# Optimizing Direct Convolutions on ARM Multi-Cores

Pengyu Wang[1,*], Weiling Yang[1,*], Jianbin Fang[1,†], Dezun Dong[1],
Chun Huang[1], Peng Zhang[1], Tao Tang[1], Zheng Wang[2]
1 College of Computer Science and Technology, National University of Defense Technology, China
2 School of Computing, University of Leeds, United Kingdom
{pengyu_wang, w.yang, j.fang, dong, chunhuang, zhangpeng13a, taotang84}@nudt.edu.cn, z.wang5@leeds.ac.uk

## ABSTRACT

Convolution kernels are widely seen in deep learning workloads and are often responsible for performance bottlenecks. Recent research has demonstrated that a direct convolution approach can outperform the traditional convolution implementation based on tensor-to-matrix conversions. However, existing approaches for direct convolution still have room for performance improvement. We present nDirect, a new direct convolution approach that targets ARM-based multi-core CPUs commonly found in smartphones and HPC systems. nDirect is designed to be compatible with the data layout formats used by mainstream deep learning frameworks but offers new optimizations for the computational kernel, data packing, and parallelization. We evaluate nDirect by applying it to representative convolution kernels and demonstrating its performance on four distinct ARM multi-core CPU platforms. We compare nDirect against state-of-the-art convolution optimization techniques. Experimental results show that nDirect gives the best overall performance across evaluation scenarios and platforms.

## CCS CONCEPTS

• **Computing methodologies → Machine learning**; • **Software and its engineering → Compilers**.

## KEYWORDS

Convolution, Direct Algorithm, Neural networks, ARMv8 Multi-Core, Performance Optimization

## 1 INTRODUCTION

Convolutional Neural Networks (CNNs) are one of the most popular deep neural network architectures and are found to be successful in a wide range of tasks, including image classification [38, 60], object detection [46, 55, 56], natural language processing [43], and semantic segmentation [58, 68]. The core component of a CNN is the convolutional (CONV) operation [23, 35, 51, 63], which is often responsible for the performance bottleneck of a CNN, accounting for over 90% of the CNN execution time [28]. As such, there has been considerable interest in optimizing convolution implementations to accelerate CNNs [1, 10, 21].

Traditionally, CONV kernels were implemented as general matrix multiplications (GEMM) [1, 3, 9, 10]. This approach maps the input tensor into a row- or column-major matrix through format conversion (also known as *im2col*) to translate CONV operations into GEMM kernels [18, 52]. By using this matrix format, the convolution operation can be performed as a single matrix multiplication to take advantage of the highly optimized GEMM kernel accelerated using heavily optimized BLAS (Basic Linear Algebra Subprograms) libraries [3, 7, 11].

However, im2col can increase the memory footprint and the tensor-to-matrix conversion can result in an irregular-shaped matrix with sub-optimal performance [31, 53, 66]. As such, more recent approaches attempt to optimize CONV operations without converting the input tensors into matrices. This strategy is known as *direct convolution* [16, 24, 27, 30–32, 45, 49, 52, 54, 65, 67, 69]. It works by sliding a CONV kernel over the input tensor and computing the dot product between the kernel and a small patch of the input at each position. This operation is repeated for every input position to produce a feature map. Direct convolution has two advantages over im2col. Firstly, direct convolution has lower memory requirements as it operates directly on the input tensor, avoiding transforming the input tensor into a larger matrix. This can improve cache locality and reduce memory usage. Secondly, direct convolution can exploit the sparsity of the convolution kernel and avoid unnecessary computations [54], leading to faster computation times.

LIBXSMM is the state-of-the-art library-based solution for implementing direct convolutions on CPUs [5, 6, 31, 33]. It uses a specialized data layout and the Batch-Reduce GEMM (BRGEMM) as the computational kernel [32, 33]. It gives improved performance over im2col+GEMM on x86 and ARM CPUs. While promising, LIBXSMM has two fundamental drawbacks. First, its data layout design is incompatible with the common data layouts (i.e., $NCHW$ or $NHWC$)[1] used in mainstreamed deep learning (DL) frameworks [8, 13, 30]. Therefore, integrating the BRGEMM routines into DL frameworks requires either code refactoring to the underlying DL framework or introducing a format conversion stage at the user code when calling and exiting each CONV operator. The latter requires changes to the standard user model code and will incur additional overhead. Second, LIBXSMM still uses a conventional GEMM-based microkernel, which fails to leverage potential data reuse opportunities in convolutions [9, 52] to improve performance. Other work on direct convolutions [27, 33, 52, 65, 67, 69] also failed to address the two aforementioned limitations within a single framework. These approaches either use a different data layout with integration issues [33, 52, 67], or sacrificing performance to maintain compatibility with standard data layouts [27, 65, 69].

This paper presents nDirect[2], a new direct convolution solution with a focus on providing high performance, high data reusability, and DL framework compatibility. Our work explicitly targets

[*] Equal contribution
[†] Corresponding author

[1] $NCHW$=[Batch Size, Input Channels, Input Height, Input Width]; $NHWC$=[Batch Size, Input Height, Input Width, Input Channels].
[2] The code and data for this paper are publically available at: https://github.com/nDIRECT/nDIRECT.

**Table 1: Summary of notations used in the paper**

| | Description | | Description |
|---|---|---|---|
| $N$ | Batch Size | $K$ | Output Channels |
| $C$ | Input Channels | $R$ | Kernel Height |
| $H$ | Input Height | $S$ | Kernel Width |
| $W$ | Input Width | $P$ | Output Height |
| $Q$ | Output Width | $str$ | Stride |
| $I$ | Input Tensor | $O$ | Output Tensor |
| $F$ | Filter Tensor | | |

ARM multi-core CPUs widely seen in smartphones and HPC systems, which are also commonly used for CNN model inference. nDirect implements new strategies for micro-kernel computation, data packing and parallelization. nDirect is designed to be compatible with mainstreamed DL frameworks and does not require code refactoring of the underlying CONV implementations or the user model code. Instead of transforming data between different data layouts [31, 67], nDirect adheres to the conventional tensor formats by converting the data layout of filter tensors on the fly, providing compatibility with existing DL frameworks. It leverages SIMD instructions to implement a CONV-friendly computation pattern. Unlike prior work's sequential data packing method, nDirect overlaps data packing memory accesses with computation operations to hide the memory access latency.

We demonstrate the benefit of nDirect by applying it to three HPC multi-cores and one embedded CPU of the ARMv8 architecture. We evaluate nDirect by measuring its performance on individual convolution layers and the end-to-end inference time of representative CNN models. We compare nDirect against four state-of-the-art convolution approaches [18, 27, 31, 70]. We show that nDirect consistently delivers better performance across hardware platforms. We showcase that, despite being a low-level library-based method and lacking high-level optimizations like operator fusion, nDirect is competitive to Ansor, an automated search framework within TVM, for the end-to-end inference optimization.

This paper makes the following contributions:

- It presents a new direct convolution algorithm that preserves the conventional data layouts used by mainstream DL frameworks;
- It proposes a new way to implement convolution computation kernels, which outperforms existing solutions;
- It provides a set of analytical models to derive the optimal algorithmic parameters.

## 2 BACKGROUND AND PROBLEM SCOPE

Table 1 summarizes the CONV notations used throughout in the paper.

### 2.1 Prior Convolution Implementations

Algorithm 1 gives a straightforward, unoptimized implementation of CONV, which has seven nested loops around a multiply-and-accumulate statement. The algorithm uses stride ($str$) to determine how to move the input tensor $I$ across the $S$ spatial space of the filter tensor $F$, to generate the output tensor $O$. As there are no dependencies across the loop iterations, the computation can be permuted and tiled to improve the performance [45].

Algorithm 1 can be typically improved using four strategies [48], including the direct convolution targeted in this work, the im2col+GEMM approach, FFT (Fast Fourier Transform) and Winograd [44]. While FFT and Winograd can reduce the computation complexity, they have limited applications [41, 50]. This is because the two methods can increase the memory pressure and reduce the prediction accuracy [42]. Since our work focuses on optimizing CONV without compromising prediction accuracy, direct convolution and im2col+GEMM are the most relevant methods.

**Algorithm 1:** Naive Direct Convolution Algorithm

```
Input: $I[N][C][H][W]$; $F[K][C][R][S]$
Output: $O[N][K][P][Q]$
for $n = 0$ to $N - 1$ do
  for $c = 0$ to $C - 1$ do
    for $k = 0$ to $K - 1$ do
      for $oj = 0$ to $P - 1$ do
        for $oi = 0$ to $Q - 1$ do
          $ij = str \cdot oj$
          $ii = str \cdot oi$
          for $r = 0$ to $R - 1$ do
            for $s = 0$ to $S - 1$ do
              $O[n][k][oj][oi]$ +=
                $I[n][c][ij + r][ii + s] * F[k][c][r][s]$
```

### 2.2 Im2col+GEMM Approach

The process of lowering the CONV kernel to GEMM is known as im2col. A GEMM computation generally contains three dimensions, referred to as $M'$, $N'$ and $K'$ in this paper. Given a convolution size, this process involves flattening and patching images according to columns and then arranging these columns into a concatenated matrix. The convolution kernels are stored in matrix format ahead of time and called upon by a GEMM routine to execute convolutions. While using optimized GEMMs can speed up convolutions, this method has additional memory overhead, and the available hardware memory bandwidth can restrict its performance. This is a particular problem for the parallel execution of CONV on multi-core CPUs with large batch sizes because the available bandwidth to each core may be insufficient to achieve optimal GEMM performance.

