YaConv: Convolution with Low Cache Footprint

IVAN KOROSTELEV and JOÃO P. L. DE CARVALHO, University of Alberta, Canada
JOSÉ MOREIRA, IBM Research, USA
JOSÉ NELSON AMARAL, University of Alberta, Canada

This article introduces YaConv, a new algorithm to compute convolution using GEMM microkernels from a Basic Linear Algebra Subprograms library that is efficient for multiple CPU architectures. Previous approaches either create a copy of each image element for each filter element or reload these elements into cache for each GEMM call, leading to redundant instances of the image elements in cache. Instead, YaConv loads each image element once into the cache and maximizes the reuse of these elements. The output image is computed by scattering results of the GEMM microkernel calls to the correct locations in the output image. The main advantage of this new algorithm—which leads to better performance in comparison to the existing im2col approach on several architectures—is a more efficient use of the memory hierarchy. The experimental evaluation on convolutional layers from PyTorch, along with a parameterized study, indicates an average 24% speedup over im2col convolution. Increased performance comes as a result of 3× reduction in L3 cache accesses and 2× fewer branch instructions.

CCS Concepts: • **Mathematics of computing** → **Mathematical software performance**; • **Software and its engineering** → *Software libraries and repositories*; • **Computing methodologies** → *Neural networks;*

Additional Key Words and Phrases: GEMM, convolution, cache performance

ACM Reference format:
Ivan Korostelev, João P. L. de Carvalho, José Moreira, and José Nelson Amaral. 2023. YaConv: Convolution with Low Cache Footprint. *ACM Trans. Arch. Code Optim.* 20, 1, Article 18 (February 2023), 18 pages.
https://doi.org/10.1145/3570305

1 INTRODUCTION

Convolutional neural networks (CNNs) deliver reliable solutions for the problems of image classification, speech recognition, recommendation, and language translation. Software frameworks such as Caffe, Tensorflow, and PyTorch have emerged to support the increasing variety of CNNs on multiple hardware architectures. Most of these frameworks introduce a middle-layer representation for the network primitives that are efficiently implemented in high-performance numerical libraries [5, 16, 24]

Training and running a CNN is a computationally-intensive task with convolution layers accounting for roughly 80% of the total CNN inference time on a contemporary CPU. More

Authors' addresses: I. Korostelev, J. P. L. de Carvalho, and J. N. Amaral, University of Alberta, Edmonton, AB, Canada; emails: {korostel, joao.carvalho, jamaral}@ualberta.ca; J. Moreira, IBM Research, Yorktown Heights, NY, USA; email: jmoreira@us.ibm.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
1544-3566/2023/02-ART18 $15.00
https://doi.org/10.1145/3570305

than three-quarters of the CNN inference market relies on CPUs because of their low inference latency [22]. **General matrix multiplication (GEMM)** is the most-used primitive in high-performance libraries. Convolution is typically computed by performing the im2col transformation on the input image and calling a library GEMM routine [3]. Other approaches suggest reducing the memory footprint of im2col or implementing convolution through efficient architecture-specific assembly kernels [1, 4, 7, 8, 11].

The convolution algorithm presented in this article targets memory hierarchy optimization on CPUs. **The core idea of YaConv** is to pack the input image into an L3-cache-resident buffer, preload a smaller chunk of the buffer into L1 cache, and use this image chunk for computation with all L2-cache-resident filter elements before switching to other image elements. To our knowledge, YaConv is the first convolution algorithm that implements all of the following:

(1) uses unchanged GEMM microkernel from a high-performance library;
(2) integrates domain-specific packing of elements the into cache with the calls to GEMM micro-kernels;
(3) avoids unnecessary additional copies of input-image elements.

Points (2) and (3) are important distinct design decisions. Several methods have been proposed to eliminate or reduce the copy of the image tensor in memory [1, 4, 11]. However, unlike YaConv, they do not address the cache reload issue encountered while calling a library GEMM routine multiple times on the same elements. Moreover, YaConv is a novel algorithm that benefits lessons learned from the seminal work by Goto and Van Geijn [9]. In particular, YaConv employes a tiling and packing strategy that allows convolution to be efficiently expressed in terms of GEMM calls, while eliminating redundant copies of the input image.

YaConv successfully repurposes GEMM building blocks to perform convolution. It aims to compute convolution at a performance level that is close to the machine's peak without requiring the writing of new assembly kernels. Different from other **Basic Linear Algebra Subprograms (BLAS)** operations, which can be directly and efficiently implemented in terms of GEMM micro-kernel calls, convolution has many variants and widely different algorithms. Therefore, it is difficult to justify the time and effort needed not only to implement, but also to maintain, multiple micro-kernel implementations for convolution. Processor-architecture industrial organizations already spend significant resources ensuring that the GEMM micro kernel is performant in their current and novel architectures. Thus, YaConv's design saves engineering and time effort by reusing the already widely used and maintained GEMM micro kernels in high-performance BLAS libraries. This article shows that a very efficient convolution algorithm can be constructed using the building blocks and lessons learned from the BLIS Project [16] to implement novel tiling and packing logic while reusing GEMM micro kernels—also a core concept in BLIS. There are several ways to implement a convolution algorithm with the same constraints. The version implemented and evaluated in this article is the most promising in terms of performance on the actual layers found in PyTorch.

For some layers, YaConv's performance is on par with a library GEMM routine, which is around 80–90% of the machine's theoretical peak performance [9, 16, 24]. In comparison with another solution that also works for multiple CPU architectures, im2col convolution, YaConv is 24% faster, measured as the geometric mean of the speedup (w.r.t. im2col) over 73 layers taken from real CNN models. Moreover, YaConv achieves this level of performance using 10× less memory than im2col convolution, requiring only a small buffer space.

An experimental study based on varying input image sizes confirms that the performance of YaConv is dependent on architecture-specific parameters within the GEMM microkernel. The results of this study point to ways to reduce this sensitivity by tuning the image height in the intermediate layers of certain CNNs. Alternatively, the insights from this study could guide design decisions in code generators for CNN, such as Ansor [26], or into compiler framework, e.g., LLVM [12] and MLIR [13]. Cache utilization evaluation on the range of inputs reveals that two parameters—the number of output channels (M) and image height (H)—affect the performance of YaConv the most, with consistent speedup over the baseline when $M < 500$ or $H > 20$.

The contributions of this article include:

- A clear description of the cache inefficiencies of the previous convolution algorithms (Section 3) that guides the design principles for YaConv;
- Explanation of the new convolution algorithm that avoids unnecessary additional copies of image elements (Section 4) and a working implementation that uses unchanged GEMM microkernels, integrated into the popular numerical library BLIS [16];
- A public and free software implementation of YaConv as an extension to BLIS [16];[1] and
- An experimental evaluation study on convolutional layers found in practice and on a grid of parameters indicating that the performance of YaConv is superior to im2col on multiple architectures due to better utilization of the memory hierarchy (Section 5).

2 CONVOLUTION USING OUTER PRODUCT

GEMM is a standard routine from BLAS that has been optimized for over 40 years and has close to peak performance implementations in the open-source libraries such as BLIS and OpenBLAS [16, 24]. In mathematical notation, GEMM is expressed by the formula

$$C = \beta \cdot C + \alpha \cdot A * B, \tag{1}$$

where α and β are scalars, $A_{M\times K}$ and $B_{K\times N}$ are the input matrices, and $C_{M\times N}$ is the output matrix. Most high-performance implementations of GEMM rely on the seminal work of Goto and Geijn [9]. Peak CPU performance for GEMM is achieved by a loop nest that optimizes data cache and TLB locality and leverages an efficient GEMM microkernel. Throughout the article, we refer to this algorithm as the conventional GEMM algorithm.

