p-im2col: Simple Yet Efficient Convolution Algorithm with Flexibly Controlled Memory Overhead

ANTON V. TRUSOV¹,², (Member, IEEE), ELENA E. LIMONOVA¹,²,³, (Member, IEEE), DMITRY P. NIKOLAEV²,⁴, (Member, IEEE) and VLADIMIR V. ARLAZAROV²,³, (Member, IEEE)

¹Moscow Institute of Physics and Technology, Dolgoprudny, Russia
²Smart Engines Service LLC, Moscow, Russia
³Institute for Systems Analysis, FRC CSC RAS, Moscow, Russia
⁴Institute for Information Transmission Problems, RAS, Moscow, Russia

Corresponding author: Anton V. Trusov (e-mail: trusov.av@smartengines.com).

This work was supported by a grant from the Ministry of Science and Higher Education of the Russian Federation, internal number 00600/2020/51896, agreement No. 075-15-2020-914. The research was conducted using the infrastructure of the Shared Research Facilities “High Performance Computing and Big Data” (CKP “Informatics”) of FRC CSC RAS (Moscow).

ABSTRACT Convolution is the most time-consuming operation in modern deep artificial neural networks, so its performance is crucial for fast inference. One of the standard approaches to fast convolution computation is to use GeMM-based convolution algorithms relying on efficient general matrix multiplication (GeMM) from optimized BLAS libraries. However, commonly used GeMM-based algorithms may cause significant memory overhead or avoid it only at the cost of worse performance. In this paper, we propose a novel convolution algorithm, p-im2col, based on a well-known im2col algorithm that avoids memory overhead by splitting a single multiplication of a large matrix into several multiplications of smaller matrices. We theoretically and experimentally compare our algorithm with two other GeMM-based algorithms: im2col, which is widely used as a baseline, and the memory-efficient kn2row-aa. We measure the inference time of these algorithms on central processing units of x86, x86_64, ARM, and MIPS architectures for a large set of convolutional parameters. The proposed algorithm demonstrates a speedup over im2col and kn2row-aa in a number of cases and a significant reduction in additional memory requirements compared to im2col. Based on our experiments, we present a new convolution algorithm selection scheme that considers memory restrictions, CPU architecture, and convolutional parameters and provides a noticeable advantage over each particular algorithm.

INDEX TERMS computational efficiency, convolutional neural networks, GeMM-based convolution, memory consumption

I. INTRODUCTION

CONVOLUTIONAL neural networks (CNNs) remain a major tool for pattern recognition, segmentation, object detection and many other tasks [1]–[5]. The most resource-consuming operation in such networks is the tensor convolution in convolutional layers, so its efficiency becomes crucial in actual applications. Efficiency is especially important in real-time recognition systems [6]–[10] that process large amounts of data, Internet of Things (IoT) environments [11], [12] and embedded systems [13]–[15].

In practical applications, it is essential to consider the memory, energy consumption, and computational power. Energy efficiency is important for autonomous systems ranging from smartphones to unmanned vehicles [16], [17]. Moreover, recently, the amount of energy required to solve deep learning problems has also raised concerns [18]–[20]. Low memory consumption improves inference time because high data locality provides fast access and reduces energy costs, as unused memory banks can be placed in low-power mode.

At present, memory and computationally efficient implementation of CNNs is crucial for on-device artificial intelligence systems and can significantly enhance their application domain. For example, it can help to utilize neural networks for visual geolocation of unmanned aerial vehicles [21] or
save energy for user equipment in edge computing networks [22] or for other purposes.

CNNs can be executed on a wide range of computing devices: central processing units (CPUs) graphic processing units (GPUs), tensor processing units (TPUs), field-programmable gate arrays (FPGAs), and application-specific circuits (ASICs). They are usually trained on either GPUs or TPUs because those devices allow for fast computations with high parallelism. However, after training, CNNs may be used in any computing environment. Since all kinds of devices have principally different architectures, efficient algorithms for linear algebra operations (for example, matrix multiplication, which is actively used in CNNs) can be noticeably different and are developed for each of them. For example, CPUs are better suited for sequential calculations and are very sensitive to memory locality, so one should attempt to reduce the number of memory cache misses [23]. GPUs contain many execution units that can simultaneously perform the same operation over a data set, so one should prepare the inputs for a highly parallel system, as well as meet memory limitations [24]. TPUs are similar to GPUs in a number of aspects, as they provide high concurrency for computations but are specialized for machine learning tasks. However, they also strictly constrain memory usage [25]. Unlike all previously discussed devices, FPGAs and ASICs have an unlimited number of execution units as long as the logic gate number and chip area limitations are satisfied; energy consumption is also important for those devices [26]. It is clear that the CNN inference algorithms should also vary to reach the full potential of a specific device.

Note that despite the availability of a variety of hardware, practical applications, especially in mobile, edge, and IoT computing often use CPUs due to their versatility and convenience for engineers and software developers. This is why this work focuses on CPU inference of CNNs.

The most computationally challenging aspect of CNNs concerns the convolutional layers. This is why efficient implementation of these layers is extremely important. The basic convolution is illustrated in Fig. 1. For a 3D input image $I$ and 4D kernel $F$, the following operation is performed to obtain a 3D output image $R$:

$$R(y, x, c_o) = \sum_{h=0}^{K_h-1} \sum_{w=0}^{K_w-1} \sum_{c_i=0}^{C_{in}-1} I(y + \Delta y, x + \Delta x, c_i) \cdot F(c_i, c_o, \Delta y, \Delta x),$$

(1)

where $H \times W \times C_{in}$ defines the input image size, $C_{in} \times C_{out} \times K_h \times K_w$ defines the kernel size and $H_{out} \times W_{out} \times C_{out}$ defines the output image size.

State-of-the-art deep CNNs widely utilize convolutions. We present a number of multiply-accumulate operations in convolutional layers and the total number of parameters for some network architectures in Table 1. As those numbers are high, methods for reducing the convolutional layer complexity are actively under development. For example, we may decrease the number of model parameters using pruning [27]–[30] and knowledge distillation [31], [32]. Additionally, we can approximate the tensor convolution operation with more computationally efficient separable convolutions [33], [34] or eliminate multiplication operations by replacing the covariance with the $L_1$ norm [35], [36] or morphological approximation [37]–[39]. However, such approximations are usually difficult to train. They may also reduce the accuracy of model predictions. Another method is to replace the convolution of floating-point tensors with their quantized approximation. In this way, we not only reduce the inference time of a network but also decrease the memory volume used with reasonable changes in accuracy. There are several widely used frameworks that conduct 8-bit quantized convolution and matrix multiplication, such as Google’s gemmlowp [40] and Facebook’s QNNPACK [41].

However, all these methods still require tensor convolution. Many works [47]–[50] show that direct implementation of tensor convolution is inefficient because it is not cache-friendly (has poor memory locality). The common practice is to transform convolution to matrix multiplication [47] using the im2col algorithm (described in detail in Section II-A), which unrolls the input tensor into a matrix, each row of which represents a single filter application. This allows us to fully exploit general matrix multiplication (GeMM) algorithms from basic linear algebra subprogram (BLAS) libraries. These libraries provide device-specific, highly efficient implementations and regular updates. When using

![FIGURE 1: Basic convolution scheme.](image-url)
them, we do not need to change a significant amount of code when moving from one device to another. One drawback of the im2col algorithm is the consumption of a vast amount of memory used to store the unrolled tensor. Several other GeMM-based convolution algorithms were presented in [48]. Some of those algorithms, such as the kn2row-aa algorithm, use significantly less memory. Instead of computing a single convolution with a $K_h \times K_w$ kernel, the algorithm computes $K_h K_w$ convolutions with $1 \times 1$ kernels as matrix multiplications and accumulates their results with shift (described in detail in Section II-B). Although this algorithm does not allow for efficient computation of convolutions with non-unit stride, it allows a significant reduction in memory and preserves approximately the same inference time on Intel Core i7 and ARM Cortex A57 processors [48].

Note that the set of efficient tensor convolution algorithms is not limited to GeMM-based convolution. One can attempt to implement a cache-friendly direct convolution algorithm by dynamically compiling convolution microkernels best suited to a specific neural network [49] or replace the intermediate buffer of values (of an unrolled image) with a smaller buffer of pointers called an “indirection buffer” [50]. However, all these algorithms require a vast amount of device-specific code and optimizations.

In this paper, we provide a novel algorithm for tensor convolution and compare it to state-of-the-art GeMM-based algorithms using inference speed and memory efficiency. We propose a simple yet effective modification of the im2col algorithm with flexibly controlled memory overhead that preserves the computational efficiency over a wide range of parameters. As a modification of im2col, the proposed method does not impose additional constraints, such as unit stride in the kn2row-aa algorithm. Additionally, it is easy to implement, does not depend greatly on the type (as long as the underlying GeMM supports matrix multiplication for this type) or structure of the data and can be adapted to quantized and decomposed convolutions. We provide a theoretical comparison of the memory consumption of the proposed algorithm with those of the original im2col and kn2row-aa and then experimentally measure the inference time on a wide variety of devices. In this way, we prove that our algorithm is perfectly suited for implementation with CNNs.