### 2.3 Direct Convolution

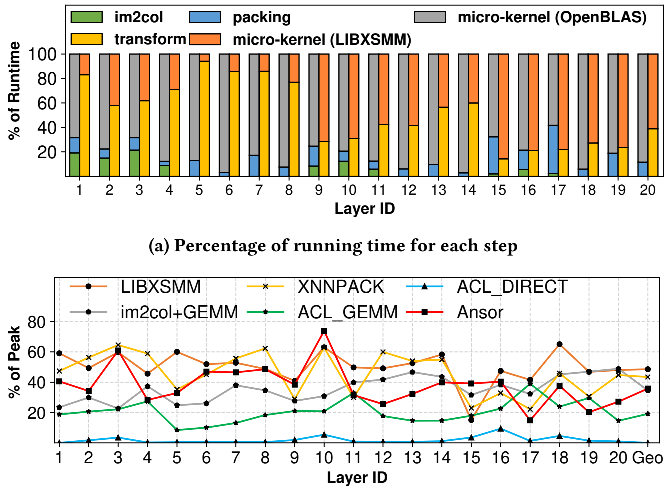
There have been several attempts to optimize direct convolutions with varying degrees of success. For example, the ARM Compute Library (ACL) [1] supports direct convolution implementation, but it gives a poor performance on our evaluation platform and only works for very limited configurations. LIBXSMM [5, 31] is most closely relevant to nDirect, but it uses a new storage format so as to enhance data locality and utilizes SIMD instructions. Additionally, it tiles loops to accommodate small matrix multiplications as the innermost micro-kernel, which is generated by just-in-time (JIT) [39] compilation. In this process, the filter with a data layout of $KCRS$ is converted into a tensor with dimensions $\lceil\frac{K}{k}\rceil \cdot \lceil\frac{C}{c}\rceil \cdot R \cdot S \cdot c \cdot k$, and the $NCHW$ input tensor is converted into a tensor with dimensions $N \cdot \lceil\frac{C}{c}\rceil \cdot H \cdot W \cdot c$.

### 2.4 Search-based Code Optimization

There is also a body of work on auto-tuning the DNN back-end code generation [15, 19, 20, 47, 64, 71]. The Ansor [70] in the TVM DL compiler [19] employs evolutionary search with a predictive model to search an optimized code schedule by looking at optimizations like loop tiling and instruction scheduling. The code schedule is then

**Table 2: Comparing nDirect against prior conv. solutions**

| | im2ol+ GEMM | XNNPACK | LIBXSMM | Ansor | nDirect |
|---|---|---|---|---|---|
| **Approach** | Library | Library | JIT | Search | Library |
| **Required format conversion?** | ✓ | ✓ | ✗ | ✓ | ✓ |
| **Low memory footprint?** | ✗ | ✓ | ✓ | ✓ | ✓ |
| **High performance?** | ★ | ★★ | ★★ | ★★ | ★★★ |



(a) Percentage of running time for each step

(b) 64-cores CONV Performance on Phytium 2000+

**Figure 1: CONV performance of (a) breakdown and (b) multi-core on the ARMv8-based Phytium 2000+ processor.**

passed to the back-end code generator (e.g., the LLVM compiler) to emit machine instructions. Ansor supports auto-tuning for direct convolution with the *NCHW* data layout. Ansor also leverages the operator fusion technique from Relay [57] into computational subgraphs for code optimizations.

## 2.5 Positioning Our Work

Table 2 summarizes prior work in format conversion, memory footprint, and performance. LIBXSMM requires format conversion, requiring code refactoring or will incur conversion overhead. Im2col+GEMM could increase memory pressure. All prior methods also leave much room for performance improvement. Therefore, nDirect aims to fill this gap by preserving the mainstream DL formats and achieving high performance.

# 3 MOTIVATION AND OVERVIEW

Our work is motivated by the observation that current convolution optimization methods have room for performance optimization or have compatibility issues with mainstreamed DL frameworks (Table 2). These methods include im2col+ OpenBLAS GEMM [11, 18], LIBXSMM [5, 6, 31, 33], XNNPACK [14, 27], tuned direct convolution by Ansor [70], direct and GEMM-based methods provided by ACL [1].

To illustrate these points, consider optimizing CONV operator on Phytium 2000+, a 64-core ARMv8 multi-core CPU [29]. We employ ResNet-50, a popular CNN utilized for object detection [38] and set the batch size to match the number of physical cores available [31], while executing various CONV layers of different sizes from ResNet-50 (see Table 4).

## 3.1 Motivation Results

Figure 1 shows the convolution performance of representative implementations, and we normalize the throughput (GFLOPS) relative to the theoretical peak performance of Phytium 2000+ with 64 cores.
**Breakdown of overhead.** Figure 1a gives a breakdown of the runtime overheads for each part of im2col+GEMM and LIBXSMM's direct convolution approaches. In the case of im2col+GEMM, the runtime overheads arise from data packing, im2col transformation and micro-kernel calls. Convolutions with $R > 1$ and $S > 1$ require im2col transformation, which causes expensive data duplication cost. Additionally, the overhead of data packing can not be ignored, accounting for up to 40% of total expenses for CONV layer 17. For LIBXSMM, assuming the adoption of conventional data formats *NCHW*, the runtime overheads originate from data format transformation and micro-kernel calls. As presented in Figure 1a, the cost of data format transformation accounts for the majority of overall overhead, with up to 90% of total execution time for CONV layer 5.
**Parallel execution.** Figure 1b displays the performance of individual CONV layers from ResNet-50 when executed using a batch size of 64 on 64 cores. It is worth noting that we only measure the performance of LIBXSMM's micro-kernels to observe the benefits of using a cache-friendly data format. Despite performing the best, LIBXSMM only delivers an average 50% of the theoretical CPU peak performance. In addition, we observe that im2col+GEMM achieves 40% of the peak performance. For convolutions without im2col transformation, such as CONV layers 19 and 20, GEMM methods achieve close to 50% of the peak performance.

## 3.2 Opportunities for Improvement

After closely examining the results and the implementation of prior work, we have identified three opportunities for improvement, described as follows.

First of all, compatibility with the mainstream data layout used in DL frameworks is important for the adoption of these approaches. While LIBXSMM has achieved promising convolution performance, it introduces new data layouts designed to improve cache locality and exploit vectorization. However, incorporating such new data layouts into mainstream DL applications would require significant redevelopment of existing frameworks and entail substantial engineering efforts. This is challenging for processors like ARM CPUs, which often lack DL software support compared to x86 and GPUs. Alternatively, without changing the underlying DL framework, data format conversion will need to be performed by the user code before and after calling each CONV operator. This not only requires user code refactoring but the expensive overhead of format

**Algorithm 2:** nDirect Convolution

```
Input: $I[N][C][H][W]; F[K][C][R][S]$
Output: $O[N][K][P][Q]$
1:  L1:  for ($n = 0; n < N; n++$) do
2:  L2:    for ($ht = 0; ht < H; ht+ = T_h$) do
3:  L3:      for ($ct = 0; ct < C; ct+ = T_c$) do
4:  L4:        for ($kt = 0; kt < K; kt+ = T_k$) do
                 /*Transform the filter's layout $T_k T_c RS$ to $\lceil \frac{T_k}{V_k} \rceil T_c RSV_k$*/
5:                 transform_filter();
6:  L5:            for ($hv = ht; hv < T_h; hv++$) do
7:  L6:              for ($wv = 0; wv < W; wv+ = V_w$) do
8:                     Input_Buffer B ← Pack_Micro-kernel();
9:  L7:                for ($kv = kt + V_k; kv < kt + T_k; kv+ = V_k$) do
10:                      Main_Micro-kernel(B);
```

conversion can also outweigh the benefit of im2col. For instance, the conversion time for CONV layers 1 in Figure 1a is around 4× of the actual computation time. Therefore, a better scheme should minimize the disruption to the existing DL software systems, making integrating them into existing DL frameworks easier without significant redevelopment effort.

Secondly, we identified opportunities to enhance the performance of GEMM-based convolution methods. Although LIBXSMM uses optimized micro-kernels and a cache-friendly data format to achieve fast direct convolution, we found that its loop tile sizes are too small to fully utilize the multi-level caches and fused multiply-accumulate (FMA) units available in modern ARMv8 multi-cores. And the sequential load instructions generated by LIBXSMM's JIT compiler can cause pipeline stall hazards.