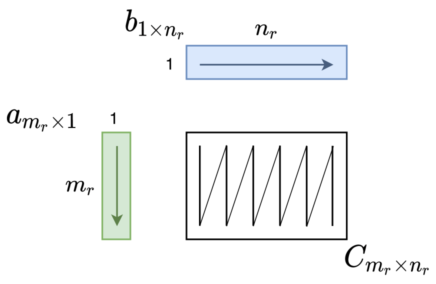
2.1 Outer Product

Matrix multiplication is often introduced as the computation of multiple inner products, as defined by the sum $C[i][j] = \sum_{k=1}^{K} A[i][k] \cdot B[k][j]$. Implementations of GEMM directly using this inner product form suffer from poor reuse of loaded register values. Instead, the GEMM microkernel in BLAS libraries is implemented as multiple outer product computations.

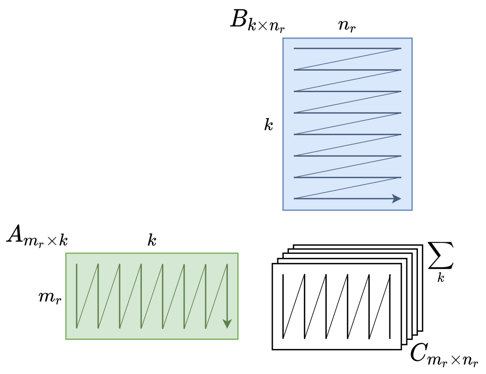
Figure 1(a) shows one outer product update (rank-1 update) that computes partial result $C_{m_r\times n_r} \mathrel{+}= a_{m_r\times 1} * b_{1\times n_r}$. In each update, elements of the vector $a_{m_r\times 1}$ and elements of the vector $b_{1\times n_r}$ are loaded into vector registers. Either elements of a or b are broadcast in registers to produce an operand tile of size $m_r \times 1$ or $1 \times n_r$. The values in registers are multiplied and accumulated to $C_{m_r\times n_r}$. To compute the full GEMM, this step is repeated for each column of $A_{m_r\times k}$ and each row of $B_{k\times n_r}$, as shown in Figure 1(b).

Unrolling the loop along k dimension and prefetching the next elements are commonly implemented to achieve better performance [9, 16, 24]. After k-loop unrolling, several columns of A and rows of B are used for each update to maximize vector-register utilization. Sizes m_r and n_r control the amount of register reuse by the outer product update [9]. These two parameters depend on the architecture and define the minimum GEMM size the microkernel will compute at peak performance. To utilize the microkernel for the full GEMM, a cache-aware strategy must tile the arrays into cache-sized buffers. Moreover, the elements should be placed in the buffers in the order in which they will be accessed by the outer-product updates, as shown in Figure 1(b).

[1]Link to YaConv full source code implementation: https://github.com/ivan23kor/yaconv.

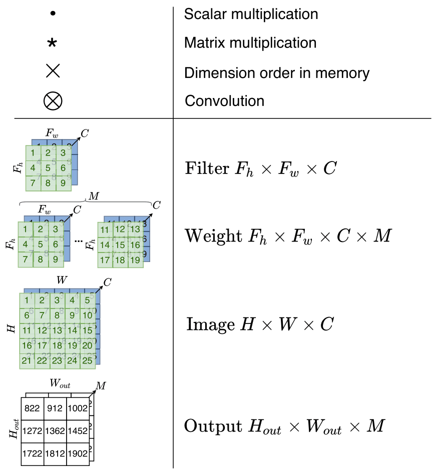


(a) One outer product update

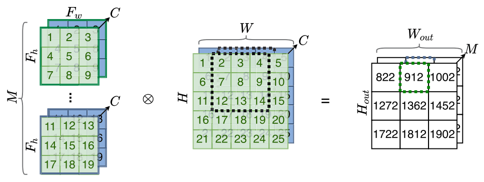


(b) GEMM microkernel is a rank-k update

Fig. 1. GEMM as outer product.


$\cdot$ Scalar multiplication
$*$ Matrix multiplication
$\times$ Dimension order in memory
$\otimes$ Convolution
Filter $F_h \times F_w \times C$
Weight $F_h \times F_w \times C \times M$
Image $H \times W \times C$
Output $H_{out} \times W_{out} \times M$

(a) Notation



(b) Naive convolution example

Fig. 2. Convolution notation and an example of naive convolution.

2.2 Convolution Notation

Convolutional neural networks consist of layers, each of which has a fixed weight tensor. Each convolutional layer can be expressed by the formula

$$W_{F_h\times F_w\times C\times M} \otimes I_{H\times W\times C} = O_{H_{out}\times W_{out}\times M}, \tag{2}$$

where $I_{H\times W\times C}$ and $O_{H_{out}\times W_{out}\times M}$ are the input and output tensors and $W_{F_h\times F_w\times C\times M}$ is the weight tensor.

Figure 2(a) illustrates the tensor notation and the meaning of symbols $\cdot$, $*$, $\otimes$, and $\times$ used throughout this article. A variable input to each layer is the tensor $I_{H\times W\times C}$ that contains an image of height H, width W, and the number of input channels C. Shown in Figure 2(a), a weight tensor has shape $F_h \times F_w \times C \times M$ and consists of M filters. Each filter is a tensor of shape $[F_h \times F_w \times C]$, where C is the number of channels, F_h is the height of the filter, and F_w its width. The output tensor has dimensions $[H_{out} \times W_{out} \times M]$, where H_{out} and W_{out} are the output image height and width. Vertical

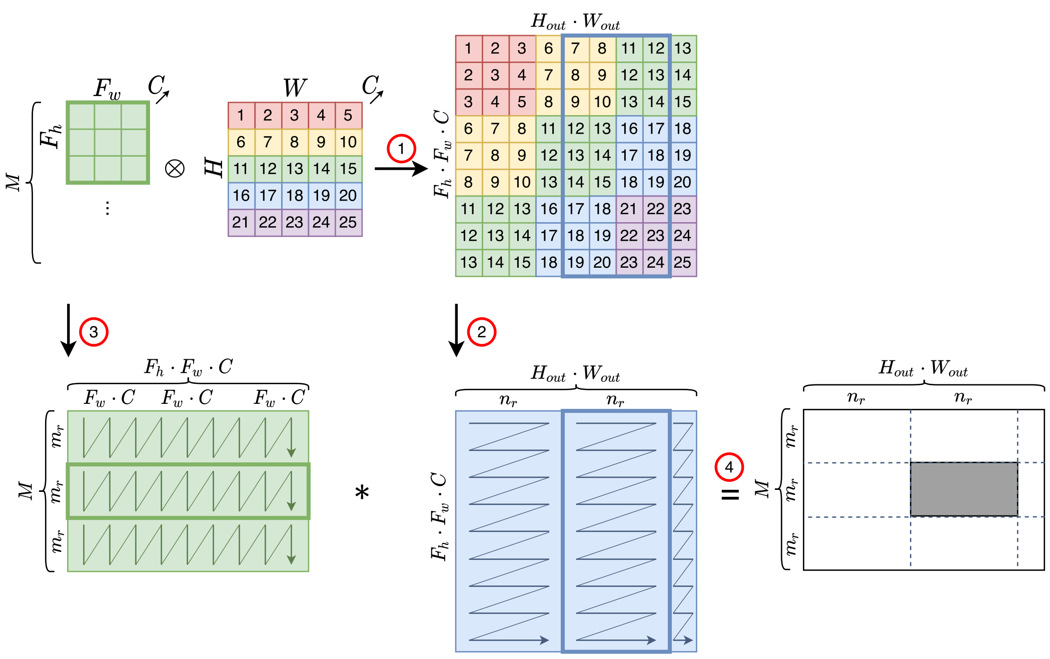


Fig. 3. `im2col` convolution.

and horizontal padding P_h, P_w with zero-elements are typically applied to enlarge the input image so that the output image is of the same size ($H_{out} = H, W_{out} = W$).