The main contributions of our work are as follows:

- we propose p-im2col – a new GeMM-based CPU-targeted convolution algorithm with flexibly controlled memory overhead;
- we provide experimental inference time measurements of the p-im2col and state-of-the-art im2col and kn2row-aa algorithms on a wide range of devices with different CPU architectures;
- since the results vary for different sets of convolution parameters, we introduce a comparison method for convolution algorithms; and
- we suggest an empirical rule for the selection of the fastest GeMM-based convolution algorithm (of the three under consideration).

The remainder of this paper is outlined as follows. In Section II, we describe the im2col and kn2row-aa GeMM-based convolution algorithms and provide our own p-im2col algorithm. In Section III, we theoretically compare these three algorithms in terms of memory and computational efficiency. In Section IV, we present the experimentally measured inference time of these algorithms on multiple devices, analyze the results of the experiments and recommend how to choose the best algorithm for specific parameters of the convolution. Finally, in Section V, we conclude our work.

II. GEMM-BASED CONVOLUTION

We presented the idea of convolution in (1); furthermore, state-of-the-art convolutional layers also support stride and dilation parameters. In general, the convolution of an image $I$ of size $H \times W \times C_{in}$ and a kernel $F$ of size $C_{in} \times C_{out} \times K_h \times K_w$ with a result $R$ of size $H_{out} \times W_{out} \times C_{out}$ can be written as:

$$R(y, x, c_o) = \sum_{\Delta v=0}^{K_h-1} \sum_{\Delta u=0}^{K_w-1} \sum_{c=0}^{C_{in}-1} I(v, u, c) \cdot F(c_o, c_i, \Delta v, \Delta u),$$  \hspace{1cm} (2)

$$y = 0, .., H_{out} - 1, \ x = 0, .., W_{out} - 1,$$
$$c_o = 0, .., C_{out} - 1,$$

where:

$$W_{out} = (W - d_x (K_w - 1))/s_x,$$
$$H_{out} = (H - d_y (K_h - 1))/s_y,$$

$$u = x \cdot s_x + \Delta u \cdot d_x,$$
$$v = y \cdot s_y + \Delta v \cdot d_y,$$

Here, $(s_x, s_y)$ is a stride, and $(d_x, d_y)$ is a dilation. To achieve a certain output size after convolution (for example, to preserve the image size), one can extend the input image with zeroes or other values. This technique is called padding.

As (2) shows, the direct convolution computation contains 6 nested loops. It has poor memory locality. Values of the input image are not loaded in the order they are stored in memory, so the CPU cannot take advantage of prefetching the values via the hierarchical cache system.

To address this problem, GeMM-based convolution algorithms transform input images and kernels into 2D matrices and then perform matrix multiplication or several matrix multiplications and accumulate their results. In this way, we transfer the responsibility of performing cache-friendly computations to the GeMM function in the BLAS library.

Before we describe details of specific GeMM-based convolutions, let us note some important facts concerning GeMM itself. In most modern BLAS libraries, GeMM is inspired by [23]. These libraries split left and right matrices into small blocks so one of them can fit in the L1 cache and the other can fit in the L2 cache. They also reorder the values in these blocks so that these values can be loaded continuously when computing the matrix multiplication of a small piece at a time. We will refer to this piece of the result matrix and the function that computes it as a matrix multiplication.
microkernel. A matrix multiplication microkernel is usually sufficiently small to store all accumulated values in CPU common or SIMD registers.

Now that we have a general idea of GeMM implementation, let us describe GeMM-based convolution algorithms.

A. IM2COL ALGORITHM

im2col is one of the most commonly used and easy-to-implement algorithms of GeMM-based convolution. It is often used in the literature as a baseline to demonstrate an increase in computation, memory or power efficiency.

In this algorithm, for each application of the convolution kernel, we copy values of the input image and save them in a single column (if the image matrix is the right matrix in matrix multiplication) or in a single row (if it is the left matrix). We will consider the unwrapped image to be the left matrix and call it the im2col buffer, and the kernel tensor represented as a matrix will be the right matrix (see Fig. 2).

\[ R(y, x, c_o) = R'(x + yW_{out}, c_o), \quad y = 0, .., H_{out} - 1, \]
\[ x = 0, .., W_{out} - 1, \quad c_o = 0, .., C_{out} - 1. \]

The convolution result \( R \) is obtained from \( R' \) by a simple unrolling:

\[ R(y, x, c_o) = R'(x + yW_{out}, c_o), \quad y = 0, .., H_{out} - 1, \]
\[ x = 0, .., W_{out} - 1, \quad c_o = 0, .., C_{out} - 1. \]

The height of the im2col buffer equals the size of the output image \( (H_{out} \times W_{out}) \), while its width is \( C_{in}K_hK_w \). The buffer requires a vast amount of memory, and initialization of the im2col buffer from the image takes some time. This is why one may want to use a more computationally and memory-efficient convolution algorithm.

B. KN2ROW-AA ALGORITHM

This algorithm is the most memory-efficient GeMM-based convolution algorithm presented in [48]. Instead of computing the convolution of an \( H \times W \times C_{in} \) image \( I \) with a \( C_{in} \times C_{out} \times K_h \times K_w \) kernel \( F \), it computes \( K_hK_w \) convolutions of the same image with \( C_{in} \times C_{out} \times 1 \times 1 \) kernels. Such an approach considers convolution with unit stride. Each of these convolutions equals the matrix multiplication of the image represented as the \( HW \times C_{in} \) matrix \( J \) by the kernel represented as the \( C_{in} \times C_{out} \) matrix \( Q \) (we consider unit dilation for simplicity):

\[ J(u + vW, c_o) = I(v, u, c_i) \]
\[ Q_{\Delta u + \Delta v, c_o} = F(c_i, c_o, \Delta v, \Delta u) \]
\[ \Theta_k = JQ_k \]
\[ v = 0, .., H - 1, \quad u = 0, .., W - 1, \quad c_o = 0, .., C_{out} - 1, \]
\[ \Delta v = 0, .., K_h - 1, \quad \Delta u = 0, .., K_w - 1. \]

The results of these multiplications are accumulated in one buffer with a shift. We will call this buffer the accumulator \( A \). Using this notation, we can present the kn2row-aa algorithm as follows:

\[ A(z, c_o) = \sum_{\Delta v = 0}^{K_h - 1} \sum_{\Delta u = 0}^{K_w - 1} f(z - [(K_h - 1 - \Delta v)W + \Delta u, \Delta u, c_o]) \]
\[ v = 0, .., H - 1, \quad u = 0, .., W - 1, \quad c_o = 0, .., C_{out} - 1, \]
\[ \Delta v = 0, .., K_h - 1, \quad \Delta u = 0, .., K_w - 1, \]
\[ z = 0, .., HW + W(K_h - 1) + K_w - 2, \]

where:

\[ f(\xi, \Delta u, \Delta v, c_o) = \begin{cases} 
0; & \xi < 0 \\
\Theta_{\Delta u + \Delta v, c_o}; & 0 \leq \xi < HW \\
0; & \xi \geq HW 
\end{cases} \]

The convolution result \( R \) can be obtained as:

\[ R(y, x, c_o) = A(x + yW + W(K_h - 1) + K_w - 1, c_o), \]
\[ y = 0, .., H_{out} - 1, \quad x = 0, .., W_{out} - 1, \quad c_o = 0, .., C_{out} - 1. \]

If the input image is zero-padded to preserve the size of the image during the convolution (which was considered in [48]),
then we need to zero out some values of the input image before each multiplication to avoid errors on the boundaries. However, if we do not use padding, then the correct result is already computed in the accumulator, but there are also some excess values (shown in gray in Fig. 3). In this algorithm, we only need to store the “tails” of the accumulator and the zeroed-out values of an image if we use padding or the excess values in the accumulator if we do not. Asymptotically, it requires $O((H + W)K^2C)$ memory, where $K$ is the linear size of the kernel and $C$ is the number of channels in the input or output, which is significantly less than the im2col value of $O(HWK^3C)$.

Furthermore, while im2col suffers from a large memory overhead, kn2row-aa cannot reach the full GeMM efficiency potential because it computes many multiplications with a smaller depth ($C_{in}$ instead of $C_{in}K_hK_w$), kn2row-aa also produces excess values if zero-padding is not used, so one may need to reorder the results after computation. Finally, it does not allow for efficient computation of a convolution with nonunit stride.

### C. PROPOSED P-IM2COL ALGORITHM

In practice, we often need an algorithm with a flexible memory/efficiency balance to identify a variant that will be more efficient than the im2col algorithm in memory but without the constraints of the kn2row-aa algorithm. This is why we propose our easy-to-implement modification of the im2col algorithm, which we call p-im2col.

Matrix multiplication is separable alongside rows of the left matrix or columns of the right matrix, so we do not need the entire large im2col buffer to compute several rows of the output. We can compute only $P$ rows of the im2col buffer at a time and use them to produce $P$ rows of the output. The result of the matrix multiplication has $H_{out}W_{out}$ rows, so we will need $H_{out}W_{out}/P$ iterations to compute it. In this way, we need to allocate memory for $PC_{in}K_hK_w$ values. We call it the left buffer (see Fig. 4). The algorithm is presented below (see Algorithm 1).