Moreover, we noticed that the im2col transformation and sequential data packing utilized in the im2col+GEMM approach can also hamper performance by generating significant memory load and store operations. This can result in slowdowns when multiple threads are competing for memory bandwidth. Therefore, an ideal micro-kernel for convolution should have high performance and no additional memory access overhead.

Thirdly, we observed that existing parallelization strategies are coarse-grained, contributing to the poor convolution performance on ARM multi-cores. For example, ACL's direct convolution achieves only 5% of the multi-core peak performance on Phytium 2000+. This is because of the strategy's naive naïve parallelization of the $K$ dimension without considering the convolution workloads characteristics, such as the batch size $N$ and input shape $H \times W$. As a result, the computations are performed sequentially over multiple batches, resulting in linear cost accumulation. Further optimization is needed to overcome this problem.

To summarize, these findings indicate that there is still considerable potential for performance improvement when optimizing convolution operations on ARM multi-core CPUs.

## 3.3 Overview

nDirect exploits the opportunities identified in Section 3.2. We achieve this by redesigning direct convolution with compatible data layouts, new micro-kernels and suitable parallelization strategies optimized for multi-core CPUs.

**Data layout.** To be compatible with mainstream DL frameworks (e.g., Tensorflow [13] and MXNet [8]), nDirect preserves the conventional *NCHW* and *NHWC* data layouts. In this paper, we explain nDirect using the *NCHW* data layout as an example.

**Algorithm implementations.** Algorithm 2[3] outlines the nDirect convolution for the *NCHW* data format, inspired by the GEMM block algorithm [34]. We tile the filter and input tensors at two levels to improve spatial data locality. The first level of tiling exploits cache usage (lines 2–4) and determines the tile size based on the capacity of each level of cache, as described in Section 4. The second tiling level uses vector registers (lines 6–9) with a tile size that maximizes floating-point arithmetic intensity (*FAI*), as detailed in Section 5. We use the outer-product method to update the output tensor $O$ since its *FAI* is higher than the inner-product method, allowing us to access elements of the filter tensor $F$ more continuously. This is also the reason for focusing on the format conversion of the tensor $F$ (line 5). Figure 2 illustrates that the input tensor's spatial data locality is poor, and the processor can only continuously access $V_w$ elements at each iteration. To address this, we map its elements to a continuous buffer (line 8).

**Road map.** In the upcoming sections, we will delve into three essential optimizations that we have implemented in nDirect to minimize data movements when permuting loops (Section 4), introduce a novel micro-kernel that is optimized for convolution (Section 5), and outline our parallelization strategy (Section 6). Our current implementation supports single floating-point (FP32) as this is the most commonly used data type for CNN, but our techniques can be applied to other data types, including FP16, FP64 and INT16.

## 4 NDIRECT DESIGN

Algorithm 2 shows the main computation kernel of nDirect, described in the following subsections. nDirect follows the design principle of the classical *Goto* algorithm for matrix multiplications [34, 62].

### 4.1 Loop Ordering

Since a CONV operator can be considered a high-dimensional GEMM, we map the CONV's dimensions to *Goto's* loop ordering as outlined in Algorithm 2. Specifically, we map the CONV dimensions to the GEMM (*i.e.*, $M'$, $N'$, and $K'$) computation dimensions of the input tensor $I$, the filter tensor $F$, and the output tensor $O$, as $K \rightarrow M'$, $N \times H \times W \rightarrow N'$, and $R \times S \times C \rightarrow K'$. The specific mapping method and data flow scenarios are as follows.

In Algorithm 2, we use loops $L2$ and $L3$ to partition $I$ into sub-blocks that can fit into the last-level cache (LLC). Unlike the *Goto* algorithm, we choose not to pack the elements of $I$ between these two levels of loops. This is because the CNN tensor is often irregular-shaped, i.e., one of the tensor dimensions can be much smaller than the others. Prior studies [67] have shown that data packing can introduce additional memory operations that cannot be amortized by the improved performance for irregularly shaped GEMMs [66].

Loops $L3$ and $L4$ partition the $F$ into a series of sub-blocks that can fit into the L2 cache. Here, we choose to transform elements of filter $F$ into continuous memory space on the fly. This is because the size of $F$ is typically much smaller than that of the $I$, i.e., $K << N \times H \times W$. During the packing step, the processor accesses filter $F$ in a pipelined manner, where the packing overhead can be hidden. Moving to loops $L5$ and $L6$, we further divide the sub-blocks of

[3] To simplify the presentation, we set $str = 1$ in the algorithm.

**Algorithm 3:** Main_Micro-kernel

```
Input: Input_Buffer IB; Transformed_Filter TF
1:  L8:  for ($cv = ct; cv < ct + T_c; cv++$) do
2:  L9:    for ($r = 0; r < R; r++$) do
3:           $I_b \leftarrow 14 \times (3 \times (cv - ct))$
4:           $(V2 - V5) \leftarrow IB[I_b : I_b + 14]$
      /*Fully unroll loop with upper bound $S$, e.g., a $3 \times 3$ convolution kernel*/
5:           $F_b \leftarrow 8 \times (9 \times (cv - ct) + 3 \times r)$
6:           $(V0 - V1) \leftarrow TF[F_b : F_b + 8]$
        /*scalar-vector multiply*/
7:           $(V8 - V19) \leftarrow FMA((V2[0] - V4[3]), V0)$
8:           $(V20 - V31) \leftarrow FMA((V2[0] - V4[3]), V1)$
9:           $(V0 - V1) \leftarrow TF[F_b + 8 : F_b + 16]$
10:          $(V8 - V19) \leftarrow FMA((V2[1] - V5[0]), V0)$
11:          $(V20 - V31) \leftarrow FMA((V2[1] - V5[0]), V1)$
12:          $(V0 - V1) \leftarrow TF[F_b + 16 : F_b + 24]$
13:          $(V8 - V19) \leftarrow FMA((V2[2] - V5[1]), V0)$
14:          $(V20 - V31) \leftarrow FMA((V2[2] - V5[1]), V1)$
15:      Store to Output
```

input $I$ into data slices that fit in the L1 data cache. Similarly, loop $L7$ partitions the sub-blocks of $F$ into data slices.

At the first iteration of loop $L4$, the elements from the $I$ are fetched from the main memory into vector registers in a non-streaming manner. For subsequent iterations of this level of the loop, the sub-block of $I$ will reside in the LLC. Since the data block of filter $F$ is preloaded into the L2 cache (line 5 of Algorithm 2), it keeps in the L2 cache when iterating loops $L5$ and $L6$. In the packing micro-kernel, elements of $I$ are fetched from the main memory to vector registers. Section 5.3 describes the design of this micro-kernel in detail. When iterating over loop $L7$, elements of $I$ will be used by the packing micro-kernel residing in the L1 data cache.

## 4.2 Determine the Tiling Size

Loop tiling is key to improving cache data locality. In this subsection, we explain how to determine the sizes of $T_h$, $T_c$, and $T_k$ in Algorithm 2. Note that we will discuss the block size of the micro-kernel in Section 5. Our design aims to take advantage of the vector FMA units while leveraging the memory hierarchy of caches and vector registers.

To optimize the L1 data cache utilization, each $R \times T_c \times (V_w + S - 1)$ slice of the input $I$ should be kept in the L1 cache during each iteration of loop $L7$. Furthermore, the L1 cache should also hold two $V_k \times T_c \times R \times S$ slices of $F$ at this loop level. Therefore, $T_c$ must satisfy the following constraints:

$$R \times T_c \times (V_w + S - 1) + 2 \times V_k \times T_c \times R \times S < C_{L1} \tag{1}$$

Section 5.2.3 show that the optimal value of $V_k$ and $V_w$ are 8 and 12 respectively on our evaluation platforms. Then we can obtain $T_c$ with Equation 1.

Similarly, we would like the L2 cache to keep one $T_k \times T_c \times R \times S$ block of filter $F$ during each iteration of loop $L6$, and two $R \times T_c \times (V_w + S - 1)$ slices of input $I$ at loop $L6$. Because the L2 cache on ARM CPUs often holds both data and instructions simultaneously, the cache needs to reserve some space for the instructions being executed and data elements of $O$. Therefore, $T_k$ and $T_c$ must satisfy:

$$T_k \times T_c \times R \times S + 2 \times R \times T_c \times (V_w + S - 1) < C_{L2} \tag{2}$$

With Equations 1 and 2, we can obtain $T_k$ and $T_c$. Likely, we can derive $T_h$ in a similar way by considering the capacity constraint of the L3 cache (should this be available on the underlying hardware).

# 5 MICRO-KERNEL DESIGN

nDirect incorporates two micro-kernels that are specifically customized to maximize $FAI$ and minimize data access latency. The first micro-kernel is designed to accelerate convolutions, corresponding to line 10 of Algorithm 2. The second micro-kernel is responsible for packing input tensor $I$ and performing calculations in the first iteration of loop $L_7$ (line 8 of Algorithm 2).

