2023-10-04

Design and Implementation of Deep Learning 2D Convolutions on modern CPUs

Kelefouras, V

10.1109/tpds.2023.3322037
IEEE Transactions on Parallel and Distributed Systems
Institute of Electrical and Electronics Engineers

Design and Implementation of Deep Learning 2D Convolutions on modern CPUs

Vasilios Kelefouras, Georgios Keramidas,

Abstract—In this paper, a new method is provided for accelerating the execution of convolution layers in Deep Neural Networks. This research work provides the theoretical background to efficiently design and implement the convolution layers on x86/x64 CPUs, based on the target layer parameters, quantization level and hardware architecture. The proposed work is general and can be applied to other processor families too, e.g., Arm. The proposed work achieves high speedup values over the state of the art, which is Intel oneDNN library, by applying compiler optimizations, such as vectorization, register blocking and loop tiling, in a more efficient way. This is achieved by developing an analytical modelling approach for finding the optimization parameters. A thorough experimental evaluation has been applied on two Intel CPU platforms, for DenseNet-121, ResNet-50 and SqueezeNet (including 112 different convolution layers), and for both FP32 and int8 input/output tensors (quantization). The experimental results show that the convolution layers of the aforementioned models are executed from $x1.1$ up to $x7.2$ times faster.

Index Terms—Deep Neural Networks, convolution, oneDNN, optimization, analytical model, vectorization, register blocking, loop tiling

1 Introduction

Convolution layers are the main performance bottleneck in many classes of Deep Neural Networks (DNNs) and especially in Convolutional Neural Networks (CNNs) which are widely used in AI applications such as computer vision.

Although GPUs are recognized as a better option than CPUs for training DNNs, modern many-core CPUs offer competitive time-to-train for distributed deep learning training applications [1]. Furthermore, CPUs might be preferred over GPUs for training DNNs in some cases as first, CPUs can be more efficient when training small sized models or datasets [2], second, the large memory capacity of CPUs makes training with large datasets and/or models easier [3], third, CPUs can reduce the total cost of ownership for the DNN market, as CPUs are already widely deployed in datacenters and edge devices [3]. Regarding the inference tasks, CPUs are normally preferred over GPUs [4].

Speeding up the convolution layer of DNNs is a challenging and non-trivial task. This is because first, the optimization process depends on all the following : the convolution layer's parameters, the target hardware (HW) architecture and the quantization level (if applied), second, different manually vectorized and manually optimized routines are needed, for each case above, third, the exploration space is massive and thus it cannot be searched; the number of different optimized routines that need to be tested in order to find the optimum (just for a specific processor and for a specific layer's parameters), is astronomical.

To address the above problem, CPU/GPU vendors provide optimized libraries, such as Intel oneDNN [5]. oneDNN is a highly optimized vendor library developed and optimized by Intel engineers over many years; oneDNN is an open-source project and supports non-Intel HW platforms too, such as RISC-V. As of 2022, oneDNN is used as the default backend for CPU optimization in TensorFlow. oneDNN uses just-in-time compilation (JIT) to generate optimal code at runtime, tailored to the input parameters.

In this paper, the design and implementation of convolution layers on x86/x64 processors is delivered, for different layer parameters, quantization levels, vectorization technologies, cache sizes, number of vector registers, and CPU cores. The proposed work achieves high speedup values over oneDNN, by applying compiler optimizations, such as register blocking and loop tiling, in a more efficient way.

A thorough experimental evaluation is applied, on two diverse Intel CPUs (a 20-core NUMA CPU with AVX-512 and a 4-core CPU with AVX-256), three popular CNNs (DenseNet-121, ResNet-50, SqueezeNet), and both FP32 and int8 input/output tensors. We show that the proposed work provides speedups from $x1.1$ up to $x7.2$ over oneDNN.

The main contributions of this paper are: a) a research work providing the theoretical background to efficiently design and implement 2D convolution layers based on the target layer parameters, quantization level and HW architecture, b) an analytical model facilitating the optimization process, c) an experimental procedure showcasing that the proposed work achieves high performance gains over oneDNN on two 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 conclusions and future work.

2 Related Work

To accelerate DNN convolution layers, many strategies have been proposed, which are briefly explained hereafter.

The first group of works includes high level compression-based techniques such as quantization, low-rank quantization, parameter pruning, knowledge distillation [6], as well as exploiting sparsity in tensors [3] [7].

The second group of works implement the convolution operation by using a) either the Fast Fourier Transform

Manuscript received Oct 19, 2021; revised XXX

(FFT) [8] or the Winograd algorithm [9] [10], to reduce the number of executed FP computations, b) the highly optimized Matrix-Matrix Multiplication (MMM) routines of Intel_MKL or BLAS optimized libraries [11]. MMM-based algorithms rely on 'im2col' or 'im2row' memory transformations [11] to convert the direct convolution algorithm into an MMM problem. However, this approach introduces a high overhead in memory storage and bandwidth, which is proportional to the kernel size. In [12] [13], authors show that an efficient direct convolution implementation attains higher performance than the MMM based methods which use the expert-implemented MMM libraries.

The Winograd-based methods are mainly used for $3x3$ kernel sizes [10], while the FFT-based methods are mainly used for larger [8]. However, the reduced number of operations does not always align with performance as several challenges make it hard to fully utilize the HW resources on modern CPUs [10]. Furthermore, both algorithms suffer from a lack of precision. [9] extends and optimizes the Winograd-class of convolutional algorithms to the N-dimensional case of CNN on x86/x64 CPUs. In [10], another Winograd implementation is presented for manycore CPUs. In [14], a distributed implementation for the IBM Cell Broadband Engine processor is proposed.

Another optimization approach includes using Polyhedral compilers, such as Diesel [15], that automatically generate multi-level tiled code for affine loop nests, specialized machine learning (ML) compilers such as TVM [16] and autotuning systems such as TVM auto-scheduler [17]. TVM auto-scheduler uses a combination of auto-tuning and a dynamically trained ML model to guide the design-space exploration process. TVM auto-scheduler has demonstrated higher performance than Polyhedral compilers and other autotuning systems but it takes several hours to converge (for each layer), while also involves a complex training environment [4]. In [4], a structured configuration space is defined that enables faster convergence. In [18], the search space of loop interchange and loop tiling is reduced by using an analytical modelling approach. In [19], a ML approach is followed to generate efficient loop ordering.

Optimized vendor libraries such as oneDNN or cuDNN are also developed. oneDNN [5] supports different quantization levels and three algorithms (direct, MMM-based and Winograd-based). According to [5], Winograd algorithm is applicable only for limited input shapes and AVX-512, while MMM-based method is not that efficient.

Another group of works (that this paper is more related to) apply low-level compiler optimizations on the above compression based techniques and different algorithms. The main optimizations used are vectorization, parallelization, register blocking, loop tiling, loop interchange, software prefetching, improving the memory layout to enable vectorization and merging DNN layers.

In [1], Intel engineers present their JIT implementation which uses the direct algorithm on x64 CPUs; in this work, all the aforementioned compiler optimizations are applied. In [20], [1] is extended to compute the 1D dilated convolution. In [21], an efficient CPU implementation for convolution-pooling in CNNs is presented, by using convolution interchange and vectorization. In [22], authors propose a method to accelerate CNNs by using Arm NEON intrinsics and 16-bit computations. In [23], a 3D CNN model is optimized on an Arm based supercomputer. In [24], new direct implementations are proposed for depth-wise convolutions on ARMv8 architectures. In our previous work [13], we have optimized the convolution operation in the context of image filtering/smoothing image processing applications.

In comparison with the aforementioned works which focus on heuristics and empirical methods, our approach aims to find an efficient solution by using an analytical modelling approach. This way, the search space is massively reduced and a higher quality solution is easier to be found.

3 Proposed Method

The optimization process of the convolution layer is complex as first, a large number of optimizations needs to be applied and second, even for a specific layer and for a specific CPU, the exploration space is massive and it cannot be searched; the number of different optimized routines that need to be tested is astronomical. To address this problem we have developed an analytical model to generate the optimization parameters. The exploration space is reduced massively, first, by well studying this optimization problem, second, by expressing the number of executed Load/Store (L/S) instructions, and the number of dL1/L2/L3 memory accesses, as mathematical equations, where the HW architecture and convolution layer parameters, serve as inputs to these equations. Thus, optimizations such as register blocking and loop tiling, are applied in a more efficient way, and high speedup gains are achieved.