3 CACHE INEFFICIENCIES OF PREVIOUS ALGORITHMS

Figure 2(b) shows how a naive algorithm computes convolution $I_{5\times5\times2} \otimes W_{3\times3\times2\times M} = O_{3\times3\times M}$. For this example, the naive method iterates over image patches of dimensions $[3 \times 3 \times 2]$ and computes the sum of products between each input image patch and each filter. The highlighted output elements in Figure 2(b) are computed using all weight elements in the filter and the input elements surrounded by dashed lines. The depth dimension of the selected output elements (output channels) corresponds to the number of filters in the weight tensor.

The naive approach underutilizes the available vector units on a CPU and suffers from poor cache locality of the patch elements. In Figure 2(b), every $F_w \cdot C = 3 \cdot 2 = 6$ elements of each image patch lie consecutively in memory. However, two consecutive rows within the same patch lie at an offset of $(W - F_w) \cdot C = (5 - 3) \cdot 2 = 4$ elements in memory, as each patch of size $I_{3\times3\times2}$ is a part of the entire image $I_{5\times5\times2}$. This causes loading of cache lines and populating TLB entries for the elements not in the order they are accessed during computation.

3.1 Convolution With `im2col` Transformation

`im2col` addresses the problems of the naive approach by placing the patch elements adjacently in memory and calling a performant `GEMM` routine to compute the output. Figure 3 provides an example convolution $W_{3\times3\times C\times M} \otimes I_{5\times5\times C} = O_{3\times3\times M}$ computed through the `im2col` + `GEMM` path. The whole algorithm is broken into four steps, indicated with red circles. Along with the description of the `im2col` transform, Figure 3 shows data movement in the cache-resident buffers of the `GEMM` routine.

Step ① in Figure 3 demonstrates how `im2col` copies the input tensor into a patch buffer as an $[F_h \cdot F_w \cdot C] \times [H_{out} \cdot W_{out}]$ matrix. Each column in this matrix is a flattened patch of the input of size $F_h \times F_w \times C$. There are $H_{out} \cdot W_{out}$ columns in the `im2col` buffer corresponding to the output image of size $H_{out} \times W_{out}$. Computing a GEMM between the reshaped weight tensor as a matrix $[M] \times [F_h \cdot F_w \cdot C]$ and the `im2col` matrix produces a matrix of shape $[M] \times [H_{out} \cdot W_{out}]$. The output of GEMM is the output tensor $H_{out} \times W_{out} \times M$ stored as a column-major matrix of leading dimension M in memory.

Step ② in Figure 3 shows the movement of the elements of the `im2col` buffer during the GEMM call. In the GEMM implementation, the second matrix is packed to an L3-cache-sized buffer in a layout that facilitates L1 cache and register reuse by the outer product microkernel (Section 2.1). The blue arrows within the packed `im2col` buffer demonstrate the order of the elements in memory, with each arrow's length capped at n_r—an architecture-dependent factor introduced in Section 2.1.

Step ③ in Figure 3 shows how the GEMM routine packs the weight tensor into an L2-cache-resident buffer as an $[M] \times [F_h \cdot F_w \cdot C]$ matrix. Packing the weight tensor after the `im2col` buffer ensures that the packed weight buffer elements are in L2 cache and the majority of the packed `im2col` buffer elements are in L3 cache (except for those that were evicted during the weight copy). Similarly to the packed `im2col` buffer, the green arrows of length m_r (Section 2.1) show the memory layout of the elements in the weight buffer.

In step ④, the outer product GEMM microkernel multiplies the tiles $[m_r] \times [F_h \cdot F_w \cdot C]$ of the packed weight and $[F_h \cdot F_w \cdot C] \times [n_r]$ of the `im2col` buffer. The result of each microkernel call is stored in the output as a block of size $m_r \times n_r$ at the corresponding tile offset in the packed buffers.

The GEMM routine applies tiling along each matrix dimension to ensure that each packed buffer fits in the respective cache level, in the case when the weight tensor and/or the `im2col` matrix are larger than L2 and/or L3 cache. Placement of the packed buffers into L2 and L3 cache is ensured by the order of the packing steps. By packing the `im2col` buffer before the weight tensor, the GEMM routine ensures that the weight elements are not evicted from L2.

Weight and `im2col` buffer elements are streamed from the respective packed weight and packed `im2col` buffers that reside in L2 and L3 cache. For instance, highlighted tiles in Figure 3—one from the packed weight buffer and one from the patch buffer—are multiplied to produce the block of output shown as a grey rectangle. Tile offsets within the weight and patch buffers directly translate into vertical and horizontal (on Figure 3) block offsets in the output. Parameters m_r and n_r are optimized for L1 cache and register reuse within the microkernel.

4 YACONV

Two principles are at the core of the new convolution algorithm: the algorithm should not require redundant copies or loads of input elements in cache and the algorithm must use unaltered GEMM microkernels. We follow the CPU cache utilization guidelines presented in Reference [9]. `YaConv` introduces a new iteration pattern for convolution and controls packing of the input tensor elements into the cache. Eliminating redundant copies in the image tensor is central to `YaConv` and it also enables the reuse of the filter tensor as discussed bellow (Figure 4, step ③). The tensor-streaming choices for both `im2col`-based convolution and `YaConv` are determined by the nesting order of the loops surrounding the micro kernel. `YaConv` nests the loops in such a way that the packing of each tensor in a given level of the memory hierarchy is based on how the operands are used in the micro kernel. Although the results show that `YaConv` significantly outperforms `im2col`-based convolutions, this article does not claim that the packing choices made are optimal. Alternative nesting orders can be found by space exploration via analytical models [15].

In naive convolution, two loops iterate over the spatial dimensions of the input image and compute the sum of products between the weight tensor and each image patch (Figure 2(b)).

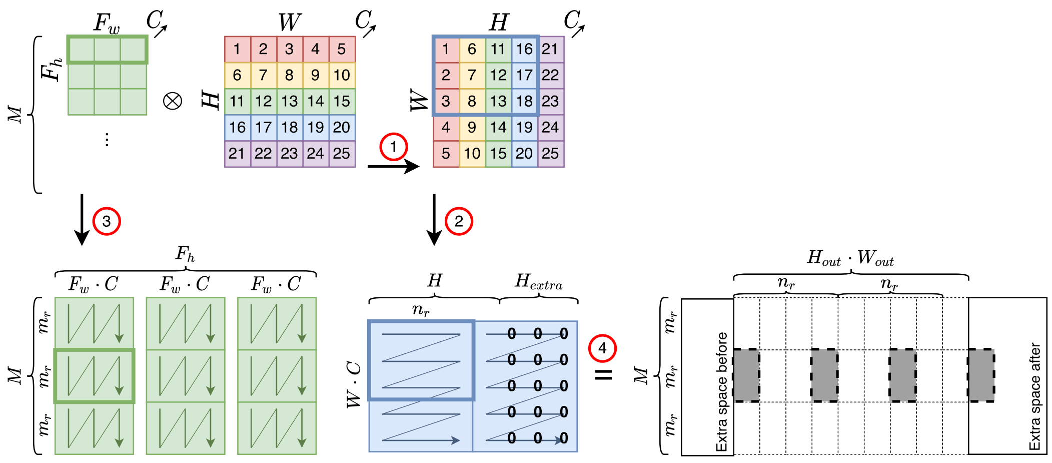


Fig. 4. The novel YaConv algorithm.

In YaConv, these sums do not happen in the same loop iteration. Instead, the YaConv computes one-row convolutions between a selected row of each filter and the corresponding image patches.

Figure 4 demonstrates how YaConv computes the same convolution as in Figure 3 using packing and outer-product microkernels from a library GEMM. In step 1, YaConv reshapes the input tensor as a column-major matrix $[W \cdot C] \times [H]$. This step does not incur any data copy overhead, because the memory layout of such a matrix matches the layout of the input tensor $I_{H \times W \times C}$.