## 5.1 Design Overview

nDirect aims to improve the data reuse of direct convolution and leverage the ARM NEON SIMD extensions to boost instruction level parallelism. Specifically, we utilize the 32 128-bit-wide vector registers ($V0-V31$) and the arithmetic fused multiply-accumulate (FMA) unit available on ARMv8 CPUs. The challenge is to select suitable vector parameters ($V_k$ and $V_w$) to maximize register multiplexing and $FAI$. To this end, we use analytical methods to guide our optimization. As a working example, we use FP32 tensor datatype and a $3 \times 3$ convolution kernel to explain our approach in this section, but our techniques can be applied to other datatype and convolution kernels by adjusting the parameters of the analytical models.

## 5.2 Main Micro-kernel

*5.2.1 Optimization constraints.* Figure 2 depicts convolution workflows of nDirect, where it applies $\lceil \frac{V_w+S-1}{4} \rceil$, $\frac{V_k}{4}$ and $\frac{V_w \times V_k}{4}$ vector registers to store single-precision floating elements from input, filter and output tensors, respectively. To make sure that the required data can fit into the available vector registers, $V_w$ and $V_k$ have to satisfy:

$$\begin{cases} \lceil \frac{V_w+S-1}{4} \rceil + \frac{V_k}{4} + \frac{V_w \times V_k}{4} \leq 32 \\ V_k \% 4 = 0 \end{cases} \tag{3}$$

Since each vector register can hold 4 FP32 elements, the vector length, $V_k$ is set to be a multiple of 4 to fully utilize the vector units.

*5.2.2 Optimization goal.* Algorithm 3 outlines the micro-kernel implementation in nDirect. Here, we unroll the loop with an upper bound of $S$ for a convolution kernel size of $S$ (lines 5-14). Our objective is to maximize $FAI$ in one iteration of loop $L9$. To illustrate the algorithm workflow, we use a $3 \times 3$ convolution kernel as an example.

During each iteration of loop $L9$, we initially load $V_w + S - 1$ input and $V_k$ filter tensor elements into vector registers. We then use scalar-vector multiplication with FMA instructions to compute $V_w \times V_k$ output elements, resulting in $2 \times V_w \times V_k$ floating-point operations. Note that each FMA instruction includes an addition operation and an multiplication operation. Figure 2(a) illustrates the first round of the calculation. After completing the first round of calculation, we update the vector registers that store the filter elements. At the same time, the input data related to the convolution operation requires an offset of step size 1 in the vector registers. We perform similar operations at the end of the second round of calculation (Figure 2(b)). Finally, we formulate the average $FAI$ in one iteration of loop $L8$ as follows:

$$FAI = \frac{2 \times 3 \times V_w \times V_k}{V_w + S - 1 + S \times V_k} \tag{4}$$

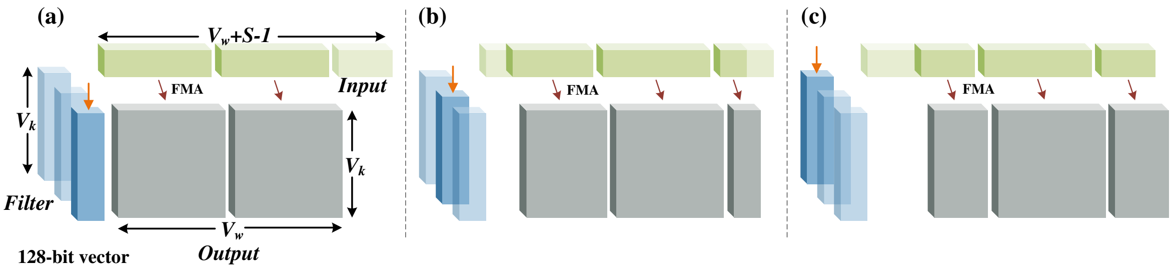


**Figure 2: The nDirect convolution workflow in one iteration of loop $L9$ in Algorithm 3 (lines 3 to 14). The input, output and nontransparent filter blocks are held in vector registers. Arrows from input to output represent FMA operations.**

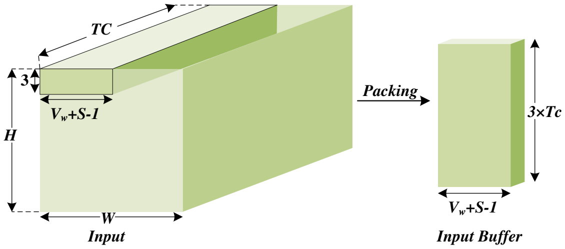


**Figure 3: nDirect packing example. Here, nDirect packs $3\times(V_w+S-1)$ elements from each of the $TC$ continuous input channels into a linear buffer.**

5.2.3 *Solving equations.* To optimize nDirect, we consider the constraints defined in Equation 3 and the goal defined in Equation 4. To maximize the *FAI*, we adopt the Lagrange multipliers method [36] to find the optimal values of $V_w$ and $V_k$ for CPU architectures used in our evaluation.

## 5.3 Micro-kernel for Packing

Conventional im2col+GEMM uses a sequential packing strategy by mapping the discontinuous input matrix elements into a linear buffer *before* performing computation. This strategy can reduce memory access latency during computation but introduces additional overhead as can be seen from Figure 1a. nDirect also aims to pack discontinuous input tensor elements into a linear buffer, which is smaller than the L1 cache, but it tries to hide the packing latency. Note that the input tensor elements used are identical in each iteration of loop $L7$ in Algorithm 2. nDirect performs data packing in the first iteration (line 8 of Algorithm 2).

Figure 3 shows how nDirect packs tensor elements. Generally, $T_c\times3\times(V_w+S-1)$ input elements accessed in loop $L7$ in Algorithm 2 are initially distributed in $T_c$ continuous channels of input tensor $I$. In the first iteration of loop $L7$, nDirect calls `Pack_Micro-kernel` to pack discontinuous $T_c \times 3 \times (V_w + S - 1)$ input elements into a linear buffer named $B$ (line 8 in Algorithm 2). Since sequential write operations with data dependencies can incur pipeline stall hazards, nDirect places store (st) instructions immediately after FMA instructions to hide packing overheads by utilizing the out-of-order instruction execution of modern CPUs. In each subsequent iteration, input data are fetched from linear buffer $B$, designed to improve the L1 data cache hit rate.

# 6 PARALLELIZATION STRATEGIES

We use OpenMP with static work partitioning to parallelize CONV operations on shared-memory multi-core CPUs. To utilize hardware parallelism, we use all available CPU cores, meaning that we will spawn $PT$ parallel threads for a CPU with $PT$ cores. Ideally, all the cores would start and finish the work simultaneously, thus not having any core idling at any point in time. However, this is not always possible due to memory access latency and application workloads. As such, we need to carefully determine how many threads are used to parallelize each of the parallel dimensions.

## 6.1 Model Thread Mapping

nDirect parallelizes the $N$, $K$, $H$ and $W$ dimensions in Algorithm 2. We do not parallelize the reduction dimensions of $C$, $R$ and $S$, because doing so can result in write conflicts since all participating threads would write to the same location in the $O$. While these conflicts can be eliminated using locks or additional memory buffers, the associated runtime overhead can be high [45].

To map threads onto computation dimensions, we use $PT_k$ threads to parallelize the $K$ dimension and $PT_n$ threads to parallelize the $N$, $H$ and $W$ dimensions, where $PT_k \times PT_n = PT$. At runtime, each thread performs $\frac{K\cdot N\cdot H\cdot W\cdot C\cdot R\cdot S}{PT\cdot str^2}$ numbers of arithmetic operations. Similarly, the number of memory accesses to filter $F$ within each parallel thread is $\frac{K\cdot C\cdot R\cdot S}{PT_k}$, which is accessed in a streaming manner meaning that the memory accesses are performed on continuous addresses. Additionally, memory access to input tensor $I$ required by each parallel thread is $\frac{N\cdot C\cdot H\cdot W}{PT_n\cdot str^2}$, which is accessed in a non-streaming manner. To model the difference in accessing latency between streaming and non-streaming memory accesses, we introduce a coefficient $\alpha$ to memory accesses to $I$. Therefore, the $FAI$ for each thread is:

$$FAI = \frac{\frac{K\cdot N\cdot H\cdot W\cdot C\cdot R\cdot S}{PT\cdot str^2}}{\frac{K\cdot C\cdot R\cdot S}{PT_k} + \alpha\cdot\frac{N\cdot C\cdot H\cdot W}{PT_n\cdot str^2}} = \frac{1}{\frac{PT_n\cdot str^2}{N\cdot H\cdot W} + \frac{\alpha}{K\cdot R\cdot S\cdot PT_n}} \qquad (5)$$

Our objective is to maximize $FAI$, which means minimizing $\frac{PT_n\cdot str^2}{N\cdot H\cdot W} + \frac{\alpha}{K\cdot R\cdot S\cdot PT_n}$.

