig. 2, the Winograd algorithm [9] contains three memory-intensive transformation stages and one compute-intensive matrix multiplication stage. The *input transformation* stage results in V iterations over output channels K while the *filter transformation* generating the filtered input U is independent of the image tiles, and can be calculated offline. The output transformation reshapes the result tensor, M, and accumulates the final results in the output S. The *tensor multiplication* stage is the bottleneck of the Winograd algorithm [15].

In the third stage, i.e., *tensor multiplication*, let $M_{k,d} = \sum_{c=0}^{C-1} U_{k,c} \odot V_{c,d}$ be the aggregate tensor multiplications along the input color channels, then $M^i_{k,d} = (U^i \circ V^i)_{k,d}$ denotes the ith entry of a Winograd tile where $0 \leq i < \theta$ and $\circ$ is a matrix product. Note that $F(t \times t, r \times r)$ produces tiles with $\theta = (t + r - 1)^2$ elements resulting in $\theta = 16$ entries in case of $F(2 \times 2, 3 \times 3)$. The coordinate representation $M^i_{k,d}$ can be reinterpreted as plain matrix multiplication over a batch of θ factors U^i and V^i:

$$M^i_{k,d} = \sum_{c=0}^{C-1} U^i_{k,c} \cdot V^i_{c,d} \quad \forall i, k, d. \tag{3}$$

Where $0 \leq k < K$, $0 \leq d < H' \times W'$ and $0 \leq i < \theta$. We identify the Winograd index $0 \leq i < \theta$ with L lanes in vector registers (shown in Fig. 4), where $\theta = 16$ in the case of $F(2 \times 2, 3 \times 3)$. Since current Arm architectures feature 128 bit vector registers storing $L = 4$ single precision floating-point values, we need $p = 4$ no-warm-up passes to compute a total of $\theta = 16$ independent contributions for $F(2 \times 2, 3 \times 3)$. The remaining loops over the output channel index k, the spatial

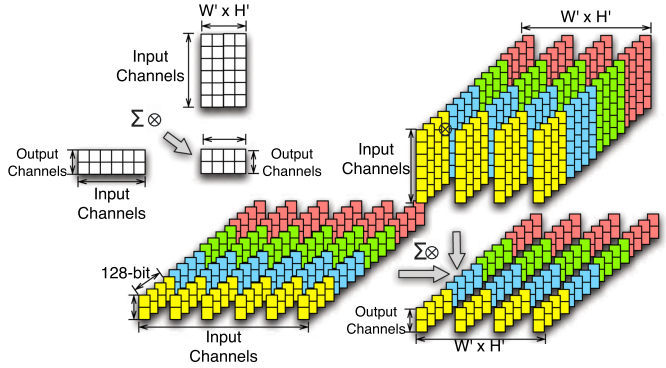


Fig. 4. Upper left: 2-dimensional illustration of Equation 3, where each element is a vector of length 16. Bottom right: a corresponding 3-dimensional illustration of $F(2 \times 2, 3 \times 3)$.

coordinates d, and the pass identifier p are parallelized via multi-threading.

We reformulate the transformations as $C = A^T(A^T B^T)^T = A^T BA$ to exploit fast transposition in registers (for matrices stored in row-major order). In case of Winograd convolution, Equation 2 can be rewritten to account for the equality $(U \odot V)^T = U^T \odot V^T$ to be:

$$S = A^T(U \odot V)A = A^T(A^T(U^T \odot V^T))^T, \tag{4}$$

According to Equation (4), we can finally reformulate four stages of the Winograd algorithm to use our TensorGEMM library for arbitrarily shaped tensor multiplication (more details on TensorGEMM in Section 3.3):

$$\text{Input Transform:} V^T = B^T(B^T d)^T \tag{5}$$

$$\text{Filter Transform:} U^T = G(Gg)^T \tag{6}$$

$$\text{TensorGEMM:} (M^T)^p = (V^T)^p \times (U^T)^p, 0 \leq p < \frac{\theta}{L} \tag{7}$$

$$\text{Output Transform:} S = A^T(A^T M^T)^T. \tag{8}$$

Our Winograd algorithm that supports arbitrary dimensions follows the above four equations (also presented in Algorithm 1). The input, kernel, and output transform use Lavin's formulas [9] listed in lines 5, 7, and 12, respectively. However, the data layout of the resulting tensors (input, weight, and output tensors in Algorithm 1) has to be reshaped before and after the execution of TensorGEMM. Those reshape routines are executed in lines 6, 7, and 11. TensorGEMM is invoked at line 10. There are p no-wArm-up passes/calls of TensorGEMM routine to calculate $M^T = V^T \times U^T$. In Fig. 4, 4 passes of TensorGEMM routine for a 128 bit vector registers of an Arm architecture are colored with yellow, blue, gray, and red. This scheme is the basis for the kernel of our Winograd algorithm for which we apply cache and register blocking in the following section.

Lavin's strategy [9], [15] can be performed using batched GEMM, i.e., θ consecutive calls to GEMM using matrices of dimensions $K \times C$ and $C \times (H' \times W')$. While this reduces the code complexity, performance can suffer from memory-bound transformations, interleaved indexing and packing for GEMM routines, and the low computational intensity for special shapes of matrices [10]. When L is set to be one, Algorithm 1 can be viewed as Lavin's strategy by substituting TensorGEMM in line 10 with a call to one GEMM routine. Using multiple GEMMs will introduce a number of performance issues on common CPU architectures: a) before calling multiple GEMMs, the data blocks generated by the input transformation are scattered into θ matrix pairs using interleaved indexing, b) after multiple GEMMs, the layout of the result needs to be packed back into a unit-strided order, and c) both operations involve significant data movement in input and output transformations and thus significantly reduce the overall performance. TensorGEMM however only issues $p = 4$ passes of TensorGEMM routine for Arm architecture with Neon intrinsics which significantly reduces the data movement cost of the Winograd algorithm.

Algorithm 1. Winograd Algorithm Using TensorGEMM. θ is 16 for $F(2 \times 2, 3 \times 3)$ or 64 for $F(6 \times 6, 3 \times 3)$. L is the Instruction Width (4 for Arm v8 Architecture). Batch Size is Set to One.

Input: input[C][H][W], kernel[K][C][3][3]
Output: output[K][E][F]
1 $E = (W + padLeft + padRight - 3)/stride + 1$
2 $F = (H + padLeft + padRight - 3)/stride + 1$
3 $H^{'} = \frac{E+r-1}{r}$, $W^{'} = \frac{F+r-1}{r}$
4 $p = \frac{\theta}{L}$
5 InputTrans(input[C][H][W], V^T[C][$H^{'}$][$W^{'}$][θ]); // Eqn 5
6 Reshape(V^T[C][$H^{'}$][$W^{'}$][θ], V^T[p][C][$H^{'}$][$W^{'}$][L])
7 KernelTrans(kernel[K][C][3][3], U^T[K][C][θ]); // Eqn 6
8 Reshape(U^T[K][C][θ], U^T[p][K][C][L])
9 **for** $i = 0$ to $p - 1$ **do**
10 TensorGEMM(U^T[i][K][C][L], V^T[i][C][$H^{'}$][$W^{'}$][L], M^T[i][K][$H^{'}$][$W^{'}$][L]).; // Eqn 7
11 Reshape(M^T[p][K][$H^{'}$][$W^{'}$][L], M^T[K][$H^{'}$][$W^{'}$][θ])
12 OutputTrans(M^T[K][$H^{'}$][$W^{'}$][θ], output[K][E][F]).; // Eqn 8

3.2 Proposed Winograd Optimization for Arbitrary Dimensions

In this section, we introduce the optimizations we use for the Winograd algorithm in order to allow for arbitrary dimensions. This is mainly driven by the need to enable performance portability for Arm CPUs with different specifications. The main optimization considerations [16], [27], [40], [46] are as follows: a) how to adjust the data layouts to minimize data movement, b) how to effectively do cache blocking for L1/L2 that vary in size from one processor to another, c) how to fully utilize the vector registers with register blocking schemes, and d) how to improve data locality by avoiding redundant memory/cache accesses with data packing, minimize cache misses by loop order, etc. A performance portable and transparent library to generate highly efficient code with all the above considerations, and without the need for manual tuning, can boost the productivity of deploying DL services/solutions onto millions of Arm SoC chips with a large space of hardware configurations. With the above considerations in mind, our optimizations for a performance portable Winograd algorithm are as follows.

