2022-04-29

Design and Implementation of 2D Convolution on x86/x64 Processors

Kelefouras, Vasileios

10.1109/tpds.2022.3171471
IEEE Transactions on Parallel and Distributed Systems
Institute of Electrical and Electronics Engineers (IEEE)

Design and Implementation of 2D Convolution on x86/x64 Processors

Vasilios Kelefouras, Georgios Keramidas,

Abstract—In this paper, a new method for accelerating the 2D direct Convolution operation on x86/x64 processors is presented. It includes efficient vectorization by using SIMD intrinsics, bit-twiddling optimizations, the optimization of the division operation, multi-threading using OpenMP, register blocking and the shortest possible bit-width value of the intermediate results. The proposed method, which is provided as open-source, is general and can be applied to other processor families too, e.g., Arm.
The proposed method has been evaluated on two different multi-core Intel CPUs, by using twenty different image sizes, 8-bit integer computations and the most commonly used kernel sizes (3x3, 5x5, 7x7, 9x9). It achieves from $2.8\times$ to $40\times$ speedup over the Intel IPP library (OpenCV GaussianBlur and Filter2D routines), from $105\times$ to $400\times$ speedup over the gemm-based convolution method (by using Intel MKL int8 matrix multiplication routine), and from $8.5\times$ to $618\times$ speedup over the vslsConvExec Intel MKL direct convolution routine. The proposed method is superior as it achieves far fewer arithmetical and load/store instructions.

Index Terms—Convolution, Gaussian blur, Code Optimization, Vectorization, AVX, OpenMP, OpenCV, Intel MKL, Intel IPP, High Performance Computing (HPC), image processing

1 Introduction

THE 2-D Convolution is a widely used operation in image and video processing for filtering, smoothing, sharpening, edge detection and differentiation. Furthermore, it is the most computationally intensive building block in Convolutional Neural Networks (CNNs).

Speeding up the convolution operation is a challenging and non-trivial task. This is due to two main reasons. First, the optimization process must take into account multiple parameters: input image dimensions and image bit-depth, kernel size, kernel values, and target hardware architecture. Second, modern compilers fail to provide optimized code, therefore different manually vectorized (e.g., using SSE/AVX intrinsics) and optimized routines are needed, and most importantly these routines must be customized to the above mentioned parameters.

Three main strategies are typically used to speedup the convolution operation: a) optimizing the direct convolution method [1], e.g., Intel IPP and Intel MKL libraries provide such routines, b) using the highly optimized Matrix-Matrix Multiplication (MMM) routines of Intel_MKL or BLAS optimized libraries [2] [3], c) using either the Fast Fourier Transform (FFT) [4] or the Winograd algorithm [5] [6]. According to [7] an efficient direct convolution implementation achieves higher performance than the MMM-based and FFT based convolution methods on CNNs, as the non-direct methods introduce computational and memory overheads. The proposed method uses the direct method.

The aim of this work is to speedup the 2D convolution in the context of image and video processing algorithms. The extension of the proposed method in CNNs is left for future work.

To speedup the convolution operation, CPU vendors provide optimized libraries such as Intel IPP (Integrated Performance Primitives) and Intel MKL that are commercial proprietary libraries working only on Intel processors. OpenCV (Open Source Computer Vision), which is a very popular library of programming functions for real time computer vision applications, uses Intel IPP (specified in the installation phase) to accelerate its routines on Intel CPUs. Its optimized routines use multiple threads and AVX/SSE intrinsics (IPP uses the direct method). Intel MKL provides a set of routines to efficiently execute the direct convolution operation for single and double precision real and complex data. Furthermore, Intel MKL provides highly optimized int8/int16/FP32 MMM routines that are widely used to compute the convolution layer of CNNs.

In this paper, the design and implementation of the 2D convolution operation on x86/x64 processors is delivered, for different kernel sizes, kernel values, vectorization technologies, number of physical CPU cores, image bit-depths and image sizes as well as for separable kernels. The proposed method achieves far fewer arithmetical and Load/Store (L/S) vector instructions than the state of the art libraries/methods as it includes efficient vectorization by using SIMD intrinsics, bit-twiddling optimizations, efficient vector division, register blocking and the shortest possible bit-width value for the intermediate results. In particular, our method offers high SIMD utilization.

Fig. 1 shows the instruction gains over the Intel libraries for an 1024x1024 image. The results are extracted by using the Valgrind tool [8]. The MMM-based method gives the highest number of instructions mainly because of the extra computation/memory overhead of the im2col operation (it is explained in Section 2). The two direct methods achieve fewer instructions as there are no overheads.

The proposed method is evaluated over a) the Intel IPP / OpenCV and in particular GaussianBlur and Filter2D routines [9], b) the MMM-based convolution method, by using Intel MKL 'cblas_gemm_s8u8s32()' int8 MMM routine [10], c) 'vslsConvExec' Intel's MKL direct convolution

Manuscript received Oct 19, 2021; revised XXX

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/TPDS.2022.3171471, IEEE Transactions on Parallel and Distributed Systems

routine (it does not support integer data, thus floating point (FP) input data are used) [11], on two multi-core Intel CPUs, by using a wide range of 8-bit greyscale images and the most commonly used kernel sizes (3x3, 5x5, 7x7, 9x9). Our method achieves high performance and energy consumption gains in all cases.

The main contributions of this paper are: a) a new method for computing the 2D convolution operation on modern CPUs, achieving far fewer arithmetical and L/S instructions compared to the state of the art software commercial libraries, b) a research work providing the theoretical background and source code [1], to efficiently design and implement the 2D convolution operation on different CPUs and for different kernels and images, c) an experimental procedure showcasing that the proposed method achieves high performance gains on two different CPUs.

The remainder of this paper is organized as follows. In Section 2, the related work is reviewed. The proposed method is presented in Section 3, while the experimental results are discussed in Section 4. Finally, Section 5 is dedicated to discussion while Section 6 is dedicated to conclusions and future work.

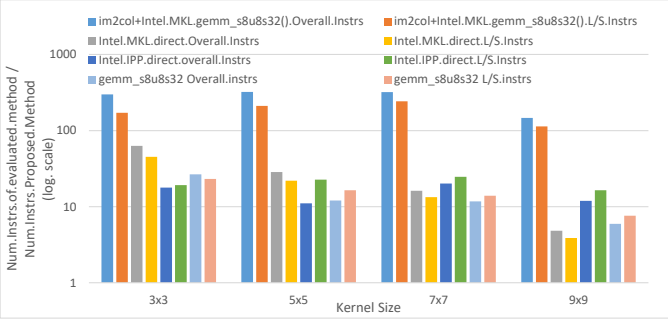

Fig. 1: Instruction gain over Intel MKL/IPP libraries

2 Related Work

The main strategies that have been followed to improve the performance of the convolution operation on CPUs can be classified into three main groups. The proposed method belongs to the first group.

The first group of works uses the direct algorithm. The main optimizations used are vectorization, improving the memory layout to enable vectorization, parallelization (by using OpenMP) and register blocking. Although parallelization can be easily applied by using OpenMP pragmas, the efficient application of vectorization is far more challenging and necessitates the manual development of low-level software; furthermore, if applied efficiently, it gives by far the highest performance gain.

There are three main vectorization strategies for the convolution operation [12] [13] [14] [7]. The first strategy (aka coefficient propagation) broadcasts each kernel's scalar coefficient to a separate vector variable [12] [13] [14] (each vector variable has multiple copies of just one scalar coefficient). In the second strategy, all the scalar coefficients which are located in a kernel's row are copied into a vector variable, multiple times [12] [13]. According to [12], the first method above is faster than the second (when using 16/32-bit instructions), as it achieves fewer arithmetical instructions. The third strategy, which is used in CNNs only, vectorizes the depth or filter loop [1] [15] [7], and data are processed in a way similar to Method2; note that in CNNs the convolution layer includes three more loops (both the input feature map and the weights are tensors) and thus a comparison with our method is hard to be realized. Most of the works use 32-bit FP instructions, some use 16-bit integer instructions [12] [13] [14] [16], but no related work uses 8-bit instructions on x86/x64 processors. 8-bit implementations are far more challenging as first, there is no x86/x64 8-bit vector multiplication instruction, second, there is no vector instruction for integer division, and third, packing/unpacking becomes complicated and extra instructions are required. Thus, none of the current methods can be directly extended to 8-bit (serious modifications are required), e.g., the first vectorization strategy cannot be efficiently implemented in 8-bit because of the first reason above.

To the best of our knowledge, the proposed method achieves fewer L/S and arithmetical instructions compared to all the currently published methods for the following reasons. First, it leverages the existing available 8-bit integer instructions. Second, the proposed method efficiently packs and stores the output pixels in memory; in Subsection 4.1.3, we show that this step boosts performance by about $x2.5$. Third, it uses optimizations that the existing works do not; the number of instructions is further reduced by optimizing the division operation and the bit-width of the intermediate results (in Subsection 4.1.3, we show that these optimizations boost performance by about $x1.6$, each).

In [17], a parallel implementation of Harris corner detection algorithm is proposed for NUMA architectures. In [18], the Harris operator is optimized using a number of optimizations such as vectorization, data interleaving and parallelization, on both x86/x64 and Arm processors. In [19], different ways of vectorizing the 3D convolution are shown. In [20], a white paper for a Gaussian Blur implementation on Intel processors is proposed, using FP computations. In [21], a performance comparison is applied between x86 SSE and Arm NEON intrinsics for four image processing routines, including Gaussian Blur. In [15], authors propose a method to accelerate CNNs by using Arm NEON intrinsics and 16-bit computations. In [22], the 2D convolution is optimized by using AVX512 and OpenMP. In [23], authors study the vectorization process in CNNs, using Matlab code. Last, in [24], an implementation for canny edge detection algorithm is delivered. The proposed method achieves fewer L/S and arithmetical vector instructions than [17-24] for the three reasons explained above.

In [1], the CNN convolution kernels are implemented via a dynamic compilation approach on x86/x64 processors. In [25], [1] is extended to compute the 1D dilated convolution. In [7], authors optimized the convolution layer of CNNs. [1] and [7] cannot be directly applied on 8-bit.

Intel IPP and Intel MKL provide high performance direct methods for computing the convolution operation; these optimized routines have been implemented by using AVX and multiple threads. A comparison with these methods is

1. https://github.com/kelefouras/2D_Convolution

provided in Section 4.

Last, there are methods that apply auto-vectorization by using image processing domain-specific languages (DSL) such as Halide [26] [27].

The second group of works implement the convolution operation by using the highly optimized Matrix-Matrix Multiplication (MMM) routines of Intel_MKL and BLAS optimized libraries [2] [3]. In these methods, convolution is implemented on top of MMM, leveraging the aforementioned highly optimized libraries. This approach is mainly used in CNNs and has been employed in mainstream deep learning frameworks [3].

MMM-based algorithms rely on 'im2col' or 'im2row' memory transformations [2] to convert the Convolution problem into an MMM problem, introducing a non-trivial overhead in memory storage and bandwidth, which is proportional to the kernel size. In the 2D convolution case, the 'im2col' / 'im2row' operation copies the $N \times M$ input image to a new 2D array of size $(N \times M) \times (k \times k)$, where $k \times k$ is the kernel size. This is a slow and memory-bound operation. Furthermore, the MMM operation is applied on the new tall and skinny matrix whose dimensions are dissimilar from the matrices arising from traditional high performance computing (HPC) applications and according to [7] the MMM routines do not achieve their peak performance in this case. Last, the memory size needed is $k \times k$ larger compared to the direct method.