Given that different manually vectorized and optimized routines are required for different quantization levels, layer parameters, and HW architecture (e.g., AVX-512, AVX-256), a significant number of optimized routines has been developed. Furthermore, a software routine is developed that generates the output optimization parameters and calls the appropriate optimized routine (Algorithm 4). Note that the development effort of this paper was high.

3.1 Vectorization

Algorithm 1 shows the un-optimized code of the forward propagation of the 2D convolution layer with ReLU. Merging the convolution and ReLU layers is a popular optimization that reduces the number of memory accesses (supported by oneDNN too). The Nomenclature used here is as follows. The upper-case letters denote dimensions, e.g., B is the number of the batches, while the lower-case letters denote indexes, e.g., $0 \leq b \leq B-1$. A graphical illustration of Algorithm 1 is also shown on the top of Fig. 1. In Fig. 1 we assume B=1, in order to avoid drawing 4D tensors.

Vectorization is strongly affected by the memory layout of in array. The three most popular memory layouts of in array are the $bdyx$, $byxd$ and $dyxb$, and they are used by default in Caffe, Tensorflow and Neon, respectively [5]. We use $byxd$ here to denote that d is the inner-most dimension, meaning that two elements adjacent in memory would share the same indices of b, y, and x, and their index of d would be different by 1 (for non-boarder elements). On the contrary, b is the outermost dimension. In Algorithm 1 and Fig. 1, $byxd$ is assumed.

Algorithm 1 Naive 2D Forward Propagation with ReLU

```
for b=0,B,1 do                                          ▷ batch
    for m=0,M,1 do                                      ▷ feature maps (channels)
        for y=0,Y,1 do                                  ▷ output's height
            for x=0,X,1 do                              ▷ output's width
                float acc=0.0;
                for k.y=0,K.Y,1 do                      ▷ kernel's height
                    for k.x=0,K.X,1 do                  ▷ kernel's width
                        for d=0,D,1 do                  ▷ input depth
                            acc+ = in[b][y*stride.y+k.y][x*stride.x+k.x][d] * filter[m][k.y][k.x][d];
                        end for
                    end for
                end for
                acc+ = bias_array[m];
                out[b][y][x][m] = ReLU(acc);
            end for
        end for
    end for
end for
```

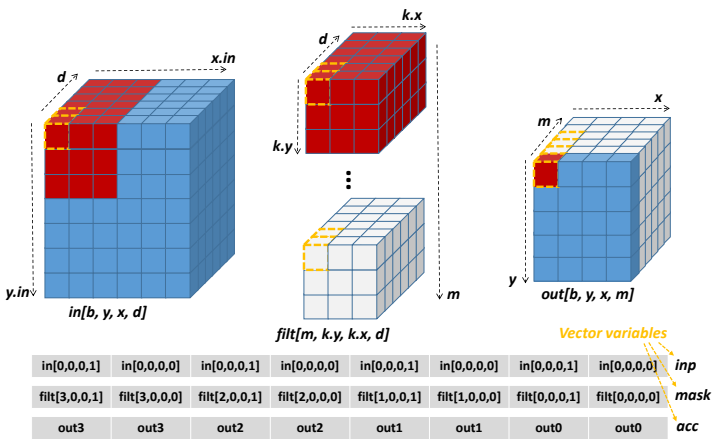


Fig. 1: Visual Illustration of Algorithm 2

Changing the layout of the in array introduces a high overhead, while changing the layout of $filter$ introduces a minor overhead as its size is normally much smaller. However, in the training process, the output of a convolution layer is the input of the next, and therefore we can avoid changing the layout of the input tensor by storing the output tensor's elements in the exact order they need to be read in the next layer. However, changing the layout in the first layer cannot be avoided and the overhead can be significant.

Vectorization can be applied to one or two loops, and thus many vectorization options arise. In this work, a theoretical analysis has been made in order to find which loops to vectorize. The criteria used to select the loops to vectorize include the parallelism exposed, the memory layout of the arrays, the number of executed instructions, and the memory footprint of the microkernel. A microkernel is a well-optimized piece of code who is called many times [4].

Let us first explore the vectorization opportunities of Algorithm 1 when only one loop is vectorized. It is efficient to vectorize a loop that is included in the subscript of out array, as in this case, first, the output elements are not stored one by one in memory but in vectors, and second, the elements in the accumulator vector are directly stored into memory and they are not horizontally added; this leads to fewer store and arithmetical instructions. Vectorizing a loop that is not included in the in subscript is advantageous as in this case vectorization is easier to be applied to different memory layouts of the input tensor. This is because only a single element (and not many) is loaded from in array at a time in this case. Although there are three loops that are not included in the in subscript (m, $k.y$ and $k.x$), only m is an efficient option since the other two do not provide adequate parallelism. Vectorizing b loop is efficient only when the memory layout of the input tensor has b as its innermost dimension (b-wise layout) and B is large. Vectorizing x or y loop is efficient only when the memory layout is x-wise or y-wise, respectively; still, vectorizing just x or y is problematic for layers with small spatial dimensions; note that CPUs with AVX-256/512 can process 8/16 FP or 32/64 int8 elements, respectively, in a single instruction. If the layout of in is not the appropriate, it is not efficient to vectorize b, x or y loops, as the overhead of changing the layout of in is significant. The alternative of vectorizing another loop in the first layer, store the output in a different way (e.g., b-wise), and then vectorize b in the remaining layers, hinders implementation issues and therefore the execution time of the first layer will be significantly degraded. Note that in the three CNNs studied in this paper, the first layer is the performance bottleneck, and thus a significant slowdown in the first layer will impact the overall execution time.

In the case of quantization, no option above is efficient and two loops need to be vectorized. Vectorizing a loop that is included in the out subscript is not feasible when the tensors contain int16/int8 values. This is because the intermediate results (IRs) need to be 32-bit and the existing x86-64 int8/int16 vector multiplication instructions, such as $maddubs$, will mix the results of different output elements in this case [13]. To sum up, when quantization is not applied, m is an efficient option for different memory layouts, while b/x are efficient only for large B/X values, and when the appropriate layout is used. However, in Subsection 3.3, we show that by vectorizing m loop, multiple and not one kernels are loaded and processed together (Fig.1), and the microkernel's data footprint might not fit into the cache in some cases, degrading performance; to address this problem two loops need to be vectorized in this case too (this is discussed in Subsection 3.3).

Let us now explore the vectorization opportunities when two loops are vectorized. Vectorizing two loops that reside in the in subscript, is not efficient as in elements from no consecutive memory locations need to be loaded (in the general case), and thus the number of load and arithmetical instructions becomes high. Although, we can store the output tensor's elements in the exact order we need to read them in the next layer, this is problematic, as first, we need to store the elements of two dimensions (and not one) in consecutive memory locations, and this constraints the optimization process in all layers, second, the first layer cannot be well optimized. Vectorizing two loops that reside in the out subscript does not solve the aforementioned problem with quantization. Therefore, it is efficient to vectorize one loop from the in subscript and another from the out subscript, which gives only one solution, i.e., m and d loops. This implies that the memory layout of in should have d as its innermost dimension. $k.y$ and $k.x$ are not selected as the vectorization process becomes very complicated in this case.

Based on the above analysis, when the memory layout

of in has d as its innermost dimension, we vectorize either m loop or both m and d loops, based on the target DNN parameters, HW architecture and quantization level. This memory layout, which is also known as pixel-wise (the RGB values of each pixel are clustered and stored consecutively), is the most popular way of storing images, and therefore this format is assumed for the rest of this paper. In the case where x is the innermost dimension of in memory layout, then vectorizing either m or x loop should be considered based on the above parameters; vectorizing x loop is not feasible when quantization is applied and therefore, the vectorization approach used in our previous work [13] should be followed to address this problem. In the case where b is the innermost dimension, then vectorizing either m or b loop should be considered.

Regarding 3D convolutions, extra constraints are introduced and the above analysis needs to be updated.