**FIGURE 4**: Proposed algorithm.

![Diagram](image)

**Algorithm 1**: p-im2col convolution algorithm

If parameter $P$ is fixed, then the size of the left block (which is the only additional memory required by our algorithm) does not depend on the size of the image. This allows
for a significant memory reduction compared to the im2col algorithm.

Now, let us consider computational efficiency. If parameter $P$ is a multiple of the size of the matrix multiplication microkernel, then there is no difference in the behavior of this microkernel between the proposed algorithm and im2col: the same values are multiplied and accumulated in the same order. However, there is a slight overhead: each time the matrix multiplication function is called, BLAS needs to allocate some memory to store the reordered values for the microkernel. However, the smaller the left matrix is, the more cache-friendly this reordering will be.

Therefore, parameter $P$ needs to be

1) a multiple of the multiplication microkernel height,
2) sufficiently small to reduce the memory overhead, and
3) sufficiently large to not increase the inference time due to the number of GeMM function calls.

These are general ideas regarding the value of parameter $P$, but a specific optimal value depends on many things, such as the microkernel code, CPU preloading strategy, and CPU cache sizes, which are difficult to estimate. This is why we experimentally investigate the dependence of the inference time of our convolution algorithm on the value of $P$ on a wide range of devices and report the obtained values in Section IV.

### III. THEORETICAL COMPARISON

Now that we have considered three GeMM-based convolution algorithms, let us highlight their main differences and similarities. The im2col algorithm computes the convolution using a single call of the GeMM function from the BLAS library but requires the allocation and initialization of a large buffer to do so. Our p-im2col algorithm splits the computation of the result and computes $P$ rows of output $H_{out}W_{out}/P$ times, reinitializing the left matrix before each GeMM call. Furthermore, the kr2row-aa algorithm uses $K_hK_w$ GeMM calls to compute $1 \times 1$ convolutions as matrix multiplications with the accumulation of shifted results. Unlike the first 2 algorithms, the latter does not require any additional initialization before the GeMM calls; however, it requires some additional memory to store the excess values in the output, and it also computes more multiplications than needed.

In Table 2, we show the exact amount of memory, the number of GeMM calls and the total number of multiplications in each algorithm for computing the convolution of an $H \times W \times C_{in}$ image with a $C_{in} \times C_{out} \times K_h \times K_w$ kernel with unit stride, unit dilation and no padding.

To illustrate these formulas, let us consider the convolution of an $S \times S \times C_{in}$ image with a $C_{in} \times C_{out} \times K \times K$ kernel, where $S \in \{20, 100\}$, $C_{in}, C_{out} \in \{8, 128\}$, and $K \in 3, 5$, and assume that values are stored in 32-bit floating point format. These values are representative of real-world applications of the convolution operation in CNNs. The amount of memory needed for the computation of a discrete convolution using the considered GeMM-based algorithms is shown in Table 3. $P$ is set to 120, which is a viable value (see Section IV).

Table 3 shows that on small images, the amount of additional memory is low for all algorithms. However, if we consider large images, im2col requires a vast amount of memory to store the im2col buffer, while neither the proposed method nor kn2row-aa requires as much memory. Usually, kn2row-aa is more memory efficient than our algorithm, but the amount of additional memory required by this algorithm still increases as the image size increases, while the requirement for the p-im2col algorithm remains the same.

### IV. MULTIPLE DEVICE RESEARCH

#### A. OVERALL EFFICIENCY EVALUATION

The memory efficiency of the p-im2col algorithm is strictly defined by the value of parameter $P$, but its computational efficiency does not allow an accurate theoretical prediction. Therefore, our first goal was to experimentally evaluate the

| Algorithm | Additional memory, values | GeMM calls | Multiplications |
|-----------|---------------------------|------------|-----------------|
| im2col    | $C_{in}K_hK_w \times (W - K_w + 1)(H - K_h + 1)$ | 1 | $C_{in}K_hK_wC_{out} \times (W - K_w + 1)(H - K_h + 1)$ |
| kr2row-aa | $(2(K_h - 1)W + (K_w - 1)H + + (K_w - 1)(K_h - 2))C_{out}$ | $K_wK_h$ | $C_{in}K_hK_wC_{out}WH$ |
| p-im2col  | $C_{in}K_hK_wP$ | $(W - K_w + 1)(H - K_h + 1)/P$ | $C_{in}K_hK_wC_{out} \times (W - K_w + 1)(H - K_h + 1)$ |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
algorithm on a wide variety of hardware. Therefore, we considered 8 CPUs of x86, ARM, and MIPS architectures. We used x86/x86_64 devices of several generations. Table 4 presents the list of these devices.

1) Experimental setup
To experimentally evaluate the computational efficiency of the p-im2col algorithm, we measured the inference time for convolutions of various sizes. The main objective of the p-im2col algorithm is lightweight neural networks used in embedded recognition since this is where memory limits are stringent, and we considered quite common parameters for such networks. We tested the convolution of an $S \times S \times C_{in}$ image with a $C_{in} \times C_{out} \times K \times K$ kernel, where $S \in \{50, 70, 85, 100\}$, $C_{in}, C_{out} \in \{8, 16, 32, 64, 128\}$, $K \in \{3, 5\}$, $P \in \{8, 1500\}$, and $P \mod 4 = 0$. We used 32-bit floating-point data and Eigen [51] as a BLAS library. It uses $4 \times 4$ microkernels, so the taken $P$ is always divided by its size. We ran our experiments several times for each set of parameters to ensure that the relative error was less than 0.5%.

2) Efficiency evaluation
The computational efficiency evaluation for the considered algorithms is slightly tricky. The inference time of the algorithm has a rather complicated dependence on the convolution parameters. It depends not only on the number of multiplications but also on the number of additions, memory load/store instructions, cache misses, etc. In our problem, we need to compare the inference time for different parameter sets, so we normalize the inference time using the number of multiplications (see Table 2).

Let us denote the set of convolution parameters for our experiment as $\theta = \{S, C_{in}, C_{out}, K\}$, the inference time of p-im2col with parameter value $P$ as $T_{A}(\theta, P)$, the time of the kn2row-aa algorithm as $T_{B}(\theta)$, and the time of the im2col algorithm as $T_{C}(\theta)$. The number of multiplications is:

$$M(\theta) = C_{out}C_{in}K^2(S - K + 1)^2. \quad (11)$$

For the kn2row-aa algorithm, it is slightly larger: $C_{out}C_{in}K^2S^2$. However, we consider the difference as an overhead. Therefore, the inference time per multiplication in the p-im2col algorithm can be measured as

$$t_{A}(\theta, P) = T_{A}(\theta, P)/M(\theta). \quad (12)$$

The same holds for the kn2row-aa and im2col algorithms:

$$t_{B,C}(\theta, P) = T_{B,C}(\theta)/M(\theta). \quad (13)$$

To find an optimal value of parameter $P$, we take the minimum of the function

$$t_{A}(P) = \mathbb{E}_{\theta}(t_{A}(\theta, P))$$

$$P_{\text{opt}} = \text{argmin}(t_{A}(P)). \quad (14)$$

Thus, $t_{A}(P)$ is the average inference time per multiplication for all convolution parameters in our experiment. These functions for all devices are presented in Fig. 5. Note that we do not compare $t_{A}(P)$ functions for different devices and only show them in the same figure for the reader’s convenience. The optimal values of the parameter $P$ are shown in Table 5.

From Fig. 5, we can see that a single $P_{\text{opt}}$ exists for all the CPUs in our experiment but can vary significantly across different CPUs. Our experiments did not show a strong correlation between the optimal value of $P$ and CPU architecture or convolution parameters. Thus, based on the experimental results, we developed an empirical solution for the suboptimal choice of parameter $P$: use $P = 120$ for x86/x86_64 CPUs and $P = 64$ for others.

### TABLE 5: Optimal values of parameter $P$

| CPU model                        | $P_{\text{opt}}$ |
|----------------------------------|-----------------|
| AMD Ryzen 9 5950X                | 200             |
| Intel Celeron 1007U              | 136             |
| Intel Core i3 8109U              | 168             |
| AMD E-450 APU                    | 56              |
| Intel Core i7 930                | 300             |
| Amlogic S922X Cortex-A53         | 56              |
| Samsung Exynos5422 Cortex-A15    | 80              |
| KOMDIV-64                        | 48              |

3) Efficiency analysis
Now that we have the optimal value of parameter $P$ in the p-im2col algorithm, we can compare its computational efficiency to those of the kn2row-aa and im2col algorithms. The inference times of p-im2col with the optimal $P$, kn2row-aa, and im2col for each test on each device are presented in B. To compare and analyze the methods, we introduce average dependencies for each convolution parameter.

Let us once again use the notations $T_{A}(\theta, P)$, $T_{B}(\theta)$ and $T_{C}(\theta)$ for the experimentally measured inference times of the p-im2col, kn2row-aa and im2col algorithms in test $\theta$,
FIGURE 5: Normalized inference time depending on $P$ for the p-im2col algorithm on different CPUs.

respectively. The number of “useful” multiplications in this test is $M(\theta)$ (see eq. (11)). Now, using the values from Table 5, we can introduce the functions:

$$t_X^A(x) = \mathbb{E} \left[ T_A(\theta|X = x, P_{opt})/M(\theta) \right]$$

$$t_X^B(x) = \mathbb{E} \left[ T_B(\theta|X = x)/M(\theta) \right]$$

$$t_X^C(x) = \mathbb{E} \left[ T_C(\theta|X = x)/M(\theta) \right],$$

where $X$ is $S$, $C_{in}$, or $C_{out}$. An example of these functions on the Ryzen 9 CPU is presented in Fig. 6.