An indirection buffer is introduced in [2] to avoid reshuffling the data. In [3], im2col conversion and MMM packing operations are merged to one, reducing the memory footprint. In [28], an efficient CPU implementation for convolution-pooling in CNNs is presented, by using convolution interchange and vectorization.

To better utilize the CPU's vector extensions in CNNs, quantization is used. In [29], authors optimize CNNs by using extremely low-bit quantization on ARM CPU (from 2 up to 8-bit) for the MMM and Winograd methods. Intel oneAPI Deep Neural Network Library (oneDNN) [30] supports 8-bit quantization and three algorithms (direct, MMM-based and Winograd-based).

Regarding the third group, either the Fast Fourier Transform (FFT) [4] or the Winograd algorithm [5] [6] are used, to reduce the number of FP computations required. The Winograd-based methods are mainly used for small kernel sizes (e.g., 3x3), while the FFT-based methods are mainly used for larger kernel sizes. However, the reduced number of operations does not always align with performance as several challenges make it hard to fully utilize the hardware resources on modern CPUs. The Winograd's based convolution algorithms cannot efficiently utilize the memory hierarchy and the wide vector registers that modern CPUs support [6]. FFT-based approaches suffer from significant performance overheads too [7]. Note that both algorithms suffer from a lack of precision.

In [31], a distributed implementation for the IBM Cell Broadband Engine processor is proposed. [5] extends and optimizes the Winograd-class of convolutional algorithms to the N-dimensional case of CNN on x86/x64 CPUs. In [6], another Winograd implementation is presented for many-core CPUs. In [32], a new class of Winograd algorithms for integer arithmetic is presented.

3 Proposed Method

The inputs to our method are the following: kernel size, kernel values, vectorization technology (e.g., AVX,SSE), number of physical CPU cores, image bit-depth and image dimensions; for different parameters, different code implementations are proposed. The proposed method is broken down into seven Subsections. For ease of presentation, we assume AVX technology (256-bit operations), 8-bit input/output images and a 2D kernel of size 3×3.

3.1 Bit-width selection of the Intermediate Results (IRs)

In 2D convolution, a $k \times k$ kernel 'slides' over the input image, performing $k \times k$ multiplication, $(k \times k - 1)$ addition and 1 division (integer coefficients are assumed here) operations and the result is stored into a single output pixel. A naive implementation is shown on the left of Fig. 2 (for ease of presentation, the border pixels are not computed here).

Most of the image formats use 8-bit pixels and therefore to achieve maximum performance 8-bit processing is needed so as to fully utilize the wide vector instructions. This means that the 'input', 'output' and 'coef' arrays in Fig. 2, should be 8-bit. However, 'tmp' variable is 32-bits.

Given that the range of the output pixel value is always within $[0, 255]$ (uses 8-bits), the range of the '$tmp/divisor$' value in Fig. 2, should always be within $0 \leq tmp/divisor \leq 255$ and as a consequence $0 \leq tmp \leq 255 \times divisor$. This means that 'tmp' needs to be defined as a 32-bit variable for large 'divisor' values only (large 'coef' values give large 'divisor' values); note that large 'coef' and 'divisor' values can be avoided in many cases by scaling down both values. The kernel values affect the optimization process.

If all the kernel values in Fig. 2 ('coef') are positive, then 'tmp' can be defined as an unsigned 16-bit variable ($[0, 65535]$), if $divisor \leq 257$. Otherwise (if $divisor \succ 257$), 'tmp' should be defined as a 32-bit variable. If the kernel contains negative values too, then 'tmp' can defined as a signed 16-bit variable, if $divisor \leq d$, where $128 \leq d \prec 257$ and d depends on the kernel values; in this case, we cannot calculate the maximum divisor value without knowing the exact kernel values.

Conclusion: In most cases, 16-bit IRs bit-width is adequate for kernels of size 3x3 and 5x5; for larger kernel sizes, 32-bit width is normally needed, depending on the coefficients' values. Note that for separable kernels, e.g., Gaussian Blur, 16-bit width is adequate for larger kernel sizes too, as fewer operations are executed in this case; two 1D kernels are used instead of one 2D (one for the X and another for the Y dimension), and thus the number of operations being performed is lower. In Section 4, we show that 16-bit width provides a speedup from $x1.05$ to $x2.0$ over 32-bit width (about $x1.5$ on average).

Hardware Limitations: Accuracy loss is likely when using 8-bit inputs, regardless of the IRs bit width; the reason follows. In x86/x64 processors there is just one 8-bit vector multiplication instruction available (maddubs); maddubs, includes both multiplication and addition of the IRs (Fig. 5). Its first operand is of type unsigned 8-bit integer (used for storing the pixels), while its second operand is of type signed 8-bit integer (used for storing the coefficients); the output contains signed 16-bit integers

and thus ($-32768 \leq i0 \times C00 + i1 \times C01 \leq 32767$). When both maddub's operand values are near to their maximum values, an overflow occurs, which leads to accuracy loss. Given that the maximum value of the input (e.g., i0, i1) is 255, an overflow will never occur if the sum of each coefficient pair (e.g., C00+C01) is smaller or equal to 128.

```
/*------------ Gaussian Blur ------------*/
for (row = 1; row < N-1; row++) {
 for (col = 1; col < M-1; col++) {
  tmp = 0;

  tmp += input[row-1][col-1] * coef[0][0];
  tmp += input[row-1][col+0] * coef[0][1];
  tmp += input[row-1][col+1] * coef[0][2];

  tmp += input[row-0][col-1] * coef[1][0];
  tmp += input[row-0][col+0] * coef[1][1];
  tmp += input[row-0][col+1] * coef[1][2];

  tmp += input[row+1][col-1] * coef[2][0];
  tmp += input[row+1][col+0] * coef[2][1];
  tmp += input[row+1][col+1] * coef[2][2];

  output[row][col] = tmp/divisor;
  }
}
```

```
/*Gaussian Blur with Register blocking*/
/*--- Assume (N-2)%2=0 ---*/
for (row = 1; row < N-1; row+=2) {
 for (col = 1; col < M-1; col++) {
  tmp = 0;
  in10=input[row][col-1];
  in11=input[row][col];
  in12=input[row][col+1];

  in20=input[row+1][col-1];
  in21=input[row+1][col];
  in22=input[row+1][col+1];

  tmp += input[row-1][col-1] * coef[0][0];
  tmp += input[row-1][col+0] * coef[0][1];
  tmp += input[row-1][col+1] * coef[0][2];

  tmp += in10 * coef[1][0];
  tmp += in11 * coef[1][1];
  tmp += in12 * coef[1][2];

  tmp += in20 * coef[2][0];
  tmp += in21 * coef[2][1];
  tmp += in22 * coef[2][2];

  output[row][col] = tmp/divisor;

  tmp = 0;
  tmp += in10 * coef[0][0];
  tmp += in11 * coef[0][1];
  tmp += in12 * coef[0][2];

  tmp += in20 * coef[1][0];
  tmp += in21 * coef[1][1];
  tmp += in22 * coef[1][2];

  tmp += input[row+2][col-1] * coef[2][0];
  tmp += input[row+2][col-0] * coef[2][1];
  tmp += input[row+2][col+1] * coef[2][2];

  output[row+1][col] = tmp/divisor; }
```

Fig. 2: On the left, a naive implementation of the 3×3 Gaussian Blur algorithm is shown. On the right, the application of register blocking optimization is shown

3.2 Vectorization

The optimization of the vectorization process is the key to achieve high performance in convolution operation. The proposed vectorization method minimizes the number of arithmetical instructions as well as the number of store and division operations. It also reduces the number of load operations; to minimize the number of load operations, the optimization in Subsection 3.3. must be also applied.

Fig. 3 shows a high level illustration of the proposed method. For a kernel of size $k \times k$, k vector load operations are needed to load the first working set of the input image (r0,r1,r2 in Fig. 3). Each vector contains $simd.length/pixel.size$ pixel elements, e.g., for AVX technology (256-bit) and 8-bit pixels, each vector contains 32 pixel elements. The first output result is calculated by processing the first k elements (0:k-1 elements) of the k previously loaded vector variables, e.g., 0:2 elements in Fig. 3. The second output result is calculated by processing the elements 1:k, the third 2:k+1, etc. Loading $k \times (simd.length/pixel.size)$ pixels and calculating just one result is very inefficient as first, the hardware vectorization unit is underutilized, and second, more vector instructions are required. The two main objectives of the proposed method are first, to utilize the vector instructions as much as possible (and as a consequence the number of vector instructions is reduced), second, to select the appropriate vector instructions (note that modern CPUs support a rich instruction set with diverse latency and throughput values).

To meet the first objective, multiple and not one output results should be computed together. To this end, we fill the vector coefficients with many copies of the scalar coefficients (Fig. 4); to calculate $out.pixels$ output results, $out.pixels$ copies of the scalar coefficients are required. In the general case, $out.pixels$ copies of the scalar coefficients are needed (Fig. 4), where $out.pixels = \lfloor (\frac{simd.length/pixel.size}{k+1}) \rfloor$, when k is odd. For a kernel of size 3×3, 12 vector coefficients are used (if all the scalar coefficients are different) (Fig. 4); the first four vector coefficients (V_C00-V_C03) are generated from the first row of the kernel, while the other eight are generated from the second and third kernel rows, respectively. By packing as many scalar coefficients as possible into vectors, the overall number of multiplications/additions is reduced. Note that the overall number of vector coefficients is $k \times (k+1)$, which becomes too high for kernels of size 7×7 and larger, and thus a slightly different algorithm is proposed in this case (explained later). The reason that the vector variables in Fig. 4 include a zero after every k elements lies in the fact that the only 8-bit vector multiplication instruction available is 'maddubs' (Fig. 5) and the IRs should not be mixed. Note that our method is more efficient when k is even.

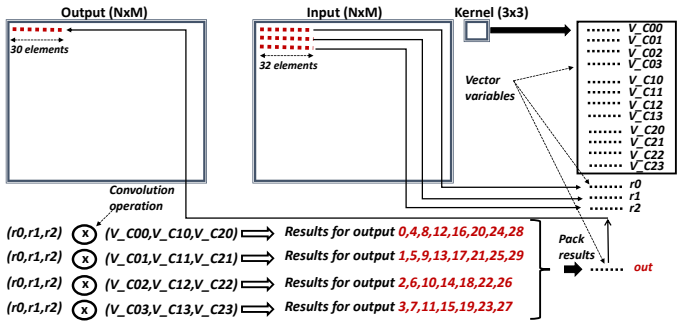

Fig. 3: High level illustration of the proposed vectorization method (Algorithm 1) for a kernel of size 3x3

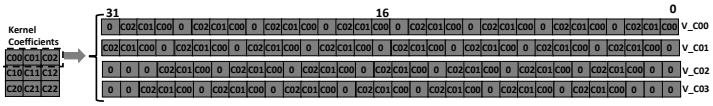

Fig. 4: Vector coefficients for Algorithm 1. The least significant elements are the rightmost.

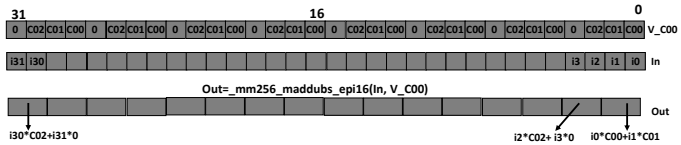

Fig. 5: $\_mm256\_maddubs\_epi16()$ vector multiplication instruction

In the previous paragraph we explained how $out.pixels$ are calculated together (Fig. 3). These output results need to be stored $k+1$ memory locations apart, and as we will be showcasing below, this is not efficient as multiple extract

Algorithm 1 Proposed vectorization algorithm for kernel sizes 5×5 and smaller (16-bit IRs). For presentation purposes the implementation of the 3×3 kernel is shown. All the x86/x64 intrinsics can be found in this link [33].