Let $m0/d0$ be the number of iterations vectorized in m/d loops, respectively; the $m0 \times d0$ value is given by Eq. 1, where $vector.length$ is the length of the target vectorization technology in bytes, and $d.length$ is the length of the tensors' data type used in bytes, e.g., for AVX-256 and FP32 $m0 \times d0 = 32/4$. $m0/d0$ are always powers of 2.

$$m0 \times d0 = vector.length/d.length \tag{1}$$

It is important to note that the memory layout of the filter array must change. A separate loop kernel is introduced where the old filter's elements are stored in a new array, in the exact order they will be accessed in the convolution layer. This routine is vectorized too and in this case, its impact on overall performance is normally insignificant.

When $d0 = 1$, only m loop is vectorized. $d0 > 1$ when a) quantization is used (lines 5-7 in Algorithm 4), e.g., $d0 \geq 4$ in int8 case, b) the microkernel's data cannot fit in L2 or dL1, c) $m0 \times Rm > M$, where Rm is the register blocking factor of m loop. The latter two cases are discussed next.

oneDNN uses a similar approach, as they vectorize m loop in the FP32 case, and $m + d$ loops when quantization is used. The difference between oneDNN and the proposed work is that we vectorize $m + d$ loops not only when quantization is used, but in cases (b)-(c) too above.

The application of vectorization is shown in Algorithm 2. Note that when $d0 = 1$, the lines 16-17 are omitted. On the contrary, when $d0 > 1$, the IRs in acc variable need to be added (line 16), e.g., in Fig. 1, the two $out0$ values need to be added. This is implemented by using shift and add instructions since the $hadd$ instructions are slower.

The results of the horizontal additions are placed into consecutive vector positions to be efficiently processed and stored into memory. It is important to note that this step is computationally expensive and its optimization is not trivial. As it is going to be explained into the next subsection, register blocking optimization is also applied and therefore the $put.consecutive()$ routine in line 17 is applied to many vector variables not just one. Thus, to optimize this step, multiple vector variables are processed together and not one (we re-order the values of multiple vectors together). Neither the optimized code nor the optimization approach is shown here, as first, the problem we are addressing is different for different $d0$ values, second, the page size is limited. To reduce the number of vector instructions, when $d0 = 2$, we change the $filt$ layout in a way where its odd and even m values are stored separately and thus they are loaded into separate vectors. When $d0 = 4$, we change the $filt$ layout in a way where the m values that are multiples of 4 are stored separately (e.g., first store 0,4,8,12 then 1,5,9,13, etc) and thus they are loaded into separate registers.

Algorithm 2 Algorithm 1 with vectorization. The $filt$ array is a new array with different memory layout

```
1: __m256 inp, mask, acc, bias                                  ▷ vector variables
2: for b=0,B,1 do
3:     for m=0,M,m0 do                                          ▷ vectorized loop
4:         for y=0,Y,1 do
5:             for x=0,X,1 do
6:                 acc = setzero();                             ▷ initialize with zeros
7:                 for k.y=0,K.Y,1 do
8:                     for k.x=0,K.X,1 do
9:                         for d=0,D,d0 do                      ▷ vectorized loop
10:                            inp = broadcast(in[b][y*stride.y+k.y][x*
   stride.x + k.x][d : d + d0]);   ▷ load and broadcast d0 elements
11:                            mask = load(filt[m : m+m0][k.y][k.x][d :
   d + d0]); ▷ load m0 × d0 elements (consecutive memory locations)
12:                            acc = fmadd(inp, mask, acc);     ▷
   acc+ = inp × mask
13:                        end for
14:                    end for
15:                end for
16:                acc = hadd(acc);  ▷ apply d0-1 horizontal additions
17:                acc = put.consecutive(acc);  ▷ put the results of the
   horizontal additions into consecutive positions
18:                bias = load(bias_array[m : m + m0]);   ▷ load bias
19:                acc = add(acc, bias);                          ▷ add bias
20:                acc = ReLU(acc);
21:                store(out[b][y][x][m : m + m0], acc); ▷ store acc into
   memory (m0 elements)
22:            end for
23:        end for
24:    end for
25: end for
```

3.2 Register Blocking and Data Reuse

Register blocking can significantly reduce the number of executed L/S instructions and therefore, a higher arithmetic intensity (AI) value can be achieved; consequently, the program achieves a better place in the roofline model, and this is why register blocking is by far the most critical optimization of all. Algorithm 3 shows Algorithm 2 with register blocking to m and x loops (loop tiling is also applied in m loop, but let's ignore this for now); lines 6-38 define the microkernel.

At this point, two important questions arise; in which loops to apply register blocking and what factors to use. The exploration space is big and it is very time consuming to test all different solutions. To address this problem, an analytical model is developed where we theoretically calculate the number of executed L/S instructions, based on the target DNN and HW parameters and vectorization/blocking factors. Thus, the best solution is found theoretically.

(Method1) Register Blocking analysis for b, y, x, m, d loops only : Let's assume that register blocking is applied to b, y, x, m, d loops in Algorithm 2 with factors Rb, Ry, Rx, Rm, Rd, respectively. Applying register blocking to $k.x$ and $k.y$ loops requires a more complicated analysis and it is explained next (Method2). The number of executed L/S instructions for each array are shown in Eq. 2.

Algorithm 3 Algorithm 2 with register blocking and loop tiling, $m0 = 8$, $d0 = 1$, $Rm = 2$, $Rx = 2$ and $d.length = 4$.

```
1: __m256 inp, mask0, mask1, acc0, acc1, acc2, acc3, bias;
2: for b=0,B,1 do
3:     for m=0,M,Tm do                                   ▷ loop tiling
4:         for y=0,Y,1 do
5:             for x=0,X,Rx do                           ▷ reg.blocking
6:                 for m2=m,m2<m+Tm,m0 x Rm do ▷ reg.blocking and vectorization
7:                     acc0 = setzero();
8:                     acc1 = acc0; acc2 = acc0; acc3 = acc0;
9:                     for k.y=0,K.Y,1 do
10:                        for k.x=0,K.X,1 do
11:                            for d=0,D,d0 do           ▷ vectorization
12:                                mask0 = load(filt[m2 : m2 + m0][k.y][k.x][d : d + d0]);
13:                                mask1 = load(filt[m2+m0 : m2+2×m0][k.y][k.x][d : d + d0]);
14:
15:                                inp = broadcast(in[b][y ∗ stride.y + k.y][x ∗ stride.x + k.x][d : d + d0]);
16:                                acc0 = fmadd(inp, mask0, acc0);
17:                                acc1 = fmadd(inp, mask1, acc1);
18:                                inp = broadcast(in[b][y ∗ stride.y + k.y][(x + 1) ∗ stride.x + k.x][d : d + d0]);
19:                                acc2 = fmadd(inp, mask0, acc2);
20:                                acc3 = fmadd(inp, mask1, acc3);
21:                            end for
22:                        end for
23:                    end for
24:                    bias0 = load(bias_array[m2 : m2 + m0]);
25:                    bias1 = load(bias_array[m2 + m0 : m2 + 2 × m0]);
26:                    acc0 = add(acc0, bias0);
27:                    acc1 = add(acc1, bias1);
28:                    acc2 = add(acc2, bias0);
29:                    acc3 = add(acc3, bias1);
30:                    acc0 = ReLU(acc0);
31:                    acc1 = ReLU(acc1);
32:                    acc2 = ReLU(acc2);
33:                    acc3 = ReLU(acc3);
34:                    store(out[b][y][x][m2 : m2 + m0], acc0);
35:                    store(out[b][y][x][m2+m0 : m2+2×m0], acc1);
36:                    store(out[b][y][x + 1][m2 : m2 + m0], acc2);
37:                    store(out[b][y][x + 1][m2 + m0 : m2 + 2 × m0], acc3);
38:                end for
39:            end for
40:        end for
41:    end for
42: end for
```

$$In.Loads = \frac{OPS}{Rb \times Rm \times Ry \times Rx \times Rd} \times Rb \times Ry \times Rx \times Rd \quad (2a)$$

$$Filt.Loads = \frac{OPS}{Rb \times Rm \times Ry \times Rx \times Rd} \times Rm \times Rd \quad (2b)$$

$$Out.Stores = \frac{(\frac{OPS}{D/d0 \times K.Y \times K.X})}{Rb \times Ry \times Rx \times Rm} \times Rb \times Ry \times Rx \times Rm' \quad (2c)$$

$$OPS = B \times M/m0 \times Y \times X \times K.Y \times K.X \times D/d0 \quad (2d)$$

$$Rm' = \lceil Rm/d0 \rceil + Z, \text{ where Z=0 when} \quad (2e)$$

$((Rm \times m0)\%(m0 \times d0))$ is a power of 2 or zero, and Z=1 otherwise.