This figure shows that the dependence of the inference time on a convolution parameter varies significantly across the different algorithms. In A, we also present these functions for other devices. We observe some common tendencies of the normalized inference time ($t_X^Y$, where $Y$ is an algorithm and $X$ is a parameter):

- As $C_{in}$ increases, the inference time of the kn2row-aa algorithm usually decreases. This is because both p-im2col and im2col have an initialization stage (the im2col transformation), which grows linearly as $C_{in}$ increases. kn2row-aa, however, does not require any initialization before matrix multiplications are computed, and the efficiency of matrix multiplication improves as the depth increases with increasing $C_{in}$ (this holds for all three algorithms).

- As $C_{out}$ increases, the normalized inference time of all three algorithms decreases as matrix multiplication becomes more efficient. However, $t_C^{out}$ demonstrates the fastest descent, probably because it has only one GeMM call.

- The dependence of the normalized inference time on $S$ is not very clear and varies across different devices. This parameter mainly affects the amount of allocated memory, especially in the im2col algorithm.

B. CONVOLUTION ALGORITHM SELECTION

We have described the theoretically and experimentally measured inference times of three GeMM-based convolution algorithms: the classic im2col, the memory-efficient kn2row-aa, and the novel p-im2col, which is, in fact, a family of algorithms since we can vary parameter $P$. Now, it would be reasonable to decide how to choose the best algorithm. Let us consider two scenarios:

- when there is a strict memory constraint and
- when we use optimal parameter $P$ to obtain a better inference time along with reduced memory usage.

1) Algorithm selection based on memory constraints

If the amount of memory is limited, then we would not recommend using the im2col algorithm because the im2col buffer requires a vast amount of memory. Instead, one should choose between the kn2row-aa and p-im2col algorithms using the formulas from Table 2 to meet the constraints. By decreasing the value of parameter $P$, we can significantly decrease the memory consumption of the p-im2col algorithm.
2) Algorithm selection based on computational efficiency

If no memory constraint is present, then we can use the optimal $P$ for the p-im2col algorithm, which is given in Table 5. Therefore, we compared the inference times of im2col, kn2row-aa, and optimal p-im2col for the set of convolutional parameters defined in Section IV-A1. The comparison is presented in B. From these results, we can conclude that the fastest convolution algorithm may significantly vary across different devices and different convolutional parameters. However, there are common tendencies:

- The im2col algorithm tends to be the fastest for a large number of output channels, especially for a larger kernel. Matrix multiplication is asymptotically more complex than reordering in the im2col transformation (it requires $O(HWC_{out}C_{in}^2 K^4)$ operations versus $O(HWC_{out}K^2)$ in initialization of the im2col buffer). Therefore, as $C_{out}$ or $K$ increases, the time required for im2col transformation becomes less impactful than the time for matrix multiplication, and the single GeMM call in the im2col algorithm works faster than multiple calls in the p-im2col or kn2row-aa algorithms.
- The kn2row-aa algorithm is often the fastest if the number of input channels is large and the number of output channels is small. For kn2row-aa, $C_{in}$ is the depth of matrix multiplication; therefore, as it increases, GeMM becomes more efficient. However, as $C_{out}$ increases, the amount of additional memory and the number of excess computations in this algorithm also increase.
- In other cases, when neither $C_{in}$ nor $C_{out}$ is large, the p-im2col algorithm tends to be the fastest, as it has the same computational complexity as im2col but better memory locality.

Considering these points, we developed the following five empirical strategies to choose a convolution algorithm:

- (i)im2col. Always use the im2col algorithm.
- (k)kn2row. Always use the kn2row-aa algorithm.
- (p)-im2col. Always use the p-im2col algorithm with $P = 120$ for x86/x86_64 CPUs and $P = 64$ for others.
- (c)ombined.
  - Use the im2col algorithm if $C_{out} \geq 128$.
  - Use the kn2row-aa algorithm if we did not choose im2col and $C_{in} \geq 128$.
  - Use the p-im2col algorithm with the same parameter $P$ as in the p-im2col strategy in all other cases.
- (a)daptive. For each specific device, we can run a set of experiments as we did in Sections IV-A and B and choose the fastest algorithm for particular convolution parameters.

In the first three strategies (i, k and p), we simply use one of the algorithms. In strategy c, we combine them based on the empirical rules deduced after analyzing the results presented in B. Strategy a is likely to achieve the best results but requires running additional tests on a target device. Fortunately, this can be done once during the configuration of a library containing this convolution algorithm, which is a
standard case for high-performance libraries. For example, the FFTW library for fast discrete Fourier transform uses such an approach [52].

Let us compare these 5 strategies. To this end, we used data obtained in the experiments from B. Let us assume that on a specific test $\theta \in \Theta$ ($\Theta$ is a set of all tests), strategy $x$ achieves an inference time of $\tau_x(\theta)$, and strategy $y$ achieves an inference time of $\tau_y(\theta)$. Then, we define the comparison function on a single test as:

$$
f(x, y, \theta, \alpha) = \begin{cases} 
1, & \text{if } \frac{\tau_x(\theta) - \tau_y(\theta)}{\tau_y(\theta)} > \alpha \\
-1, & \text{if } \frac{\tau_x(\theta) - \tau_y(\theta)}{\tau_x(\theta)} > \alpha \\
0, & \text{otherwise}
\end{cases}
$$

where $\alpha$ denotes a certain threshold value. For the purpose of our comparison, we used $\alpha = 0.05$, which means that if the difference between $\tau_x(\theta)$ and $\tau_y(\theta)$ was less than 5%, we considered that strategies $x$ and $y$ have approximately the same inference time on test $\theta$.

Table 6 presents the average value of $f(x, y, \theta, \alpha)$ on all tests $\theta \in \Theta$ and on all devices that we used in our experiments.

**TABLE 6: Strategy comparison of $x$ vs. $y$**

The higher the score is, the better $x$ is than $y$.

| $x$ | $y$ | $i$ | $k$ | $p$ | $c$ | $a$ |
|-----|-----|-----|-----|-----|-----|-----|
| $i$ | 0   | 0.093 | -0.378 | -0.503 | -0.549 |
| $k$ | 0.093 | 0   | -0.538 | -0.561 | -0.668 |
| $p$ | 0.378 | 0.538 | 0   | -0.028 | -0.154 |
| $c$ | 0.503 | 0.561 | 0.028 | 0   | -0.128 |
| $a$ | 0.549 | 0.668 | 0.154 | 0.128 | 0   |

As Table 6 shows, strategy $a$ outperforms all other strategies, but it is more difficult to implement because it requires an additional test for each device. Instead, we can use strategy $c$ or even the simpler $p$ without considerable performance loss. However, strategies $i$ and $k$ show similar results that are noticeably worse than the results of the remaining three strategies. This means that strategies $p$ and $c$ are more universal than $i$ and $k$. Therefore, we can give the following recommendations for the choice of the convolution algorithm:

- If there are no memory limitations and we can run tests on a specific device during the configuration of the algorithm, then one should use strategy $a$.
- If there are no memory limitations but we need an out-of-the-box solution with satisfactory performance, then one should use strategy $c$.
- If we need to keep memory consumption low but the limitation is not strict, then one should use strategy $p$.
- If the memory constraint is strict, then one should choose an algorithm according to this constraint, as we discussed in Section IV-B1.
- Recall that we conducted the experiment with Eigen as a BLAS library, so it would be sensible to experimentally find an optimal value of $P$ if another library is used.

**V. CONCLUSION**

In this paper, we considered two standard GeMM-based convolution algorithms that can be used in CNNs: im2col and kn2row-aa. We proposed the novel algorithm p-im2col, which is an easy yet efficient modification of the im2col algorithm. In this algorithm, we can flexibly control the memory requirements using the parameter value. p-im2col can be used on any CPU architecture and with any BLAS library that implements GeMM operations for this architecture; however, parameter $P$ should be fine-tuned for this combination of a CPU and a library. Although the proposed algorithm does not achieve the memory efficiency of the kn2row-aa algorithm, the computation of kn2row-aa can take more time, especially when the number of input channels is low.

We performed extensive experiments on a wide range of x86/x86_64, ARM, and MIPS CPUs using Eigen as a BLAS library. We compared the required amount of memory and computational complexity of the considered algorithms and measured the corresponding inference times. Using these results, we developed several strategies for choosing the convolution algorithm for specific parameters based on memory limitations or minimizing the inference time. The p-im2col algorithm is a more universal solution than kn2row-aa or the classic im2col. It falls slightly behind the adaptive strategy (that optimally chooses p-im2col, kn2row-aa, or im2col for a specific device and specific convolution parameters) in terms of computational efficiency.