1: unsigned short int a=65535 //in binary $a = (11111111\ 11111111)_2$
2: $\_\_m256i$ r0,r1,r2,m0,m1,m2,out_even, out_odd
3: $\_\_m256i\ mask1 = set\_epi16(0,a,0,a,0,a,0,a,0,a,0,a,0,a,0,a)$ //odd positions in mask1 are zero
4: $\_\_m256i\ mask2 = set\_epi16(a,0,a,0,a,0,a,0,a,0,a,0,a,0,a,0)$ //even positions in mask2 are zero
5: $\_\_m256i\ V\_C[3][4]$//Although an array of coeffs is shown here, we have implemented this by using 12 variables instead
6: **for** $(row = 1; row < N-1; row++)$ **do**
7: **for** $(col = 30; col <= M-32; col += 30)$ **do**
8:
9: //Load 32 8-bit pixels
10: $r0 = loadu\_si256(\&input[row-1][col-1])$//un-aligned load
11: $r1 = loadu\_si256(\&input[row][col-1])$
12: $r2 = loadu\_si256(\&input[row+1][col-1])$
13:
14: //This loop must be fully unrolled to avoid the if-conditions below
15: **for** $(X = 0; X < 4; X++)$ **do**
16:
17: //Multiply by the mask (16 16-bit results)
18: $m0 = maddubs\_epi16(r0$,V_C[0][X]) // 'maddubs' explained in Fig.4
19: $m1 = maddubs\_epi16(r1$,V_C[1][X])
20: $m2 = maddubs\_epi16(r2$,V_C[2][X])
21:
22: //Vertical addition
23: $m0 = add\_epi16(m0, m1)$ //16 16-bit additions
24: $m0 = add\_epi16(m0, m2)$
25:
26: //Horizontal addition
27: $m1 = srli\_si256(m0, 2)$ //right shift by 2 bytes (each element is 2 bytes)
28: $m0 = add\_epi16(m0, m1)$
29:
30: //Pack the 16-bit results
31: **if** $(X == 0)$ **then**
32: //keep m0 elements in positions 0,2,4,6,8,10,12,14 only
33: $out\_even = and\_si256(m0, mask1)$
34: **else if** $(X == 1)$ **then**
35: //keep m0 elements in positions 0,2,4,6,8,10,12,14 only
36: $out\_odd = and\_si256(m0, mask1)$
37: **else if** $(X == 2)$ **then**
38: //keep m0 elements in positions 1,3,5,7,9,11,13 only
39: $m0 = and\_si256(m0, mask2)$
40: $out\_even = add\_epi16(out\_even, m0)$
41: **else if** $(X == 3)$ **then**
42: //keep m0 elements in positions 1,3,5,7,9,11,13 only
43: $m0 = and\_si256(m0, mask2)$
44: $out\_odd = add\_epi16(out\_odd, m0)$
45: **end if**
46: **end for**
47:
48: //16-bit Vector Division (16-bit inputs, 8-bit output)
49: $out\_even = DIV\_16(division\_case, out\_even, div\_vector)$ //see Alg.5 (out_even / div_vector)
50: $out\_odd = DIV\_16(division\_case, out\_odd, div\_vector)$
51:
52: //Extract and blend the 8-bit final results
53: $out\_odd = slli\_si256(out\_odd, 1)$ //left shift by 1 bytes (each element is 1 byte)
54: $out\_even = add\_epi8(out\_even, out\_odd)$//32 8-bit additions
55:
56: //store the 8-bit final values to memory
57: $storeu\_si256(\&output[row][col], out\_even)$//un-aligned store
58: **end for**
59: //calculate loop reminder
60: **end for**

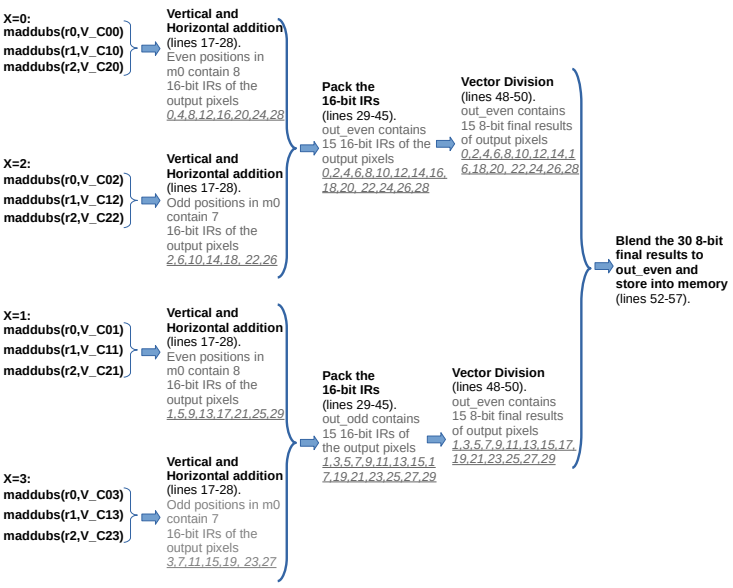

Fig. 6: A higher level illustration of Algorithm 1.

and store instructions are needed. Thus, we repeat the previous process k+1 times to calculate all the output values, and then pack the results into a single vector variable, which is stored into memory by using a single store instruction.

The proposed vectorization algorithm for kernels of size 3x3 is shown in Algorithm 1 (for ease of presentation, the calculation of the border pixels is not shown here). To ease the presentation of Algorithm 1, Fig. 6 is also provided. Firstly, the following coefficients (V_C00, V_C10, V_C20) are multiplied by the input vectors, by using maddubs instruction (lines 18-20 in Algorithm 1). Then, the results are vertically and horizontally added (lines 23-28). The Horizontal addition is needed so as to add all the pairs in the 'out' variable in Fig.5; it is implemented by using 'shift' and 'add' instructions as the easy 'hadd' instruction to use is an expensive operation. Note that for larger kernel sizes, more than one horizontal addition is needed.

The reason that 12, instead of 3 (one for each kernel's row), vector coefficients are used, follows. By using multiple vector coefficient variables (Fig. 4), less load (or no shift) vector operations are required. Instead of loading the input image again and again (e.g., loading elements 0:31, then 1:32, then 2:33 etc) and multiplying the input vectors by a single vector coefficient (e.g.,V_C00), the input is loaded once and it is multiplied by the four pre-computed shifted coefficients (Fig. 4). It is preferable to use multiple coefficient variables rather than loading the input image multiple times because first, the coefficients can be loaded just once and remain into hardware registers and second, less memory address calculations are required. Furthermore, instead of loading the input to a vector and then shifting it multiple times (AVX shift instructions are expensive), the pre-computed shifted coefficients are used instead (Fig. 4). Note that shift is an expensive operation in AVX because the existing 256-bit shift instructions are implemented as two 128-bit shift instructions and therefore the 16th vector position (there are 32 in total) is always filled with zero (two separate 128-bit shift operations occur); therefore additional instructions

are needed to fill that empty position with the appropriate value. However, using multiple vector coefficients is not efficient for large kernel sizes, as their number becomes too high; therefore a slightly different algorithm (Algorithm 2) is proposed in this case (explained later).

When the vertical and horizontal additions are completed (line 28), the 16-bit results, which are located in the even positions of 'm0' (0,2,4,6,8,10,12,14), need to be divided (eight 16-bit elements). However, dividing 'm0' at this point is inefficient as a) division is an expensive operation and only 8 out of 16 elements of 'm0' contain useful data, b) the division results, which are 8-bit, must be stored into positions (col, col+4, col+8, col+12, col+16, col+20, col+24, col+28) and thus multiple store operations are required. So, instead of dividing 'm0' just after line 28, more IRs are calculated (X loop is added to address this issue, line 15) and then packed in an elegant and efficient way (lines 31-46). In Subsection 4.1.3, we show that if multiple store instructions are used here, performance is highly degraded. Although X loop is fully unrolled in our implementations, X is not shown unrolled in Algorithm 1, so as to fit in a single page. In $X = 0$ and $X = 1$, the IRs of the following 8 output pixels are performed (Fig. 6), while in $X = 2$ and $X = 3$, the IRs of the 7 following output pixels are computed:

1: X=0: col,col+4,col+8,col+12,col+16,col+20,col+24,col+28
2: X=1: col+1,col+5,col+9,col+13,col+17,col+21,col+25,col+29
3: X=2: col+2,col+6,col+10,col+14,col+18,col+22,col+26
4: X=3: col+3,col+7,col+11,col+15,col+19,col+23,col+27

The last two output pixels (col+30,col+31) are not computed here as an extra load operation is needed (to compute 32 output pixels, 34 input pixels are needed).

Note that after line 28, either the odd or the even elements of 'm0' must be discarded (depending on the X value), as these elements do not contain any useful data; this is realized by using the 'and' operations in lines 33,36,39,43; two different masks are needed, i.e., 'mask1' and 'mask2', because in $X = 0$ and $X = 1$, the results are located in the even positions of 'm0', while in $X = 2$ and $X = 3$ the results are located in the odd positions of 'm0'. Note that the least significant elements are the rightmost.

As far as the number of division operations is concerned, instead of applying division just after the horizontal addition step in line 28 (which is the easiest thing to do), the results are efficiently packed into out_even and out_odd vector variables, and division is applied outside X loop (lines 49-50). This way, the number of division operations is reduced from four to two (the IRs are first packed and then divided).

The IRs are packed in an elegant and efficient way, and therefore a significant amount of arithmetical vector instructions is saved. In Subsection 4.1.3, we show that this step has a high positive impact on performance. By separating the even and odd results of X loop into separate vectors (out_even and out_odd), makes the process of extracting and blending the output 8-bit values into one vector variable (lines 53,54) efficient, saving lots of arithmetical vector instructions. Last, the X for loop is fully unrolled to avoid using the if-conditions in lines 31-46.

For kernels of size 7×7 or larger, a slightly modified algorithm is used instead (Algorithm 2), because in this case Algorithm 1 gives a very high number of vector coefficients, e.g., 56 vector coefficients are required for the general case of 7×7. When the number of vector variables is higher than the number of the CPU's hardware registers, then the variables cannot remain into the registers and they are loaded/stored many times from/to memory (a.k.a. register spills), degrading performance. Algorithm 2 uses just k coefficients (when the kernel is of size $k \times k$ - one vector coefficient is used for each kernel's row), addressing this problem. Algorithm 2 loads the input more times (lines 12-14) than Algorithm 1 (more load instructions), but it computes more output pixels per iteration (32 instead of 30 in Algorithm 1). Algorithm 2 is always faster than Algorithm 1 for kernels of size 7x7 and larger, as in this case Algorithm 1 gives a very high number of vector coefficients. On the contrary, Algorithm 1 is normally faster for kernels of size 5x5 and smaller.

32-bit IRs: The bit-width of the IRs has a strong impact on vectorization. The most efficient way to make the 16-bit IRs 32-bit, is by using $madd\_epi16$ instruction. In Algorithm 3, we show the implementation of Algorithm 2 for 32-bit IRs. First, maddubs instruction multiplies 8-bit numbers and outputs signed 16-bit integers (lines 16-18 in Algorithm 3). Then, $madd\_epi16$ instruction multiplies the 16-bit output of maddubs by the value of '1' and adds the adjacent results (lines 21-23), generating 32-bit integers. The results are then vertically added and afterwards directly divided as 'm0' contains only useful data now (eight 32-bit values). In the 32-bit case the number of division operations doubles. The implementation of Algorithm 1 using 32-bit IRs is similar.

The aforementioned algorithms can be easily extended to larger kernel sizes, 16-bit computations and different vectorization technologies, e.g., AVX-512, SSE, Arm SVE, Arm NEON (see Section 5).