Step 2 in Figure 4 shows how YaConv packs the input tensor into an L3-cache-resident buffer. Zeroed-out elements in the right tile of size $[W \cdot C] \times n_r$ artificially extend the size of the packed input tensor buffer. The reason for doing so lies within the GEMM microkernel—it performs poorly when the matrix sizes are not multiples of n_r. By first packing the input tensor, we bring the input tensor elements to all cache levels, ensuring that the buffer is in L3 cache.

In step 3, YaConv packs the weight tensor as F_h separate matrices of size $M \times [F_w \cdot C]$ into an L2-cache-resident buffer. This copy operation might evict some input tensor elements—as it happens in the traditional GEMM algorithm—but ensures that $M \cdot F_w \cdot C$ weight tensor elements are loaded to L2 cache. To perform partial convolutions for each filter row, one $M \times [F_w \cdot C]$ matrix is packed in the weight buffer at a time.

In step 4, YaConv computes each convolution $W_{1 \times F_w \times C \times M} \otimes I_{H \times W \times C} = O_{H_{out} \times W_{out} \times M}$ as W_{out} GEMMs with sizes $[H] \times [F_w \cdot C] \times [M]$. Having packed the input tensor as contiguous tiles of size $[W \cdot C] \times [n_r]$, YaConv passes portions of size $[F_w \cdot C] \times [n_r]$ of the packed input buffer as a second operand to the GEMM microkernel—one of such portions is highlighted by the blue rectagle in Figure 4. The first operand of the microkernel call is a tile of size $[m_r] \times [F_w \cdot C]$ from the packed weight buffer.

Each GEMM with sizes $[m_r] \times [F_w \cdot C] \times [n_r]$ computes n_r output elements for m_r output channels. All of these elements correspond to the same column of the output image (W_{out}-dimension), given by the vertical offset of the tile of size $[F_w \cdot C] \times [n_r]$ within the packed input tensor. One GEMM call computes the result of applying one row of m_r filters to n_r rows of the input image at the same column offset. The corresponding weight and input tensor elements are highlighted by green and blue rectangles in Figure 4. Each result of size $m_r \times n_r$ is placed in the output array (column-major $[M] \times [H_{out} \cdot W_{out}]$ matrix) as n_r columns at stride $W_{out} \cdot M$, because every GEMM call corresponds to applying one filter row over n_r input image rows at a fixed column.

4.1 Extra Memory Usage

Because YaConv computes GEMM between every possible combination of the weight and image tiles, and some of these elements are not part of the convolution result, they are not accumulated in the output. In the example in Figure 4, the first filter row is multiplied with elements of the blue portion of the input image as part of the GEMM microkernel call with width $n_r = 4$. However, only the first three columns of this highlighted image area should be multiplied with the first filter row and the result of this multiplication to be stored in the output tensor. The result of the dot product between the elements 16, 17, 18 of the input tensor and the first filter row is a part of applying the filter on the image when the last filter row is outside of the image bounds.

One way to discard these elements is to adjust the width of every GEMM microkernel call to smaller values than n_r, leading to significant performance degradation. Instead, YaConv reserves a larger buffer for the output elements than the $H_{out} \cdot W_{out} \cdot M$ elements needed by convolution. The actual output tensor $H_{out} \times W_{out} \times M$ is located at an offset within this buffer space, while the extra space is used for the spillover elements of the GEMM calls.

The output buffer contains the output array and two extra parts *before* and *after* the actual output tensor. For a given problem, YaConv uses space for $(F_h - 1 - P_h) \cdot W_{out} \cdot M$ extra elements before the output array. Because the image buffer is enlarged to the optimal microkernel size and to account for the spillover results from GEMM, YaConv also reserves some space for the elements immediately after the actual output. In Figure 4, H_{extra} is the H enlargement required until the next multiple of n_r. Therefore, extra space required after the output array is $(H_{extra} + F_h - 1 - P_h) \cdot W_{out} \cdot M$, where the second part comes from the GEMM spillover elements. In total, YaConv requires $(H_{extra} + 2 \cdot (F_h - 1 - P_h) \cdot W_{out} \cdot M$ extra space, at that $H_{extra} < n_r$. Asymptotically, extra space complexity for YaConv is $O((F_h - P_h) \cdot W_{out} \cdot M)$, which is sublinear on the output size $H_{out} \cdot W_{out} \cdot M$.

4.2 Tiling and Block Sizes

In real applications, weight and image tensors are large enough to not fit in the cache. We add loop tiling to our algorithm to improve cache locality and TLB entry usage, as suggested by Goto and Geijn. The conventional GEMM algorithm [9] uses three cache block sizes MC, KC, NC for two packed buffers $\tilde{A}_{MC \times KC}$ and $\tilde{B}_{KC \times NC}$:

(1) KC is calculated to fill L1 cache with tiles $m_r \times KC$ and $KC \times n_r$ of the operands of each microkernel call;
(2) MC is set to fill the L2 cache with $MC \times KC$ elements of the packed buffer $\tilde{A}$;
(3) NC determines the size of the buffer $\tilde{B}$ that resides in L3 cache.

High-performance BLAS libraries set these sizes more conservatively as temporary variables and output tile $m_r \times n_r$ take some L1 cache space, and TLB is typically a more limiting factor than L2 cache [9, 16, 24].

The sizes for weight and image buffers in YaConv are taken from the BLIS implementation of the GEMM routine. The order of calls to packing routines determines the cache-level residence for each buffer. YaConv packs a portion of the image in the outermost loop, thus loading its elements to all cache levels. The packing of weight elements into the buffer follows image packing. Such an order, coupled with adequate tile sizes, ensures that the weight elements will not be evicted from L2 cache by the image elements during packing. Two innermost loops around the microkernel (with steps n_r and m_r) iterate over L1-sized tiles of weight and image buffers and call the microkernel code to compute partial results.

Additionally, when $W = F_h = F_w = 1, P_h = P_w = 0$, the weight and image tensors can be thought of as matrices and convolution degenerates into a GEMM with sizes $M \times C \times H_{out}$. The

Table 1. Clockrate, Cache Sizes, Output Tile Dimensions of the GEMM Microkernel and Linux Kernel Version of the Machines Used for the Experiments

Architecture	Clock, GHz	L1, KiB	L2, KiB	L3, MiB	GEMM tile size	Kernel
Intel® Cascade Lake	3.5	32	1024	36 / 24	32×12	4.15.0
AMD Zen 2	2.0	32	512	16 / 4	6×16	4.15.0
IBM Power10	4.0	32	2048	64 / 8	8×16	5.11.0
Intel® Haswell	2.7	32	256	30 / 12	6×16	5.6.13

L1 and L2 cache sizes are per core. L3 size is followed by the number of cores sharing L3 cache.

loops over F_h and W_{out} contain only one iteration and YaConv becomes the conventional matrix multiplication algorithm.

5 COMPARING YACONV WITH IM2COL ON MULTIPLE MACHINES

The experimental results in this section indicate that:

(1) YaConv outperforms the im2col baseline on PyTorch layers by 23-25% on multiple architectures.
(2) The superior performance of YaConv is explained by better L3 cache usage. Moreover, in most cases, YaConv reduces L1 cache usage as compared to im2col-based convolution.
(3) As expected, the performance of YaConv compares unfavourably with im2col for small image heights H, which are not a multiple of architecture-dependent GEMM microkernel sizes. Better performance for YaConv can be achieved by adjusting the image size to match architecture-dependent values.

5.1 Experimental Methodology

Table 1 provides hardware information about the four machines used for the experiments. The cache sizes are given per node, i.e., L3 cache is shared among some cores on each platform. Each binary runs the same convolution on a batch of N images, where N is adjusted to ensure that each execution lasts at least 1 s on a 100 GFLOPS machine. FLOPS are calculated as the number of single-floating-point operations, given by $2 \cdot N \cdot H \cdot W \cdot C \cdot F_h \cdot F_w \cdot M$, divided by the wall clock time of the respective convolution routine on the whole image batch. The results presented for each experiment are mean values of ten runs. Unless explicitly specified, the relative standard deviation observed is less than 5%.