First, *tile transformation and fusion* is used to adjust the data layout to minimize data movement. As shown in Algorithm 1, to reduce the cache miss rate, the data layout after input,

filter, and output transform should be reshaped to ensure a continuous memory access pattern for TensorGEMM's multiplication kernels. However, explicit implementation of transformations and reshaping routines result in extra data movement and waste time on memory accesses. Thus we fuse the data packing with filter, input, and output transformations to relieve the memory access pressure. As shown in Algorithm 2, we fuse the transformation and reshape routine into one transformation routine, and write the result tensor directly into the target data layout. This modification is presented in lines 2, 3, and 6 in Algorithm 2 on input, kernel, and output transformations. Finally, $H^{'}$ and $W^{'}$ can also be fused into one dimension of $tiles$ by setting $tiles = H^{'} \times W^{'}$: this further simplifies the following strategies.

Algorithm 2. Optimized Winograd Using Tile Transformation and Fusion

Input: input[C][H][W], kernel[K][C][3][3]
Output: output[K][E][F]
13 tiles = $H^{'} \times W^{'}$
14 InputTrans(input[C][H][W],V^T[p][C][tiles][L]); // Eqn 5
15 KernelTrans(kernel[K][C][3][3],U^T[p][K][C][L]); // Eqn 6
16 **for** $i = 0$ to $p - 1$ **do**
/* Eqn 7 */
17 TensorGEMM(U^T[i][K][C][L], V^T[i][C][tiles][L], M^T[i][K][tiles][L])
18 OutputTrans(M^T[p][K][tiles][L],output[K][E][F]) // Eqn 8

Algorithm 3. Optimized Winograd With Cache Blocking on Output Channel and Tiles Dimension.

Input: input[C][H][W], kernel[K][C][3][3]
Output: output[K][E][F]
19 $oB_{num} = K/oB$
20 $tB_{num} = tiles/tB$
21 **for** $u = 0$ *to* oB_{num} **do**
22 **for** $v = 0$ to tB_{num} **do**
23 InputTrans(input[C][H][W],V^T[p][C][tB][L]); // Eqn 5
24 **if** *onoffKernel* **then**
25 KernelTrans(kernel[u·oB:(u+1)·oB][C][3][3], U^T[p][oB][C][L])
26 **for** $i = 0$ *to* $p - 1$ **do**
27 TensorGEMM(U^T[i][oB][C][L], V^T[i][C][tB][L], M^T[i][oB][tB][L]).; // Eqn 7
28 OutputTrans(M^T[i][oB][tB][L], output[u · oB : (u+1) · oB][E][F])

Second, *cache blocking* can be applied on the output channel K and $tiles$ dimensions to increase the data reuse in L1/L2 cache. It is similar to Goto's strategy [16] but applied on a new TensorGEMM routine. When the complete matrices U^T and V^T can not be stored in a cache, we block (i.e. tile) the U^T matrix on the dimension of the output channel, and block V^T on the dimension of tile. The dimension of the input channel on U^T and V^T is not blocked for two reasons: a) blocking the input channel may interrupt the instruction pipeline of TensorGEMM's multiplication kernel, b) the value of input channels ranges between 3 to 512 in most neural networks. For values greater than 512, the data volume in the cache can be held by adjusting the input channel and tiles. For oB and tB being the block sizes of output channel K and input $tiles$, respectively, the cache blocking strategy is illustrated in Algorithm 3. The number of blocks is calculated in lines 2 and 3. Each block is processed with the code from line 6 to line 11. In line 6, tB slides a window of shape $(t + r) \times (t + r)$ to form the tensor V^T. Thus the working space footprint for $input$ is limited to only tB sliding windows instead of whole tensor. There is a similar data locality optimization done on kernel transformation. Additionally, one can also pre-process the weight tensor offline since the weight tensor is constant during the inference computation, but at the cost of accessing $\frac{(t+r)^2}{9}$ more data. Thus, it is important to balance the trade-off between memory accesses and computational costs for the offline kernel transformations. The data layout of the above two transformation routines ensures a continuous memory access pattern in TennsorGEMM along the input channel C dimension. We can adjust oB and tB to block V^T, U^T, and M^T for the L2 cache (based on the L2 size of a specific target). Finally, the output tensor M^T is accumulated and transformed to the output in line 11.

Algorithm 4. Optimizing TensorGEMM With Register Blocking, m is the Register Block Size of Tiles, n is the Register Block Size of Output Channels.

Input: inTensor[C][tB][L], kerTensor[oB][C][L]
Output: outTensor[oB][tB][L]
29 $n_{num} = \frac{oB}{n}, m_{num} = \frac{tB}{m}$
30 **for** $i = 0$ *to* n_{num} **do**
31 **for** $j = 0$ *to* m_{num} **do**
32 U_r^T[m][C][L]=U^T[i·n:(i+1)·n][C][L]
33 V_r^T[C][n][L]=V^T[C][j·n:(j+1)·n][L]
/ * as a kernel in Listing 1 * /
34 **Innerkernel_mxn**(U_r^T[m][C][L], V_r^T[C][n][L], M_r^T[n][m][L])

Third, TensorGEMM is a GEMM-like matrix multiplication routine, thus we use *register blocking* in TensorGEMM. The register blocking we use in TensorGEMM is described in Algorithm 4. It is noteworthy that the kernel of *Innerkernel_mxn* is called by Algorithm 4 and an example of the Neon-optimized kernel can be found in Listing 1. The register blocks (tiles) are shown in the loops in lines 30 and 31. Each block is processed with the code from line 32 to line 34. We extract the register blocks V_r^T and U_r^T from the corresponding cache blocks V^T and U^T. Then in line 34, TensorGEMM's multiplication kernel routine M^T is calculated. A C++ temple-based code generation method is applied to generate a series of efficient multiplication kernels for TensorGEMM with high compute intensity (we elaborate on this in the following subsection).

Fourth, in order to determine the optimal packing of different transformations into different cache levels (L2/L3, etc.), a similar way [40] but deep coupling *loop reordering* is used for controlling the memory access pattern. The loops in lines 21 and 22 in Algorithm 3 for cache blocking, and the loops in lines 30 and 31 in Algorithm 4 for register blocking are first unrolled to simplify loop reordering. The loop reordering on the cache blocking loops should assure that V^T will be scanned once and held in the L1 cache whereas U^T would be scanned multiple times and stored in the L2

cache, or vice versa. The flexibility of loop reordering provides the possibility of removing redundant data movement in cache/register blocking optimizations, depending on the cost of repeated scanning of U^T or V^T.

Listing 1. An Example of Generating $m \times 4$ Inner Kernels With C++ Template. Kernels Such as 4×4, 3×4, ..., 1×4 are Generated at Compile Time for Corner Cases Without Any Runtime Overhead.

```
template<int m>
void Innerkernel_mx4(float A[k][m][4], float B[k
  ][4][4], float C[m][4][4], int k, int offset){
  // c00, a0, ... are 128bit NEON registers
  if (m > 0) { //load result C's 1st row
    c00 = vld1q_f32(C);      //load 1st vector
    c01 = vld1q_f32(C + 4); //load 2nd vector
    c02 = vld1q_f32(C + 8); //load 3rd vector
    c03 = vld1q_f32(C + 12);//load 4th vector
    C += offset;//move pointer to C's 2nd row
  }
  if (m > 1) { //load result C's 2nd row
    c10 = vld1q_f32(C);      //load 1st vector
    c11 = vld1q_f32(C + 4); //load 2nd vector
    c12 = vld1q_f32(C + 8); //load 3rd vector
    c13 = vld1q_f32(C + 12);//load 4th vector
    C += offset;//move pointer
  }
  ......     //load other N-2 rows of result C
  for (int i = 0; i < k; i++) {
    b0 = vld1q_f32(B);      //load 1st vector of B
    b1 = vld1q_f32(B + 4); //load 2nd vector of B
    b2 = vld1q_f32(B + 8); //load 3rd vector of B
    b3 = vld1q_f32(B + 12);//load 4th vector of B
    B += 16;
    if (m > 0) { //compute C's 1st row
      a0  = vld1q_f32(A);     //load A's 1st row
      c00 = vfmaq_f32(c00,a0,b0); //fma
      c01 = vfmaq_f32(c01,a0,b1); //fma
      c02 = vfmaq_f32(c02,a0,b2); //fma
      c03 = vfmaq_f32(c03,a0,b3); //fma
    }
    if (m > 1) { //compute C's 2nd row
      a1  = vld1q_f32(A + 4);//load A's 2nd row
      c10 = vfmaq_f32(c10,a1,b0); //fma
      c11 = vfmaq_f32(c11,a1,b1); //fma
      c12 = vfmaq_f32(c12,a1,b2); //fma
      c13 = vfmaq_f32(c13,a1,b3); //fma
    }
    ......//compute other N-2 rows of result C
    A += m*4;
  }
}
```

3.3 Generation of TensorGEMM Multiplication Kernels

TensorGEMM is a compute-intensive step in our Winograd implementation and requires more than 80% of the total time [15]. Therefore a set of multiplication kernels of arbitrary shapes ($m \times n$) should be developed to optimally fit different problem sizes, especially for skinny tall matrices.