Future research on the p-im2col algorithm could take one of the following directions:

- one could theoretically deduce or at least find strong empirical patterns for $P_{opt}$ and its relations to convolution parameters, since in this paper we experimentally selected a constant value for each device;
- experimentally obtain memory and inference time measurements with different BLAS libraries and data types of the image and kernel, for example, for quantized neural networks;
- or investigate the potential of p-im2col algorithm on non-CPU architectures.
APPENDIX A DEPENDENCE OF THE INFECTION TIME ON THE CONVOLUTION PARAMETERS

In this section, we present the dependence of the normalized inference speed of the p-im2col \( t^X_A(x) \), kn2row-aa \( t^X_B(x) \) and im2col \( t^X_C(x) \) algorithms (according to (15)) on particular convolution parameters for a set of considered devices (experimental setup described in Section IV-A).

APPENDIX B TIME MEASUREMENTS FOR A SET OF DEVICES

In this section, we provide the experimentally measured inference times of the convolution of an \( S \times S \times C_{in} \) image with a \( C_{in} \times C_{out} \times K \times K \) kernel using different algorithms. Each cell of the tables below contains the inference time of the p-im2col with optimal parameter \( P(A) \), kn2row-aa (B) and im2col (C) algorithms in milliseconds. The color represents the fastest algorithm: light-green means that p-im2col is 3-6% faster than the others, green means that it is 6-10% faster, and dark-green means that it is more than 10% faster. Similarly, the colors light-yellow, yellow and dark-yellow are used to show that the kn2row-aa algorithm is the fastest, and the colors light-red, red and dark-red are used when both p-im2col and kn2row-aa are slower than standard im2col. If there is no algorithm that is at least 3% faster than two others correspondig sell remains white.

FIGURE 7: Dependence of the inference time on the convolution parameters for an AMD Ryzen 9 5950X x86_64 CPU.
FIGURE 8: Dependence of the inference time on the convolution parameters for an Amlogic S922X Cortex-A53 ARMv8-A CPU.

FIGURE 9: Dependence of the inference time on the convolution parameters for an Intel Celeron 1007U x86_64 CPU.
FIGURE 10: Dependence of the inference time on the convolution parameters for an Intel Core i7 930 x86_32 CPU.

FIGURE 11: Dependence of the inference time on the convolution parameters for an Intel Core i3 8109U x86_64 CPU.
FIGURE 12: Dependence of the inference time on the convolution parameters for an AMD E-450 x86_64 APU.

FIGURE 13: Dependence of the inference time on the convolution parameters for a KOMDIV-64 MIPS CPU.
FIGURE 14: Dependence of the inference time on the convolution parameters for a Samsung Exynos 5422 Cortex-A15 ARMv7-A CPU.
### TABLE 7: AMD Ryzen 9 5950X (x86_64)

| S  | C_{in} | C_{out} | 8   | 16  | 32  | 64   | 128  | 8   | 16  | 32  | 64   | 128  |
|----|--------|---------|-----|-----|-----|------|------|-----|-----|-----|------|------|
| 30 |        |         |     |     |     |      |      |     |     |     |      |      |
| 8  | A 0.10 | B 0.10 | 0.16| 0.16| 0.28| 0.52 | 1.00 | 0.23| 0.38| 0.67| 1.28 | 2.5  |
|    | C 0.09 |        | 0.15| 0.27| 0.51| 1.04 |      | 0.23| 0.38| 0.67| 1.26 | 2.5  |
|    |        |         |     |     |     |      |      |     |     |     |      |      |
| 50 |        |         |     |     |     |      |      |     |     |     |      |      |
| 8  | A 0.17 | B 0.18 | 0.29| 0.30| 0.53| 1.00 | 1.95 | 0.42| 0.72| 1.33| 2.5  | 4.9  |
|    | C 0.18 |        | 0.29| 0.53| 0.99| 1.98 |      | 0.43| 0.72| 1.32| 2.5  | 5.0  |
|    |        |         |     |     |     |      |      |     |     |     |      |      |
| 70 |        |         |     |     |     |      |      |     |     |     |      |      |
| 8  | A 0.33 | B 0.33 | 0.56| 0.92| 1.12| 2.2  | 4.4  | 0.91| 1.64| 3.1  | 6.0  | 11.9 |
|    | C 0.34 |        | 0.57| 1.03| 1.97| 3.9  |      | 0.86| 1.45| 2.7  | 5.2  | 10.1 |
|    |        |         |     |     |     |      |      |     |     |     |      |      |
| 85 |        |         |     |     |     |      |      |     |     |     |      |      |
| 8  | A 0.64 | B 0.64 | 1.13| 1.15| 2.1 | 4.0  | 7.8  | 1.67| 2.9  | 5.3  | 10.2 | 19.9 |
|    | C 0.66 |        | 1.14| 2.1 | 4.0  | 7.9  |      | 1.94| 3.4  | 6.1  | 11.2 | 20.0 |
|    |        |         |     |     |     |      |      |     |     |     |      |      |
| 100|        |         |     |     |     |      |      |     |     |     |      |      |
| 8  | A 1.31 | B 1.30 | 2.3 | 2.3 | 4.2 | 8.0  | 15.6 | 3.4 | 5.9  | 10.8 | 20.0 | 40   |
|    | C 1.41 |        | 2.5 | 4.6 | 8.6 | 16.3 |      | 6.2 | 8.9  | 14.0 | 23.4 | 45   |
|    |        |         |     |     |     |      |      |     |     |     |      |      |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
### TABLE 8: Amlogic S922X Cortex-A53 (ARMv8-A)

| S   | R      | C\textsubscript{in} | C\textsubscript{out} | 8     | 16   | 32   | 64   | 128  | 8     | 16   | 32   | 64   | 128  |
|-----|--------|---------------------|----------------------|-------|------|------|------|------|-------|------|------|------|------|------|
| 8   | A      | 0.77                | 1.18                 | 2.1   | 3.9  | 7.8  | 2.3  | 3.2  | 5.6   | 11.2 | 24   |      |
|     | B      | 0.88                | 1.50                 | 2.8   | 5.8  | 14.8 | 2.4  | 4.1  | 7.9   | 15.4 | 38   |      |
|     | C      | 0.95                | 1.38                 | 2.4   | 4.4  | 8.2  | 4.9  | 6.2  | 8.3   | 13.3 | 22   |      |
| 16  | A      | 1.56                | 2.3                  | 4.2   | 7.9  | 16.2 | 4.0  | 6.4  | 12.7  | 23   | 47   |      |
|     | B      | 1.62                | 2.7                  | 5.0   | 9.5  | 22   | 4.4  | 7.7  | 14.5  | 25   | 62   |      |
|     | C      | 2.9                 | 3.3                  | 5.3   | 9.3  | 16.2 | 7.8  | 11.1 | 16.6  | 25   | 42   |      |
| 32  | A      | 3.1                 | 4.6                  | 8.5   | 17.3 | 35   | 9.2  | 14.5 | 25    | 45   | 92   |      |
|     | B      | 3.2                 | 5.1                  | 9.2   | 17.5 | 39   | 9.4  | 14.3 | 25    | 47   | 109  | 27   |
|     | C      | 5.9                 | 6.7                  | 11.5  | 20.0 | 32   | 13.8 | 19.2 | 29    | 47   | 82   |      |
| 64  | A      | 6.2                 | 9.3                  | 17.9  | 35   | 69   | 21   | 33   | 53    | 92   | 187  | 29   |
|     | B      | 7.9                 | 11.1                 | 19.1  | 35   | 65   | 22   | 31   | 56    | 97   | 183  | 33   |
|     | C      | 10.3                | 12.4                 | 20.3  | 37   | 65   | 25   | 40   | 60    | 94   | 165  | 45   |
| 70  | A      | 16.9                | 23                   | 39    | 70   | 140  | 48   | 64   | 106   | 191  | 370  | 52   |
|     | B      | 17.5                | 25                   | 43    | 73   | 132  | 50   | 67   | 125   | 200  | 370  | 57   |
|     | C      | 19.7                | 29                   | 45    | 72   | 127  | 52   | 76   | 127   | 191  | 330  |      |
| 128 | A      | 1.55                | 2.3                  | 4.3   | 7.7  | 15.5 | 4.6  | 6.6  | 11.6  | 22   | 48   |      |
|     | B      | 1.81                | 3.2                  | 6.3   | 11.5 | 28   | 5.1  | 8.5  | 16.1  | 29   | 73   |      |
|     | C      | 2.2                 | 3.0                  | 5.0   | 8.6  | 16.4 | 10.2 | 12.5 | 17.0  | 26   | 45   |      |
| 16  | A      | 3.1                 | 4.7                  | 8.4   | 15.4 | 32   | 8.0  | 12.7 | 24    | 48   | 130  | 28   |
|     | B      | 3.3                 | 5.5                  | 10.3  | 18.3 | 43   | 9.3  | 14.9 | 28    | 50   | 125  | 27   |
|     | C      | 4.7                 | 6.5                  | 11.2  | 17.7 | 32   | 12.9 | 19.7 | 32    | 51   | 89   |      |
| 32  | A      | 6.2                 | 9.2                  | 17.0  | 34   | 69   | 17.4 | 29   | 56    | 97   | 197  | 36   |
|     | B      | 6.5                 | 10.0                 | 18.0  | 33   | 76   | 18.7 | 28   | 51    | 93   | 229  | 50   |
|     | C      | 10.2                | 13.3                 | 22    | 38   | 65   | 24   | 41   | 66    | 100  | 270  | 58   |
| 64  | A      | 12.6                | 18.9                 | 38    | 72   | 143  | 44   | 63   | 113   | 197  | 380  | 72   |
|     | B      | 16.1                | 21                   | 38    | 71   | 131  | 44   | 61   | 112   | 197  | 350  | 70   |
|     | C      | 19.1                | 24                   | 44    | 75   | 131  | 51   | 84   | 131   | 200  | 340  | 77   |
| 70  | A      | 31                 | 48                   | 77    | 141  | 280  | 96   | 143  | 220   | 380  | 760  | 128  |
|     | B      | 35                 | 48                   | 86    | 145  | 260  | 93   | 138  | 250   | 340  | 730  | 168  |
|     | C      | 38                 | 60                   | 92    | 148  | 250  | 108  | 182  | 270   | 400  | 690  |      |
| 85  | A      | 9.2                 | 13.7                 | 25    | 49   | 104  | 27   | 46   | 80    | 142  | 290  |      |
|     | B      | 9.6                 | 14.8                 | 26    | 49   | 115  | 27   | 42   | 75    | 137  | 330  |      |
|     | C      | 15.1                | 20                   | 32    | 56   | 98   | 42   | 64   | 97    | 150  | 360  |      |
| 128 | A      | 18.6                | 28                   | 57    | 110  | 200  | 72   | 99   | 159   | 280  | 570  |      |
|     | B      | 21                 | 32                   | 55    | 105  | 192  | 67   | 90   | 163   | 280  | 520  |      |
|     | C      | 30                 | 37                   | 65    | 114  | 196  | 80   | 125  | 195   | 300  | 510  |      |
| 100 | A      | 49                 | 75                   | 115   | 210  | 420  | 153  | 210  | 320   | 560  | 1140 | 52   |
|     | B      | 51                 | 73                   | 123   | 210  | 380  | 142  | 200  | 340   | 580  | 1060 | 128  |
|     | C      | 60                 | 96                   | 134   | 210  | 380  | 168  | 270  | 380   | 580  | 1040 | 256  |