Eq. 2a is explained hereafter. Prior register blocking (see Algorithm 2), there are OPS load/broadcast instructions for in array; OPS gives the number loop iterations in Algorithm 2. After register blocking, the iterators are incremented with a larger step, e.g., Rx instead of 1 (line 5 in Algorithm 3), and thus there are $OPS/(Rb \times Rm \times Ry \times Rx \times Rd)$ load/broadcast instructions for in array (first term in Eq. 2a). However, register blocking generates multiple load instructions in the innermost loop body (Algorithm 3), and not one; in particular there are $Rb \times Ry \times Rx \times Rd$ load instructions in the general case (2nd term in Eq. 2a), because these iterators are in the subscript of in array; in Algorithm 3, $Rb = 1$,$Ry = 1$,$Rd = 1$,$Rx = 2$ and thus 2 load instructions occur. Eq. 2b is extracted the same way.

Regarding the out array, Eq. 2c is extracted in a similar way when d0=1, but when $d0 > 1$, Eq. 2c becomes more complex and this is why Rm' is introduced. As it was described in Subsection 3.1, $m0 \times d0$ elements are vectorized, but $m0 \times Rm$ elements need to be stored into consecutive memory locations, and as a consequence fewer than $m0 \times d0$ elements are stored when $Rm < d0$ ($Rm < d0$ only when $d0 > 1$ as $Rm \geq 1$), e.g., when Rm=1, m0=8 and d0=1, 8 elements (256-bit) are stored into memory, while when d0=2 and m0=4, 4 elements are stored into memory (128-bit). Now consider the case where $Rx = 2$, $Rm = 2$, $m0 = 4$, $d0 = 2$ (FP32 and AVX-256). As in the previous example, 128-bit and not 256-bit store instructions are required; however, albeit there are four ($Rm \times Rx = 4$) 128-bit store instructions, they can be realized by using two 256-bit instructions instead, and thus $Rm' = 1$ in Eq. 2c, not $Rm' = 2$.

Based on Eq. 2, the overall number of L/S instructions is given by Eq. 3. The number of load instructions for the bias array is insignificant and thus to ease presentation it is not included in Eq. 3. Eq. 3 gives the number of L/S instructions based on the DNN input parameters and vectorization / register blocking factors. The vectorization factors ($m0, d0$) are found in lines 5-7 (Algorithm 4), while the register blocking factors are found by Eq. 4c (explained after).

$$L/S.overall = OPS/Rm + OPS/(Rb \times Ry \times Rx) + (OPS/(D/d0 \times K.Y \times K.X)) \times (\lceil (Rm/d0 \rceil + Z)/Rm) \quad (3)$$

$$0.85 \times Regs \leq Rb \times Rm \times Ry \times Rx + Rb \times Ry \times Rx + Rm + ext \leq Regs \quad (4a)$$

$$0.85 \times Regs \leq Rm \times Rx + Rx + Rm + ext \leq Regs \quad (4b)$$

$$0.85 \times Regs \leq Rm \times Rx + min(Rm, Rx) + 1 + ext \leq Regs \quad (4c)$$

Based on Eq. 3, the number of L/S instructions does not depend on the Rd value and thus register blocking is not applied to d loop ($Rd = 1$). Furthermore, the number of L/S instructions in Eq. 3 does not depend on the individual register blocking factors, but on the following two quantities: ($Rb \times Ry \times Rx$) and Rm. Therefore, ($Rb \times Ry \times Rx$) is studied as a whole and as a consequence, there is no need to apply register blocking to all the b, y, x loops, e.g., applying register blocking just to x loop with $Rx = 8$ is equivalent to ($Rb = Ry = Rx = 2$), in the general case. This statement is false only in special cases, i.e., when $Rx > X$ or when Rx cannot perfectly divide X. In such cases, register blocking is also applied to y loop.

Based on the above analysis, we apply register blocking to x and m loops only. y loop is blocked only in the special cases above. To make our analysis easier to follow, for the rest of this Section we assume that $Rm < M$, $Rx < X$ and that Rm/Rx perfectly divide M/X, respectively.

The register blocking factors are given by Eq. 4c. To ease presentation Eq. 4a and Eq. 4b are also provided which are used to generate Eq. 4c. The first term in Eq. 4a refers to the vector variables needed for the accumulators ($acc0 - acc3$ in Algorithm 3), the second term in Eq. 4a refers to the vector variables needed for loading the in array and the third term for the $filt$ array; these values are generated

based on the arrays' subscripts (note that $Rd = 1$). Based on the above analysis $Rb = Ry = 1$ and thus Eq. 4a gives Eq. 4b. The number of vector variables used in the innermost loop body should not exceed the number of available HW registers ($Regs$ value in Eq. 4), which equals to 16/32 for AVX-256/512, respectively. Otherwise, register spills occur (additional L/S instructions) and performance is degraded. The constant value 0.85 is found experimentally.

The ext in Eq. 4 refers to any other additional vector variables required in the innermost loop body, and in most cases $ext = 0$, e.g., in Algorithm 3, $ext = 0$. Additional vector variables are required only when quantization is applied (but not always), e.g., for AVX-256 and int8, there is no special $fmadd$ instruction (like the one in line 12 Algorithm 2) that takes as input 8-bit values and generates 32-bit IRs and thus the instruction in line 12 is substituted by three instructions in this case; thus, more variables are required for storing the IRs. Note that in the case where AVX-512 and int8 still $ext.vars = 0$.

Eq. 4b can be optimized in a way that fewer vector variables are used (Eq. 4c). According to Eq. 4b, Rx/Rm vector variables are allocated for $in/filt$ arrays, respectively. As it can be observed in Algorithm 3, Rx variables of in are multiplied by Rm variables of $filt$. However, there is no need loading all these values prior to their processing. Instead, the number of variables required can be further reduced by overwriting either the values of in or $filt$, e.g., in Algorithm 3, we load the first value of in, multiply it by all the filter values and then load the next value of in on the same variable (Algorithm 3). This saves variables and thus Eq. 4c gives larger register blocking factors than Eq. 4b, and as a consequence fewer L/S instructions are achieved. However, if we use two variables for storing in in Algorithm 3, four fmadd instructions are executed one after another and not two, and in this case the fmadd instructions are executed faster (better throughput is achieved). Although there is a trade-off between Eq. 4b and Eq. 4c, we have experimentally observed that Eq. 4c is more efficient and therefore only Eq. 4c is used in this work.

An example follows. Consider AVX-512, $Regs = 32$, $m0 = 16$, $d0 = 1$, and FP32 data (thus $ext = 0$). Eq. 4c is satisfied by several solutions, some of which are shown below (note that $c = D \times K.Y \times K.X$):

(Rm,Rx) = (1,30), Eq. 3 gives $1.033 \times OPS + OPS/c$ (5a)

(Rm,Rx) = (2,14), Eq. 3 gives $0.571 \times OPS + OPS/c$ (5b)

(Rm,Rx) = (3,9), Eq. 3 gives $0.444 \times OPS + OPS/c$ (5c)

(Rm,Rx) = (4,6), Eq. 3 gives $0.417 \times OPS + OPS/c$ (5d)

(Rm,Rx) = (5,5), Eq. 3 gives $0.400 \times OPS + OPS/c$ (5e)

(Rm,Rx) = (6,4), Eq. 3 gives $0.417 \times OPS + OPS/c$ (5f)

(Rm,Rx) = (8,3), Eq. 3 gives $0.458 \times OPS + OPS/c$ (5g)

Although Eq. 5d and Eq. 5f give the same number of instructions, Eq. 5d is always more efficient as its memory footprint is smaller (it has a smaller Rm value).

The solution achieving the minimum number of L/S instructions might not be the fastest in the following special cases : a) $Rm \times m0 > M$ (addressed in Subsection 3.3), b) $Rx > X$, c) $Rm \times m0 < M < 2 \times Rm \times m0$ or $Rx < X < 2 \times Rx$, d) the microkernel's data cannot fit in L2 or dL1 (addressed in Subsection 3.3). To deal with the cases (a) and (d), the method in Subsection 3.3 is used. To address cases (b) and (c), we might need to select a solution where the Rm/Rx perfectly divide M/X, respectively, or use more than one microkernels with different blocking factors.