Benchmarking cache performance is difficult because of the complex hierarchy of modern CPU memory systems. Although im2col does extra work copying input image elements to another buffer, it loads the elements into the cache and populates the TLB entries. This data preparation by im2col reduces data access time within the packing routines of the library GEMM. Thus, for a fair comparison, this experimental evaluation compares the performance of the whole im2col-based convolution routine with YaConv. There should be fewer accesses to the last levels of the memory hierarchy by YaConv, because YaConv loads each input image element exactly once into an L3-cache-resident buffer. Both YaConv and im2col-based convolution implementations allocate temporary space once for the whole batch of images.

The implementation of im2col from Caffe is used for the baseline, followed by a call to BLIS GEMM [10]. BLIS[2] was built using the default library-provided flags for each platform and the same flags were used to build the microkernels for integration into YaConv. All platforms run a 64-bit

[2]BLIS version b3e674db3c05ca586b159a71deb1b61d701ae5c9.

Table 2. Values for Selected Convolution Parameters from 218 Layers in PyTorch, from Most to Least Common

Parameter(s)	Common values
H, W	14, 7, 28, 56
F_h, F_w	3, 5
C	64, 192, 32, 128
M	128, 256, 64, 192

Linux kernel and all benchmarks are compiled using gcc with -O2 with -mtune=native. **perf** was used to collect cache and TLB counters [23].

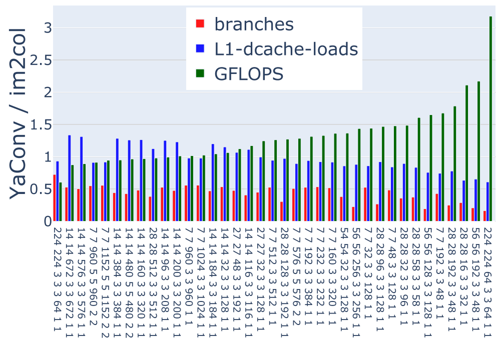
5.2 Performance on PyTorch Layers

This evaluation uses convolutional layer parameters from thirteen pre-trained CNN models in Torchvision, which is a part of the PyTorch project [17]. Of 661 layers, 400 layers are a special case of convolution with filter size 1, which both PyTorch and Tensorflow compute as a direct call to the library `GEMM`. Also, `YaConv` cannot handle convolutions with non-unit strides because of the element ordering restrictions imposed by the `GEMM` microkernel. Thus, another 42 non-unit-stride layers were eliminated leaving 218 layers with unit stride and filter size greater than 1. Some of these layers have the same geometry: 73 unique layers can be used for the evaluation of `YaConv`. The most common values for H, W, F_h, F_w, C, M are provided in Table 2.

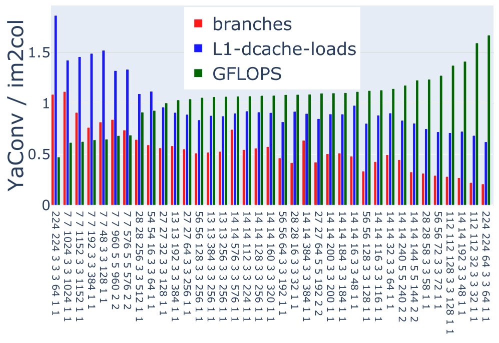
Figure 5 presents the ratios between `YaConv` and `im2col` for the following measurements provided by `perf`: (a) number of L1 cache accesses; (b) number of total branch instructions executed; (c) total routine GFLOPS. Layers are shown on the x-axis in the format $H\ W\ C\ F_h\ F_w\ M\ P_h\ P_w$ and are sorted by the GFLOPS ratio between `YaConv` and the baseline. To improve visibility, the graph presents only half of the 73 layers. Layers are selected by choosing every second layer from the sorted list. Figure 5 indicates that `YaConv` reduces the number of branches taken by the `im2col` convolution algorithm. For all machines, the ratio of L1 cache loads between the two algorithms decreases with a larger speedup of `YaConv` over `im2col` convolution, pointing to increased reuse of elements in L1 cache as the source of the observed speedup. Moreover, `YaConv` achieves up to 24% speedup over `im2col`-based convolution across all four evaluated machines—over 3× on Intel Cascade Lake and Power10 for some input sizes.

The reduction in the number of branches taken in comparison with `im2col`-based convolution is due to the elimination of the `im2col` transform. `YaConv` does not make a copy of an image element for each of the filter elements. `YaConv` loads each image element into L3 cache exactly once, resulting in fewer L3 cache loads and misses. However, for some layers, the L3 cache performance measured in the experiments does not support this hypothesis. Cache design in some architectures is complex and L3 performance requires further investigation. Additionally, the slowest layers in Figure 5 are also the ones where `YaConv` accesses L1 cache more than the baseline. The two following sections present a detailed study on Intel Cascade Lake, where parameters are varied to identify performance trends. The same studies were conducted on all four machines, exhibiting similar trends in algorithm runtimes and cache utilization. Detailed results are presented for one machine for brevity.

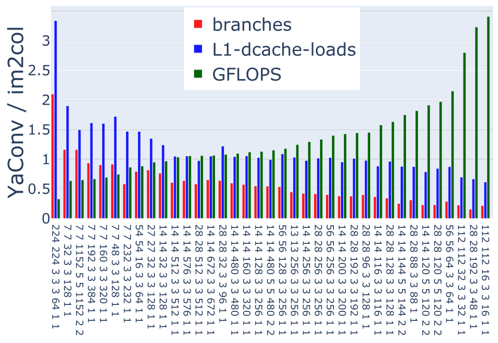
For a few small image sizes ($H = W = 7$, 13, and 14), `im2col`-based convolution exhibits better cache and TLB usage and thus performs better than `YaConv`. This performance difference occurs because `im2col`-based convolution computes `GEMM`s over an input image of width $H \times W$—with several full `GEMM` tiles and only a few non-full tiles due to replication of input pixels. In contrast, Yaconv



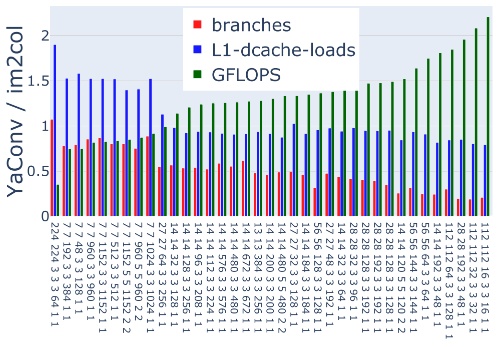
(a) Intel® Cascade Lake



(b) AMD Zen 2



(c) IBM Power10



(d) Intel® Haswell

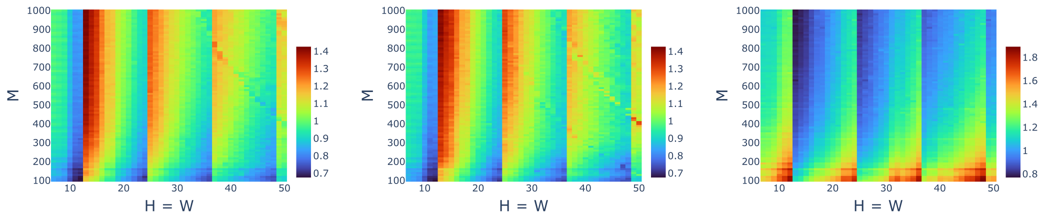
Fig. 5. Ratios for L1 cache, branches, and GFLOPS between YaConv and im2col on layers from PyTorch across four machines.

does not replicate pixels and has to compute GEMMs over packed buffers of width H—which need to be padded to the size of the GEMM micro kernel, thus exhibiting poor cache and TLB utilization. Adopters of YaConv may easily build a solution that selects between YaConv with an im2col-based convolution based on the value of H and W to always deliver the best performance. This happens for 9 out of 36 input shapes, as Figure 5 shows. Section 5.3 further contrasts im2col-based convolution and YaConv for many image sizes and gives insight on how to address these edge cases.