We use Armv8 Neon primitives [29], [51] which support Fused-Multiply-Add (FMA) instructions for 32-bit floating-point numbers. Note that for most inner tiling sizes $m \times n$, the TensorGEMM's multiplication kernel sets up a group of accumulator registers, loads m tensors from a column in A, and n tensors from a row in B (as shown in Fig. 5). Subsequently, the tensors are multiplied and accumulated in the blue box D and are stored in $(m \times n)$ registers. When the computation progresses to the bottom border of A and the right of B, the register contents, namely the accumulators, are written back to the memory. Fig. 5 shows a schematic view of the described computational pattern, each tensor (an Armv8 register) holds 4 floats.

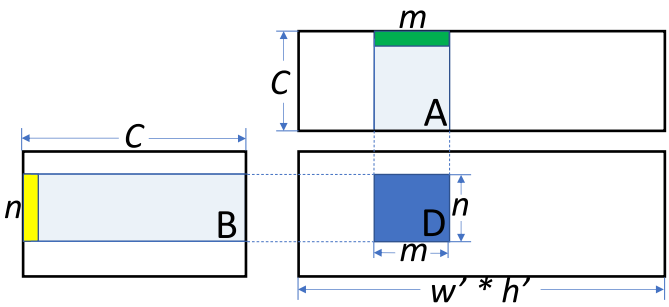


Fig. 5. Illustration of the processing order in TensorGEMM. Each element in the matrix is a tensor, which maps to a 128-bit vector register containing 4 floats on Arm.

We analyze the ratio of computation to memory operations [16] (known as Arithmetic Intensity (AI)). First, the arithmetic computation requires $2 \cdot m \cdot n$ FMA instructions. Only loads of operands from A and B are incurred at each compute iteration. The write-back only occurs when it traverses the entire loop. We estimate the lower bound of AI as the following formula:

$$\mathrm{AI} = \frac{2 \cdot m \cdot n}{m + n}, \tag{9}$$

Which monotonically increases with larger m and n. Generally speaking, a kernel function with larger AI performs better, i.e., can run closer to the device's peak performance. However, we require m and n registers to load operands from A and B, and $m{\cdot}n$ registers for the accumulators, respectively. Therefore, m and n have to satisfy the following constraint:

$$(m + n + m \cdot n) \leq R.$$

Where R is the number of accessible registers on each processor core (32 for Armv8 architecture).

A *multiplication kernel* is the smallest computational unit of TensorGEMM. When designing the optimal multiplication kernel, there are two principles we rely on to use the 32 Neon vector registers, as follows:

(I) Make full use of the compute units, fill the pipeline with instructions, and reduce pipeline stalls.
(II) Increase the arithmetic intensity to improve efficiency.

The goal of the first principle is to exploit instruction-level parallelism (ILP) for the multiplication kernel. Accordingly, we focus on designing a series of different shapes of multiplication kernels to cover various skinny tall and long rectangle shapes, whereas LIBXSMM [39] and OpenBLAS [52] employ a single kernel shape of highest AI while ignoring the skinny tall cases. We list seven feasible candidates for multiplication kernel shapes having AI values above or equal to four (listed in Table 2), and calculate the respective register usage. We implement those kernel functions with C++ templates, in order to ensure that code is generated at the compilation phase. We carefully tuned five kernel function templates by hand, and the other two multiplication kernels of the shapes (4×4) and (6×3) can be generated by the other optimized templates of the shapes (5×4) and (7×3). In addition to the aforementioned shapes, we also generate 22 auxiliary multiplications kernels with the aforementioned five kernel templates to better handle the corner cases, especially skinny tall matrices.

In Listing 1, the implementation of the multiplication kernel template of the shape$(m \times 4)$ is presented. The

TABLE 2
Seven Typical Shapes of Multiplications Kernels for TensorGEMM

m	3	3	4	4	5	6	7
n	7	6	5	4	4	3	3
Number of registers	31	27	29	24	29	27	31
AI (as in Eqn 9)	4.2	4	4.44	4	4.44	4	4.2

parameter m determines the shape of the generated kernels. We can get a multiplication kernel of the shape (5×4) by setting $m = 5$ in Listing 1. The lines 3 to 17 are used to control the load instruction with m. Lines 18 to 38 are the most compute-intensive part of this TensorGEMM kernel, the number of instructions used in this loop is also determined by m. Finally, m controls the number of instructions used to write the data back. All the conditional branching will not exist in the generated kernels.

The selection of inner kernels depends on the shape of input TensorGEMM matrices, the cache blocking size, and the characteristics of the multi-level memory-cache hierarchy of the given SoC chip. The multiplications kernel with the best performance will be selected through auto-tuning. To solve corner cases, when the size of the multiplication kernel is not divisible by the cache blocking factor in the column dimension. Zero padding is applied to account for the vector register and cache-line sizes. When there is a misalignment in the row dimension, the C++ template (e.g, *Innerkernel_mx4* in Listing 1) is used to generate more multiplications kernels during compile time. For example, with the kernel of shape $(m \times 4)$, our template will generate extra five kernels, such as (5×4), (4×4), ..., (1×4), to handle all corner cases on the row dimension.

3.4 Auto-Tuning in FastConv

The runtime parameters and their range of values listed in Table 3 are used for tuning our library (FastConv). Let S denote the size of the search space for the parameter:

$$S = \rho \cdot \psi \cdot \sum_{m,n \in Table\ 2} \left(\left\lfloor \frac{tiles}{m} \right\rfloor \cdot \left\lfloor \frac{K}{n} \right\rfloor \right). \tag{10}$$

Where ρ and ψ are the number of feasible values for the variants *loopReorder* and *onoffKernel*, respectively. Here, ρ denotes the four cases of loop order in cache blocking (line 21 and 22 in Algorithm 3) and register block (line 30 and 31 in Algorithm 4); ψ dennotes the onoffKernel Flag in line 24 for Algorithm 3. Thus the values of $\rho = 4$ and $\psi = 2$ are used in our implementation. To ensure divisibility of the register and cache block size, and to avoid misalignment in both the row and column dimensions, the tile cache block size tB must be a multiple of m and less than the total number of tiles $tiles$. The output channel block size oB needs to be multiple of n and will be less than K. As there are seven multiplication kernel shapes in Table 2, we accumulate them on each case, and then the total number of available choices for our parameters can be computed with Equation (10).

With these parameters, the actual execution pattern of the whole Winograd algorithm can be controlled with the generated multiplication kernels of TensorGEMM. The optimal parameter configuration with the best performance can be obtained by tuning over this parameter space. The optimal parameter configurations for a given problem size and hardware configuration are gathered and stored from offline runs, and the code with the best performance can be regenerated with those parameters.

TABLE 3
Runtime Parameters and the Search Space of the Our Portable Winograd Implementation

Parameters	Description	Value range
C	input channels	N/A
K	output channels	N/A
H, W	height and width of input image, respectively.	N/A
m	m tensors for register block on V_T	[2, 7]
n	n tensors for register block on U_T	[2, 7]
tB	tile cache block size	[0, tiles/m]
oB	output channel block size	[0, K/n]
$onoffKernel$	on/offline kernel transform tag (ψ in Eqn 10)	0, 1
$loopReorder$	loop reorder tag(ρ in Eqn 10)	0, 1, 2, 3

There are several steps in auto-tuning. First, the configurations of all possible ranges (intervals) of parameters define the parameter search space. Second, A tuning database is constructed to record the configurations of parameters' values and their corresponding performance results. Additionally, several algorithms such as grid search, random selection, and model-based prediction strategies, can be used along with the tuning database as a configuration generator to generate alternatives of configurations, for performance evaluation on a given target. Third, the code is generated using the parameters in the new configuration, then an offline performance test is done and the results are recorded in the tuning database. Finally, after a round of evaluations, the configuration with the best performance is used to auto-generate the code for the library and can be deployed to be used in production.

Searching the entire parameter space for parameter configurations yielding the best performance is time-consuming. One would usually not search the entire space, and instead use grid search, random selection, or even model-based prediction strategies to evaluate a small subset of the parameter space. In our auto-tuning module, the type and the granularity of the search strategies can be customized by the user. By default we use grid search as the default strategy: at most 4096 configurations are evaluated and stored in our tuning database. Finally, the configuration with the best performance will be selected and used for code generation with the best performance.

The auto-tuning strategy described above is performed offline. The time spent on auto-tuning for each convolution case varies from several minutes to several hours, depending on the input shape of convolution, the computing capability of the hardware, and the granularity of the grid search strategy.