3.3 Register Blocking

Convolution is a data intensive loop kernel and therefore the number of load/store (L/S) operations performed is critical. So far, the number of load operations is reduced by packing as many scalar coefficients as possible into the vector coefficients and by using multiple coefficient vector variables (Algorithm 1). The number of store operations is minimized by efficiently extracting and blending the IRs. In Algorithm 1, 30 output pixels are calculated and stored by using just 3 load operations and a single store instruction.

To further reduce the number of load operations of Algorithm 1, register blocking optimization is applied. Register blocking consists of loop unroll and scalar replacement optimizations. For ease of presentation, we show the register blocking optimization using scalar code (Fig. 2), but its application on Algorithm 1 is trivial. In Fig. 2, the application of register blocking is shown with a factor of 2. Firstly, loop unrolling is applied to 'row' loop, exposing common array references in the loop body. Then, the common array references are replaced by variables (scalar replacement). On the left of Fig. 2, every time the row iterator is incremented, two of the three input image rows are loaded again, which is not efficient; on the right, the number of load operations is reduced by about $1.5\times$.

Register blocking optimization is applied to Algorithm 1 only; it cannot be applied to Algorithm 2, as different pixel values are loaded in each X iteration. We found experimentally that register blocking can boost performance by

Algorithm 2 Proposed vectorization algorithm for kernel sizes 7×7 and larger (16-bit IRs). For presentation purposes the implementation of the 3×3 kernel is shown. All the x86/x64 intrinsics can be found in this link [33].

unsigned short int a=65535 //in binary $a = (11111111\ 11111111)_2$
__$m256i$ r0,r1,r2,m0,m1,m2,out_even, out_odd
__$m256i\ mask1 = set_epi16(0,a,0,a,0,a,0,a,0,a,0,a,0,a,0,a)$ //odd positions are zero
__$m256i\ V_C00, V_C10, V_C20$ //just three vector coefficients are required

for $(row = 1; row < N-1; row++)$ **do**
  for $(col = 32; col <= M-34; col += 32)$ **do**

    //This loop must be fully unrolled to avoid the if-conditions below
    for $(X = 0; X < 4; X++)$ **do**
      //Load 32 8-bit pixels
      $r0 = loadu\_si256(\&input[row-1][col-1+X])$//unaligned load
      $r1 = loadu\_si256(\&input[row][col-1+X])$
      $r2 = loadu\_si256(\&input[row+1][col-1+X])$

      //Multiply by the mask (16 16-bit results)
      $m0 = maddubs\_epi16(r0,V\_C00)$ // 'maddubs' is explained in Fig.4
      $m1 = maddubs\_epi16(r1,V\_C10)$
      $m2 = maddubs\_epi16(r2,V\_C20)$

      //Vertical addition
      $m0 = add\_epi16(m0, m1)$//16 16-bit additions
      $m0 = add\_epi16(m0, m2)$

      //Horizontal addition
      $m1 = srli\_si256(m0, 2)$ //right shift by 2 bytes (each element is 2 bytes)
      $m0 = add\_epi16(m0, m1)$

      //Pack the 16-bit results
      if $(X == 0)$ **then**
        //keep m0 elements in positions 0,2,4,6,8,10,12,14 only
        $out\_even = and\_si256(m0, mask1)$
      else if $(X == 1)$ **then**
        //keep m0 elements in positions 0,2,4,6,8,10,12,14 only
        $out\_odd = and\_si256(m0, mask1)$
      else if $(X == 2)$ **then**
        //keep m0 elements in positions 0,2,4,6,8,10,12,14 only
        $m0 = and\_si256(m0, mask1)$
        $m0 = slli\_si256(m0, 2)$ //left shift by 2 bytes (each element is 2 bytes)
        $out\_even = add\_epi16(out\_even, m0)$
      else if $(X == 3)$ **then**
        //keep m0 elements in positions 0,2,4,6,8,10,12,14 only
        $m0 = and\_si256(m0, mask1)$
        $m0 = slli\_si256(m0, 2)$ //left shift by 2 bytes (each element is 2 bytes)
        $out\_odd = add\_epi16(out\_odd, m0)$
      end if
    end for

    //16-bit Vector Division (16-bit inputs, 8-bit output)
    $out\_even = DIV\_16(division\_case, out\_even, div\_vector)$ //see Alg.5 (out_even / div_vector)
    $out\_odd = DIV\_16(division\_case, out\_odd, div\_vector)$

    //Extract and blend the 8-bit final results
    $out\_odd = slli\_si256(out\_odd, 1)$ //left shift by 1 bytes (each element is 1 byte)
    $out\_even = add\_epi8(out\_even, out\_odd)$

    //store the 8-bit final values to memory
    $store\_si256(\&output[row][col], out\_even)$//aligned store
  end for
  //calculate loop reminder
end for

Algorithm 3 Implementation of Algorithm 2 for 32-bit IRs

__$m256i$ r0,r1,r2,m0,m1,m2, $ones = set1\_epi16(1)$
__$m256i\ V_C00, V_C10, V_C20$ //vector coefficients
__$m256i\ out[4]$ //For ease of presentation an array is shown here, but four variables must be used instead

for $(row = 1; row < N-1; row++)$ **do**
  for $(col = 32; col <= M-34; col += 32)$ **do**

    //This loop must be fully unrolled and four vector variables are used instead of $out[4]$
    for $(X = 0; X < 4; X++)$ **do**
      //Load 32 8-bit pixels
      $r0 = loadu\_si256(\&input[row-1][col-1+X])$//unaligned load
      $r1 = loadu\_si256(\&input[row][col-1+X])$
      $r2 = loadu\_si256(\&input[row+1][col-1+X])$

      //Multiply by the mask (16 16-bit results)
      $m0 = maddubs\_epi16(r0,V\_C00)$
      $m1 = maddubs\_epi16(r1,V\_C10)$
      $m2 = maddubs\_epi16(r2,V\_C20)$

      //Horizontal addition - 16-bit input, 32-bit output
      $m0 = madd\_epi16(m0, ones)$//Multiplies signed 16-bit integers and horizontally adds the results
      $m1 = madd\_epi16(m1, ones)$
      $m2 = madd\_epi16(m2, ones)$

      //32-bit Vertical addition
      $m0 = add\_epi32(m0, m1)$//8 32-bit additions
      $m0 = add\_epi32(m0, m2)$

      //32-bit Vector DIVISION - m0 now contains 8 32bit values
      $out[X] = DIV\_32(division\_case, m0, div\_vector)$ //see Alg.6 32-bit Division (32-bit inputs, 8-bit output)
    end for

    //Extract and blend the 8-bit final results
    $out[1] = slli\_si256(out[1], 1)$//left shift by 1 byte (each element is 1 byte)
    $out[0] = add\_epi8(out[0], out[1])$
    $out[2] = slli\_si256(out[2], 2)$//left shift by 2 bytes (each element is 1 byte)
    $out[0] = add\_epi8(out[0], out[2])$
    $out[3] = slli\_si256(out[3], 3)$//left shift by 3 bytes (each element is 1 byte)
    $out[0] = add\_epi8(out[0], out[3])$

    //store the 8-bit final values to memory
    $store\_si256(\&output[row][col], out[0])$//aligned store
  end for
  //calculate loop reminder
end for

up to 1.2×. The register blocking factor needs to be found experimentally and it depends on the kernel size, image size and hardware platform. The kernel size affects the blocking factor as the smaller the kernel size is, the fewer the vector coefficients are; if the number of the vector variables is larger than the number of the available hardware vector registers (in x64 there are normally 16 registers), then the reused variables cannot remain into the registers and they are spilled to L1 data cache (aka register spilling). However, it is performance efficient to apply register blocking even when register spills occur as the number of saved load operations is high, normally outweighing the cost of register spilling (this is hardware dependent). The image size also affects the blocking factor; when the width of the input image is very large (e.g., M=5184), then L1 data cache almost fills up (with the input/output image rows) and thus register spilling

causes useful data to be evicted from the cache, degrading performance. Note that more image rows are processed at a time when register blocking is applied.

Algorithm 4 Unsigned scalar division with fixed divisor (x/d)

```
1: x = dividend
2: d = divisor
3: w = word size in bits
4: b = floor(log2(d))
5: f = 2^(w+b)/d
6:
7: if f is an integer then
8:    // DIV Case A
9:    result = x >> b // d is a power of 2
10: else if the fractional part of f is lower than 0.5 then
11:   // DIV Case B
12:   result = ((x + 1) * floor(f)) >> (w + b)
13: else
14:   // DIV Case C
15:   result = (x * ceil(f)) >> (w + b)
16: end if
17: }
```

3.4 Optimizing the division operation

Division is a long-latency operation and its optimization highly impacts the overall performance. Its optimization is broken down into two steps.

Firstly, the division operation is transformed into multiplication and shift operations. The (x/d) operation is transformed into $(x * ceil(f)) >> (w + b)$ [34], where w is the word size in bits (e.g., $w = 16$, if 16-bit division is applied), $b = floor(log_2(d))$ and $f = 2^{(w+b)}/d$. The unsigned scalar division with fixed divisor is shown in Algorithm 4. Note that the divisor is always a constant value; therefore the computationally expensive procedure that calculates the $ceil(f)$ and b values, is applied just once for each input image.

The second step includes the optimization of the multiplication operation in $(x * ceil(f)) >> (w + b)$. Let us assume that 16-bit width is used for the IRs. In this case, the aforementioned multiplication is a 16-bit multiplication and its result needs 32-bits to be stored. Therefore, two vector multiplication instructions are needed ('mullo' and 'mulhi'). 'Mullo' is needed to calculate the low 16-bits, while 'mulhi' is needed to calculate the high 16-bits; then, extra instructions are also needed to pack the results and process the 32-bit results. This procedure, which is computationally expensive, is optimized by using the bit twiddling optimizations shown in Algorithm 5; this way, only one multiplication operation is performed and no packing of the results is needed [34]. All the x86/x64 intrinsics can be found in [33]. The 32-bit division is shown in Algorithm 6.

3.5 Multi-Threading

The 'row' loop in Algorithm 1-Algorithm 3 is parallelized by using the 'for' directive of OpenMP framework. The number of the threads used equals to the number of the physical CPU cores.