Table 3: Hardware platforms used in evaluation

| | Phytium 2000+ | KP920 | ThunderX2 | RPi 4 |
|---|---|---|---|---|
| **Number of Cores** | 64 | 64 | 32 | 4 |
| **Peak FP32 GFLOPS** | 1126.4 | 2662.4 | 1279.7 | 56.8 |
| **Frequency** | 2.2 GHz | 2.6 GHz | 2.5 GHz | 1.8 GHz |
| **Max Bandwidth** | 143.1 GiB/s | 190.7 GiB/s | 158.95 GiB/s | 16.8 GiB/s |
| L1 cache | 32 KB | 64 KB | 32 KB | 32 KB |
| L2 cache | 2 MB | 512 KB | 256 KB | 1 MB |
| L3 cache | None | 64 MB | 32 MB | None |

## 6.2 Solving the Equation

By applying the inequality of arithmetic and geometric mean method to Equation 6, we have:

$$FAI \le \frac{\sqrt{N \cdot H \cdot W \cdot K \cdot R \cdot S}}{2 \cdot \sqrt{\alpha} \cdot str} \tag{6}$$

where both sides of the equation will equal if $\frac{PT_n \cdot str^2}{N \cdot H \cdot W} = \frac{\alpha}{K \cdot R \cdot S \cdot PT_n}$. In other words, when $PT_n = \sqrt{\frac{\alpha \cdot N \cdot H \cdot W}{K \cdot R \cdot S \cdot str^2}}$, $FAI$ would reach its maximum value. Since the micro-kernel for packing (Section 5.3) has little overhead, we take the up-bound value of $PT_n$, i.e, $PT_n = \lceil \sqrt{\frac{\alpha \cdot N \cdot H \cdot W}{K \cdot R \cdot S \cdot str^2}} \rceil$. Note for dimensions of $N$, $H$ and $W$, the priority of parallelization is $N$, $H$ and $W$. Specifically, if $\frac{PT_n}{N} > 1$, we will use $N$ threads to parallelize dimension $N$, and $\frac{PT_n}{N}$ threads to parallelize dimension $H$. For the targeting hardware platform, we use microbenchmarks to determine $\alpha$ by accessing the memory space in a streaming and non-streaming manner. Since the value is determined offline and is a one-off cost, it does not affect the runtime performance.

# 7 EXPERIMENTAL SETUP

We evaluate nDirect by comparing it against four existing convolution implementations described in Section 7.3. Our evaluation includes layer-wise performance comparison and end-to-end inference of the entire CNN network.

## 7.1 Hardware Platforms

Our experiments were performed on three HPC systems and one embedded system with ARM multi-cores. Our evaluation platforms include Phytium 2000+ [62], Kunpeng 920 (KP920) [4], ThunderX2 [37], and a Raspberry Pi 4 (RPi 4) [12]. Table 3 provides an overview of the specifications for these platforms. It is worth noting that the L2 cache on Phytium 2000+ is shared between a cluster of four cores, while it is private to a processor core on KP920 and ThunderX2.

## 7.2 Convolution Workloads

We use convolution layers from two representative CNNs: ResNet-50 [38] and VggNet-16 [60]. They are widely used for large-scale image recognition. Table 4 gives the experimental parameters used for each layer. We set the batch size to match the number of physical cores to evaluate the performance of multi-batch CONV operations and CNNs end-to-end inference.

## 7.3 Baseline Implementations

We compare nDirect against the following baselines:

Table 4: Configurations of convolution operators in ResNet-50 (IDs 1-23) and VGG-16 (IDs 24-28)

| **ID** | C | K | H/W | R/S | str | **ID** | C | K | H/W | R/S | str |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 64 | 224 | 7 | 2 | 15 | 512 | 14 | 3 | 3 | 2 |
| 2 | 128 | 128 | 56 | 3 | 2 | 16 | 256 | 14 | 3 | 3 | 1 |
| 3 | 64 | 64 | 56 | 3 | 1 | 17 | 1024 | 2048 | 14 | 1 | 2 |
| 4 | 256 | 512 | 56 | 1 | 2 | 18 | 256 | 1024 | 14 | 1 | 1 |
| 5 | 64 | 64 | 56 | 1 | 1 | 19 | 1024 | 512 | 14 | 1 | 1 |
| 6 | 64 | 256 | 56 | 1 | 1 | 20 | 1024 | 256 | 14 | 1 | 1 |
| 7 | 256 | 64 | 56 | 1 | 1 | 21 | 512 | 512 | 3 | 3 | 1 |
| 8 | 256 | 128 | 56 | 1 | 1 | 22 | 512 | 2048 | 7 | 1 | 1 |
| 9 | 256 | 256 | 28 | 3 | 2 | 23 | 2048 | 512 | 7 | 1 | 1 |
| 10 | 128 | 128 | 28 | 3 | 1 | 24 | 64 | 64 | 224 | 3 | 1 |
| 11 | 512 | 1024 | 28 | 1 | 2 | 25 | 128 | 128 | 112 | 3 | 1 |
| 12 | 512 | 256 | 28 | 1 | 1 | 26 | 256 | 256 | 56 | 3 | 1 |
| 13 | 512 | 128 | 28 | 1 | 1 | 27 | 512 | 512 | 28 | 3 | 1 |
| 14 | 128 | 512 | 28 | 1 | 1 | 28 | 512 | 512 | 14 | 3 | 1 |

**im2col+GEMM.** We use the im2col implementation from MXNet and the OpenBLAS GEMM routine [11], and OpenMP for multi-threading parallelization. Besides, we use MXNet with im2col+GEMM as the baseline when evaluating the end-to-end inference. We use MXNet 1.6.0.

**LIBXSMM.** The direct convolution provided by LIBXSMM utilizes small GEMM-based micro-kernel generated by JIT. It requires converting the input tensor into a specified format. We excluded this transformation time from the execution time for a fair comparison. We use LIBXSMM 1.17.

**XNNPACK.** Google's XNNPACK is a highly optimized solution for neural network inference and frequently utilized in mobile systems. It provides the indirect convolution algorithm, a modification of GEMM-based convolution algorithms but with a smaller memory footprint and elimination of im2col transformation cost.

**Ansor.** This optimizer [70] is part of the TVM DL compilation framework [19]. To generate high-performance tensor program, Ansor searches in a large search space to find the optimal computational subgraphs. We use Ansor in TVM version 0.12.0 and deploy it to tune convolution layers and CNN models. We use the default number of executed trials of Ansor. Specifically, we use the number of executed trials to 1,000, 15,000 and 20,000 when tuning a single layer, VggNet and ResNet variants, respectively. We exclude the tuning overhead from our measurement.

For layer-wise evaluation, we compare nDirect against multiple schemes: im2col+GEMM, LIBXSMM, XNNPACK and Ansor. We also integrated nDirect with MXNet and evaluated the end-to-end performance of CNN models by comparing our approach with im2col+GEMM used by MXNet and CNN models tuned by Ansor and the TVM back-end code generator.

## 7.4 Evaluation Methodology

To ensure a fair comparison, we adopt the same experimental setups used in the source publications or utilize the default settings of the baseline methods. Specifically, we use $NHWC$ and $KRSC$[4] data formats for XNNPACK's indirect convolution and $NCHWc$[5] for LIBXSMM's direct convolution. For other methods, we use $NCHW$ for input tensors and $KCRS$[6] for filters. Additionally, we include all the layout transformation overhead of nDirect when measuring

[4] $KRSC$=[Output Channels, Kernel Height, Kernel Width, Input Channels].

[5] $NCHWc$=[Batch Size, Input Channels/$c$, Input Height, Input Width, $c$], where $c$ refers to the vector length.

[6] $KCRS$=[Output Channels, Input Channels, Kernel Height, Kernel Width].

its performance. We run each experiment 20 times and report the geometric mean GFLOPS. We found the variances across different runs to be minor, less than 5%.

# 8 EXPERIMENTAL RESULTS

## 8.1 Multi-core Convolutions

Figure 4 reports the multi-core convolution throughput (measured in GFLOPS) on each of the evaluation platforms. The x-axis corresponds to layer ids given in Table 4. The line chart shows nDirect's performance with respect to the hardware's theoretical peak performance (see the y-axis on the right).

Compared with the best-performing baseline, nDirect improves the throughput by 1.32×, 1.34× and 1.07× respectively, on average, on Phytium 2000+, KP920 and ThunderX2, which highlights the effectiveness of our new convolution computation mode. For most layers with $str = 1$ (Section 2.1), nDirect delivers 70%-80% of the CPU peak performance. For example, on layers with $R = 3$ and $S = 3$, nDirect achieves up to ≈ 80% of the peak performance, exceeding layers with $R = 1$ and $S = 1$ because it can utilize more vector registers to achieve a higher $FAI$ according to Equation 3. For $str = 2$, each time the micro-kernel is called, the amount of data fetched into the vector registers is consistent with when $str = 1$, but the quantity of computation is reduced by half, resulting in a decrease in $FAI$. Hence, there is a partial performance penalty. Nonetheless, nDirect performs best overall and consistently outperforms the baseline methods across CONV layers and platforms.