3.5 User Interface and Implementation

A transparent and easy-to-use programming interface is designed for the proposed library. This library is header-only and therefore can be embedded into other third-party software stacks to accelerate inference on Arm CPUs with optimal convolution performance. Our convolution class contains three public function members: *Init()*, *Tuning()*, and *Forward()*. For library users, the user only needs to call three predefined routines:

TABLE 4
The Hardware Specifications of Our Test Platforms

CPU Name	Cores	#CPUs (GHz)	L1 Cache (Bytes)	L2 Cache (Bytes)	L3 Cache (Bytes)	Type
Snapdragon 835	4+4	4@2.45+4@1.90	-	4@2M-share+4@1M-shared	none	SoC/mobile
Snapdragon 855	4+4	(1@2.84+3@2.42)+4@1.80	-	(1@512K+3@256K)+4@128K	8@4MB-shared	SoC/mobile
Snapdragon 888	4+4	(1@2.84+3@2.42)+4@1.80	-	(1@1M+3@512K)+4@128K	8@4MB-shared	SoC/mobile
Apple M1	4+4	4@3.204+4@2.064	4@128K+4@64K	4@12M-shared+4@4M-shared	none	Consumer PC
Kunpeng 920	8	8@2.60	8@64K	8@512K	8@32MB-shared	Datacenter/server
AWS Graviton2	64	64@2.50	64@64K	64@1M	64@32MB-shared	Datacenter/server

Arm big.LITTLE is a heterogeneous computing architecture with a performance cluster of cores and power-efficient cluster of cores, Energy saving is ensured with clustered switching mechanism. This means that in most cases Arm SoC devices have two clusters of CPUs, and can only use one cluster at a time. "-" denotes that L1 cache size is not released by the vendor and also can not be measured by tools such as likwid [53].

(I) *Init()*: Construction of the *Conv* with a given shape on a target Arm CPUs.
(II) *Tuning()*: Configuring algorithms and parameters via offline auto-tuning before the actual deployment of the model.
(III) *Forward()*: In production phase, a call to *Init()* reads the configurations generated in the previous step, and the auto-generated kernel(s). Subsequently, *Forward()* is called to deliver the optimal computing performance on the target Arm CPU.

FastConv is the first work, to the authors' knowledge, that uses a reconfigurable design. The library internally generates the highest performing code variant for the given target Arm CPU. The code variants, optimized for different targets and convolution shapes, cover a wide range, and combinations, of optimizations: tuning the data layout for unit-strided access patterns, loop reordering, packing strategies for data blocks to interleave indexing, packing the layout back to match the auto-generated TensorGEMM inner kernel of different shapes, and register/cache blocking. These optimizations are combined together, in a transparent fashion, to deliver the optimal performance for the given convolution shape on the target chip, thus enabling its performance portability on different types of Arm CPUs.

4 Evaluation

FastConv is developed with C++ and Neon intrinsics. The correctness of our implementation has been verified against the naive implementation. The performance of the Winograd implementation is compared against FeatherCNN [10], NNPACK [18], Arm NN inference engine[28] and other backend libraries[27], [39], [40], [52] supporting GEMM routines. We evaluate six Arm processors. Three flagship mobile/SoC chips: Snapdragon 835, 855 and 888 from Samsung Galaxy S8, Xiaomi 9, and Xiaomi 11, respectively. Two data center servers: Huawei Kunpeng 920 and AWS Graviton2 M6g instance. One consumer PC: Apple MacBook Pro M1. Snapdragon 835 is equipped with four performance cores (Cortex-A73) with 2MB cluster sharing L2 cache, and four energy-efficient cores (Cortex-A53) with 1MB cluster sharing L2 cache. Snapdragon 855 is designed with one performance core (Cortex-A76) with 512KB L2 cache, three performance cores with 256KB L2 cache, four efficient cores with 128KB L2 cache, and DynamIQ 4MB cluster shared L3 cache. Snapdragon 888 shares a similar architecture with 855 but with double the L2 cache size. The hardware specifications of test platforms are detailed in Table 4.

We use VGG–16 [2], Resnet–50 [4], Densenet–121 [54] and Inception V4 [4] networks for performance evaluation. For VGG–16, the dimensions of TensorGEMM multipliers using Winograd are $K \times (C \cdot \theta)$ and $(C \cdot \theta) \times (H' \cdot W')$, as described in Table 5. The computational load together with its multiplication stage's Arithmetic Intensity (AI) for each layer is also calculated and listed in the Table. VGG–16's convolutional layers cover a wide variety of representative shapes generally composed of two typical patterns: large images with relatively few channels and small images with more channels. Similar shapes also appear in Resnet [4], Densenet [54], Squeezenet [55], and many other frequently used neural network architectures.

4.1 Performance Evaluation

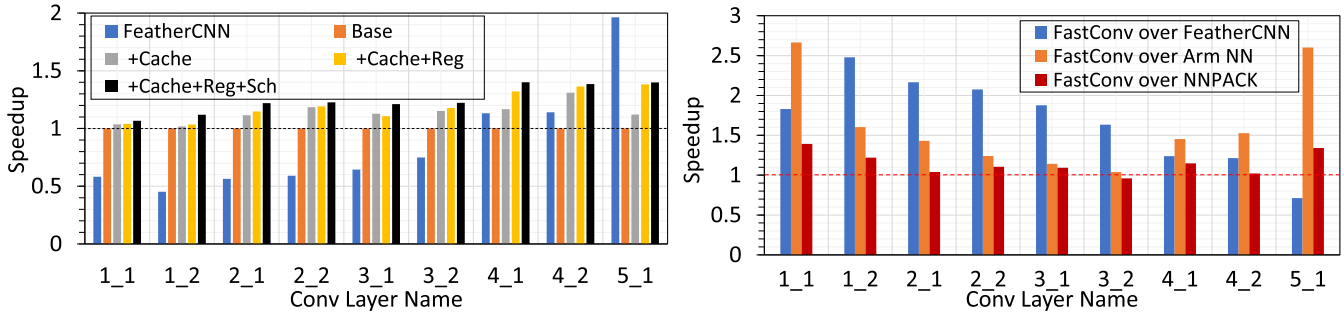
According to the introduced optimization techniques, the performance is evaluated for the following:

(I) A step-by-step evaluation of individual optimizations of the FastConv Winograd over the Winograd kernels of other libraries.
(II) A scalability evaluation of the multi-threaded FastConv Winograd.
(III) A roofline comparison and analysis for the multi-threaded Winograd implementation.
(IV) Performance portability evaluation of FastConv using different convolution shapes over different Arm CPUs.

We first conduct a step-wise evaluation on Winograd using VGG–16 convolution layers. We use our library with default settings (i.e. no optimizations) as a baseline in Fig. 6a. We use the default setting of $oB = 40$, $tB = 3$ and a fixed multiplication kernel of the shape 4×4. We enable the following optimizations one after the other (i.e. step-wise): cache blocking tuning (+*Cache*), register blocking tuning

TABLE 5
Shape, Computational Load and Arithmetic Intensity (AI) in Winograd $F(6 \times 6, 3 \times 3)$'s Multiplication Stage of *VGG–16* Conventional Layers

Layer	C	K	H, W	H′×W′ F(2×2, 3×3)	H′× W′ F(6×6, 3×3)	GFLOP	AI for F(6×6, 3×3)
1_1	3	64	224	12544	1444	0.17	0.747
1_2	64	64	224	12544	1444	3.70	11.24
2_1	64	128	112	3136	361	1.85	12.44
2_2	128	128	112	3136	361	3.70	19.86
3_1	128	256	56	784	100	1.85	15.91
3_2	256	256	56	784	100	3.70	21.63
4_1	256	512	28	196	25	1.85	9.44
4_2	512	512	28	196	25	3.70	10.25
5_1	512	512	14	49	9	0.92	2.77



(a) Step-wise speedup evaluation. (b) Speedup comparison.

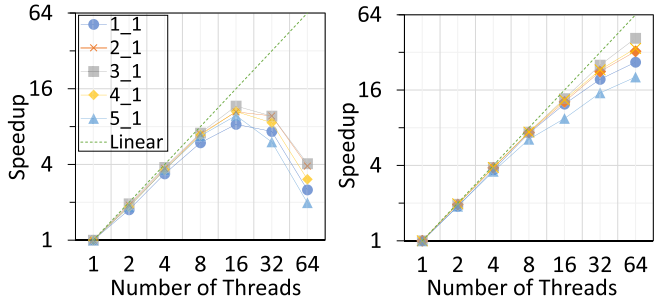
Fig. 6. Step-wise speedup evaluation and speedup comparison of FastConv on Kunpeng 920 using VGG-16. (a) Individual optimizations are added one by one in a step-wise evaluation. (b) Speedup comparison against Winograd kernels from FeatherCNN, Arm NN inference engine, and NNPACK. The baseline for (a) is the untuned FastConv initialized with a default setting of $oB = 40, tB = 3$, with a fixed multiplication kernel of the shape 4×4.