Algorithm 5 16-bit vector division with unsigned divisor $(m2/f\_vector)$

```
__m256i DIV_16(const unsigned int division_case, __m256i m2, const
__m256i f_vector) {
//f_vector is a pre-computed vector containing multiple copies of
the scalar value ceil(f)
// b is a pre-computed scalar value (Algorithm 4)
__m256i m1, m3

if (division_case == 1) then
   // DIV Case A
   return (srli_epi16(m2, b)) //m2 >> b
else if (division_case == 2) then
   // DIV Case B
   m2 = add_epi16(m2, set1_epi16(1))
   m3 = mulhi_epu16(m2, f_vector)
   m1 = sub_epi16(m2, m3)
   m1 = srli_epi16(m1, w) //m1 >> w
   m3 = add_epi16(m3, m1)
   return srli_epi16(m3, b) //m3 >> b
else
   // DIV Case C
   m3 = mulhi_epu16(m2, f_vector)
   m1 = sub_epi16(m2, m3)
   m1 = srli_epi16(m1, w) //m1 >> w
   m3 = add_epi16(m3, m1)
   return srli_epi16(m3, b) //m3 >> b
end if
}
```

Algorithm 6 32-bit vector division with unsigned divisor $(m2/f\_vector)$

```
__m256i DIV_32(const unsigned int division_case, __m256i m2, const
__m256i f_vector) {
//f_vector is a pre-computed vector containing multiple copies of
the scalar value ceil(f)
// b is a pre-computed scalar value (Algorithm 4)
__m256i m1, m3

if (division_case == 1) then
   // DIV Case A
   return (srli_epi32(m2, b)) //m2 >> b
else if (division_case == 2) then
   // DIV Case B
   m2 = add_epi32(m2, set1_epi32(1))
   m1 = mul_epu32(m2, f)
   m1 = srli_epi64(m1, w)
   m3 = srli_epi64(m2, w)
   m3 = mul_epu32(m3, f)
   m3 = blend_epi16(m1, m3, 0xCC)
   m1 = sub_epi32(m2, m3)
   m1 = srli_epi32(m1, w) //m1 >> w
   m1 = add_epi32(m3, m1)
   return srli_epi32(m1, b) //m1 >> b
else
   // DIV Case C
   m1 = mul_epu32(m2, f)
   m1 = srli_epi64(m1, w)
   m3 = srli_epi64(m2, w)
   m3 = mul_epu32(m3, f)
   m3 = blend_epi16(m1, m3, 0xCC)
   m1 = sub_epi32(m2, m3)
   m1 = srli_epi32(m1, w) //m1 >> w
   m1 = add_epi32(m3, m1)
   return srli_epi32(m1, b) //m1 >> b
end if
}
```

Algorithm 7 Implementation of Algorithm 2 for a Separable kernel

1: __$m256i$ r0,r1,r2,m0,m1,m2,even,odd,V_Cy0,V_Cy1,V_Cy2,V_Cy0_sh, V_Cy1_sh,V_Cy2_sh,V_Cx
2: unsigned char temp[M] //temp storage for the output of 3 x 1 kernel. M is the image width size
3: //**3 x 1 kernel Coefficients**
4: unsigned char Cy0,Cy1,Cy2 //there are just 3 different scalar coefficients for the 3 x 1
5: $V\_Cy0 = set\_epi8(0, Cy0, 0, Cy0, ..., 0, Cy0, 0, Cy0)$ //multiple copies of Cy0 into even positions
6: $V\_Cy1 = set\_epi8(0, Cy1, 0, Cy1, ..., 0, Cy1, 0, Cy1)$ //multiple copies of Cy1 into even positions
7: $V\_Cy2 = set\_epi8(0, Cy2, 0, Cy2, ..., 0, Cy2, 0, Cy2)$ //multiple copies of Cy2 into even positions
8: $V\_Cy0\_sh = set\_epi8(Cy0, 0, Cy0, 0, ..., Cy0, 0, Cy0, 0)$ //multiple copies of Cy0 into odd positions
9: $V\_Cy1\_sh = set\_epi8(Cy1, 0, Cy1, 0, ..., Cy1, 0, Cy1, 0)$ //multiple copies of Cy1 into odd positions
10: $V\_Cy2\_sh = set\_epi8(Cy2, 0, Cy2, 0, ..., Cy2, 0, Cy2, 0)$ //multiple copies of Cy2 into odd positions
11: //**1 x 3 kernel Coefficients**
12: unsigned char Cx0,Cx1,Cx2//there are just 3 different scalar coefficients for the 1 x 3
13: $V\_Cx = set\_epi8(0, Cx2, Cx1, Cx0, ..., 0, Cx2, Cx1, Cx0)$ //vector coefficient - like the first coefficient in Fig.3
14:
15: **for** $(row = 1; row < N - 1; row++)$ **do**
16: //——- **1D Convolution, 3 x 1 kernel** ———
17: **for** $(col = 0; col <= M - 32; col += 32)$ **do**
18: //Load 32 8-bit pixels
19: $r0 = load\_si256(\&input[row - 1][col])$//aligned load
20: $r1 = load\_si256(\&input[row][col])$
21: $r2 = load\_si256(\&input[row + 1][col])$
22:
23: //Multiply even pixels by the mask (16 16-bit results)
24: $m0 = maddubs\_epi16(r0$,V_Cy0) //explained in Fig.4
25: $m1 = maddubs\_epi16(r1$,V_Cy1)
26: $m2 = maddubs\_epi16(r2$,V_Cy2)
27:
28: //Vertical additions
29: $m0 = add\_epi16(m0, m1)$; $m0 = add\_epi16(m0, m2)$
30:
31: $even = DIV\_16(division\_case, m0, div\_vector)$ //see Alg.5 16-bit Division (16-bit inputs, 8-bit output)
32:
33: //Multiply odd pixels by the mask (16 16-bit results)
34: $m0 = maddubs\_epi16(r0$,V_Cy0_sh) //explained in Fig.4
35: $m1 = maddubs\_epi16(r1$,V_Cy1_sh)
36: $m2 = maddubs\_epi16(r2$,V_Cy2_sh)
37:
38: //Vertical additions
39: $m0 = add\_epi16(m0, m1)$; $m0 = add\_epi16(m0, m2)$
40:
41: $odd = DIV\_16(division\_case, m0, div\_vector)$ //see Alg.5 16-bit Division (16-bit inputs, 8-bit output)
42:
43: //pack even and odd to one register and store
44: $odd = slli\_si256(odd, 1)$//left shift by 1 byte
45: $r0 = add\_epi8(even, odd)$;
46: $store\_si256(\&temp[col], r0)$
47: **end for**
48: //calculate loop reminder
49:
50: //——-**1D Convolution, 1 x 3 Kernel** ———
51: **for** $(col = 32; col <= M - 32; col += 32)$ **do**
52: **for** $(X = 0; X < 4; X++)$ **do**
53: $r0 = loadu\_si256(\&temp[col - 1 + X])$ //Load 32 8-bit pixels
54:
55: $m0 = maddubs\_epi16(r0$,V_Cx) //Multiply by the mask (16 16-bit results)
56:
57: $m1 = srli\_si256(m0, 2)$ //Horizontal addition
58: $m0 = add\_epi16(m0, m1)$
59: //code used in lines 30-58 of Algorithm 2
60: **end for**
61: **end for**
62: //calculate loop reminder
63: **end for**

3.6 Separable kernels

A separable $k \times k$ 2D kernel can be broken down into two 1D kernels, a kernel of size $k \times 1$ and another of size $1 \times k$; this can reduce the number of arithmetical instructions, especially for medium/large kernel sizes (Gaussian Blur is separable). The most efficient way of implementing the separable kernels is by merging the two 1D convolution operations. First, the $k \times 1$ kernel is applied on the first row and its results are stored into a new array ('temp' in Algorithm 7) which is then processed by the $1 \times k$ (Algorithm 7). The procedure is repeated row by row.

The optimization process of the $1 \times k$ kernel is a simplified version of Subsection 3.2. Regarding the $k \times 1$ kernel, there is no 8-bit multiplication vector instruction and thus the multiplication of the even and odd elements is performed separately (lines 24-26 and 34-36).

It has been found experimentally that separable kernels perform better for kernels of size 7×7 or larger (as the gain in arithmetical instructions is high), worse for kernels of size 3×3, and both methods perform almost equally for kernels of size 5×5.

3.7 Image Boundaries

In the aforementioned algorithms, the calculation of the border pixels (first and last columns/rows) is not included, to ease presentation. The first columns are processed by shifting the vector coefficients by one position and filling the gap (first position element) with a special value depending on the border type policy needed, e.g., zero. Regarding the last column pixels, they are processed separately to avoid the cost of array padding. A separate routine is used for the loop reminder that stores the output pixel elements either in shorter vectors or one by one. Note that to achieve best performance, different loop reminder routines are needed for different M values, e.g., if just one column needs to be computed, an optimized scalar version would be faster. Last, the computation of the first and last row output pixels can be simplified as normally the border pixel policy is filling with zeros.

4 Performance Evaluation

The experimental results are performed on two host PCs; an Intel quad-core i7-4790 CPU at 3.60GHz with DDR3 1600MHz (PC1) and an Intel quad-core i5-7500 CPU at 3.40GHz with DDR4 2666MHz (PC2), both running Ubuntu 20.04. Both processors support AVX2 instructions. PC2 achieves higher memory bandwidth than PC1 and this is why all the implementations run faster on PC2, apart from the largest three image sizes where PC1 performs better in some cases; this is because PC1 has a larger L3 cache, i.e. 8MB compared to 6MB of PC2 (this is explained below).

The proposed method is compared to the latest version of OpenCV (version 4.5.1) with Intel IPP, AVX and multithreading enabled as well as to the latest version of Intel MKL library (version 2021.3). Note that the OpenCV routines being evaluated use the Intel IPP library. All the codes are compiled by using gcc 9.3.0 and '-O3' option.

The performance metrics used are speedup and Calculated Pixels per Second (CPS); we have used CPS instead of execution time, as CPS better shows how well the hardware is utilized. The CPS formula is given by $(M \times N \times times.run)/measured.ex.time$, where M is the

image width, N is the image height and $times.run$ is the number of times each code version has run. To get accurate execution time values, each version runs multiple times ($times.run$) so as its execution time is about one minute.

The register blocking factors used are 2 and 3, for the 5x5 and 3x3 case, respectively. Register blocking is not efficient for the largest image size (5184x3456) for the reason explained in the end of Subsection 3.3. Note that register blocking is not applicable to Algorithm 2 (7x7 and 9x9 case).

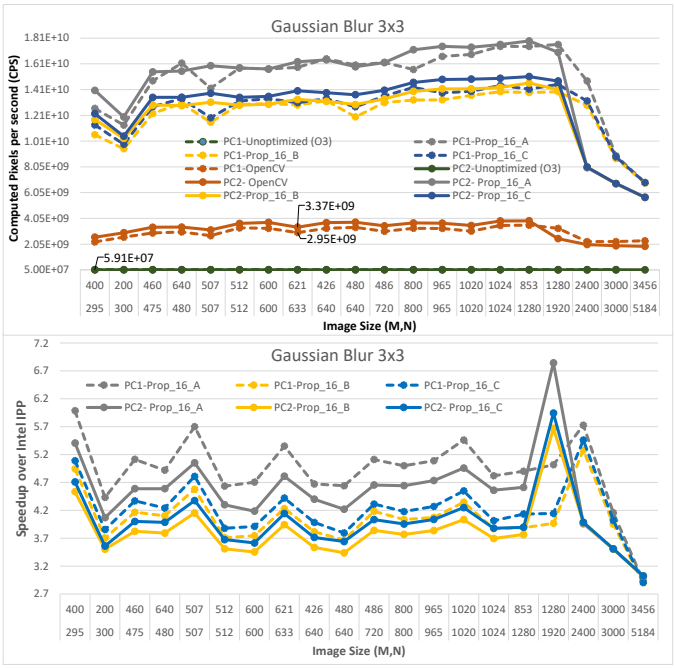

Fig. 7: Evaluation over Intel IPP / OpenCV on 3×3 Gaussian Blur. M is width and refers to the bottom value, while N is height and refers to the top value

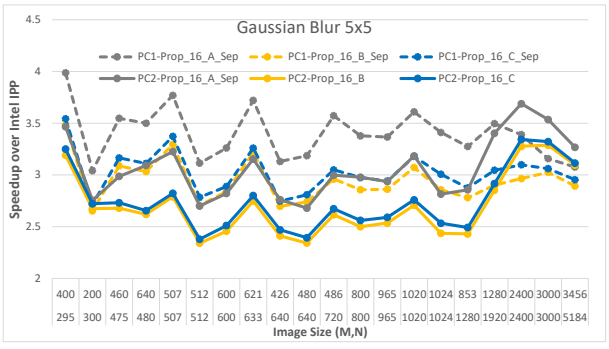

Fig. 8: Evaluation over Intel IPP / OpenCV on 5×5 Gaussian Blur.