Figure 5 quantifies our packing optimization to the end performance improvement using five convolution layers from VggNet. The technique demonstrates different levels of performance benefits on different architectures. This is because the cache-replacing policy on Phytium 2000+ is pseudo-random, differing from the other two platforms, which utilize the Least Recently Used (LRU) replacement policy.

## 8.2 Direct Convolution Tuned by Ansor

In this experiment, we take the throughput of individual convolutional layers tuned by Ansor as the baseline and report the performance improvement of nDirect over Ansor. The results are given in Figure 6. We found that the Ansor auto-tuning for each convolution layer can coverage in 1,000 execution trials, suggesting that we have given a sufficient search budget to Ansor.

nDirect outperforms Ansor-tuned direct convolution on individual layers across evaluation platforms, giving an average performance improvement of 1.92×, 1.82×, and 1.51× on Phytium 2000+, KP920, ThunderX2 respectively. On some layers like layer 10, Ansor delivers comparable performance to nDirect. However, nDirect still outperforms Ansor on all individual layers by offering better data packing and parallelization strategies.

## 8.3 End-to-end Inference Time

We evaluate the end-to-end inference performance of nDirect under different ResNet and VGGNet variants on Phytium 2000+ and ThunderX2. We choose to compare with Ansor as LIBXSMM and XNNPACK are not compatible with MXNet to run the entire network.

As shown in Figure 7, we normalized the inference performance to that of Ansor. nDirect, as a library-based approach, can deliver comparable performance to Ansor, but without the expensive search overhead of Ansor. Specifically, on Phytium 2000+, nDirect delivers a speedup of 1.19× to 1.45× over Ansor. On ThunderX2, nDirect delivers slightly lower performance for the end-to-end inference compared to Ansor, with a speedup of 0.88× to 0.98×. The better performance of Ansor on the whole CNN is due to its ability to optimize across CNN layers through operator fusion [67, 72]. This technique can write back operations for intermediate results and fetch operations in the CNNs pipeline, further reducing memory access latency and bandwidth pressure to improve CNNs end-to-end performance. Because ThunderX2 has a lower bandwidth than Phytium 2000+, such optimization becomes more important. As nDirect is designed to optimize individual CONV operators, it does not support operator fusion. Our future work will look into integrating nDirect into TVM to take advantage of the higher-level operator fusion optimization. Nonetheless, nDirect delivers comparable performance to Ansor despite lacking operator fusion optimizations.

## 8.4 Embedded Platform

We now evaluate nDirect on an embedded system with lower computation capabilities than HPC systems. Figure 8 reports the results of nDirect and alternative implementations on RPi 4. nDirect outperforms the alternatives both in single-threaded and multi-thread scenarios. Specifically, the best-performing baselines are XNNPACK for single-core execution and LIBXSMM for multi-core executions. However, nDirect delivers a geometric mean speedup of 1.15× and 1.19× over XNNPACK and LIBXSMM, respectively, confirming the effectiveness of our optimization.

## 8.5 Impact of Hyper-threading

Our evaluation was conducted by turning off the hardware hyper-threading (HT). In this experiment, we enable HT on ThunderX2 to exploit HT hardware parallelism. Here, we run 4 threads per core and set the batch size to match the number of logical cores. The results are given in Figure 9. nDirect outperforms XNNPACK, the best-performing baseline, by delivering a geometric mean speedup of 1.28×.

# 9 RELATED WORK

The DL stack often relies on vendor-specific libraries to take advantage of hardware performance. Existing strategies for optimizing convolution operators can be broadly categorized into three approaches. The first involves customizing cache- and vector-friendly data layouts [24, 31–33, 42, 52, 61, 67]. The second approach involves transforming loops with efficient search strategies [19, 25, 45, 70]. The third approach involves generating innermost micro kernels [16, 17, 26, 27, 45, 52, 54, 69].

**Specialized data layouts.** Many prior works have sought to optimize convolution operations by introducing specialized data formats that allow for continuous memory accesses and direct use of SIMD instructions and FMA units [24, 31, 61, 67, 69]. These approaches have demonstrated promising convolution performance

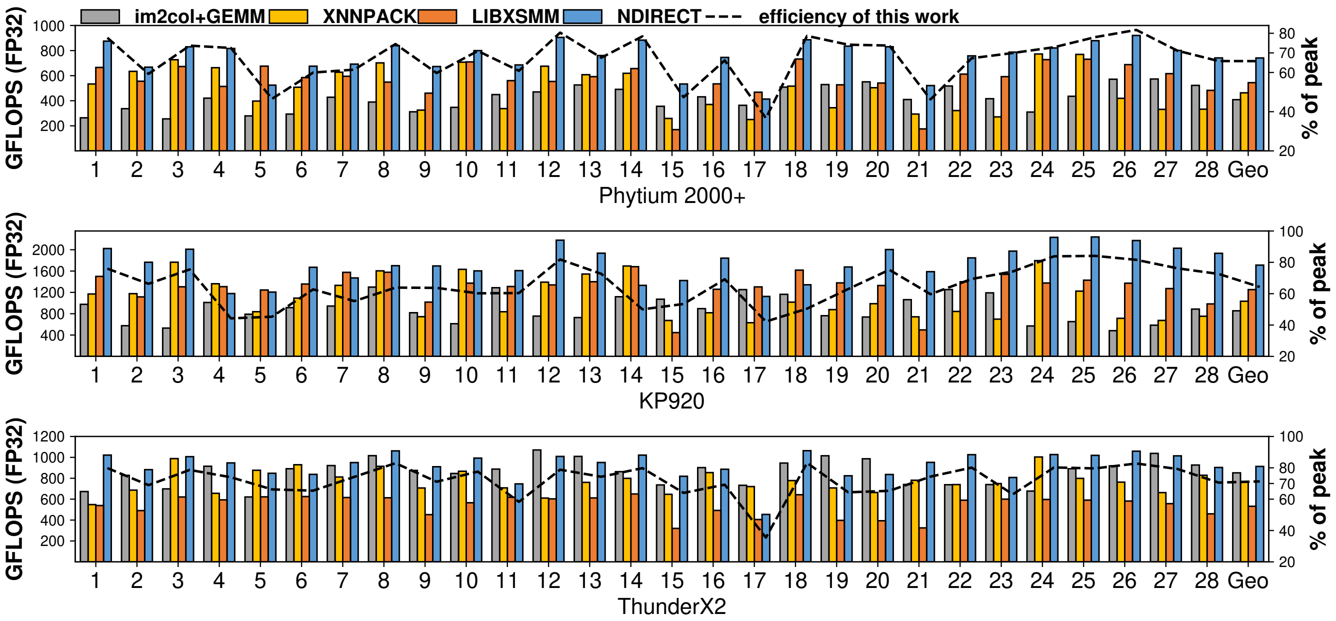


**Figure 4: Convolution algorithms performance on three representative ARMv8 multi-cores. Top: Phytium 2000+, Middle: KP920, Bottom: ThunderX2. The x-axis is indexed based on the layer ids in Table 4.**

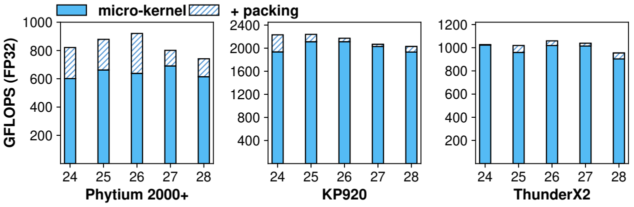


**Figure 5: Quantification of packing optimization.**

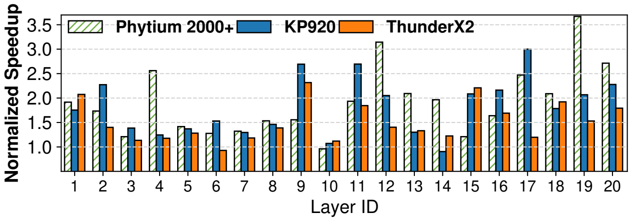


**Figure 6: Performance comparison for convolution operators with respect to Ansor.**

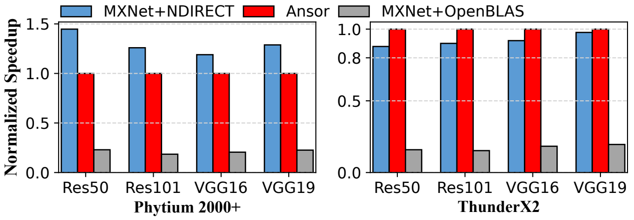
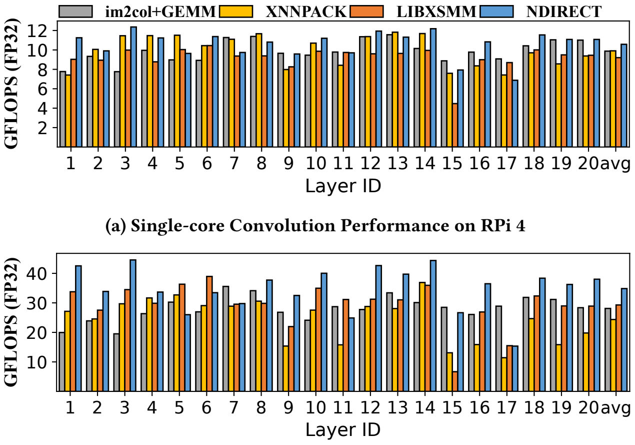


**Figure 7: End-to-End inference evaluations on Phytium 2000+ ($N$ = 64) and ThunderX2 ($N$ = 32).**

by enabling stride-1 memory access and hardware-specific optimizations. However, one significant drawback is that they often require new, specialized data formats that cannot be easily integrated into mainstream DL frameworks that use conventional formats. This limitation either requires changing the underlying DL frameworks or the user code that can also result in additional computation overhead for format conversion when invoking the standard CONV operator. nDirect is designed to avoid this pitfall by operating on the standard data layouts used by mainstreamed DL frameworks.