5.3 YaConv Performance Varies with Image Sizes

The values of H, W, and C in the layers in Figure 5 were selected from the real CNNs from PyTorch. A more comprehensive understanding of the performance of YaConv can be gained by varying these values. Figure 6 presents cache and runtime profile collected on the Intel Cascade Lake machine using the same methodology as in the Section 5.2. All subfigures have fixed parameters $C = 300, F_h = F_w = 3, P_h = P_w = 1$ and vary square image sizes $H = W$ and M in the range of values found in actual CNNs. The color of points in the heatmaps represents the value of the ratio $\frac{\texttt{YaConv}}{\texttt{im2col}}$ for each metric.

Two insights can be gained from Figure 6. First, the repetitive pattern with step size 12 on the plots confirms that computing full GEMM microkernel on partially filled tiles significantly affects performance. For this machine, the microkernel uses $n_r = 12$ to maximize the use of the vector unit. Therefore, when H is a multiple of 12, Figure 6 shows the fewest L1 cache and TLB accesses and the best runtime for YaConv.



(a) L1 cache loads (b) TLB loads (c) GFLOPS

Fig. 6. Varying image size $H = W$ and number of input channels with fixed $M = 200, F_h = F_w = 3, P_h = P_w = 1$ on Intel Cascade Lake (where H and W are the height and width of the image and M is the number of output channels).

Second, YaConv performs better with smaller values for M. With the fixed value $C = 300$, relative speedup of YaConv over im2col convolution gradually decreases until $M = 480$. After that point, the relative performance stabilizes with further increase in M. As shown in Figure 8(a) in the following Section 5.4, this effect can be explained by the gradual increase in the number of L3 cache loads that depends on the memory stride M of the weight tensor elements. However, performance metrics do not show any trend while varying the number of input channels C.

In Figure 5(a), the worst-performing layers on Intel Cascade Lake have $H = W = 7, 13, 14$, most of which come from one CNN (GoogLeNet). In im2col-based convolution, the width of the image buffer is $H \cdot W$, which includes several full GEMM tiles even for the smallest images $H = W = 7$. In YaConv, the same input image is packed into a buffer of width H and padded to the full tile with zeroes. For larger image sizes this padding does not play a big role, because the partially filled tiles are a small portion of the whole computation. However, for small H, the extra zero elements require more usage of the cache and the TLB entries and the result of GEMM for these elements is not used for accumulation of the output. This is confirmed by sharp vertical edges in Figures 6(a) and 6(b). The performance patterns shown in Figure 6 are also observed in other architectures and can be used to improve the performance of YaConv by adjusting the image size for layers in the middle of the network according to architecture-specific values.

5.4 YaConv Improves L3 Cache Performance

The vertical axis in Figure 7 presents the ratios of L3 accesses and misses and GFLOPS between YaConv and im2col collected on Intel Cascade Lake. Thus, GFLOPS values greater than one indicate that YaConv performs better than im2col-based convolution. Values of cache access and misses lower than one also indicate that YaConv performs better than im2col-based convolution. The horizontal axis contains the same layers as Figure 5(a). While most layers experience a reduction in L3 cache usage, YaConv brings an increase in cache accesses and misses for certain sizes of the parameters C and M, e.g., 128, 512, 1024. These increases in L3 accesses and misses seem to occur more often when $H = W = 7$. Thus, a parameterized performance study could reveal more information about the correlation of these misses with performance.

The set of experiments shown in Figure 8 varies the number of input channels C on the horizontal axis, the number of output channels M on the vertical axes with fixed image and filter sizes $H = W = 7, F_h = F_w = 3$. The ratio of L3 cache loads is given as $\frac{\texttt{im2col}}{\texttt{YaConv}}$, whereas the ratio for GFLOPS is $\frac{\texttt{YaConv}}{\texttt{im2col}}$. Values for both of these ratios in Figure 8 are presented with the corresponding color on the heatmap so that larger numbers are better for YaConv. The same color scales are used for both subfigures with each color scale adjusted to show an interval between the minimum and the maximum values on each heat map. Some locally large values in Figure 8(a) were clipped to

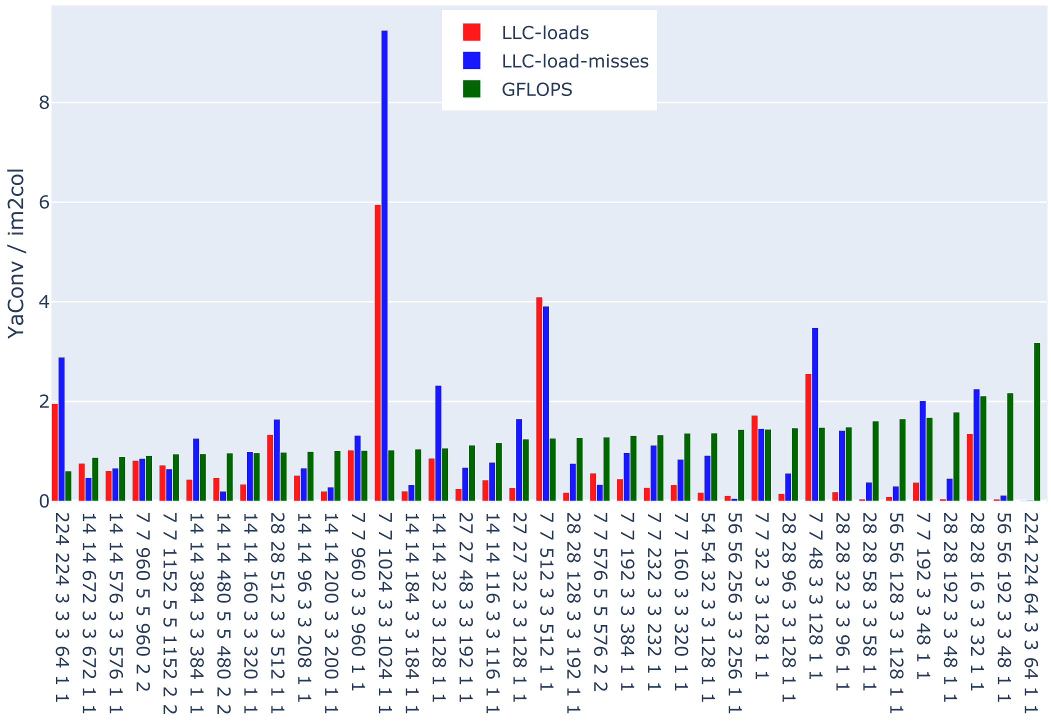


Fig. 7. L3 cache usage and GFLOPS on PyTorch layers on Intel Cascade Lake.

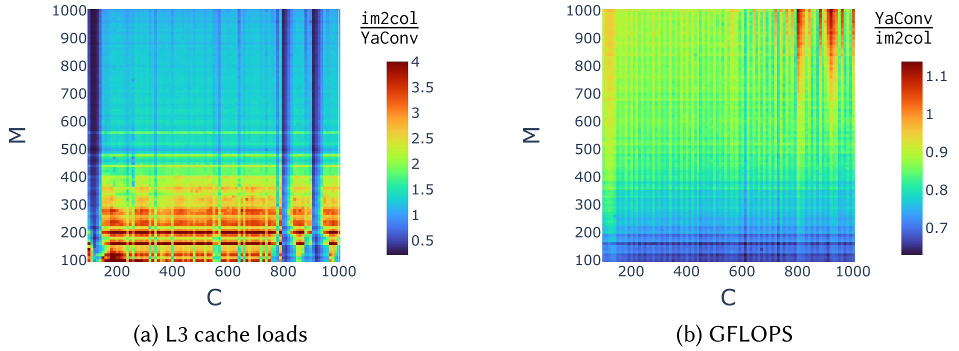


Fig. 8. Varying the number of input and output channels with fixed image and filter size on Intel® Cascade Lake.