4.1 Evaluation over Intel IPP / OpenCV

The proposed method is evaluated using two different convolution loop kernels for image smoothing. The first one is the Gaussian Blur (OpenCV GaussianBlur routine [9]); the Gaussian Blur is separable and its coefficient values are symmetrical, e.g., in a 3×3 kernel the coefficients of the 1st and 3rd row are identical. The second loop kernel is non-symmetrical and non-separable (OpenCV filter2D routine

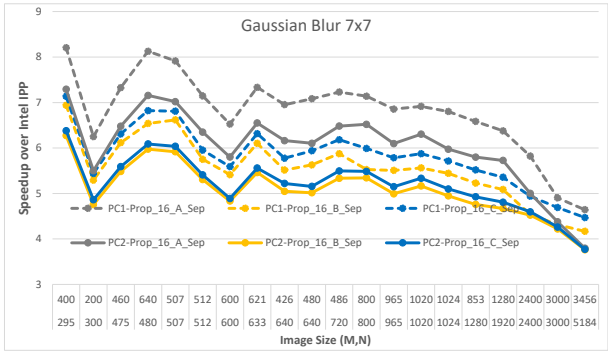

Fig. 9: Evaluation over Intel IPP / OpenCV on 7×7 Gaussian Blur

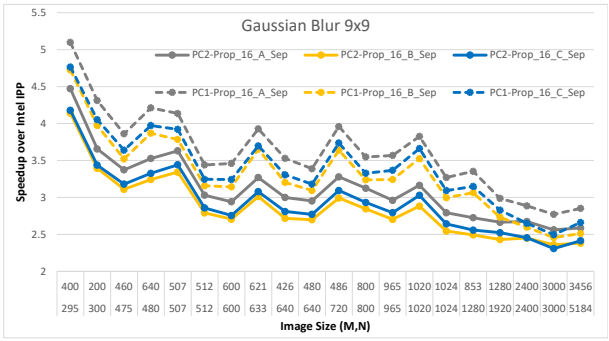

Fig. 10: Evaluation over Intel IPP / OpenCV on 9×9 Gaussian Blur

[9]) and all its coefficients are different. Four different kernel sizes have been used for each case, i.e., 3x3, 5x5, 7x7 and 9x9, and a wide range of greyscale 8-bit image sizes. Algorithm 1 is used for the 3x3 and 5x5 cases, while Algorithm 2 for the 7x7 and 9x9.

4.1.1 Gaussian Blur

Fig. 7- 10 show the evaluation of the proposed method for the Gaussian Blur. To improve readability, the results on 'PC1' and 'PC2' are shown using the same colours; 'PC2' uses a continuous line while 'PC1' a discontinuous line. The 'Unoptimized (O3)' line refers to a naive single threaded implementation. Although OpenMP gives average speedup of $3.4\times$ here, we did not use OpenMP in the unoptimized version as we would like to evaluate the code that a non-expert would use. One implementation is used for all the kernel sizes. '16/32' refers to the bit-width used for the IRs, 'A/B/C' refers to the division case (see Algorithm 5/6) while 'sep' refers to separable kernels (in this case Algorithm 7 has been used). The Gaussian Blur is separable and thus Algorithm 7 can be used to reduce the number of arithmetical instructions. Algorithm 7 performs better than Algorithm 1 for the 7x7 and 9x9 kernels, but Algorithm 1 performs better for the 3x3 case as in this case there is no gain in arithmetical instructions. For the 5x5 case, both algorithms provide roughly the same performance.

Regarding the 3x3 case (Fig. 7), all the implementations achieve their highest CPS values, as the 3x3 case includes fewer calculations than the 5x5,7x7 and 9x9 cases. The unoptimized single-threaded code ('O3' option is used) achieves

about 69 and 59 Mega CPS (MCPS) and is about 233 and 272 times slower than the proposed method, on 'PC1' and 'PC2', respectively. IPP achieves from 2.22 to 3.52 Giga CPS (GCPS) and from 1.88 to 3.86 GCPS, on 'PC1' and 'PC2', respectively, delivering high speedup values over the unoptimized method (Fig. 7). Note that IPP uses separable kernels for the GaussianBlur routine.

Gcc cannot auto-vectorize the code because of the if-condition in the loop body; an if-condition is required to process the border pixels differently. Note that if we process the border pixels separately (in another loop kernel), then no if-condition is needed and gcc is capable of auto-vectorizing the code; still, the speedup achieved is quite low (around 1.8×). The 'Ofast' option does not provide any further performance gain here. The divisor value of the unoptimized code is always a power of two and therefore gcc replaces the division operation with a shift operation.

The performance of the proposed method is shown for different divisor values (Fig. 7). As it was explained in Subsection 3.4, there are three different division cases (Algorithm 5/Algorithm 6). If the divisor is a power of 2 (case A), then the division is realized by using just a single shift instruction, while in case B and case C, 7/11 and 5/9 instructions are executed, for the 16/32 case, respectively. This is why case A is faster than case C and case C is faster than case B, at all times.

Regarding the three largest input sizes, there is a performance drop for both IPP and the proposed method, because the input/output images are larger than L3 cache. Performance is degraded more on 'PC2' than on 'PC1' because 'PC2' has a smaller L3 size (6MB compared to 8MB). The image size 2400x2400 is the most extreme case where PC1 is 1.8× faster than PC2; in this case, the input image fits into the PC1's L3 cache, but it cannot remain into the PC2's L3 cache. The drop is much higher for the 3x3 case compared to the other kernel sizes because the arithmetic intensity (the ratio of arithmetical instructions and bytes loaded/stored) of the 3x3 case is lower than the others, which means that the 3x3 case is more memory-bound; for large input sizes, the overall performance is bounded by the DDR bandwidth. The proposed method achieves high speedup values in all cases.

Both the proposed method and IPP, cannot achieve their best performance for very small images, as the scalability is lower. The number of threads used is four at all cases. Note that performance is degraded by using either more or fewer threads, even in the smallest studied image sizes.

Fig. 8 shows the evaluation of the proposed method for the kernel size of 5x5. We have not included a figure showing the CPS because of the limited page size. As it was expected, all methods are slower in the 5x5 case (compared to the 3x3 case), as the overall number of instructions is higher. IPP achieves from 1.34 to 1.59 and from 1.79 to 2.76, Giga CPS (GCPS), on PC1 and PC2, respectively. The proposed method achieves from 5.61 to 7.74 and from 5.62 to 7.97 GCPS, on PC1 and PC2, respectively. The proposed method achieves high speedup values in all cases.

In the 5x5 case (Fig. 8), some of the proposed implementations include a separable kernel (Algorithm 7) and some not (Algorithm 1). However, both methods provide roughly the same performance here. It is important to note that the separable case (Algorithm 7) executes more division operations and therefore its performance is degraded more in the 'B/C' division cases.

Regarding the 7x7 case (Fig. 9), our method achieves more CPS than the 5x5 case. It achieves from 6.0 to 9.1 and from 6.0 to 9.2 GCPS, on PC1 and PC2, respectively. Although the 5x5 case theoretically requires fewer arithmetical instructions than the 7x7 case, in practice it executes more, as the process of the last horizontal addition (line 26 in Algorithm 2) becomes complicated. In the 3x3 and 7x7 cases, sets of 4 and 8 scalar coefficients (including the zeros) are used to fill the vector coefficients (Fig.3), and thus each set of scalar coefficients is located either on the lower or on the upper half of the AVX registers. However, in the 5x5 and 9x9 case a set always uses both AVX parts, and this makes the horizontal-add operation more complicated, requiring extra vector instructions. This is because in AVX, the existing 256-bit shift instructions are implemented as two 128-bit shift instructions and therefore the 16th vector position is always filled with zero; therefore additional instructions are needed to fill that empty position with the appropriate value. This is the reason that the 5x5 and 9x9 cases achieve lower speedup values over IPP, compared to the 3x3 and 7x7 cases.

For the 9x9 case, the proposed method achieves from 2.72 to 3.25 and from 2.72 to 3.38 GCPS, on PC1 and PC2, respectively.

4.1.2 Filter2D

Fig. 11- 12 show the evaluation of the proposed method for the non-symmetrical convolution kernel (Filter2D). The kernels are not separable in this case and therefore 32-bit IRs are needed for the 7x7 and 9x9 case. We have not included a figure showing the CPS because of the limited page size; this is why we also show the speedup values for the 5x5,7x7 and 9x9 cases in a single figure.

We were surprised when we found out that Filter2D routine is several times slower than the GaussianBlur routine, e.g., about x10 and x15 times slower for the 3x3 and 5x5 case, respectively. The main reasons follow. Firstly, Filter2D routine is single-threaded; we are not aware of the reason that Filter2D routine is not parallelized. Secondly, the GaussianBlur routine uses separable kernels.

The proposed method achieves speedup from $16\times$ to $40\times$ over Filter2D routine. The performance of the proposed method is lower for small M values and large images, for the reason explained in Subsection 4.1.1.

The proposed method achieves fewer CPS for the non-symmetrical Filter2D kernel compared to the Gaussian Blur since a) all the scalar coefficients are different now and as a consequence the number of vector coefficients required by the proposed method is higher, b) Algorithm 7 is not applicable, c) 32-bit IRs are needed for the 7x7 and 9x9 case. Regarding the 3x3 case, the number of vector coefficients has been increased from 8 to 12 and this why performance is degraded by about $1.76\times$ and $1.4\times$, on PC1 and PC2, respectively. Note that the arithmetic intensity is even lower for non-symmetrical kernels. In the 5x5 case, the number of coefficients has been increased from 18 to 30; by using more coefficients, a larger number of register spills occurs, and for this reason the proposed method achieves better performance in the Gaussian Blur case (about $1.46\times$ on

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/TPDS.2022.3171471, IEEE Transactions on Parallel and Distributed Systems

PC1 and 1.25× on PC2). Regarding the 7x7 and 9x9 cases, Algorithm 3 is used instead of Algorithm 7; our method performs about 3× times faster on the GaussianBlur case because of the (b)-(c) reasons above.

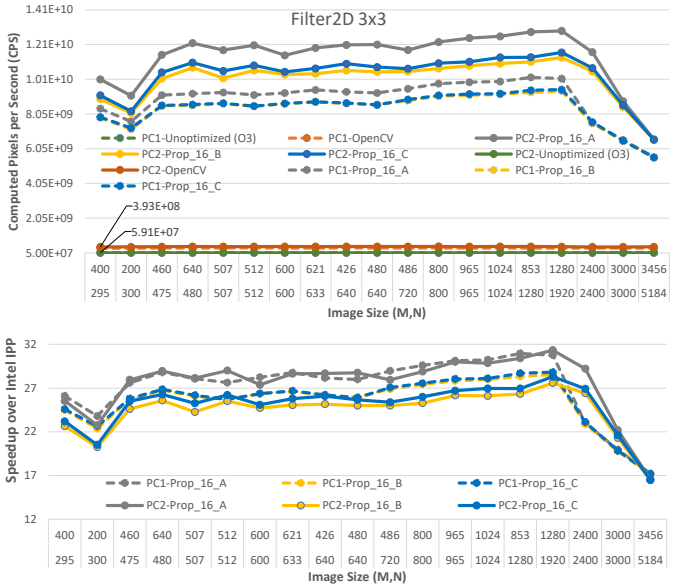

Fig. 11: Evaluation over Intel IPP / OpenCV on 3×3 Filter2D

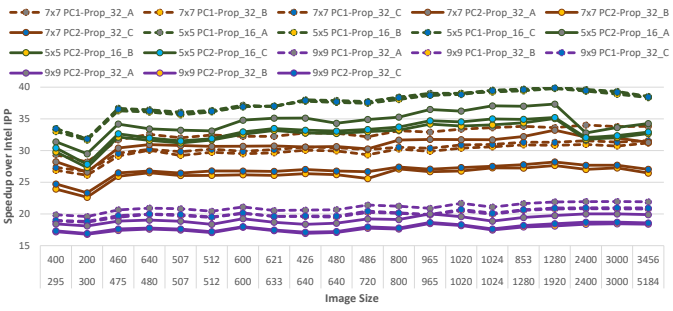

Fig. 12: Evaluation over Intel IPP / OpenCV for Filter2D routine and kernels of size 5x5,7x7,9x9

4.1.3 Performance Breakdown

In this Subsection, we provide a performance analysis of different optimizations of the proposed method on PC1. First, a performance evaluation of different IRs bit-width values is performed (Fig. 13); the first bar always refers to the 'A' case, the second to the 'B', while the third to the 'C' case. As it was expected, the implementation using 32-bit IRs is always slower than the one using 16-bit, as extra arithmetical instructions are being executed. Performance degradation is lower when the divisor is a power of 2 (case A), as in this case both 16-bit and 32-bit division is realized via a single shift instruction. However, when the actual division is computed (case B/C), an extra overhead occurs for the 32-bit case. This is because the 32-bit division (Algorithm 6) uses more arithmetical instructions compared to the 16-bit division (shown in Algorithm 5). For large input images and 3x3 kernel sizes, the performance drop is insignificant, as the code becomes memory-bound and the execution time highly depends on the time needed to load/store the data.