![Image](https://example.com/image.png)
| S | K | $C_{in}$ | $C_{out}$ | T | T | T |
|---|---|---|---|---|---|---|
| 8 | A | 0.53 | 0.78 | 1.28 | 2.3 | 4.3 |
|   | B | 0.53 | 0.89 | 1.61 | 3.1 | 6.1 |
|   | C | 0.51 | 0.75 | 1.23 | 2.2 | 4.3 |
| 16 | A | 0.91 | 1.37 | 2.3 | 4.3 | 8.3 |
|   | B | 0.93 | 1.53 | 2.7 | 5.1 | 10.0 |
|   | C | 0.89 | 1.36 | 2.3 | 4.4 | 8.3 |
| 50 | A | 1.67 | 2.5 | 4.5 | 8.4 | 16.3 |
|   | B | 1.72 | 2.8 | 5.0 | 9.4 | 18.4 |
|   | C | 1.79 | 2.7 | 4.8 | 8.6 | 16.4 |
| 64 | A | 3.1 | 5.2 | 9.0 | 16.9 | 32 |
|   | B | 3.3 | 5.4 | 9.6 | 17.9 | 35 |
|   | C | 3.7 | 6.0 | 9.7 | 17.3 | 32 |
| 70 | A | 6.3 | 11.8 | 19.0 | 33 | 65 |
|   | B | 6.3 | 10.4 | 18.3 | 35 | 68 |
|   | C | 7.6 | 13.1 | 20 | 34 | 64 |
| 85 | A | 1.07 | 1.57 | 2.6 | 4.6 | 8.8 |
|   | B | 1.05 | 1.78 | 3.2 | 6.1 | 12.3 |
|   | C | 1.03 | 1.52 | 2.5 | 4.6 | 8.8 |
| 100 | A | 1.84 | 2.7 | 4.6 | 8.8 | 16.9 |
|    | B | 1.83 | 3.0 | 5.4 | 10.2 | 19.9 |
|    | C | 1.90 | 2.9 | 4.8 | 8.9 | 16.7 |
| 128 | A | 3.4 | 5.3 | 9.5 | 17.2 | 32 |
|    | B | 3.4 | 5.6 | 9.8 | 18.6 | 36 |
|    | C | 3.8 | 5.7 | 10.0 | 17.6 | 32 |