4 to prevent the figure from giving a distorted image of performance. The range of values was chosen to cover most of the poorly-performing layers from Figure 5.

When the number of output channels does not exceed the cache block size along that dimension ($M < 480$ in Figure 8(a)), the geometric mean of the reduction of L3 accesses of YaConv w.r.t im2col is 3×. With further increase in M, relative improvement of L3 usage in YaConv gets worse down to 1.6×. The heatmaps in Figures 8(a) and 8(b) indicate a correlation between the number of L3 cache loads and YaConv performance. Figure 8 exhibits a pattern of significantly worse L3 cache utilization by YaConv around specific values, e.g., $C < 60, C \approx 800, C \approx 940$. While performing experiments for a smaller range of parameters but with step size 1, we found that the same pattern persists on a finer scale and some input values cause a sudden increase in the whole memory hierarchy utilization. This could be explained by the parameter sizes (leading dimensions of tensors), coinciding with cache associativity stride and causing more hit conflicts than an average case. We observed similar trends on other machines and for the heat maps produced with other fixed image sizes.

6 RELATED WORK

Many convolution algorithms are not expressed in terms of GEMM [2]. For instance, prior work explored the use of Fast-Fourier Transform [18] and Winograd's algorithm [14]. Winograd-based convolution algorithms are designed to minimize arithmetic intensity and aim to be efficient for convolutions over small tiles for small filters and small batch sizes. YaConv does not reduce the arithmetic intensity of convolutions, but does eliminate unnecessary copies performed by traditional GEMM-based convolutions. Moreover, YaConv is not sensitive to small filters and batch sizes, but YaConv is sensitive to the number of filters and performs better with small number of filters. FFT-based convolution algorithms are known to be faster then direct convolution algorithms for large filter sizes [19]. Although the complexity of convolution algorithms is established, there is no thorough empirical study contrasting standardized and well-maintained implementations of such algorithms on modern hardware. This work enables such study by making available an implementation of YaConv that extends and re-uses building blocks from BLIS—a widely used, well-documented and maintained linear algebra library[16]—publicly available and as free software.[3] The remainder of this section focuses on previous work closely related with YaConv, which is a novel convolution algorithm aimed at reusing GEMM micro kernels from high-performance libraries.

A common approach to deal with the performance inefficiencies of naïve convolution is im2col convolution. The method was first introduced by Chellapilla et al. and it became popular due to its use in Caffe [3, 10]. im2col is a data-copy procedure that prepares patches of the input image in a separate patch matrix. Each patch is a portion of the input image of the same size as the filter. Each input element is copied up to $F_h \cdot F_w$ times into this patch matrix—one time for each patch-filter product in which the element takes part. (The exact number of copies depends on the stride of the convolution.) The goal of the im2col transformation is to place patch elements adjacent in memory to optimize cache locality and to take advantage of the highly optimized GEMM routines readily available for most architectures. im2col has three drawbacks: (1) memory footprint: the patch matrix requires additional storage of $O(F_h \cdot F_w)$ of the image size; (2) an additional copy operation; and (3) enlarged inner dimension of the GEMM, given by the size $F_h \cdot F_w \cdot C$ of the patch matrix, leads to lower reuse of elements loaded into the cache. This effect is pronounced when $K >> M$ and $K >> N$.

A way to reduce the overhead of im2col consists in not creating the patch matrix explicitly [6, 11]. One solution to avoid the creation of a patch matrix consists in creating an extra level of indirection in the form of a buffer of pointers to input image patches. This extra indirection reduces the space and time needed to prepare the patches for access—there is no need to have multiple copies of the same element. However, it increases the access time for each element, because each access requires one additional pointer dereference [6]. This Indirect Convolution Algorithm requires a modified GEMM microkernel that allows arbitrary strides between elements [6]. Another solution integrates im2col with the packing step of GEMM in BLIS [16] to prevent the extra data copying [11]. Although neither of these methods keeps redundant copies of input elements in memory, each matrix element is loaded into cache $F_h \cdot F_w$ times. In contrast, YaConv changes the

[3]Link to YaConv full source code implementation: https://github.com/ivan23kor/yaconv.

packing routine and the iteration pattern to eliminate the need to copy the input elements either directly or via pointers.

The same approach can be implemented in hardware [21]. Soltaniyeh et al. design a new accelerator that integrates the `im2col` transform and `GEMM`. To decrease the overhead of the data transformation, the authors design interconnected patch units based on systolic arrays. These units act as buffers for holding common patch elements and reduce redundant accesses to the same elements. Although `YaConv` could be implemented in hardware, `YaConv`'s key design points allow it to be implemented on any commodity hardware without special instructions or specific hardware support.

Another way to reduce the memory footprint of `im2col` is to call the library `GEMM` routine several times [4]. **Memory Efficient Convolution (MEC)** creates a different patch matrix that reduces the size of the patch matrix by a factor of F_w in comparison with the original `im2col` transformation [4]. The `GEMM` routine is called W_{out} times on this transformed patch matrix, each time storing its output at an offset in the output tensor. Although the MEC algorithm reduces the space overhead and can be faster in some cases, it was shown to underperform `im2col` on a wide range of input parameters [1]. A reason for the MEC's inefficiency could be the cache overwrite between consecutive `GEMM` calls. The number of algebraic operations needed to compute a `GEMM` with sizes $100 \times 100 \times 100$ is the same as the number of operations needed to compute a `GEMM` with sizes $100 \times 100 \times 10$ ten times. However, the latter can be several times slower, because fewer elements are reused while they are available in the caches.

Reducing the memory bandwidth per element can be achieved by manipulating data in registers [25]. Zhao et al. optimize convolution training for the Sunway TaihuLight supercomputer that features CPE vector units organized in a mesh. The authors propose to map the whole convolution to tiles that fit in each CPE and move the overlapping patch elements between adjacent CPEs with specialized register communication instructions. In the same way as `YaConv`, this approach eliminates the need for extra buffer space and uses elements in the cache until no longer needed. However, the solution depends on the instruction-level optimizations available on the specific architecture under evaluation [25].

Anderson et al. propose a way to repeatedly call the `GEMM` routine without copying elements of the input tensor. Their low-memory algorithm separates the weight tensor $F_h \times F_w \times M \times C$ into $F_h \cdot F_w$ tensors, each of shape $1 \times 1 \times M \times C$. Anderson et al. notice that convolution with 1×1 filters is the same as a `GEMM` with sizes $M \times C \times [H_{out} \cdot W_{out}]$. Convolution with any other filter size is a sum of $F_h \cdot F_w$ such convolutions, each corresponding to scaling the whole input image by one of the $F_h \cdot F_w$ filter elements [1]. To account for the relative position of the 1×1 filter elements within the $F_h \times F_w \times M \times C$ weight tensor, $F_h \cdot F_w$ scalings are accumulated in the output array at corresponding offsets. Although MEC and the approach of Anderson et al. solve the memory problem of `im2col` convolution, they do not address the problem of redundant cache reloads. `YaConv` makes full use of each element present in the cache before the element is evicted, because `YaConv` integrates the loading of elements into cache with computation.

While the aforementioned methods either explicitly copy the input tensor or call a `GEMM` routine several times, direct convolution methods optimize the naive convolution loop nest. As naive convolution suffers from cache misses and an unoptimized multiply-add pattern, one solution is to call specific efficient assembly code for each convolution size at run-time. Georganas et al. present a JIT-optimized implementation for convolution that is aimed at whole-model performance optimization. They design efficient assembly kernels for direct convolution using hardware-specific vector instructions and cache prefetching [7]. To alleviate memory bandwidth issues, cache and register blocking are applied to change the layout of elements in memory [7]. This approach attains up to 90% of the peak machine performance but requires an enormous amount of engineering for each instruction set (10K+ code lines in assembly). However, YaConv introduces a generic cache utilization strategy and relies on the existing GEMM microkernels. With its new approach, YaConv achieves the same convolution performance as the work of Georganas et al. without requiring hand-crafted assembly programming.