(+*Reg*), and scheduling loop reordering (+*Sched*). The combinations of all these optimizations (i.e. *Cache+Reg+Sched*) together is *FastConv*. The performance results of FeatherCNN are included in this evaluation. We plot its speedup against our baseline. The step-by-step single-thread optimization results are averaged over 10 runs on Kunpeng 920 in Fig. 6a. The results show that an average of 1.13x, 1.19x and 1.25x speedups are contributed by the tuning optimization of *Cache*, *Cache+Reg*, and *Cache+Reg+Sched*. FeatherCNN is slower than the baseline and FastConv on the layers before 4_1 and 5_1, respectively. FeatherCNN results show higher performance on extremely small input tensor sizes on layer 5_1 by using the *External Packing* strategy at which the data is packed in an extra memory buffer with a contagious memory access pattern [10]. This simplifies the implementation and improves the efficiency for smaller input shapes. Finally, FastConv achieves speedups between 1.07x to 1.40x, depending on different layers of VGG–16.

In addition, we conduct a comparison with three other Winograd implementations on Arm CPU, which includes FeatherCNN [10], Arm NN [28], and NNPACK [18] (Fig. 6b). In comparison to FeatherCNN, FastConv is 1.22x and 2.48x times faster (except for layer 5_1). The decreasing speedup with the shrinking input image size can be explained by the fact that the strided read pattern for both input and output transform does not work well with 4-way skewed associative cache on small image sizes with close memory access distances, across both different image rows and channels. This could be fixed by adjusting the memory access pattern and data layout. A comparison to NNPACK is also performed and the results show that FastConv is close to 1.40x times faster on both terminal layers, and is 1.02x to 1.15x better than NNPACK on the middle layers. When compared with Arm NN inference engine including kernel transformation, FastConv is 1.14x to 2.17x faster as there is a run-time overhead of the input tensor reshaping from "NHWC" to "NCHW", and another overhead for the strided data scattering and gathering operation before and after the GEMM routines. Additionally, the Arm NN inference engine employs a $F(4 \times 4, 3 \times 3)$ shape for its Winograd implementation, which may also degrade performance.

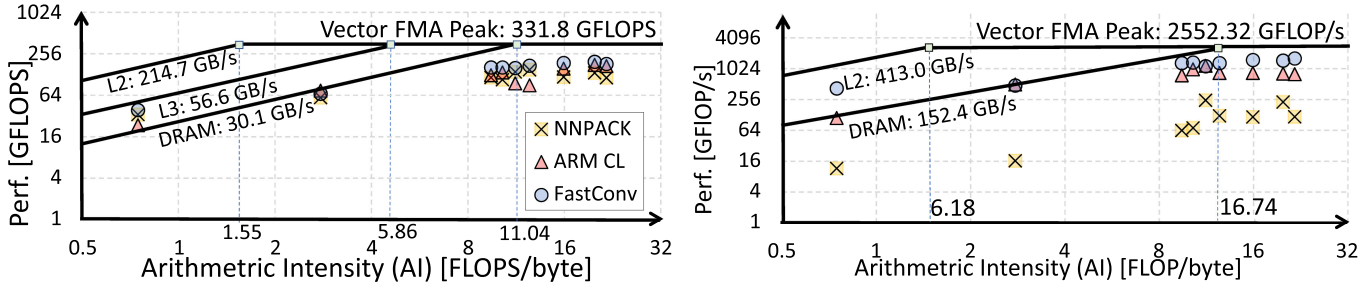
We perform a layer-wise scalability test on the Winograd implementation with VGG–16. Our multi-threaded implementation of the Winograd algorithm is compared with NNPACK. The scaling results on AWS Graviton2 Arm server with 64 cores are presented in Fig. 7. According to Fig. 7, FastConv and NNPACK can scale to 64 and 16 cores, respectively. There is an almost linear speedup with FastConv when running the middle layers of VGG–16. When dividing the speedup value by the number of threads, we can get the parallel efficiency numbers. When using all 64 cores, FastConv achieves 50% to 65% percent of parallel efficiency on the middle layers and between 32% to 42% on the first and last layers. The paralleling efficiency of NNPACK on 64 cores, however, ranges between 3% to 6%, on average. In comparison to FastConv, NNPACK shows poor efficiency and scalability performance in our experiments.

We do a roofline analysis for the most time-consuming step (i.e. multiplication stage) in the Winograd algorithm. In our Winograd optimization, we have minimized the memory movement overhead in transformation. At the same time, we are also trying to improve the computational efficiency in the most time-consuming step. We report the roofline results of the implementation of the multiplication stage in FastConv/TensorGEMM, NNPACK, and Arm NN inference engine with multiple GEMMs. The results are presented in Fig. 8. The roofline analysis on Kunpeng 920 with all 8 CPU cores for the three libraries is presented by Fig. 8a. Besides the first and last layers in VGG–16, FastConv/TensorGEMM is much closer to the peak in comparison with the other two libraries (note: Y-axis is log-scale). For the first and last VGG–16 layers, the multiplication using FastConv/TensorGEMM and GEMMs are all bounded by DRAM and L3, respectively. In these two cases, FastConv/TensorGEMM is still closer to the DRAM or L3 peaks. The results also indicate there is room for further performance improvement for the long rectangular or skinny tall cases for the Winograd algorithm. For the AWS Graviton2 Arm server, its per core cluster-shared L3 cache size is less



(a) Scaling of NNPACK's Winograd. (b) Scaling of FastConv's Winograd.

Fig. 7. Layer-wise scalability of VGG–16 on Winograd implementations of NNPACK and FastConv on AWS Graviton2 Arm Server.



(a) Roofline comparison using Kunpeng 920 with 8 cores.

(b) Roofline comparison using AWS Graviton2 with 64 cores.

Fig. 8. Layer-wise multi-core roofline comparison for the bottleneck multiplication stage of Winograd algorithm with VGG–16 on Kunpeng 920 and AWS Graviton2 (Y-axis log-scale).

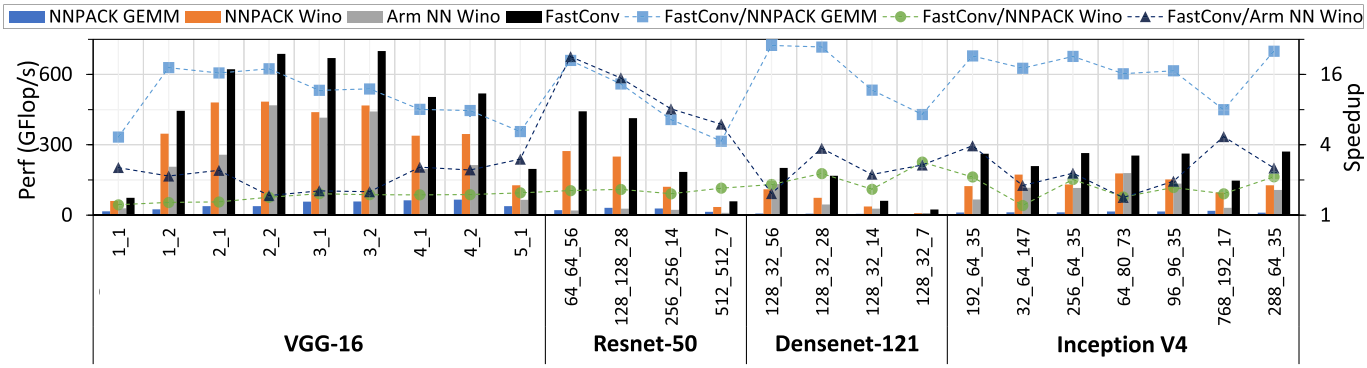


Fig. 9. Layer-wise multi-threaded performance comparison for convolution layers in VGG–16, Resnet–50, Densenet–121, and Inception V4 on Kunpeng 920 with all 8 cores. The gray, green and blue lines are the speedup of FastConv over the implementation of NNPACK's GEMM-Based algorithm, NNPACK, and Arm NN's Winograd algorithm, respectively. Besides VGG–16, the shape of each convolution layer in Resnet–50, Densenet–121, and Inception V4 is denoted with an ordered tuple (input channel size, output channel size, and width/height of the input 2D tensor).

than its L2 cache size. Thus we don't plot the L3 roofline in Fig. 8b. FastConv/TensorGEMM and Arm NN inference engine have the same performance in the last VGG–16 layer (5_1). On other layers with all 64 cores, FastConv/TensorGEMM achieves an order of magnitude improvement over the other libraries. More importantly, for the long rectangular or skinny tall convolution shapes in the first and last layer, TensorGEMM achieves a performance close to the L2 peak, in comparison to the other libraries (appearing to be bounded by memory, and not L2). Those performance improvements are attributed to our blocking, scheduling, and auto-tuning optimizations.