To sum up, the approach used to select the register blocking factors is shown in lines 8-11 in Algorithm 4. We select the solution that achieves the minimum number of L/S instructions and the solution that gives up to 8% more instructions but with a lower Rm value (if any), e.g., in the example above the (5,5) and (4,6) are selected only. This is because solutions with smaller Rm value give microkernels that use less memory; the higher the $m0 \times Rm$ value is, the more the kernels being processed together (Fig. 1) and as a consequence, the higher their memory size will be. The data footprint of in and $filt$ are ($D \times Rx \times K.Y \times d.length$) and ($D \times K.X \times K.Y \times Rm \times m0 \times d.length$), respectively.

(Method2) Extend Method1 for $k.x$ and $k.y$ loops too: When the size of the kernel is not 1×1, the number of load instructions in in array can be significantly reduced by exploiting the data reuse on the kernel, e.g., when a 3×3 kernel is shifted by one position to the right (assume $stride.x = 1$), six out of nine in elements are reused. This type of data reuse can be exploited and further reduce the number of load instructions in in array.

To this end, register blocking should be applied to both x and $k.x$ loops, and $Rk.x$ value should be fixed, i.e., $Rk.x = K.X$. Based on the arrays' subscripts, $(K.X + (Rx - 1) \times Stride.X)$ vector variables are loaded from in and multiplied by $K.X \times Rm$ variables of $filt$ in this case. Thus, Eq. 2a becomes Eq. 6. In this method, each element of in array is loaded once and processed multiple times, and therefore, we allocate only one variable for in. Thus, Eq. 4c becomes Eq. 7. In Eq. 7, only one variable is used for in and $K.X \times Rm$ variables for $filt$. In this case, the data access pattern of in array changes and far fewer load instructions are executed in in array (Eq. 6). The number of L/S instructions in the other arrays remain unchanged.

$$In.Loads = \frac{OPS \times (K.X + (Rx-1) \times Stride.X)}{(K.X \times Rx \times Rm)} \quad (6)$$

$$0.85 \times Regs \leq Rm \times Rx + 1 + K.X \times Rm + ext \leq Regs \quad (7)$$

Following up the previous example (see Eq. 5), the number of overall L/S instructions is highly reduced now, as we get 0.28 OPS and 0.22 OPS, for 3×3 and 7×7, respectively (assuming $stride.x = 1$).

Applying register blocking to both $(Rx + k.x)$ and $(Ry + k.y)$ is not efficient because first, too many vector variables are required for $filt$ array ($K.X \times K.Y \times Rm$) and this is not feasible for kernels larger than 3×3 , second, the AI value is reduced.

Although Method2 achieves far fewer load instructions, Method1 gives better performance in most cases. Method2 achieves better data reuse at a register level (and thus achieves fewer load instructions), while Method1 achieves better data reuse at a cache level. In Method1, $D \times (Rx - 1)$ elements in in array are reused in the cache every time the $k.x$ loop increases its value (each element is loaded from the cache multiple times). On the other hand, in Method2, each in element is loaded just once but there is no data reuse in the cache. Method2 always gives a smaller Rm value and as a consequence microkernels with smaller data footprint,

compared to Method1, and thus Method2 is preferred over Method1 when the data of Method1 cannot fit into the cache (this is discussed below).

3.3 Reducing the microkernel's memory footprint

As it was explained in the previous Subsection, first, the register blocking solution achieving the minimum number of L/S instructions might not be the fastest when the microkernel's data footprint cannot fit into the cache, and second, the selected register blocking factors are not valid when $Rm \times m0 > M$. This Subsection addresses these two cases.

Let us further explain the first point with an example. The solution in Eq. 5d gives $0.417 \times OPS$ while the Eq. 5b gives $0.57 \times OPS$, L/S instructions. In the case where a 3×3 kernel is used and $D = 256$, Eq. 5d uses data of $608KB$ while Eq. 5b of $338KB$. Thus, if we need to reduce the data footprint (to fit into the cache), we need either a) to select a solution with $Rm < 4$, and as a consequence a higher number of L/S instructions will be executed, degrading performance, or b) use Method2; however, even in this case, data might not fit into the cache.

There is a more elegant way of reducing the microkernel's data footprint without sacrificing the number of L/S instructions. To this end, we keep the register blocking factors unchanged and we decrease the $m0$ value according to the memory size we need to save. Note that $m0$ is given by Eq.1 and thus when $m0$ is reduced, $d0$ is increased accordingly ($m0$ and $d0$ are powers of 2). The data footprints of in and $filt$ are ($D \times Rx \times K.Y \times d.length$) and ($D \times K.X \times K.Y \times Rm \times m0 \times d.length$), respectively, and thus, the overall size is given by $D \times K.Y \times d.length(Rx + K.Y \times Rm \times m0)$. The data footprint of $filt$ is several times larger in most cases, as $Rx < k.Y \times Rm \times m0$. By reducing the $m0$ value by a factor of 2,4 or 8, the overall memory size is reduced almost equally. This way, the number of L/S instructions remains unchanged as the (Rm, Rx) values remain unchanged. Note that the OPS value in Eq. 2d depends on the $m0 \times d0$ value, which remains unchanged.

On the negative side, the higher the $d0$ value, the higher the overhead in arithmetical instructions in lines 16-17 (Algorithm 2), as more horizontal additions are required and the results need to be stored into consecutive positions (line 17). The impact of these instructions on performance is not high, as first, they are located outside of the three innermost loops ($d, k.x, k.y$), second, the $D \times K.X \times K.Y$ value is large when the data footprint does not fit in dL1. The second drawback, which impacts performance more, is that more store instructions might be required when $d0 > 1$ (this was explained in Subsection 3.1, see Eq. 2e).

To conclude, the microkernel's footprint should always fit in either L2 or dL1, at all times. Whether it should fit in L2 or dL1 depends on the target HW and DNN parameters and quantization level, and the selection is being made in lines 12-51 in Algorithm 4. When quantization is used, there are cases where multiple and not one instructions are required to implement line 12 in Algorithm 2. In this case, the instruction in line 12 is substituted by three instructions and therefore the AI of the program is increased (becomes more compute bound). In this case, the optimization approach should change too. Based on our experimental analysis, when line 12 in Algorithm 2 cannot be realized by a single instruction, the tiles should fit in dL1. Otherwise, loop tiling for dL1 is not efficient as long as the data fit in L2.

Last, in the case where $Rm \times m0 > M$, it is not feasible to use the current Rm value, and thus the above technique is applied to achieve an efficient solution for small M values.

3.4 Parallelization, loop tiling and loop permutation

Parallelization is realized by using the OpenMP framework. The number of the threads used equals to the number of physical CPU cores (let c). The loops being parallelized here are either b or y, based on the target DNN and HW parameters. The loop being parallelized is always set as the outermost. For the reminder of this paper we assume that b loop provides sufficient parallelism; otherwise, both b and y loops (or even a third loop) need to be parallelized.

Loop tiling and loop permutation are also applied to further reduce the number of data accesses in memory hierarchy. The exploration space is massive and therefore a theoretical approach need to be followed in this case too.

Regarding loop permutation, $(k.y, k.x, d)$ are set as the innermost loops; this way, the out array is not loaded but just stored into memory once, reducing the number of both arithmetical and L/S instructions. b loop is parallelized and thus is set as the outermost. The remaining loops are (m, y, x) which can be permuted. However, the number of different permutations can be reduced by using reasoning and thus only two candidate loop permutations are propagated, i.e., (m, y, x) and (y, x, m).

Applying loop tiling to d-loop is not efficient, as first, the out array is loaded and stored multiple times in this case, second, the tiles of in contain non-consecutive memory locations (with the current layout) and thus they cannot entirely fit into the cache [25]. $k.y/k.x$ loops are too small for applying tiling. b-loop is not appropriate for loop tiling (with the existing in layout), as each b iteration brings a large amount of data and thus data cannot fit into the cache. On the other hand, m, x and y loops are the best candidates for loop tiling. However, the tiles contain duplicates (common array elements), when either x or y is tiled and $K.X > 1$ or $K.Y > 1$, respectively.