(a) Single-core Convolution Performance on RPi 4

(b) 4-cores Convolution Performance on RPi 4

**Figure 8: Convoltion performance of (a) single-core and (b) multi-core on the ARMv8-based RPi 4.**

**Loop transformations.** Developing appropriate loop-tiling strategies in explosive search space can effectively enhance data reusability [25, 45]. Ansor [70] constructs a hierarchical search space using efficient pruning techniques and employs evolutionary search with a learned cost model to generate optimized programs. As nDirect

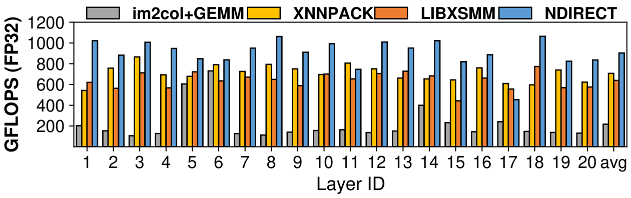


**Figure 9: Convolution performance with enabling Hyper-Threading technique.**

offers a lower-level, library-based optimization for individual CONV operators, it can benefit from the high-level tensor-graph schedule optimization like operator fusion.

**Micro-kernel implementations.** GEMM-based micro kernels are widely used for innermost computations, with examples in [16, 17, 26, 27, 45, 52, 54, 69]. LIBXSMM's direct convolution [31] employs JIT compilation to generate small GEMM code and exploit parallelism through instruction-level optimizations. However, the load instructions of the generated micro-kernels are sequentially arranged, leading to suboptimal performance. Though existing highly optimized BLAS libraries (e.g., OpenBLAS and Intel's MKL) have successfully accelerated GEMM, we have empirically confirmed that the limited *FAI* bound in GEMM mode requires redesigning micro-kernels for convolution operations. nDirect avoids this pitfall by carefully overlapping the memory access operations with computation instructions.

## 10 DISCUSSION

Our work specifically targets ARMv8 multi-cores. In this section, we discuss how to extend our techniques to other architectures and convolution kernels.

### 10.1 Architecture Portability

Our approach is generally applicable and can be easily migrated to other architectures. All our discussions so far target the ARMv8 architecture with 128-bit vector register. The latest ARM Scalable Vector Extension(SVE) [2] provides a variable vector length, which is any multiple of 128 bits between 128 and 2048 bits. Our techniques can be applied to this extension with modified $V_w$ and $V_k$ according to the available length and number of vector registers. In addition to ARM-based CPUS, our techniques are also applicable to modern CPU architectures with SIMD extensions, like Intel AVX-512. Porting our techniques to other hardware architectures requires modifying the micro-kernels according to the constraints defined in Equation 3. These constraints can vary depending on the data type and the vector register width of the target architecture. Furthermore, our approach can be combined with auto-tuning to search for tile sizes and permutation orders to match different cache hierarchies.

### 10.2 Integrating with Other Kernels

Our techniques can be directly applied to standard convolution kernels commonly used in mainstream applications without requiring any modifications to the user code. Here, we discuss how our approach can be integrated with Depthwise Separable Convolution (DSC) [59] and 3D Convolution. DSC, consisting of Depthwise Convolution and Pointwise Convolution, is the building block for two representative CNN models, Xception [40] and MobileNet [22]. nDirect can be directly called to compute the Pointwise Convolution since it can be seen as the 1 × 1 convolution kernel with vectorizable dimension $K$. To support Depthwise Convolution, we only needs removing the reduction operations of dimension C in micro-kernels. Since 3D Convolution can be seen as 2D Convolution with additional reduction dimensions, we can directly use the micro-kernels of nDirect for acceleration and further optimize the outer loops for better cache locality.

## 11 CONCLUSIONS

We have presented nDirect, a new direct convolution solution to provide high performance, high data reusability, and deep learning (DL) framework compatibility on ARM multi-core CPUs. nDirect complies with the conventional data formats used by mainstream DL framework but offers new optimizations for micro-kernel design, data packing and parallelization. We evaluate nDirect by testing its performance on individual convolution layers and the end-to-end inference time of representative CNN models. We conduct our evaluation on four platforms: three HPC multi-cores and one embedded CPU of the ARMv8 architecture. We also compare nDirect against state-of-the-art convolution libraries and a DL tuning framework. Experimental results show that nDirect outperforms the competing baselines on most test cases, achieving better overall performance across all hardware platforms.

## ACKNOWLEDGMENT

This work was funded in part by the National Key Research and Development Program of China under Grant No. 2021YFB0300101, the National Natural Science Foundation of China under Grant No. 61972408, and a UK Royal Society International Exchange Program. For the purpose of open access, the author has applied a Creative Commons Attribution (CCBY) licence to any Author Accepted Manuscript version arising from this submission.

## REFERENCES

[1] Acl. https://github.com/ARM-software/ComputeLibrary.
[2] Armv9. https://www.arm.com/company/news/2021/03/arms-answer-to-the-future-of-ai-armv9-architecture.
[3] cublas. https://developer.nvidia.com/cublas.
[4] Kunpeng-920. https://www.hisilicon.com/cn/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920.
[5] Libxsmm. https://github.com/libxsmm/libxsmm.
[6] libxsmm-dnn. https://github.com/libxsmm/libxsmm-dnn.
[7] Mkl. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.
[8] Mxnet. https://github.com/apache/mxnet.
[9] ncnn. https://github.com/Tencent/ncnn.
[10] onednn. https://github.com/oneapi-src/oneDNN.
[11] Openblas. https://github.com/xianyi/OpenBLAS.
[12] Raspberry-pi-4-model-b. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.
[13] Tensorflow. https://github.com/tensorflow/tensorflow.
[14] Xnnpack. https://github.com/google/XNNPACK.
[15] Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., and Ragan-Kelley, J. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph. 38*, 4 (2019), 121:1–121:12.
[16] Amir, O., and Gil, B. Smm-conv: Scalar matrix multiplication with zero packing for accelerated convolution. In *IEEE/CVF Conference on Computer Vision and*

*Pattern Recognition Workshops, CVPR Workshops 2022, New Orleans, LA, USA, June 19-20, 2022* (2022), IEEE, pp. 3066–3074.

[17] Barrachina, S., Castelló, A., Dolz, M. F., Low, T. M., Martínez, H., Quintana-Ortí, E. S., Sridhar, U., and Tomás, A. E. Reformulating the direct convolution for high-performance deep learning inference on ARM processors. *J. Syst. Archit. 135* (2023), 102806.

[18] 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.

[19] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), A. C. Arpaci-Dusseau and G. Voelker, Eds., USENIX Association, pp. 578–594.

[20] Chen, T., Zheng, L., Yan, E. Q., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada* (2018), S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., pp. 3393–3404.

[21] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *CoRR abs/1410.0759* (2014).

[22] Chollet, F. Xception: Deep learning with depthwise separable convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 1800–1807.

[23] Cong, J., and Xiao, B. Minimizing computation in convolutional neural networks. In *Artificial Neural Networks and Machine Learning - ICANN 2014 - 24th International Conference on Artificial Neural Networks, Hamburg, Germany, September 15-19, 2014. Proceedings* (2014), S. Wermter, C. Weber, W. Duch, T. Honkela, P. D. Koprinkova-Hristova, S. Magg, G. Palm, and A. E. P. Villa, Eds., vol. 8681 of *Lecture Notes in Computer Science*, Springer, pp. 281–290.

[24] de Limas Santana, A., Armejach, A., and Casas, M. Efficient direct convolution using long SIMD instructions. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023* (2023), M. M. Dehnavi, M. Kulkarni, and S. Krishnamoorthy, Eds., ACM, pp. 342–353.

[25] Demmel, J., and Dinh, G. Communication-optimal convolutional neural nets. *CoRR abs/1802.06905* (2018).

[26] Dolz, M. F., Martínez, H., Alonso, P., and Quintana-Ortí, E. S. Convolution operators for deep learning inference on the fujitsu a64fx processor. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (2022), pp. 1–10.