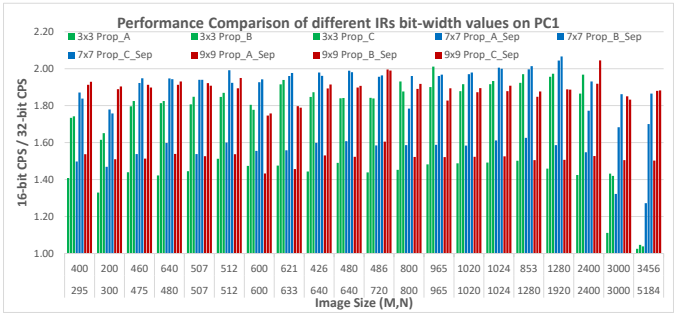

Fig. 13: Performance evaluation of different IRs values.

Second, a performance evaluation of the optimizations shown in Subsection 3.4, Subsection 3.2, and Subsection 3.1 is made, for kernels of size 3x3 (Fig. 14).

To evaluate the performance gain of Subsection 3.4, we have replaced the optimized division operation with a routine that uses the '$\_mm256\_div\_ps()$' instruction. We did that because this is the only division instruction supported in x86/x64. '$\_mm256\_div\_ps()$' divides FP values and thus, the 16-bit integer intermediate results are converted to FP to perform the division and then they are converted to integer values again. The division operation described in Subsection 3.4 gives a speedup of about $x1.6$. For large input images the code becomes memory-bound and this is why there is no performance gain; however, this is not true for larger kernel sizes where performance gain occurs even for large images.

Next, we have evaluated the performance of the last step of Algorithm 1 (lines 31-57) (see 'L/S not optimized' line in Fig. 14). When the vertical and horizontal additions are completed (line 28), the results ($m0$), need to be first divided and then stored to memory into positions (col, col+4, col+8, col+12, col+16, col+20, col+24, col+28); the latter includes multiple store instructions as the elements are not stored into consecutive memory locations (this is further explained in Subsection 3.2). To optimize this process, in Algorithm 1, more IRs are calculated and then packed in an elegant and efficient way (lines 31-46); this allows for a single store operation as well as to less division operations.

To evaluate the impact of lines 31-57 in Algorithm 1, instead of executing the code in lines 31-57, we divide $m0$ just after line 28 and then its useful data (e.g., data in positions (col, col+4, col+8, col+12, col+16, col+20, col+24, col+28)) are extracted and stored into memory. This results into multiple extract/store operations which degrade performance by a factor of about $x2.5$. Note that dividing $m0$ just after line 28, also results into extra division operations, but this does not really affect the results here as we have chosen Case A only, where division is realized by using a single shift instruction.

4.2 Evaluation over Intel MKL library

The proposed method is evaluated over Intel MKL library on PC2, by using two different convolution methods, an Intel MKL MMM based method and an Intel MKL direct method (Fig. 15, Fig. 16). For a fair comparison, non-symmetrical kernels are used here. Note that in the case

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/TPDS.2022.3171471, IEEE Transactions on Parallel and Distributed Systems

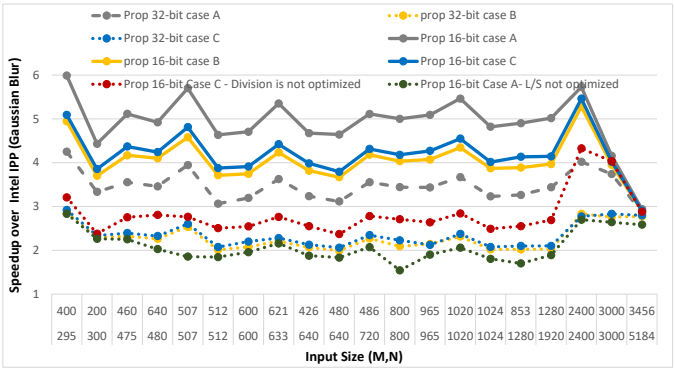

Fig. 14: Performance analysis of the proposed optimizations

where symmetrical kernels are used, the proposed method is significantly faster, especially for the 7x7 and 9x9 cases.

Firstly, the proposed method is evaluated over the MMM-based convolution method. This process consists of two steps. First, the input array (of size $N \times M$) is copied into another bigger 2d array of size $((N \times M) \times (k \times k))$, a.k.a. im2col operation ($k \times k$ is the size of the kernel). Second, the 'cblas_gemm_s8u8s32()' Intel MKL routine [10] multiplies the new array (2d array of size $(N \times M) \times (k \times k)$) by the kernel (1d array of size $(k \times k)$). 'cblas_gemm_s8u8s32()' routine takes as input 8-bit integer values and returns 32-bit integer values; this is a highly optimized routine which includes AVX intrinsics and multiple threads. The first routine (im2col) has been manually developed and is not optimized. Although, the convolution operation cannot be realized without the 'im2col' step, we show the performance of the MMM-based convolution with and without 'im2col' step in order to show that the proposed method achieves high speedup values even without running the 'im2col' step. The MMM-based method does not perform division and therefore just one speedup line is shown for each case.

The proposed method provides high speedup values over the MMM-based convolution method in all cases (Fig. 15); it achieves speedup values from $105\times$ to $400\times$, from $261\times$ to $372\times$, from $298\times$ to $375\times$ and from $160\times$ to $181\times$, for kernels of sizes 3x3, 5x5, 7x7 and 9x9, respectively. The proposed method achieves a lower speedup value for the 9x9 case, because a) the process of the horizontal additions (Algorithm 2) requires more arithmetical instructions in this case, b) fewer pixels are calculated per iteration. Note that the performance bottleneck of the MMM-based convolution is the 'im2col' routine, which is not optimized. Although, the performance of the 'im2col' routine can be improved (e.g., by parallelizing it), the proposed method provides superior performance even when this routine is not used at all (Fig. 15). The proposed method achieves speedup values from $9.7\times$ to $35\times$, from $10.0\times$ to $14.8\times$, from $6.2\times$ to $14.7\times$ and from $2.1\times$ to $6.3\times$, over 'cblas_gemm_s8u8s32()' routine when the 'im2col' routine is not used, for kernels of sizes 3x3, 5x5, 7x7 and 9x9, respectively (Fig. 15).

The proposed method achieves high performance gains for three main reasons. First, it uses the direct method and thus there are no computational/memory overheads ('im2col'). Second, 'cblas_gemm_s8u8s32()' routine cannot achieve its peak performance for tall and skinny matrices [7]. Third and most importantly, the proposed method achieves far fewer L/S and arithmetical instructions than 'cblas_gemm_s8u8s32()' routine (and less memory); this is because a) 'cblas_gemm_s8u8s32()' routine operates on an array which is $k \times k$ times larger than the input image, b) the proposed method better utilizes the AVX instructions and computes multiple output pixels in each iteration.

The second Intel MKL routine that we evaluated our work is 'vslsConvExec()' [11] (Fig. 16). Intel MKL provides a set of routines to perform linear convolution for single and double precision real and complex data. Given that there is no convolution routine for integer numbers, we have used the 'vslsConvExec()' routine, which uses numbers of type 'float' (2d convolution is applied). 'vslsConvExec()' is a highly optimized routine which supports AVX intrinsics and multiple threads. The experimental results show that the 'vslsConvExec()' direct convolution method achieves high performance gains over the MMM-based, in all cases (Fig. 15/Fig. 16). It is important to note that the execution time of the 'vslsConvExec()' routine is not affected by the kernel size but it is stronly affected by the input size; it achieves from 23 MCPS to 131 MCPS. The proposed method achieves speedup values from $98\times$ to $618\times$, from $50\times$ to $291\times$, from $23\times$ to $146\times$ and from $8.5\times$ to $47\times$, for kernels of sizes 3x3, 5x5, 7x7 and 9x9, respectively.

The proposed method achieves high performance gains for the following reasons: a) it uses int8 instead of FP input data; int8 can give a performance gain up to $4\times$, compared to FP, b) it achieves fewer L/S and arithmetical instructions as it better utilizes the AVX instructions and computes multiple output pixels in each iteration.

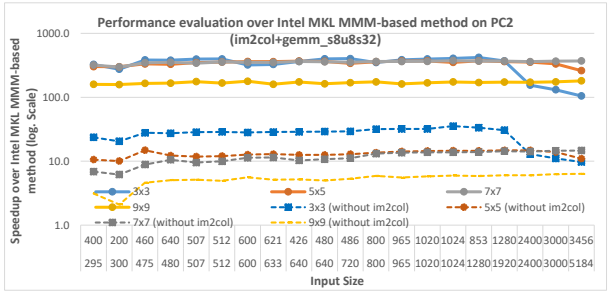

Fig. 15: Evaluation over Intel_MKL MMM-based method

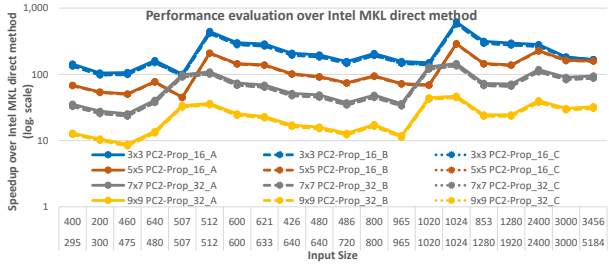

Fig. 16: Evaluation over Intel_MKL direct method

4.3 Energy Consumption

The proposed method is also evaluated over Intel IPP and Intel MKL in terms of power and energy consump-

tion (Fig. 17), on PC2 (a power meter is used). The average power/energy consumption and execution time reduction values are calculated by using the 20 studied image sizes and the following formula (value1-value2)/value2 x 100%. The UnOpt.(O3), Filter2D and im2col implementations consume on average 32, 35 and 40.5 Watts, respectively. $gemm\_s8u8s32$ and vslsConvExec consume on average 56.6 and 57.2 Watts, respectively. The latter two implementations run on all the four cores and this is why power consumption is higher. The Gaussian.Blur routine (symmetrical case), which is the second fastest routine after the proposed, consumes more power (on average 73.8 Watts) as it better utilizes the processing capacity of the processor. The proposed method consumes on average 74.8 and 70 Watts, for the non-symmetrical and symmetrical case, respectively. For symmetrical kernels, the proposed method consumes less power than the non-symmetrical, as there are fewer load instructions and fewer memory accesses in memory hierarchy. The proposed method achieves high energy gains in all cases.