In this work, when b-loop is parallelized, loop tiling is applied to m loop only (Algorithm 3), while when y-loop is parallelized both m and y loops are tiled. Parallelizing or tiling y-loop can further reduce the number of L3 or DDR accesses in some cases. When y-loop is parallelized, each thread processes an in tile of size $((Y.in/c) \times X.in \times D)$. Each core processes its own tile which must fit into its L2 private cache. In the general case, each thread processes an in tile of size $(Y.in/Ty \times X.in \times D)$, where $(Ty = c \times \lceil batch/(c \times L2.size) \rceil)$ ($batch$ is defined in Algorithm 5). Note that when $K.Y > 1$, the tiles of in contain duplicates and their number increases according to the $K.Y$ value. This might be problematic for large kernels, especially for NUMA architectures. The effect of this problem has not been studied yet for kernels larger than 5×5.

So far, there are several candidate solutions (different loop permutations as well as loops to tile and parallelize). To select the most efficient solution, we have generated the mathematical equations that approximate the number of data accesses in each memory, based on the target DNN and HW parameters; based on these equations, we have developed Algorithm 5. To justify and better explain Algorithm

Algorithm 4 Calculate the optimization parameters

1: **procedure** CALCULATE THE OPTIMIZATION PARAMETERS
2: **Input:** B,M,Y,X,K.X,K.Y,D,dL1.size,L2.size,Regs, vector.length, d.length
3: **Output:** m0, d0, Rm, Ry, Rx, Tm, Ty, par.loop
4:
5: ▷ **//find the initial vectorization parameters**
6: Calculate d0 based on the quantization level, e.g., d0=1/d0=4, for FP32/int8, respectively
7: Calculate m0 from Eq. 1
8: ▷ **//find the initial Register Blocking parameters**
9: Calculate (Rm,Ry,Rx) achieving the min number of L/S instructions (Method1 only)
10: $List1 \leftarrow$ the (Rm,Ry,Rx) solution giving up to 8% more L/S instructions but with smaller Rm value (if any)
11: $List2 \leftarrow$ the (Rm,Ry,Rx) solution achieving the min number of L/S instructions (Method 2 only)
12: ▷ **//find the tile sizes (Tm,Ty), the loop to be parallelized (par.loop), and update d0,m0,Rm,Ry,Rx if needed**
13: **if** (multiple and not one instructions are required to implement line 12 in Algorithm 2) **then** ▷ *in this case (e.g., AVX-256 and int8), line 12 is substituted by three instructions and thus the AI is increased (our optimization approach reflects this)*
14: **if** (the microkernel's data footprint cannot fit in dL1) **then**
15: (m0, d0, Rm, Rx, Ry, Tm)=**Reduce_Footprint_&_Find_Tile**(m0, d0, Rm, Rx, Ry,dL1.size, List1, List2)
16: **else**
17: (Tm,Ty,par.loop)=**Find_Tile**(B,M,Y,...,d.length)
18: **end if**
19: **else** ▷ *one instr. is required to implement line 12 in Algorithm 2*
20: **if** (the microkernel's data footprint cannot fit in L2) **then**
21: (m0, d0, Rm, Rx, Ry, Tm)=**Reduce_Footprint_&_Find_Tile**(m0, d0, Rm, Rx, Ry,L2.size, List1, List2)
22: **else**
23: **if** (the microkernel's data footprint cannot fit in dL1) **then**
24: **if** (List1 not empty) **then**
25: Use (Rm,Ry,Rx) solution from List1
26: **end if**
27: **if** (filt fits in L2) **then**
28: Tm=M;Ty=-; par.loop=b;
29: **else**
30: (Tm,Ty,par.loop)=**Find_Tile**(B,M,Y,...,d.length)
31: **end if**
32: **else**
33: (Tm,Ty,par.loop)=**Find_Tile**(B,M,Y,...,d.length)
34: **end if**
35: **end if**
36: **end if**
37: **Return** (m0, d0, Rm, Ry, Rx, Tm, Ty, par.loop)
38: **end procedure**
39:
40: **procedure Reduce_Footprint_&_Find_Tile**
41: **Input:** m0, d0, Rm, Rx, Ry, Li.size, List1, List2
42: **Output:** m0, d0, Rm, Rx, Ry, Tm
43:
44: **if** (data footprint of List1 fit in Li) **then**
45: Use (Rm,Ry,Rx) solution from List1
46: **else**
47: either a) reduce the data footprint to fit in Li, by calculating the new $d0, m0$ values, or b) use the List2 solution (if its footprint fits in Li) ▷ *in Section 4, the first option is used, but both options are efficient*
48: **end if**
49: $Tm = Rm \times m0$ ▷ *the tile gets its min value so as to fit in Li*
50: **Return** (m0, d0, Rm, Ry, Rx, Tm)
51: **end procedure**

Algorithm 5 Calculate the (Tm,Ty,par.loop) so as to reduce the number of accesses in memory hierarchy (see Table 1)

1: **procedure Find_Tile**
2: **Input:** B,M,Y,X,K.X,K.Y,D,m0,d0,Rm,Ry,Rx,dL1.size,L2.size, Regs, vector.length, d.length **Output:** Tm, Ty, par.loop
3: ▷ $in.size = B \times D \times X.in \times Y.in \times d.length$
4: ▷ $filt.size = D \times K.X \times K.Y \times M \times d.length$
5: ▷ $in.footprint = D \times Rx \times K.Y \times d.length$
6: ▷ $filt.footprint = D \times K.X \times K.Y \times Rm \times m0 \times d.length$
7: ▷ $batch = D \times X.in \times Y.in \times d.length$
8: ▷ $mem.access.A = in.size \times M/Tm + filt.size \times B$
9: ▷ $mem.access.B = in.size + filt.size \times B \times X/Rx \times Y$
10: ▷ *Here we assume that b-loop can well distribute the workload*
11:
12: $Tm \leftarrow$ calculate Tm so as $in.footprint$ and $(D \times K.x \times K.y \times Tm \times d.length)$ fit in dL1 $(Rm \times m0 \leq Tm \leq M)$
13:
14: **if** (filt and in.footprint fit in dL1) **then**
15: **return** (M,-,b) ▷ *arrays are accessed from L2 once*
16: **else if** (batch fits in L2) **then**
17: **if** (filt fits in L2) **then**
18: **if** $(mem.access.A \geq mem.access.B)$ **then**
19: **return** (M,-,b)
20: **else**
21: **return** (Tm,-,b)
22: **end if**
23: **else** ▷ *Filt cannot fit in L2*
24: **return** (Tm,-,b) ▷ *minimize L3 accesses*
25: **end if**
26: **else if** (batch fits in L3) **then**
27: **if** (filt fits in L2) **then**
28: **if** $(mem.access.A \geq mem.access.B)$ **then**
29: **return** (M,-,b)
30: **else**
31: **if** (y-loop can well distribute the workload) **then**
32: **return** (Tm,Ty,y)
33: **else**
34: **return** (M,-,b)
35: **end if**
36: **end if**
37: **else if** (filt fits in L3) **then**
38: **if** (y-loop can well distribute the workload) **then**
39: **return** (Tm,Ty,y)
40: **else**
41: **if** $(mem.access.A \geq mem.access.B)$ **then**
42: **return** (M,-,b)
43: **else**
44: **return** (Tm,-,b)
45: **end if**
46: **end if**
47: **else** ▷ *filt fits in DDR*
48: **if** (y-loop can well distribute the workload) **then**
49: **return** (Tm,Ty,y)
50: **else**
51: **return** (Tm,-,b)
52: **end if**
53: **end if**
54: **else** ▷ *batch fits in DDR*
55: **if** (filt fits in L2) **then**
56: as in lines 27-34 above
57: **else if** (filt fits in L3) **then**
58: as in lines 30-34 above
59: **else** ▷ *filt fits in DDR*
60: as in lines 37-45 above
61: **end if**
62: **end if**
63: **end procedure**

TABLE 1: Number of times the arrays are accessed from L2 and L3 cache, $expr = (B \times X/Rx \times Y)$.