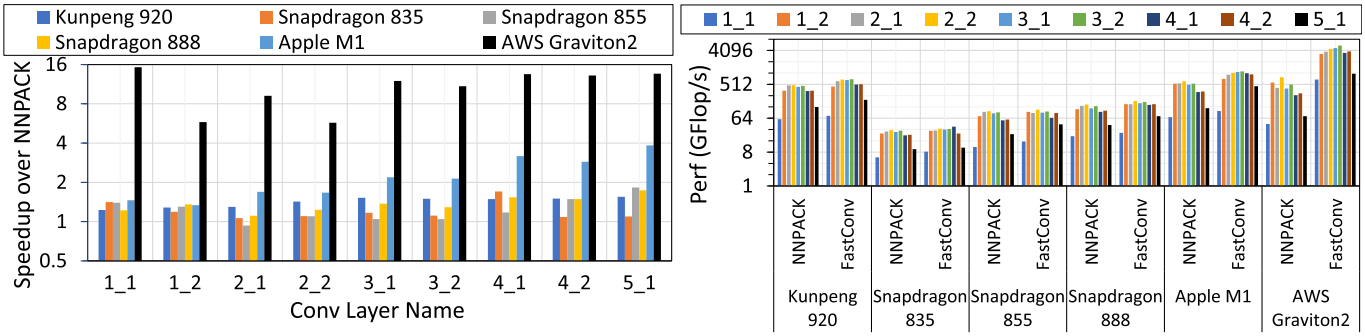
4.2 Portability Evaluation

In this section, we evaluate the performance portability of FastConv on various convolution shapes and six Arm CPU devices. The convolution layers from VGG–16 [2], Resnet–50 [4], Densenet–121 [54], and Inception V4 [4] are evaluated on all six platforms.

We evaluate the portability of FastConv (w.r.t. to input shapes) on various shapes of input layers on Kunpeng 920 with all 8 cores. The layer-wise efficiency and speedup results with nine layers from VGG–16 and convolution layers from Resnet–50, Densenet–121, and Inception V4 are shown in Fig. 9. Note that the seven middle layers of VGG–16 generate more square-shaped matrices than the two layers on both ends. The convolution layers from Resnet–50, Densenet–121, and Inception V4 generate small input image sizes with fewer input and output channels, that results in skinny tall and long rectangular GEMM/TensorGEMM input shapes. The default input tensor layout is set to be "NCHW", the input data layout transformation for GEMM-based convolution is included in the reported time to make a consistent comparison with the Winograd algorithm. We report the absolute performance in the unit of GFlop/s [1] The speedup results of FastConv over the other three libraries are presented in Fig. 9. FastConv with the Winograd algorithm is 4.30x to 28.36x faster than NNPACK's GEMM-based algorithm. In most cases, it is beyond Winograd's algorithmic speedup of 5.04, mainly due to the optimization and auto-tuning of our reconfigurable Winograd algorithm with auto-generated TensorGEMM kernels. For NNPACK's Winograd kernel, FastConv is 1.23x to 2.84x faster, which highlights the gains from our optimizations efforts in FastConv. When compared with the Arm NN inference engine, FastConv is approximately 1.47x to 3x faster on layers from VGG–16 layers, and 1.41x to 22.7x faster on layers from the three other networks. FastConv gains a larger speedup over the other libraries on Resnet–50, in comparison to VGG–16. That demonstrates our approach for auto-generating optimized convolution kernels is portable to various types of convolution shapes.

We test the performance portability on six Arm devices with nine layers from VGG–16. As multi-threaded deployment over multi-cores in mobile phones is not controllable on Android, we evaluate a single-thread on the big (performance) core on the three mobile processors. For the other

1. The performance in the unit of GFlop/s may be written as $2{\cdot}K{\cdot}C{\cdot}H{\cdot}W{\cdot}R{\cdot}S{\cdot}\tau^{-1}{\cdot}10^{-9}$, where τ is the runtime in seconds, the other symbols are listed in Table 5.



(a) Speedup comparison of FastConv over NNPACK's Winograd.

(b) Performance comparison on FastConv and NNPACK's Winograd.

Fig. 10. Layer-wise multi-threaded performance results with VGG–16 on six ARM CPU SoC devices. The X-axis shows the layer name in VGG–16. The Y-axis is the speedup and absolute performance value in GFlop/s of FastConv and NNPACK on Winograd algorithm.

three devices, multi-threading on all cores is evaluated. FastConv can automatically select the best parameters and configure the C++ template to generate optimized code for the target convolution shape and Arm architecture. NNPACK however does not have this functionality. We have measured the speedup of FastConv over NNPACK on their best performing Winograd implementation (in Fig. 10. In comparison with NNPACK, FastConv achieves speedups of 1.42x, 1.21x, 1.26x, 1.37x, 2.26x, and 11.02x on average over Kunpeng 920, Snapdragon 835, 855, 888, Apple M1, and AWS Graviton2, respectively. A notable observation is that with newer chips, FastConv gains better speedups. Huawei Kunpeng 920 is based on Cortex-A57 (released in 2012), Snapdragon 835, 855, and 888 are based on Cortex-A73, Cortex-A76, and Cortex-X1 (released in 2015, 2018, and 2020 respectively), and Apple M1 with Firestorm architecture released in 2021. An important fact to consider is that existing libraries are highly hand-tuned to target some special architectures released several years ago. With new chips being introduced, and the high pace at which the field of deep learning evolves, developers are faced with the futile task of redoing the hand-tuned optimizations. For instance, NNPACK's kernel has not been updated for years, and the latest architecture ported by OpenBLAS is Cortex-A73 (released in 2016). Thus, it is vital to have a performance portable library (FastConv) that can support both old and new Arm SoCs and servers. Furthermore, the whole porting process is fully automated in FastConv and can save enormous engineering effort for deploying DL models on billions of Arm SoC chips in phones and Internet of Things (IoT) devices [56], [57].

4.2.1 Comparison With TVM

TVM can only auto-tune GEMM kernels for convolution operations by using the Im2col algorithm, thus we compare to TVM by including the Im2col in our call to the GEMM routines.

We implemented a reconfigurable Im2col algorithm combined with automatically generated GEMM code that is optimized with the techniques described in this paper. We refer to the GEMM based convolution implementation as FastConv-GEMM. It is open-sourced, and we include it in the same GitHub repo https://github.com/Mengjintao/FastConv. FastConv–GEMM has been compared to AutoTVM, AutoTVM + LIBXSMM, AutoTVM + OpenBLAS [40], [58], OpenBLAS [52], and LibShalom [27]. The overhead of the Im2col transformation is excluded. The reconfigurable library for GEMM with default parameters on cache block size and inner kernel shapes of 8×8 is labeled as FastConv without auto-tuning, while the auto-tuning enabled version is labeled as FastConv + tuning. The achieved performance on Kunpeng 920 is shown in Fig. 11. OpenBLAS shows better performance than AutoTVM, and AutoTVM that is tuned over OpenBLAS and LIBXSMM. LibShalom [27] shows performance improvement over both autoTVM and OpenBLAS. It is worth mentioning that LibShalom is optimized for small and irregular-shaped GEMMs with start-of-art expert hand-tuning methodologies. Our reconfigurable library FastConv–GEMM with default settings is comparable to LibShalom and also outperforms other libraries. With auto-tuned FastConv, we further gain between 2% to 17% performance improvement for different layers and rank first on all layers (except layer 5_1). Note that the matrices generated by the middle layers of VGG16 are more square-shaped, whereas the matrices of the terminal layers are mostly long rectangular or skinny tall. This explains why the first three layers and the last layer in VGG–16 benefit more from auto-tuning, in comparison to middle layers. It can be concluded that our auto-tuning methodology on GEMM with the reconfigurable library can improve the performance of long rectangular and skinny tall matrices, and at the same time has no negative effects on square matrices.

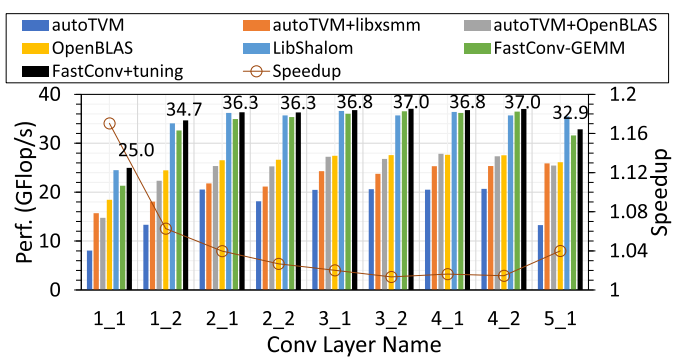


Fig. 11. Step-wise evaluation of FastConv's GEMM implementation for Im2col on Kunpeng 920. The x-axis is layer names from VGG–16 (Table 5). The left y-axis plots the performance in GFlop/s and the right y-axis plots FastConv's auto-tuning speedups over FastConv–GEMM. Note that the theoretical peak performance of a single core in Kunpeng 920 is 41.6 GFlop/s.