A novel *batch-reduce* GEMM *kernel* may be better suited for convolution [8]. The suggested microkernel introduces additional optimization opportunities for convolution, as contrasted with the traditional GEMM microkernel from Equation (1). Among several deep-learning primitives, the convolution implementation of Georganas et al. matches the performance of their previous work [7] and significantly reduces the amount of manual assembly engineering. While this work achieves the best performance of all convolution algorithms found in the literature, it introduces a new microkernel that has to be written for each architecture. YaConv relies on the GEMM microkernel that is also used for most level-3 BLAS routines and machine learning workloads, e.g., kMeans, PCA, SVM.

As compared to GEMM-based algorithms, YaConv uses asymptotically less extra memory [1, 3, 4]. Convolution based on the im2col transform creates a patch matrix of size $O(F_h \cdot F_w \cdot C \cdot H_{out} \cdot W_{out})$ [3]. MEC requires $O(F_h \cdot C \cdot H_{out} \cdot W_{out})$ extra space [4]. The algorithm of Anderson et al. also requires extra memory immediately before and after the output array, which they estimate as $O(M \cdot H \cdot W)$. Considering that $C \approx M$ and $H \approx H_{out} \approx W \approx W_{out}$, YaConv uses the same or less extra space in comparison with GEMM-based methods.

Optimal tiling size can be found through space exploration or by using an analytical model [15]. Convolution algorithms such as YaConv and the analytical model by Li et al. are complementary. Li et al.'s solution also relies on BLIS-like micro kernels (Section 6, second paragraph in Reference [15]). The analytical-model approach proposed by Li et al. can be used in an automated-tuning approach to discover good tile sizes for YaConv, and similar algorithms, for specific convolutions.

7 FUTURE WORK

The convolution algorithm design proposed in this work is not the only solution to cache reload problems of previous approaches. As it was shown in the experimental evaluation, YaConv is sensitive to image height, which can be addressed in the future by, e.g., changing the order of the dimensions of the packed image buffer. This change will bring a new iteration pattern that can support non-unit strides.

Once the edge cases for single-threaded performance are fully addressed, YaConv can be parallelized similarly to the parallel implementation of the conventional GEMM algorithm [20]. Because YaConv's loop nest degenerates into the library GEMM algorithm when $W = F_h = F_w = 1$, the same loops of the convolution algorithm can therefore be parallelized. YaConv is well-positioned for multithreaded scaling because of the algorithm's focus on minimizing the number of L3 cache accesses, which is shared among cores, by improving the reuse of elements in the cache.

Last, as the aforementioned performance improvements are implemented, the algorithm can be ported to a compiler framework that supports matrix multiplication building blocks, e.g., LLVM or MLIR [12, 13].

8 CONCLUSION

The main idea behind the design of YaConv is to prevent unnecessary copies of image elements, improve cache utilization, and make direct use of unmodified GEMM building blocks from high-performance numerical libraries. Achieving these goals required thinking of the input tensor as an $[W \cdot C] \times [H]$ matrix and packing it in the contiguous layout for outer-product GEMM matrices; packing the weight tensor in the same way that is done in the GEMM routine of the im2col-based convolution; calling the GEMM microkernel on these contiguously packed filter and image elements, and cleverly scattering the results of this computation into the resulting image. The resulting algorithm, `YaConv`, eliminates explicit image transformations, has a smaller memory footprint, and delivers superior performance than `im2col` by improving cache usage. `YaConv` is the first convolution algorithm that uses unmodified packing and `GEMM` microkernels from a numerical library to compute convolution without multiplying the number of input tensor elements kept in memory or the number of times these elements are loaded into the cache. A detailed performance study, varying parameters on multiple machines, helps understand the superior performance of `YaConv` in comparison with `im2col`: It reduces the number of branches and LLC cache accesses.

REFERENCES

[1] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2020. High-performance low-memory lowering: GEMM-based algorithms for DNN convolution. In *Proceedings of the IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'20)*. 99–106. https://doi.org/10.1109/SBAC-PAD49847.2020.00024

[2] C. Sidney Burrus and T. Parks. 1985. *Convolution Algorithms*. Citeseer, New York, NY.

[3] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Proceedings of the 10th International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, Suvisoft, La Baule (France). Retrieved from https://hal.inria.fr/inria-00112631; http://www.suvisoft.com.

[4] Minsik Cho and Daniel Brand. 2017. MEC: Memory-efficient convolution for deep neural network. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 815–824. Retrieved from https://proceedings.mlr.press/v70/cho17a.html.

[5] Intel Corporation. 2016. oneAPI Deep Neural Network Library (oneDNN). Retrieved from https://github.com/oneapi-src/oneDNN.

[6] Marat Dukhan. 2019. The indirect convolution algorithm. Retrieved from https://arXiv:1907.02129.

[7] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on SIMD architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*. 830–841. https://doi.org/10.1109/SC.2018.00069

[8] Evangelos Georganas, Kunal Banerjee, Dhiraj Kalamkar, Sasikanth Avancha, Anand Venkat, Michael Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. 2020. Harnessing deep learning via a single building block. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'20)*. 222–233. https://doi.org/10.1109/IPDPS47924.2020.00032

[9] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. Retrieved from https://arXiv:1408.5093.

[11] Pablo San Juan, Adrián Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí. 2020. High performance and portable convolution operators for ARM-based multicore processors. Retrieved from https://abs/2005.06410.

[12] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's Thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. Retrieved from http://llvm.cs.uiuc.edu.

[13] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'21)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[14] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 4013–4021. https://doi.org/10.1109/CVPR.2016.435

[15] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. 2021. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. Association for Computing Machinery, New York, NY, 928–942. https://doi.org/10.1145/3445814.3446759

[16] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (Aug. 2016), 18 pages. https://doi.org/10.1145/2925987

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison,

Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, 8024–8035. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[18] Victor Podlozhnyuk. 2007. FFT-based 2D convolution. *NVIDIA White Paper* 32 (2007), 1.

[19] Steven W Smith et al. 1997. The Scientist and Engineer's Guide to Digital Signal Processing.

[20] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium.* 1049–1059. https://doi.org/10.1109/IPDPS.2014.110

[21] Mohammadreza Soltaniyeh, Richard P. Martin, and Santosh Nagarakatte. 2022. An accelerator for sparse convolutional neural networks leveraging systolic general matrix-matrix multiplication. *ACM Trans. Archit. Code Optim.* (Apr. 2022). https://doi.org/10.1145/3532863

[22] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Bharat Kaul, Gagandeep Goyal, and Ramakrishna Upadrasta. 2021. PolyDL: Polyhedral optimizations for creation of high-performance DL primitives. *ACM Trans. Archit. Code Optim.* 18, 1, Article 11 (Jan. 2021), 27 pages. https://doi.org/10.1145/3433103

[23] Vincent M. Weaver. 2013. Linux Perf Event Features and Overhead. (2013).

[24] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *Proceedings of the IEEE 18th International Conference on Parallel and Distributed Systems* (2012), 684–691.

[25] Wenlai Zhao, Haohuan Fu, Jiarui Fang, Weijie Zheng, Lin Gan, and Guangwen Yang. 2018. Optimizing convolutional neural networks on the sunway TaihuLight supercomputer. *ACM Trans. Archit. Code Optim.* 15, 1, Article 13 (Mar. 2018), 26 pages. https://doi.org/10.1145/3177885

[26] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. *Ansor: Generating High-Performance Tensor Programs for Deep Learning.* USENIX Association.

Received 11 May 2022; revised 14 August 2022; accepted 19 October 2022