**TABLE 9: Intel Celeron 1007U (x86_64)**
### TABLE 10: Intel Core i7 930 (x86_32)

| S  | C_{in} | C_{out} | 8   | 16  | 32  | 64  | 128 | 8   | 16  | 32  | 64  | 128 |
|----|--------|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 50 | A      | 0.43    | 0.71 | 1.30 | 2.5 | 4.9 | 1.05 | 1.75 | 3.3  | 6.2  | 11.8 |
|    | B      | 0.59    | 1.08 | 2.1  | 4.1 | 8.7 | 1.62 | 3.0  | 5.7  | 11.2 | 23  |
|    | C      | 0.43    | 0.73 | 1.32 | 2.5 | 4.9 | 1.12 | 1.81 | 3.2  | 6.2  | 12.0 |
| 64 | A      | 0.88    | 1.35 | 2.5  | 4.9 | 9.4 | 1.97 | 3.4  | 6.4  | 12.2 | 23  |
|    | B      | 0.93    | 1.71 | 3.3  | 6.4 | 12.6 | 2.6 | 4.7  | 9.2  | 17.6 | 34  |
|    | C      | 0.87    | 1.43 | 2.5  | 4.9 | 9.5 | 2.1  | 3.5  | 6.4  | 12.0 | 23  |
| 70 | A      | 1.58    | 2.7  | 5.1  | 9.5 | 18.7 | 4.0  | 7.0  | 12.5 | 24   | 47  | 92  |
|    | B      | 1.63    | 3.0  | 5.8  | 11.1 | 21  | 4.5  | 8.3  | 15.9 | 30   | 60  |
|    | C      | 1.62    | 2.7  | 5.0  | 9.5 | 18.3 | 4.4  | 7.3  | 12.8 | 24   | 46  |
| 85 | A      | 2.9     | 5.5   | 9.9 | 18.8 | 36  | 7.8  | 13.5 | 24   | 47   | 92  |
|    | B      | 3.0     | 5.6   | 10.6 | 20  | 40  | 8.3  | 15.3 | 29   | 56   | 113 |
|    | C      | 3.0     | 5.6   | 10.0 | 18.9 | 36  | 10.1 | 18.7 | 26   | 49   | 93  |
| 100| A      | 5.9     | 10.8  | 19.6 | 39  | 72  | 15.7 | 26   | 49   | 94   | 184 |
|    | B      | 5.8     | 10.5  | 20  | 40  | 80  | 16.0 | 29   | 55   | 111  | 220 |
|    | C      | 7.5     | 12.3  | 21  | 38  | 74  | 19.9 | 30   | 53   | 97   | 186 |
|    | A      | 0.86    | 1.44  | 2.6  | 5.0 | 9.9 | 2.0  | 3.8  | 6.7  | 12.7 | 24  |
|    | B      | 1.18    | 2.1   | 4.1  | 7.9 | 17.3 | 3.2  | 5.9  | 11.1 | 21   | 46  |
|    | C      | 0.88    | 1.47  | 2.6  | 4.9  | 9.8 | 2.3  | 3.7  | 6.7  | 12.7 | 24  |
| 70 | A      | 1.77    | 2.8   | 5.1  | 9.8 | 18.8 | 4.3  | 7.2  | 13.1 | 24   | 48  |
|    | B      | 1.86    | 3.4   | 6.4  | 12.5 | 24  | 5.1  | 9.4  | 17.7 | 34   | 68  |
|    | C      | 1.77    | 2.9   | 5.1  | 9.8 | 19.1 | 4.3  | 7.2  | 13.3 | 25   | 49  |
| 85 | A      | 3.3     | 5.5   | 10.2 | 19.2 | 37  | 8.1  | 13.8 | 23   | 47   | 99  |
|    | B      | 3.2     | 5.8   | 11.2 | 21  | 42  | 8.8  | 16.2 | 30   | 60   | 118 |
|    | C      | 3.1     | 5.3   | 10.0 | 19.0 | 36  | 10.4 | 16.1 | 27   | 50   | 95  |
| 100| A      | 6.0     | 10.9  | 19.7 | 37  | 73  | 16.2 | 27   | 50   | 97   | 190 |
|    | B      | 5.9     | 10.8  | 20  | 39  | 80  | 16.3 | 29   | 56   | 110  | 220 |
|    | C      | 7.8     | 12.6  | 21  | 38  | 73  | 20   | 32   | 54   | 100  | 193 |
|    | A      | 12.0    | 21    | 39  | 77  | 144  | 32  | 56   | 102  | 193  | 380 |
|    | B      | 11.3    | 20    | 39  | 80  | 156  | 31  | 57   | 110  | 220  | 440 |
|    | C      | 15.9    | 25    | 42  | 77  | 147  | 41  | 63   | 109  | 200  | 380 |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
| S | $C_{in}$ | $C_{out}$ | 8 | 16 | 32 | 64 | 128 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | A | 0.19 | 0.28 | 0.48 | 0.87 | 1.71 | 0.39 | 0.65 | 1.15 | 2.2 | 4.2 |
|   | B | 0.18 | 0.31 | 0.57 | 1.09 | 2.1 | 0.50 | 0.85 | 1.55 | 2.9 | 5.7 |
|   | C | 0.21 | 0.30 | 0.49 | 0.89 | 1.70 | 0.45 | 0.70 | 1.19 | 2.2 | 4.2 |
| 64 | A | 0.31 | 0.50 | 0.89 | 1.69 | 3.3 | 0.75 | 1.24 | 2.3 | 4.3 | 8.3 |
|   | B | 0.31 | 0.53 | 0.99 | 1.89 | 3.7 | 0.85 | 1.47 | 2.7 | 5.2 | 10.1 |
|   | C | 0.33 | 0.53 | 0.92 | 1.73 | 3.3 | 0.97 | 1.57 | 2.6 | 4.6 | 8.6 |
| 70 | A | 0.57 | 0.97 | 1.81 | 3.4 | 6.4 | 1.47 | 2.4 | 4.5 | 8.4 | 16.5 |
|   | B | 0.56 | 1.00 | 1.85 | 3.6 | 7.0 | 1.57 | 2.8 | 5.1 | 9.9 | 19.3 |
|   | C | 0.68 | 1.09 | 1.95 | 3.6 | 6.8 | 2.1 | 3.1 | 5.3 | 9.3 | 18.1 |
| 85 | A | 1.14 | 2.0 | 3.5 | 6.6 | 13.1 | 2.9 | 5.1 | 9.1 | 17.0 | 33 |
|   | B | 1.09 | 1.93 | 3.6 | 7.0 | 13.7 | 3.0 | 5.3 | 9.9 | 19.1 | 37 |
|   | C | 1.59 | 2.5 | 4.0 | 7.2 | 14.0 | 4.8 | 7.0 | 10.9 | 20.0 | 38 |
| 100 | A | 0.73 | 1.39 | 2.8 | 5.6 | 11.2 | 2 | 3.3 | 6.8 | 13.1 | 27 |
|   | B | 0.71 | 1.38 | 2.9 | 6.1 | 11.7 | 2.1 | 3.4 | 7.1 | 13.1 | 26 |
|   | C | 1.06 | 1.91 | 3.5 | 7.0 | 13.2 | 3.1 | 5.4 | 9.6 | 18.6 | 37 |
| 8 | A | 0.38 | 0.58 | 0.96 | 1.80 | 3.4 | 0.83 | 1.34 | 2.4 | 4.5 | 8.7 |
|   | B | 0.36 | 0.61 | 1.12 | 2.1 | 4.2 | 0.99 | 1.67 | 3.0 | 5.8 | 11.3 |
|   | C | 0.40 | 0.59 | 1.01 | 1.83 | 3.5 | 1.06 | 1.60 | 2.7 | 4.8 | 9.0 |
| 16 | A | 0.63 | 1.01 | 1.80 | 3.4 | 6.6 | 1.57 | 2.6 | 4.7 | 8.8 | 17.4 |
|   | B | 0.61 | 1.06 | 1.93 | 3.7 | 7.4 | 1.68 | 2.9 | 5.3 | 10.2 | 20.8 |
|   | C | 0.76 | 1.20 | 2.0 | 3.7 | 7.0 | 2.3 | 3.3 | 5.4 | 9.6 | 18.6 |
| 32 | A | 1.18 | 1.95 | 3.6 | 6.7 | 13.2 | 3.0 | 5.1 | 9.3 | 17.4 | 34 |
|   | B | 1.12 | 1.96 | 3.6 | 7.0 | 13.9 | 3.1 | 5.4 | 10.0 | 19.3 | 38 |
|   | C | 1.64 | 2.4 | 4.0 | 7.3 | 14.2 | 5.2 | 7.3 | 11.5 | 20 | 37 |
| 64 | A | 2.3 | 3.9 | 7.1 | 13.3 | 26 | 6.0 | 10.3 | 18.7 | 34 | 67 |
|   | B | 2.1 | 3.8 | 7.0 | 13.6 | 27 | 5.9 | 10.4 | 19.5 | 37 | 75 |
|   | C | 3.7 | 5.4 | 8.5 | 15.7 | 28 | 10.7 | 14.9 | 24 | 41 | 73 |
| 128 | A | 4.6 | 8.0 | 14.2 | 26 | 52 | 12.2 | 21 | 38 | 71 | 144 |
|   | B | 4.3 | 7.5 | 14.1 | 27 | 54 | 11.7 | 20 | 39 | 76 | 151 |
|   | C | 8.1 | 11.5 | 18.7 | 31 | 56 | 21 | 29 | 48 | 81 | 146 |
| S     | K   | \( C_{\text{in}} \) | \( C_{\text{out}} \) | 3 | 5 |
|-------|-----|---------------------|---------------------|---|---|
| 8     | A   | 1.68 2.7 4.8 9.0 17.9 | 3.9 6.5 11.8 22 43 |   |   |
|       | B   | 1.86 3.1 5.8 11.7 23  | 4.9 8.5 15.8 30 60 |   |   |
|       | C   | 1.97 3.0 5.0 9.1 17.4  | 4.7 7.1 12.2 22 42 |   |   |
| 16    | A   | 3.1 5.1 9.2 17.8 34  | 7.5 12.7 23 44 92 |   |   |
|       | B   | 3.2 5.5 10.4 20 39  | 8.7 15.0 28 55 107 |   |   |
|       | C   | 3.7 5.6 9.6 17.8 33  | 9.3 14.1 25 45 84 |   |   |
| 32    | A   | 6.1 10.2 18.5 35 69 | 15.3 25 47 93 186 |   |   |
|       | B   | 5.8 10.3 19.7 37 73 | 16.1 28 54 103 200 |   |   |
|       | C   | 7.3 11.2 19.9 35 66 | 18.6 28 51 90 167 |   |   |
| 50    | A   | 11.9 20 37 74 147 | 30 56 101 191 370 |   |   |
|       | B   | 12.3 20 37 72 142 | 33 57 104 199 390 |   |   |
|       | C   | 14.2 24 39 69 131 | 36 62 101 180 330 |   |   |
| 64    | A   | 23 43 79 152 290 | 71 119 200 364 750 |   |   |
|       | B   | 23 40 73 142 270 | 64 111 200 390 760 |   |   |
|       | C   | 28 48 79 141 260 | 73 125 200 360 670 |   |   |
| 70    | A   | 3.3 5.5 9.7 18.3 36 | 8.1 13.5 24 46 90 |   |   |
|       | B   | 3.5 6.2 12.1 23 45 | 9.6 16.8 31 60 118 |   |   |
|       | C   | 4.0 6.0 10.2 18.4 35 | 9.7 14.9 25 46 87 |   |   |
| 16    | A   | 6.3 10.3 18.5 35 69 | 15.6 26 48 92 199 |   |   |
|       | B   | 6.4 11.5 21 40 78 | 17.6 31 57 108 210 |   |   |
|       | C   | 7.5 11.5 19.4 35 67 | 19.1 29 52 92 172 |   |   |
| 32    | A   | 12.4 20 37 70 140 | 31 52 97 192 380 |   |   |
|       | B   | 12.7 21 39 74 144 | 34 58 106 200 390 |   |   |
|       | C   | 14.7 22 39 71 133 | 38 58 104 186 340 |   |   |
| 64    | A   | 23 40 74 150 290 | 64 117 200 390 760 |   |   |
|       | B   | 24 40 74 141 270 | 67 112 200 390 760 |   |   |
|       | C   | 28 47 78 140 260 | 75 128 200 370 690 |   |   |
| 128   | A   | 48 87 160 300 580 | 146 240 420 820 1540 |   |   |
|       | B   | 46 78 144 270 540 | 127 210 390 770 1490 |   |   |
|       | C   | 57 98 160 280 520 | 151 250 420 740 1590 |   |   |