4.3 Discussion

In this section, we briefly discuss some insights we observe from analyzing the experimental results.

4.3.1 Effect of Architecture Features on Optimizations

FastConv's offline tuner can generate auto-tuning logfiles of the best configurations during our evaluation experiments. We highlight three optimization patterns from the analysis of these log files using a single thread. First, the log files generated by the cache blocking step confirm ideal blocking with Algorithm 3, i.e. only compulsory cache misses, in L2/L3 cache (depending on which cache level the auto-tuning blocking is for). Second, inspecting the auto-tuning logfiles of register blocking with Algorithm 4 revealed that for most kernel shapes the best decomposition of the input shape was used (w.r.t. to the reduction in register pressure) while avoiding the cost for data padding. For example, the kernel shape of (7×3) is used for the last layer of VGG–16 and thus successfully avoids the data padding operations; for other big square input tensor shapes the register kernel shape of (5×4) with highest arithmetic intensity is selected by our auto-tuner. Finally, the offline kernel transformation in Algorithm 3 reduced the computational requirements for kernel transformations, in all cases, when using a single thread and enabled us to saturate the memory bus. The above cache-aware decomposition/blocking of input shapes, extended from Goto's work [16], optimally selects the register blocking for kernel shapes. The loop reordering method, inspired by TVM [40], [58], also provides an advantage over NNPACK [18] and Arm NN inference engine [28].

We also analyzed the multi-threaded auto-tuning logfiles generated by FastConv's offline tuner on Kunpeng 920, AWS Graviton2, and Apple M1. We highlight four optimization patterns for multi-threaded auto-tuning. First, effective cache blocking is observed, similar to the case of a single core. Second, the loop reordering step imitates the parallelism over the inner-kernel [59] by scattering threads on register blocking loops instead of cache blocking loops and tends to keep the blocking size on the dimension of the input channel as large as possible to saturate the instruction pipeline. This is consistent with our analysis in Section 3.2 on cache blocking optimizations. Third, the most frequently used kernels that are selected by the auto-tuner have the shape of (4×5) or (4×4), which provides the highest arithmetic intensity for both common and corner case kernels. Finally, offline kernel transform is disabled for multi-thread cases to save memory bandwidth and avoid memory access conflicts; this is a different pattern from the results of a single thread.

4.3.2 Lower and Mixed Precision

Parts of the Arm processor families, such as Snapdragon 835 and Kunpeng 920, we experiment with in this paper do not belong to the ARMV8.2–A [60] architecture that supports FP16 and Int8. Arm CPUs adopting the ARMV8.2–A architecture started to appear in market in 2020. Considering there is still a large number of ARM devices not supporting ARMV8.2–A, this work is focused on FP32 to ensure their compatibility. When using FP16 and Int8 precisions we can reduce the required storage of DL models and also improve the inference performance. If we assume independence from the restrictions of NEON instructions, our work can use Int4/8 and FP16 in our TensorGEMM templates, where the number of issued passes p (as in Equation 7) of TensorGEMM can be reduced exponentially. This would be helpful in further reducing, or even eliminating, the required interleaved packing data movement in the Winograd algorithm.

5 Conclusion

We have presented a library named *FastConv* that is performance portable for Winograd convolution operations on many types of recent Arm CPUs. A combination of several technologies is used to deliver transparency and performance portability. We use C++ templates to generate multiple shapes of manually tuned multiplication kernels fully optimized for high arithmetic intensity. FastConv is designed to search for the best combination of register and cache blocking sizes, scheduling of loop iterations, packing strategies, access patterns, and online/offline computations. Auto-tuning is also applied to search the configurations for the best performance for the considered target devices and problem sizes. Our experimental layer-wise evaluation on VGG–16 confirms that after tuning our Winograd reconfigurable Library, speedups of 2.0x and 1.1x can be achieved on average over Arm Inference engine and NNPACK, respectively, when running VGG–16 layers on Kunpeng 920. Our performance portability evaluations on different models further show that an average speedup of 1.21x, 1.55x, 1.72x, and 2.08x is achieved on Snapdragon 835, 855, 888, and Apple M1, respectively. The entire porting process is fully automated and can thus save enormous engineering work for the deployment of DL models on millions of Arm SoC chips in mobile phones and IoT devices.

Acknowlegments

We appreciate the long-term guidance from Prof. Tong Zhang at the Hong Kong University of Science and Technology, the General Manager Assistant Mr. YongSheng Liu, General Manager Assistant Mr. Wei Yang, and Prof. Junzhou Huang at the University of Texas at Arlington. We also want to thank the editors and reviewers for their professional comments which have greatly improved this manuscript.

References

[1] S. Kim, S. Oh, and Y. Yi, "Minimizing GPU kernel launch overhead in deep learning inference on mobile GPUs," in *Proc. 22nd Int. Workshop Mobile Comput. Syst. Appl.*, 2021, pp. 57–63.

[2] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015, *arXiv:1510.00149*.

[3] Z. Zhong, L. Jin, and Z. Xie, "High performance offline handwritten chinese character recognition using googlenet and directional feature maps," in *Proc. 13th Int. Conf. Document Anal. Recognit.*, 2015, pp. 846–850.

[4] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 4278–4284.

[5] Z. Qin, Z. Zhang, X. Chen, C. Wang, and Y. Peng, "FD-mobilenet: Improved mobilenet with a fast downsampling strategy," in *Proc. 25th IEEE Int. Conf. Image Process.*, 2018, pp. 1363–1367.

[6] X. Xia, C. Xu, and B. Nan, "Inception-v3 for flower classification," in *Proc. 2nd Int. Conf. Image Vis. Comput.*, 2017, pp. 783–787.

[7] E. Georganas *et al.*, "Anatomy of high-performance deep learning convolutions on simd architectures," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, pp. 830–841.

[8] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.

[9] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.

[10] H. Lan *et al.*, "FeatherCNN: Fast inference computation with tensorgemm on arm architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 580–594, Mar. 2020.

[11] P. Maji, A. Mundy, G. Dasika, J. G. Beu, M. Mattina, and R. D. Mullins, "Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus," 2019, *arXiv:1903.01521v1*.

[12] L. Jia, Y. Liang, X. Li, L. Lu, and S. Yan, "Enabling efficient fast convolution algorithms on gpus via megakernels," *IEEE Trans. Comput.*, vol. 69, no. 7, pp. 986–997, Jul. 2020.

[13] R. Mulder, V. Radu, and C. Dubach, "Fast optimisation of convolutional neural network inference using system performance models," in *Proc. 1st Workshop Mach. Learn. Syst.*, 2021, pp. 104–110.

[14] Z. Jia, A. Zlateski, F. Durand, and K. Li, "Optimizing N-dimensional, winograd-based convolution for manycore CPUs," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 109–123.

[15] D. Yan, W. Wang, and X. Chu, "Optimizing batched winograd convolution on GPSs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 32–44.

[16] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, 2008, Art. no. 12.

[17] ARM, "ARM compute library." Accessed: Feb. 26, 2022. [Online]. Available: https://github.com/ARM-software/ComputeLibrary

[18] M. Dukhan, "The NNPACK library." Accessed: Dec. 22, 2020. [Online]. Available: https://github.com/Maratyszcza/NNPACK

[19] NVIDIA, "The NVIDIA CUDA deep neural network library (cuDNN)." Accessed: Jan. 25, 2022. [Online]. Available: https://developer.nvidia.com/cudnn

[20] Intel, "Intel oneAPI deep neural network library (oneDNN)." Accessed: Mar. 09, 2022. [Online]. Available: https://github.com/oneapi-src/oneDNN

[21] Wikipedia, "Cache hierarchy," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Cache_hierarchy

[22] Wikipedia, "Arm big.little," 2021. [Online]. Available: https://en.wikipedia.org/wiki/ARM_big.LITTLE

[23] ARM, "Arm dynamiq redefines multi-core computing," 2021. [Online]. Available: https://www.arm.com/why-arm/technologies/dynamiq

[24] J. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge, U.K.: Cambridge Univ. Press, 2010. [Online]. Available: https://books.google.com.sg/books?id=EgMZEqnvLzsC

[25] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[26] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Comput. Archit. Lett.*, vol. 13, no. 1, pp. 21–24, Jan.-Jun. 2014.

[27] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, "LIBSHALOM: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores," in *Proc. Int. Conf. High Perf. Comput. Netw. Storage Anal.*, 2021, pp. 1–14.

[28] A. Tools, "ARM NN Inference Engine," 2021, Accessed: Sep. 01, 2021. [Online]. Available: https://github.com/ARM-software/armnn

[29] ARM, "Arm neon technologies," 2021. [Online]. Available: https://www.arm.com/why-arm/technologies/neon

[30] N. Stephens, "ARMv8-a next-generation vector architecture for HPC," in *Proc. IEEE Hot Chips 28 Symp.*, 2016, pp. 1–31.