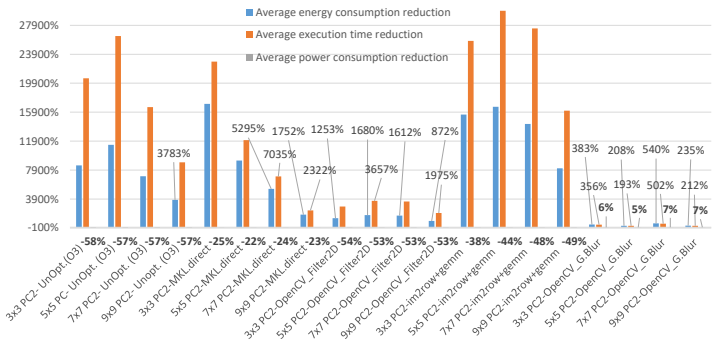

Fig. 17: Average energy and power consumption reduction

5 Discussion

Different kernel sizes and kernel values: The proposed vectorization method is more efficient for even kernel sizes, as no zeros are inserted in this case and thus more pixels are calculated by using the same number of instructions.

In the general case, the kernel values cannot fit into 8-bit variables for kernels of size 11×11 or larger, and in this case 16-bit computations are needed. This is not true for the separable kernels (e.g., Gaussian Blur) as their kernel values are smaller and thus 8-bit width is adequate for large kernel sizes too. Extending Algorithm 1 and Algorithm 2 by using 16-bit computations is straightforward. Although we have evaluated the proposed method by using 8-bit kernel values only, we expect the 16-bit case to be up to two times slower, and therefore significant performance gains are expected in the 16-bit case too. Note that the Intel IPP and the MMM-based methods are expected to run slower in this case too, e.g., 'cblas_gemm_s16s16s32()' is needed instead of 'cblas_gemm_s8u8s32()'.

The kernel size and values also affect the bit-width of the intermediate results; 16-bit width is adequate for the 3×3 and 5×5 case, while for larger kernel sizes 32-bit width is normally needed (see Subsection 3.1). For separable kernels, 16-bit width is adequate for larger kernel sizes too. The proposed method provides impressive speedup values in both cases.

Furthermore, for kernel sizes larger or equal to 17x17/9x9 when 8-bit/16-bit inputs are used, the vector coefficients in Fig. 4 can hold just one set of scalar coefficients. In this case just one output pixel is computed in each iteration and thus Algorithm 1 and 2 are simplified.

CNN Convolution Layers: The proposed method can be extended to compute the CNN convolution layer. However, CNNs introduce extra parameters that need to be considered in the optimization process. First, CNNs include different stride and dilation values. The application of Algorithm 1/Algorithm 2 is straightforward when stride=1 and dilation=1, but for different stride and dilation values, the proposed algorithms need to be appropriately extended. Furthermore, the number of instructions required to calculate an output value is way higher in CNNs, and this affects the vectorization process. This also affects the bit-width selection process in Subsection 3.1; we expect that 16-bit IRs would not be adequate in most cases, unless the quantization level is high. The CNN layer introduces three additional loops, where vectorization and register blocking can be applied; in CNNS, the depth or filter loop is normally vectorized [1] [15] [7]. Another difference relates to the size of the kernel. In CNNs, there are many kernels and not one, and each kernel requires more memory. Given that multiple kernels exist, the following question arises, should we traverse each kernel on the input array separately, or traverse multiple kernels together?

Different Processors: The proposed algorithms can be easily extended to different vector lengths (e.g., AVX-512) and vectorization technologies (e.g., Arm NEON, Arm SVE); however, different manually vectorized routines are needed for each case. The proposed method is also expected to scale well to processors with many CPU cores, as the studied algorithm is data parallel.

6 Conclusions and Future Work

In this paper, a fast method for computing the direct 2D convolution operation on x86/x64 processors is delivered. We provide important insight on how to design and implement the 2D convolution for different kernel sizes, kernel values, vectorization technologies, number of physical CPU cores, image bit-depths and image sizes. The optimized routines, which are provided as open-source, include efficient vectorization using SIMD intrinsics, bit-twiddling optimizations, the optimization of the division operation, multi-threading using OpenMP and others.

The proposed method has been evaluated by using twenty different size images, both 16-bit and 32-bit intermediate results, four different kernel sizes (3x3,5x5,7x7,9x9) and two different processors, and it provides high speedup values over Intel IPP and Intel MKL in all cases. To the best of our knowledge, the proposed method is superior as it achieves far fewer arithmetical and L/S instructions.

As far as our future work is concerned, the first step includes the evaluation of the proposed method to larger kernel sizes. In the longer term, we are planning to apply and evaluate the proposed method to the convolution layer of CNNs. Although the proposed method can be extended to multiple channels and different dilation/stride values

(with a reasonable effort), we expect that several changes might be required in order to achieve peak performance.

REFERENCES

[1] 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 *SC'18 IEEE International Conference*, 2018.

[2] M. Dukhan, "The indirect convolution algorithm," *CoRR*, vol. abs/1907.02129, 2019.

[3] Q. Wang, S. Mei, J. Liu, and C. Gong, "Parallel convolution algorithm using implicit matrix multiplication on multi-core cpus," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–7.

[4] Q. Wang, D. Li, X. Huang, S. Shen, S. Mei, and J. Liu, "Optimizing fft-based convolution on armv8 multi-core cpus," in *Euro-Par*, 2020.

[5] D. M. Budden, A. Matveev, S. Santurkar, S. R. Chaudhuri, and N. Shavit, "Deep tensor convolution on multicores," *CoRR*, vol. abs/1611.06565, 2016.

[6] Z. Jia, A. Zlateski, F. Durand, and K. Li, "Optimizing n-dimensional, winograd-based convolution for manycore cpus," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18, 2018, p. 109–123.

[7] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," *CoRR*, vol. abs/1809.10170, 2018.

[8] N. Nethercote, R. Walsh, and J. Fitzhardinge, "Building workload characterization tools with valgrind." in *IISWC*, 2006, p. 2.

[9] O. S. C. Vision), "Image filtering functions," May 2021. [Online]. [Online]. Available: https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html

[10] Intel, "Developer reference for intel oneapi math kernel library - c," Aug 2021. [Online]. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/blas-and-sparse-blas-routines/blas-like-extensions/cblas-gemm-1.html

[11] ——, "Developer reference for intel oneapi math kernel library - c," Aug 2021. [Online]. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/statistical-functions/convolution-and-correlation.html

[12] H. Amiri and A. Shahbahrami, "High performance implementation of 2d convolution using intel's advanced vector extensions," in *Artificial Intelligence and Signal Processing Conference (AISP)*, 2017, pp. 25–30.

[13] M. Moradifar and A. Shahbahrami, "Performance improvement of gaussian filter using simd technology," in *International Conference on Machine Vision and Image Processing*, 2020, pp. 1–6.

[14] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency and flexibility in specialized computing," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 24–35, Jun. 2013.

[15] S.-J. Lee, S.-S. Park, and K.-S. Chung, "Efficient simd implementation for accelerating convolutional neural network," in *Proceedings of the 4th International Conference on Communication and Information Processing*, ser. ICCIP '18, New York, NY, USA, 2018, p. 174–179.

[16] A. Frickenstein, M. R. Vemparala, C. Unger, F. Ayar, and W. Stechele, "Dsc: Dense-sparse convolution for vectorized inference of convolutional neural networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2019, pp. 1353–1360.

[17] O. Haggui, C. Tadonki, L. Lacassagne, F. Sayadi, and B. Ouni, "Harris corner detection on a numa manycore," *Future Generation Computer Systems*, vol. 88, pp. 442–452, 2018.

[18] L. Lacassagne, D. Etiemble, A. Hassan Zahraee, A. Dominguez, and P. Vezolle, "High level transforms for simd and low-level computer vision algorithms," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '14, New York, NY, USA, 2014, p. 49–56.

[19] Y. Maeda, N. Fukushima, and H. Matsuo, "Taxonomy of vectorization patterns of programming for fir image filters using kernel subsampling and new one," *Applied Sciences*, vol. 8, p. 1235, 2018.

[20] B. Bharti, "Iir gaussian blur filter implementation using intel advanced vector extensions," Intel, Tech. Rep., 2015.

[21] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1107–1116.

[22] M. Isamail and C. Panyayot, "High performance 2d convolution utilizing the avx512on a multi-core architecture," *Songklanakarin Journal of Science and Technology (SJST)*, pp. 1230–1236, 2021.

[23] J. S. J. Ren and L. Xu, "On vectorization of deep convolutional neural networks for vision tasks," *CoRR*, vol. abs/1501.07338, 2015. [Online]. Available: http://arxiv.org/abs/1501.07338

[24] V. I. Kelefouras, A. Kritikakou, and C. Goutis, "A methodology for speeding up edge and line detection algorithms focusing on memory architecture utilization," *Supercomputing, Springer*, 2013.

[25] C. Narendra, M. Sanchit, K. Dhiraj D., H. Alexander, G. Evangelos, Z. Barukh, A. Menachem, and K. Bharat, "Efficient and generic 1d dilated convolution layer for deep learning," *CoRR*, vol. abs/2104.08002, 2021.

[26] O. Reiche, C. Kobylko, F. Hannig, and J. Teich, "Auto-vectorization for image processing dsls," *SIGPLAN*, vol. 52, no. 5, p. 21–30, 2017.

[27] P. Chatarasi, S. Neuendorffer, S. Bayliss, K. Vissers, and V. Sarkar, "Vyasa: A high-performance vectorizing compiler for tensor convolutions on the xilinx ai engine," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–10.

[28] H. Kataoka, K. Yamashita, Y. Ito, K. Nakano, A. Kasagi, and T. Tabaru, "An efficient multicore cpu implementation for convolution-pooling computation in cnns," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 548–556.

[29] Q. Han, Y. Hu, F. Yu, H. Yang, B. Liu, P. Hu, R. Gong, Y. Wang, R. Wang, Z. Luan, and D. Qian, "Extremely low-bit convolution optimization for quantized neural network on modern computer architectures," in *ICPP '20 International Conference*, 2020.

[30] Intel, "oneapi deep neural network library (onednn)," July 2021. [Online]. [Online]. Available: https://oneapi-src.github.io/oneDNN/index.html

[31] L. Ismail and D. Guerchi, "Performance evaluation of convolution on the cell broadband engine processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 337–351, 2011.

[32] L. Meng and J. Brothers, "Efficient winograd convolution via integer arithmetic," *CoRR*, vol. abs/1901.01965, 2019.

[33] Intel, "Intel intrinsics guide," May 2021. [Online]. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

[34] A. Fog, "Optimizing subroutines in assembly language. an optimization guide for x86 platforms," Technical University of Denmark, Tech. Rep., 2021.

Vasilios Kelefouras Dr. Vasilios Kelefouras is an Assistant Professor in computer science at University of Plymouth. His main research interests are in the areas of code optimization, High Performance Computing and optimizing compilers. He has strong R&D experience in optimizing software applications, in terms of execution time, energy consumption and memory size, in a wide range of different hardware platforms. He has published more than 35 research papers.

Georgios Keramidas Dr. Georgios Keramidas is an Assistant Professor at the School of Informatics of the Aristotle University of Thessaloniki, Greece. His main research interests are in the areas of low-power processor/memory design,multicore systems, VLIW/multi-threaded architectures, graphic processors, power modelling methodologies, FPGA prototyping, and compiler optimizations techniques. He has published more than 70 papers, two books, and he also holds 11 US patents.