	Lines 16-25 in Algorithm 5		Lines 26-46 in Algorithm 5	
	in	filter	in	filter
	No tiling		No tiling	
L2	$\frac{M}{m0 \times Rm \times c}$	B/c	$\frac{M}{m0 \times Rm \times c}$	B/c
L3	1	1 or B	$\frac{M}{m0 \times Rm}$	1 or B
	(Tm,Ty,y)		(Tm,Ty,y)	
L2	$M/(Tm \times c)$	B/c	$M/(Tm \times c)$	B/c
L3	1	1 or B	1	1 or B
	(Tm,-,b)		(Tm,-,b)	
L2	$M/(Tm \times c)$	B/c	$M/(Tm \times c)$	B/c
L3	1	1 or B	M/Tm	1 or B
	(M,-,b)		(M,-,b)	
L2	1	$expr/c$	1	$expr/c$
L3	1	1 or $expr$	1	1 or $expr$

5, Table 1 is provided. Table 1 shows how many times the in and $filt$ arrays are loaded from L2 and L3 cache, when the if-conditions in lines 16-25 and 26-46 are executed, respectively. Because of the limited page size, we do not provide a table for all the cases in Algorithm 5. Note that two values are provided for $filt$ array, one when it fits in L2 and another when it fits in L3. The out array is stored just once and thus not shown here. Algorithm 5 selects the solution achieving the minimum number of DDR accesses. If the solutions provide the same number of DDR accesses, then the one achieving the minimum number of L3 accesses is selected, etc. In our future work we are planning to improve this step, by selecting the solution that minimizes the following value $\max_i\{Li.accesses/Li.bandwidth\}$, where $i = [1, 4]$ (in our case four memories exist).

3.5 Backward propagation and Weight gradient update

The backward propagation and the weight gradients update algorithms are slightly different to the forward propagation algorithm studied in this paper. However, there are two scenarios (which cover the majority of CNNs) where we can transform the weight tensors, and then we can reuse the existing forward propagation routines, i.e., when either a) the stride value equals to one, or b) the kernel size is 1×1. In our future work we will extend this work to the backward propagation phase when the (a) and (b) conditions above are not met, as well as to the weights gradient update phase.

4 Experimental Results

4.1 Experimental Setup

The experimental results are performed on two diverse Intel platforms, namely HW1 and HW2. HW1 consists of a two socket 10-core NUMA Intel Xeon Silver 4210 CPU (20 physical cores in total) at 2.20GHz with AVX-512, 32KB dL1 and 1MB L2 per core, 27.5MB L3 (in total), 128GB DDR4 3200MHz, running Ubuntu 20.04.4 LTS, oneDNN v2.6.0 and icc version 2021.6.0. HW2 consists of a quad-core Intel i5-7500 CPU at 3.40GHz with AVX-256, 32KB dL1 and 256KB L2 per core, 6MB L3, 16GB DDR4 2666MHz, running Ubuntu 20.04, oneDNN v2.3.0 and icc version 2021.5.0. The theoretical peak performance on HW1/HW2 is 1408/435 GFLOPs, respectively.

The proposed work is evaluated over Intel oneDNN library by using Intel's best practices and the fused implementation provided in [26] (executes the convolution and

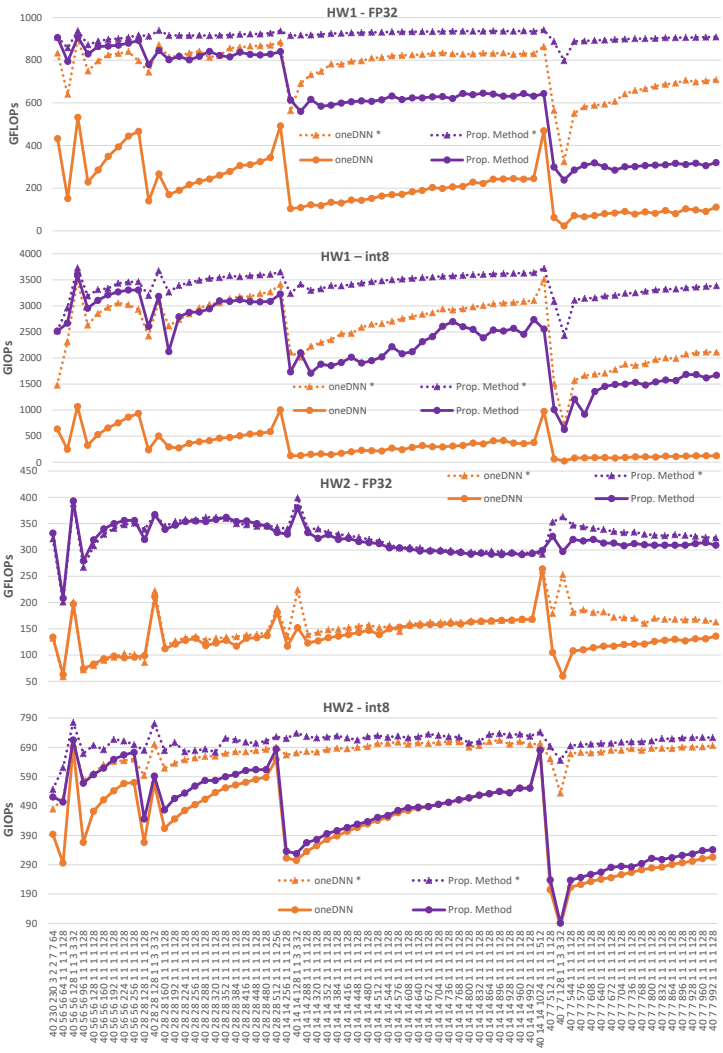


Fig. 2: Performance evaluation for DenseNet-121

ReLU layers, on blocked data format). The routine's name is $conv\_relu\_fused()$ and uses the $convolution\_direct$ algorithm and forward inference option. The memory layout of the input tensor is $byxd$. The cost of changing the layout of the tensors (if needed) is included in the above routine.

Our evaluation also contains quantization where all the input/output tensors contain 8-bit values. The IRs contain int-32 values. When quantization is used the 32-bit IRs need to be converted to 8-bit and therefore the 8-bit results are not located into consecutive vector positions. The process of re-arranging the output values into consecutive vector locations is not trivial and is optimized (Subsection 3.1). Vectorization is more challenging in the int-8 case and the code is different than the FP32 case. Furthermore, different routines are required for the AVX-256 and AVX-512 case.

The performance metrics used here are FLOPs for the FP32 case and IOPs (integer tensor operations per second) for the int8 case. Both FLOPS and IOPs are given by : $FLOPs/IOPs = B \times Y \times X \times M \times (2 \times K.Y \times K.X \times D + 1)/run.time$.

The evaluation is made on three popular CNNs, i.e., DenseNet-121, ResNet-50, SqueezeNet. Just the convolution and ReLU layers are evaluated (112 different convolution