[31] S. Flur *et al.*, "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Languages*, 2016, pp. 608–621.

[32] H. Ni, "Ncnn," [Online]. Available: https://github.com/Tencent/ncnn

[33] P. S. Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, "High performance and portable convolution operators for arm-based multicore processors," 2020, *arXiv:2005.06410*.

[34] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.

[35] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched GEMM for GPUs," in *Proc. Int. Conf. High Perform. Comput.*, 2016, pp. 21–38.

[36] C. Cecka, 2017. [Online]. Available: https://devblogs.nvidia.com/cublas-strided-batched-matrix-multiply/

[37] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Batched matrix computations on hardware accelerators based on GPUs," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 2, pp. 193–208, 2015.

[38] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, and M. Zounon, "Optimized batched linear algebra for modern architectures," in *Proc. Eur. Conf. Parallel Process.*, 2017, pp. 511–522.

[39] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating small matrix multiplications by runtime code generation," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 981–991.

[40] T. Chen *et al.*, "TVM: End-to-end optimization stack for deep learning," 2018, *arXiv:1802.04799*.

[41] Google, "JAX: Autograd and XLA," 2021. [Online]. Available: https://github.com/google/jax

[42] X. Jiang *et al.*, "Mnn: A universal and efficient inference engine," 2020, *arXiv:2002.12418*.

[43] S. Joo *et al.*, "A memory-aware performance optimization of tensor programs for embedded devices," in *Proc. IEEE Int. Conf. Consum. Electronics-Asia*, 2020, pp. 1–4.

[44] W. Niu, X. Ma, Y. Wang, and B. Ren, "26ms inference time for resnet-50: Towards real-time execution of all DNNs on smartphone," 2019, *arXiv:1905.00571*.

[45] X. Zhang, J. Xiao, and G. Tan, "I/O lower bounds for auto-tuning of convolutions in CNNs," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 247–261.

[46] L. Zheng and T. Chen, "Optimizing deep learning workloads on arm GPU with TVM," in *Proc. 1st Reproducible Qual.-Efficient Syst. Tournament Co-Designing Pareto-Efficient Deep Learn.*, 2018, Art. no. 1.

[47] J.-K. Lee, A. Lu, Y.-M. Chang, C.-L. Lee, P. Chen, and S.-C. Wang, "Supporting TVM on risc-V architectures," in *Proc. TVM Deep Learn. Compiler Conf.*, 2018, pp. 1–4.

[48] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[49] Z. Xianyi, W. Qian, and Z. Chothia, "Openblas," 2012. [Online]. Available: http://xianyi. github. io/OpenBLAS

[50] Wikipedia, "Honor of Kings — Wikipedia, the free encyclopedia," 2021, Aug. 31, 2021. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Honor%20of%20Kings&oldid=1040445127

[51] V. G. Reddy, "Neon technology introduction," *ARM Corporation*, vol. 4, no. 1, p. 1, 2008.

[52] X. Zhang, "OpenBLAS library," [Online]. Available: https://github.com/xianyi/OpenBLAS

[53] T. Gruber, J. Eitzinger, G. Hager, and G. Wellein, "likwid 5: Lightweight performance tools," in *Proc. SC19 Conf.*, 2019, pp. 1–2.

[54] F. N. Iandola, M. W. Moskewicz, S. Karayev, R. B. Girshick, T. Darrell, and K. Keutzer, "DenseNet: Implementing efficient convnet descriptor pyramids," 2014, *arXiv:1404.1869*.

[55] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," 2016, *arXiv:1602.07360*.

[56] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.

[57] M. A. Khan and K. Salah, "IoT security: Review, blockchain solutions, and open challenges," *Future Gener. Comput. Syst.*, vol. 82, pp. 395–411, 2018.

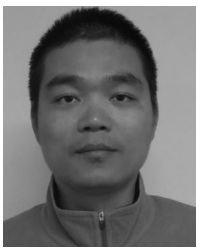
[58] T. Chen *et al.*, "Learning to optimize tensor programs," 2018, *arXiv:1805.08166*.

[59] T. M. Smith, R. Van De Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 1049–1059.

[60] D. Brash, "Armv8-a architecture evolution," 2016. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-a-architecture-evolution

Jintao Meng received the BS and MS degrees in computer science from Central China Normal University, Wuhan, in 2005 and 2008, respectively, and the PhD degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2016. He is currently an associate researcher with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include high-performance computing, bioinformatics, and graph computing.

Chen Zhuang received the BS degree in Internet of Things engineering from the Guangdong University of Technology, Guangzhou, China, in 2019. He is currently working toward the MS degree in computer technology with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research focuses on high-performance computing, with a focus on the acceleration of the parallel AI systems.



Peng Chen received the BE degree in navigation from Dalian Maritime University, China, in 2005, the ME degree in traffic information engineering and control from Shanghai Maritime University, China, in 2007, and the PhD degree from the Tokyo Institute of Technology, Japan, in 2020. He is currently a researcher with the National Institute of Advanced Industrial Science and Technology. He is also a visiting scientist with the RIKEN Center for Computational Science, Japan. His research interests include parallel computing, image processing, and machine learning.

Mohamed Wahib received the PhD degree in computer science from Hokkaido University, Japan, in 2012. He is currently a senior scientist with AIST/TokyoTech Open Innovation Laboratory, Tokyo, Japan. Prior to that, he was a researcher with the RIKEN Center for Computational Science. Prior to his graduate studies, he was a researcher with Texas Instruments (TI) R&D Labs, Dallas, TX, USA, for four years. His research interests include the central topic of performance-centric software development in the context of HPC.

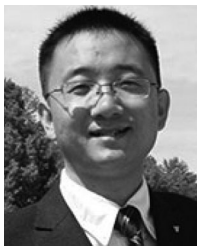
Bertil Schmidt (Senior Member, IEEE) is currently a tenured full professor and the chair of parallel and distributed architectures with the University of Mainz, Germany. Prior to that, he was a faculty member with Nanyang Technological University, Singapore, and with the University of New South Wales. His research group has designed a variety of algorithms and tools for bioinformatics, mainly focusing on the analysis of large-scale sequence and short read datasets, and data mining. For his research work, he was the recipient of CUDA Research Center Award, CUDA Academic Partnership Award, CUDA Professor Partnership Award, and Best Paper Award at IEEE ASAP 2009 and IEEE ASAP 2015.

Xiao Wang received the PhD degree in electrical and computer engineering from Purdue University in 2017, under the supervision of professor Charles Bouman and Samuel Midkiff. He is currently a research staff with Oak Ridge National Laboratory. From 2017 to 2021, he was a postdoc research fellow with Harvard Medical School, working on medical imaging research.

Haidong Lan received the BS and PhD degrees in computer science from Shandong University, Jinan, China, in 2013 and 2018, respectively. He is currently an senior engineer with Tencent AI Platform Department. His research interests include high-performance computing, especially in the areas of bioinformatics and deep learning.

Dou Wu received the BS degree in computer science and technology from Jilin University, Changchun, China, in 2020. He is currently working toward the MS degree in electronic science and technology with the Southern University of Science and joint training with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include high-performance computing and deep learning.

Minwen Deng received the BS degree in computer science from Sun Yat-sen University in 2006 and the MS degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences in 2009. From 2009 to 2010, he was an engineer with Alibaba Cloud. Since 2010, he has been a senior engineer and principal investigator with Tencent, leading the work on AI infrastructure construction.



Yanjie Wei received the BS degree in applied physics from Sichuan University in 2004 and the PhD degree in computational biophysics from Michigan Technological University in 2007. He is currently a researcher with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include computational biology and bioinformatics, focusing on protein folding and structure prediction, and gene sequence analysis.

Shengzhong Feng received the bachelor's degree from the University of Science and Technology of China in 1991 and the PhD degree from the Beijing Institute of Technology in 1997. He is currently the director of National Supercomputing Center, Shenzhen. Before joining SIAT in 2009, he was an associate professor with the Institute of Computing Technology, Chinese Academy of Sciences and a visiting professor with the University of Toronto. He has authored or coauthored more than 60 research papers which are indexed by SCI/EI, and applied more than 20 patents in resent five years. His research interests include high-performance computing, cloud computing, and bioinformatics. He is the technology leader, is responsible for the establishment of the National Supercomputing Center, Shenzhen. He also participated in the development of Dawning series supercomputer and as the principle investigator of many national level projects, such as 863 program, NSFC projects, knowledge innovation project of the Chinese Academy of Sciences. He was the recipient of many awards, such as the Hundred-Talent Program from Chinese Academy of Sciences, Outstanding Science and Technology Progress Award from Chinese Academy of Sciences, and second prize of the National Science and Technology Progress Award.