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

[28] Fang, J., Fu, H., Zhao, W., Chen, B., Zheng, W., and Yang, G. swdnn: A library for accelerating deep learning applications on sunway taihulight. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017* (2017), IEEE Computer Society, pp. 615–624.

[29] Fang, J., Liao, X., Huang, C., and Dong, D. Performance evaluation of memory-centric armv8 many-core architectures: A case study with phytium 2000+. *J. Comput. Sci. Technol. 36*, 1 (2021), 33–43.

[30] Ferrari, V., Sousa, R., Pereira, M., de Carvalho, J. P., Amaral, J. N., Moreira, J., and Araujo, G. Advancing direct convolution using convolution slicing optimization and isa extensions. *arXiv preprint arXiv:2303.04739* (2023).

[31] Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D. D., Henry, G., Pabst, H., and Heinecke, A. Anatomy of high-performance deep learning convolutions on SIMD architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018* (2018), IEEE / ACM, pp. 66:1–66:12.

[32] Georganas, E., Banerjee, K., Kalamkar, D. D., Avancha, S., Venkat, A., Anderson, M. J., Henry, G., Pabst, H., and Heinecke, A. Harnessing deep learning via a single building block. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020* (2020), IEEE, pp. 222–233.

[33] Georganas, E., Kalamkar, D. D., Avancha, S., Adelman, M., Anderson, C., Breuer, A., Bruestle, J., Chaudhary, N., Kundu, A., Kutnick, D., Laub, F., Md, V., Misra, S., Mohanty, R., Pabst, H., Ziv, B., and Heinecke, A. Tensor processing primitives: a programming abstraction for efficiency and portability in deep learning workloads. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021* (2021), B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds., ACM, p. 14.

[34] Goto, K., and van de Geijn, R. A. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3 (2008), 12:1–12:25.

[35] Hadjis, S., Abuzaid, F., Zhang, C., and Ré, C. Caffe con troll: Shallow ideas to speed up deep learning. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud, DanaC 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015* (2015), A. Katsifodimos, Ed., ACM, pp. 2:1–2:4.

[36] Haeser, G., Hinder, O., and Ye, Y. On the behavior of lagrange multipliers in convex and nonconvex infeasible interior point methods. *Math. Program. 186*, 1 (2021), 257–288.

[37] Hammond, S. D., Hughes, C., Levenhagen, M. J., Vaughan, C. T., Younge, A. J., Schwaller, B., Aguilar, M. J., Pedretti, K. T., and III, J. H. L. Evaluating the marvell thunderx2 server processor for HPC workloads. In *17th International Conference on High Performance Computing & Simulation, HPCS 2019, Dublin, Ireland, July 15-19, 2019* (2019), IEEE, pp. 416–423.

[38] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016* (2016), IEEE Computer Society, pp. 770–778.

[39] Heinecke, A., Henry, G., Hutchinson, M., and Pabst, H. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016* (2016), J. West and C. M. Pancake, Eds., IEEE Computer Society, pp. 981–991.

[40] 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 abs/1704.04861* (2017).

[41] Huang, X., Wang, Q., Lu, S., Hao, R., Mei, S., and Liu, J. Evaluating fft-based algorithms for strided convolutions on armv8 architectures? *SIGMETRICS Perform. Evaluation Rev. 49*, 3 (2022), 28–29.

[42] Jia, Z., Zlateski, A., Durand, F., and Li, K. Optimizing n-dimensional, winograd-based convolution for manycore cpus. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018* (2018), A. Krall and T. R. Gross, Eds., ACM, pp. 109–123.

[43] Kim, Y. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL* (2014), A. Moschitti, B. Pang, and W. Daelemans, Eds., ACL, pp. 1746–1751.

[44] Lavin, A., and Gray, S. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016* (2016), IEEE Computer Society, pp. 4013–4021.

[45] Li, R., Xu, Y., Sukumaran-Rajam, A., Rountev, A., and Sadayappan, P. Analytical characterization and design space exploration for optimization of cnns. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 928–942.

[46] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S. E., Fu, C., and Berg, A. C. SSD: single shot multibox detector. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I* (2016), B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., vol. 9905 of *Lecture Notes in Computer Science*, Springer, pp. 21–37.

[47] Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing CNN model inference on cpus. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), D. Malkhi and D. Tsafrir, Eds., USENIX Association, pp. 1025–1040.

[48] Mittal, S., and Vaishay, S. A survey of techniques for optimizing deep learning on gpus. *J. Syst. Archit. 99* (2019).

[49] Mogers, N., Radu, V., Li, L., Turner, J., O'Boyle, M. F. P., and Dubach, C. Automatic generation of specialized direct convolutions for mobile gpus. In *GPGPU@PPoPP '20: 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit colocated with 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 23, 2020*, A. Jog, O. Kayiran, and A. Pattnaik, Eds., ACM, pp. 41–50.

[50] Pan, J., and Chen, D. Accelerate non-unit stride convolutions with winograd algorithms. In *ASPDAC '21: 26th Asia and South Pacific Design Automation Conference, Tokyo, Japan, January 18-21, 2021* (2021), ACM, pp. 358–364.

[51] Park, H., Kim, D., Ahn, J., and Yoo, S. Zero and data reuse-aware fast convolution for deep neural networks on GPU. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016* (2016), ACM, pp. 33:1–33:10.

[52] Park, J., Bin, K., and Lee, K. mgemm: low-latency convolution with minimal memory overhead optimized for mobile devices. In *MobiSys '22: The 20th Annual International Conference on Mobile Systems, Applications and Services, Portland, Oregon, 27 June 2022 - 1 July 2022* (2022), N. Bulusu, E. Aryafar, A. Balasubramanian, and J. Song, Eds., ACM, pp. 222–234.

[53] Qin, E., Samajdar, A., Kwon, H., Nadella, V., Srinivasan, S., Das, D., Kaul, B., and Krishna, T. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), IEEE, pp. 58–70.

[54] Rajbhandari, S., He, Y., Ruwase, O., Carbin, M., and Chilimbi, T. M. Optimizing cnns on multicores for scalability, performance and goodput. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017* (2017), Y. Chen, O. Temam, and J. Carter, Eds., ACM, pp. 267–280.

[55] Redmon, J., and Farhadi, A. Yolov3: An incremental improvement. *CoRR abs/1804.02767* (2018).

[56] Ren, S., He, K., Girshick, R. B., and Sun, J. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell. 39*, 6 (2017), 1137–1149.

[57] Roesch, J., Lyubomirsky, S., Kirisame, M., Pollock, J., Weber, L., Jiang, Z., Chen, T., Moreau, T., and Tatlock, Z. Relay: A high-level IR for deep learning. *CoRR abs/1904.08368* (2019).

[58] Shelhamer, E., Long, J., and Darrell, T. Fully convolutional networks for semantic segmentation. *IEEE Trans. Pattern Anal. Mach. Intell. 39*, 4 (2017), 640–651.

[59] Sifre, L., and Mallat, S. Rigid-motion scattering for texture classification. *ArXiv abs/1403.1687* (2014).

[60] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[61] Sridhar, U., Tukanov, N., Binder, E., Low, T. M., McMillan, S., and Schatz, M. D. Small: A software framework for portable machine learning libraries. *CoRR abs/2303.04769* (2023).

[62] Su, X., Liao, X., Jiang, H., Yang, C., and Xue, J. SCP: shared cache partitioning for high-performance GEMM. *ACM Trans. Archit. Code Optim. 15*, 4 (2019), 43:1–43:21.

[63] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. Fast convolutional nets with fbfft: A GPU performance evaluation. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[64] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR abs/1802.04730* (2018).

[65] Wang, Q., Mei, S., Liu, J., and Gong, C. Parallel convolution algorithm using implicit matrix multiplication on multi-core cpus. In *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019* (2019), IEEE, pp. 1–7.

[66] Yang, W., Fang, J., and Dong, D. Characterizing small-scale matrix multiplications on armv8-based many-core architectures. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021* (2021), IEEE, pp. 101–110.

[67] Zhang, J., Franchetti, F., and Low, T. M. High performance zero-memory overhead direct convolutions. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018* (2018), J. G. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 5771–5780.

[68] Zhao, H., Shi, J., Qi, X., Wang, X., and Jia, J. Pyramid scene parsing network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017* (2017), IEEE Computer Society, pp. 6230–6239.

[69] Zhao, T., Hu, Q., He, X., Xu, W., Wang, J., Leng, C., and Cheng, J. ECBC: efficient convolution via blocked columnizing. *IEEE Trans. Neural Networks Learn. Syst. 34*, 1 (2023), 433–445.

[70] Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 863–879.

[71] Zheng, S., Liang, Y., Wang, S., Chen, R., and Sheng, K. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (2020), J. R. Larus, L. Ceze, and K. Strauss, Eds., ACM, pp. 859–873.

[72] Zheng, Z., Zhao, P., Long, G., Zhu, F., Zhu, K., Zhao, W., Diao, L., Yang, J., and Lin, W. Fusionstitching: Boosting memory intensive computations for deep learning workloads. *CoRR abs/2009.10924* (2020).