**TABLE 12: AMD E-450 APU (x86_64)**
| S     | K    | \( C_{vin} \) | \( C_{out} \) | \( S \) | \( 8 \) | \( 16 \) | \( 32 \) | \( 64 \) | \( 128 \) | \( 8 \) | \( 16 \) | \( 32 \) | \( 64 \) | \( 128 \) |
|-------|------|--------------|-------------|-------|------|------|------|------|-------|------|------|------|------|-------|
| 3     | C     | 15.7  | 28   | 54   | 108  | 210  | 38   | 70   | 138  | 270  | 540  |     |     |       |
|       | B     | 20    | 37   | 76   | 160  | 320  | 55   | 105  | 210  | 440  | 910  |     |     |       |
|       | C     | 17.2  | 30   | 55   | 118  | 230  | 41   | 74   | 143  | 280  | 560  |     |     |       |
| 16    | A     | 29    | 55   | 109  | 210  | 420  | 76   | 142  | 270  | 550  | 1150 |     |     |       |
|       | B     | 35    | 67   | 133  | 270  | 540  | 98   | 188  | 370  | 740  | 1490 |     |     |       |
|       | C     | 33    | 58   | 112  | 210  | 430  | 99   | 168  | 310  | 580  | 1150 |     |     |       |
| 32    | A     | 58    | 111  | 210  | 420  | 850  | 159  | 290  | 580  | 1180 | 2300 |     |     |       |
|       | B     | 65    | 126  | 240  | 480  | 1060 | 185  | 340  | 680  | 1350 | 2900 |     |     |       |
|       | C     | 64    | 127  | 230  | 450  | 880  | 200  | 340  | 610  | 1180 | 2200 |     |     |       |
| 50    | A     | 120   | 220  | 440  | 870  | 1820 | 340  | 610  | 1190 | 2300 | 4700 |     |     |       |
|       | B     | 126   | 230  | 460  | 990  | 2000 | 350  | 650  | 1280 | 2700 | 5500 |     |     |       |
|       | C     | 159   | 270  | 480  | 920  | 1790 | 410  | 690  | 1230 | 2400 | 4600 | 9300 |     |       |
| 64    | A     | 240   | 450  | 900  | 1780 | 3600 | 700  | 1260 | 2300 | 4600 | 9300 |     |     |       |
|       | B     | 240   | 460  | 990  | 1840 | 3900 | 780  | 1280 | 2500 | 5000 | 11000|     |     |       |
|       | C     | 320   | 590  | 970  | 1840 | 3500 | 850  | 1420 | 2400 | 4700 | 9100 |     |     |       |
| 70    | A     | 31    | 57   | 109  | 210  | 420  | 78   | 145  | 280  | 560  | 1110 |     |     |       |
|       | B     | 39    | 78   | 159  | 320  | 660  | 110  | 210  | 440  | 890  | 1790 |     |     |       |
|       | C     | 35    | 60   | 112  | 230  | 480  | 86   | 153  | 300  | 600  | 1160 |     |     |       |
| 128   | A     | 119   | 220  | 430  | 860  | 1720 | 320  | 590  | 1200 | 2400 | 4800 |     |     |       |
|       | B     | 133   | 250  | 490  | 960  | 2000 | 370  | 700  | 1350 | 2600 | 5600 |     |     |       |
|       | C     | 129   | 250  | 470  | 920  | 1820 | 420  | 710  | 1270 | 2400 | 4600 |     |     |       |
| 85    | A     | 240   | 450  | 890  | 1770 | 3500 | 670  | 1260 | 2400 | 4800 | 10000|     |     |       |
|       | B     | 240   | 460  | 900  | 1840 | 3900 | 690  | 1290 | 2500 | 5400 | 10000|     |     |       |
|       | C     | 320   | 540  | 970  | 1850 | 3600 | 860  | 1450 | 2500 | 4800 | 9400 |     |     |       |
| 128   | A     | 490   | 940  | 1800 | 3500 | 7100 | 1460 | 2500 | 4900 | 9500 | 18700|     |     |       |
|       | B     | 470   | 900  | 1800 | 3500 | 7700 | 1330 | 2500 | 5000 | 10600| 21600|     |     |       |
|       | C     | 650   | 1110 | 1940 | 3700 | 7100 | 1730 | 2900 | 5100 | 9700 | 18900|     |     |       |

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3135690, IEEE Access
TABLE 14: Samsung Exynos 5422 Cortex-A15 (ARMv7-A)

| S   | K     | C_in   | C_out  | K     | C_in   | C_out  | K     | C_in   | C_out  | K     | C_in   | C_out  |
|-----|-------|--------|--------|-------|--------|--------|-------|--------|--------|-------|--------|--------|
|     |       | 8      | 16     | 32    | 64     | 128    | 8      | 16     | 32    | 64    | 128    |
| 50  |       |        |        |       |        |        |       |        |        |       |        |        |
| A   | 2.8   | 4.9    | 9.4    | 19.3  | 38     | 6.9    | 12.5  | 24     | 53    | 107   |        |        |
| B   | 3.8   | 7.6    | 17.5   | 41    | 79     | 10.5   | 21    | 49     | 117   | 230   |        |        |
| C   | 3.4   | 5.6    | 10.2   | 20    | 38     | 9.2    | 14.2  | 27     | 50    | 97    |        |        |
| 70  |       |        |        |       |        |        |       |        |        |       |        |        |
| A   | 5.3   | 9.9    | 19.1   | 39    | 91     | 14.6   | 25    | 51     | 109   | 210   |        |        |
| B   | 6.4   | 12.6   | 26     | 55    | 110    | 18.4   | 35    | 74     | 159   | 310   |        |        |
| C   | 6.6   | 10.7   | 20     | 38    | 75     | 21     | 34    | 58     | 106   | 210   |        |        |
| 85  |       |        |        |       |        |        |       |        |        |       |        |        |
| A   | 10.8  | 19.8   | 38     | 82    | 161    | 33     | 59    | 114    | 230   | 460   |        |        |
| B   | 12.5  | 22     | 45     | 87    | 189    | 37     | 65    | 124    | 240   | 520   |        |        |
| C   | 13.2  | 24     | 48     | 80    | 152    | 42     | 76    | 136    | 250   | 470   |        |        |
| 100 |       |        |        |       |        |        |       |        |        |       |        |        |
| A   | 25    | 43     | 86     | 177   | 340    | 70     | 123   | 230    | 470   | 910   |        |        |
| B   | 24    | 42     | 80     | 171   | 320    | 69     | 119   | 220    | 460   | 920   |        |        |
| C   | 36    | 58     | 107    | 186   | 350    | 94     | 152   | 240    | 480   | 900   |        |        |
| 128 |       |        |        |       |        |        |       |        |        |       |        |        |
| A   | 50    | 86     | 171    | 350   | 680    | 156    | 260   | 470    | 930   | 1810  |        |        |
| B   | 47    | 81     | 168    | 310   | 630    | 132    | 230   | 470    | 890   | 1740  |        |        |
| C   | 72    | 113    | 210    | 350   | 710    | 180    | 300   | 500    | 940   | 1700  |        |        |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
Trusov et al.: p-im2col: Simple Yet Efficient Convolution Algorithm with Flexibly Controlled Memory Overhead

[44] M. Tan and Q. V. Le, “Efficientnetv2: Smaller models and faster training,” ArXiv, vol. abs/2104.00298, 2021.

[45] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 779–788.

[46] P. Adarsh, P. Rathi, and M. Kumar, “Yolo v3-tiny: Object detection and recognition using one stage improved model,” in 2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS). IEEE, 2020, pp. 687–694.

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

[48] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, “High-performance low-memory lowering: GEMM-based algorithms for DNN convolution,” in 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, 2020, pp. 99–106.

[49] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, “Anatomy of high-performance deep learning convolutions on simd architectures,” in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018, pp. 830–841.

[50] M. Dukhan, “The indirect convolution algorithm,” arXiv preprint arXiv:1907.02129, 2019.

[51] G. Guennebaud, B. Jacob et al., “Eigen: a c++ linear algebra library,” 2010. [Online]. Available: http://eigen.tuxfamily.org

[52] D. Padua, FFTW. Boston, MA: Springer US, 2011, pp. 671–671. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_397
ANTON V. TRUSOV (M’20) was born in Moscow, Russia in 1997. He obtained his Master’s degree in 2021 from the Moscow Institute of Physics and Technology. Since 2017, Anton has worked as a programmer and technician at Smart Engines Service LLC, and since 2021, he has been a PhD student at the Moscow Institute of Physics and Technology (National Research University). His research interests are in the area of device-efficient neural network implementations.

ELENA E. LIMONOVA (M’20) was born in Dolgoprudny, Moscow region, Russia in 1993. She studied physics, mathematics and computer science at the Moscow Institute of Physics and Technology and obtained her Master’s degree in 2017. Since 2016, Elena has worked as a programmer and technician at Smart Engines Service LLC, and since 2017, she has been a PhD student at FRC CSC RAS. Her research activities are neural network compression and image recognition on mobile devices.

DMITRY P. NIKOLAEV (M’20) was born in Moscow, USSR in 1978. He obtained his Master’s degree in physics in 2000 from Moscow State University, Moscow, Russia and his Ph.D. degree in computer science from Moscow State University, Moscow, Russia, in 2004. Since 2007, he has been a Head of the Vision Systems Laboratory at the Institute for Information Transmission Problems of Russian Academy of Sciences, Moscow, Russia, and since 2016, he has been a CTO of Smart Engines Service LLC, Moscow, Russia. Since 2016, he also has been an Associate Professor at the Moscow Institute of Physics and Technology (National Research University), Moscow, Russia, teaching the “Image Processing and Analysis” course. He is an author of over 220 papers and 6 patents. His research activities are in the area of computer vision with primary application to color image understanding.

VLADIMIR V. AR LazAROV (M’19) was born in Moscow, USSR in 1976. He received a Specialist degree in applied mathematics from the Moscow Institute of Steel and Alloys in 1999 and a Ph.D. degree in computer science in 2005. Since 1999, he has been working at the Institute for Systems Analysis of Russian Academy of Sciences (currently Federal Research Center “Computer Science and Control” of the Russian Academy of Sciences), Moscow, Russia, as a Researcher, Senior Researcher and Head of Laboratory. Since 2016, he has been a General Director of Smart Engines Service LLC, Moscow, Russia. Since 2018, he has also worked at the Institute for Information Transmission Problems of the Russian Academy of Sciences as a Senior Researcher, and since 2012, he has worked at the Moscow Institute of Physics and Technology (National Research University), Moscow, Russia, as an Associate Professor. He has published over 90 papers and authored 7 patents. His primary fields of study are computer vision and document analysis systems.