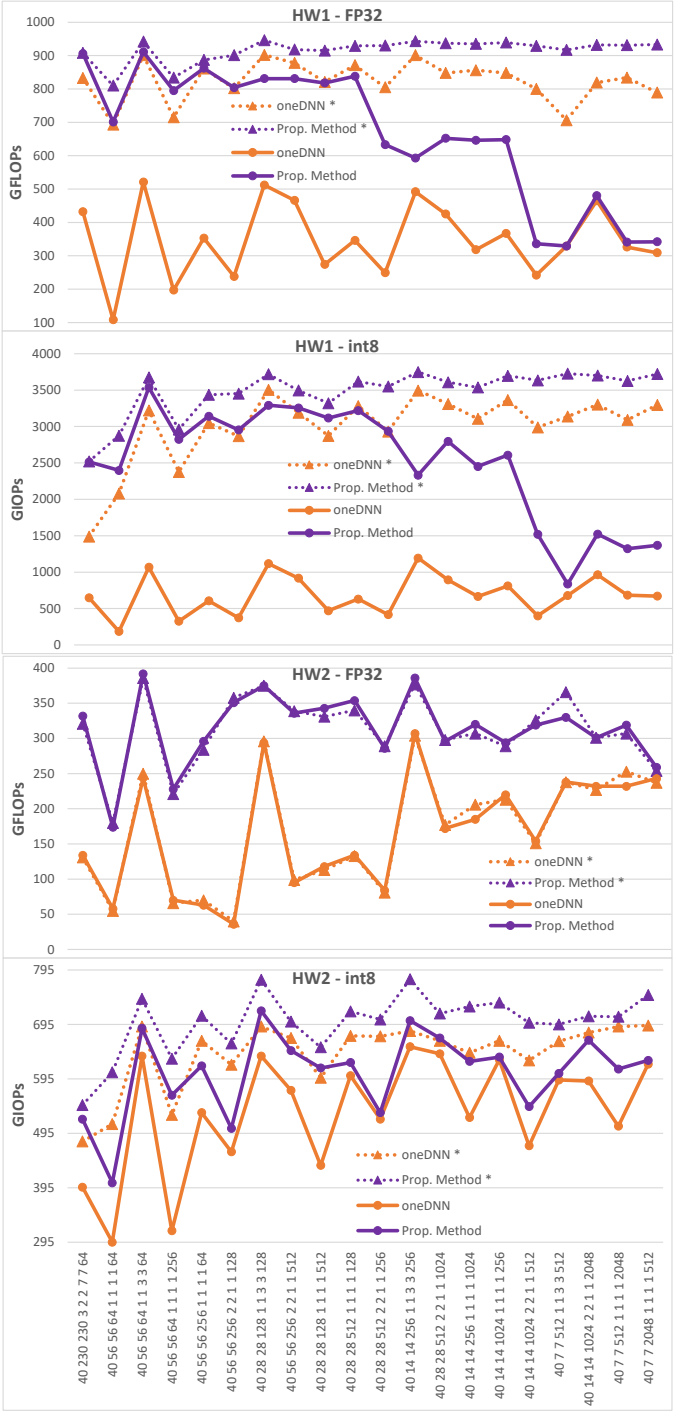


Fig. 3: Performance evaluation for ResNet-50

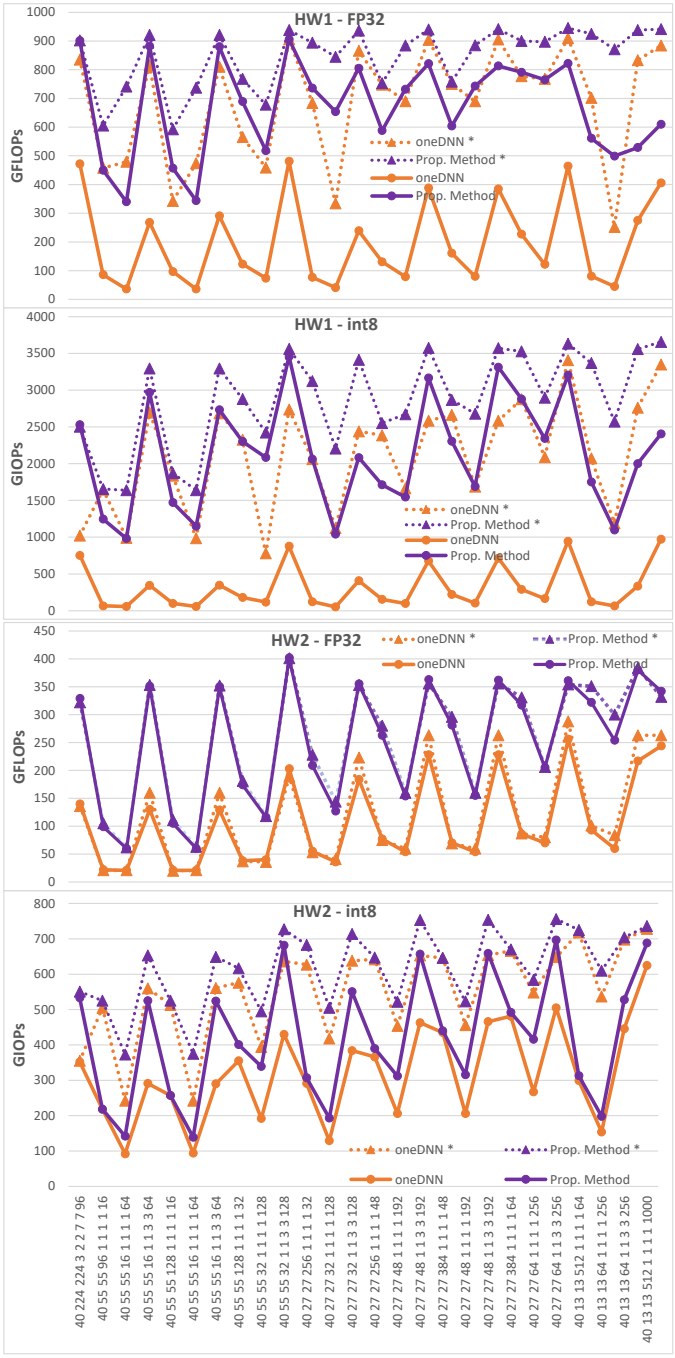


Fig. 4: Performance evaluation for SqueezeNet

layers). The x-axes in Fig. 2- 4 show the layers' parameters ($B, Y, X, D, Stride.Y, Stride.X, K.Y, K.X, M$); layers are shown in order and identical layers are shown only once.

oneDNN uses a JIT feature to generate optimized kernels at runtime, tailored to the inputs provided. One of the advantages of this method is that the input parameters are known when the code is generated and this allows for a better optimization process. The drawback is that generating JIT kernels requires some time; the kernel generation cost is amortized when either a) the workload of the layer is large enough, or b) a layer with the exact same input parameters runs many times (on different tensors).

A thorough experimental evaluation is applied, in two different ways: a) by taking into account the kernel generation cost (general case), b) by not considering the kernel generation cost (special case where a layer with the exact same input parameters runs many times). The latter is realized by putting the $conv\_relu\_fused()$ routine in a loop and running it many times (dashed lines in Fig. 2- 4); note that the kernel generation cost can also be amortized by saving the generated schedule (by using oneDNN persistent caches) and then re-using it. The routine is run as many times needed so as the execution time is close to 1 sec. This procedure is repeated for 20 times and the minimum

execution value is taken. This is done twice and thus the program run 40 times in overall. Regarding the first way of evaluation, the binary is run 20 times and the minimum value of $conv\_relu\_fused()$ routine is taken; this procedure is repeated three times and thus each binary runs 60 times. In this case, the kernel generation cost is included to the extracted FLOPs value. To minimize the 'noise' from the other running processes, we report the minimum and not the average execution time, as this is more suitable for processes that run for short periods of time (msecs); this way, we make sure we only time the target process and nothing else. Using the average execution time includes more 'noise' (especially for the 1st way of evaluation), and therefore every time we run the experiments, we get slightly different FLOPs values. On the contrary, we get approximately the same FLOPs when we use the minimum execution time.

Last, our routine that generates the optimization parameters (Algorithm 4) as well as the routine that changes the layout of the filter/bias arrays, are included in the evaluation; the overhead of the first is inconsiderable while the overhead of the second ranges from 0.1% to 3%.

4.2 Performance Evaluation

The performance evaluation is shown in Fig. 2-Fig. 4 and in Table 2. Putting the target routine in a loop and running it multiple times (2nd way of evaluation) gives either better or approximately the same performance in all cases, for both methods, as first, the data are hot into the cache, and second, the OpenMP overhead is lower. The gap between the 1st and 2nd way of evaluation is higher for oneDNN as the cost of generating the kernel is zero in the 2nd way of evaluation. The only case where the two different ways of evaluating performance give approximately the same performance is the HW2-FP32 case; in HW2-FP32 case, the workload per thread is way higher compared to the other three cases, and therefore both the OpenMP overhead and the cost of generating the kernel, are low. Note that the workload is distributed into 20/4 threads on HW1/HW2, respectively, and the int-8 case achieves fewer instructions than the FP32 case; additionally, there are more executed instructions per thread on the HW2 case compared to the HW1, as on HW2, 256-bit of data are processed and not 512-bit. Nevertheless, in the last 'small' layers of DenseNet, the workload per thread is not good enough and thus running the routine many times gives higher performance.

Quantization gives up to $x4$ times higher performance on HW1 as the int-8 case achieves about $x4$ fewer tensor operations. The cost of converting the 32-bit IRs into 8-bit and storing the 8-bit results into consecutive vector locations is not high, because, first, this process is not applied inside the three innermost loops, second, this process is well optimized. However, the quantization gain is lower on HW2 compared to HW1, as no $dpbusds$ instruction is supported on HW2 and therefore three AVX intrinsics are needed to realize the $fmadd$ instruction. This gives a higher number of arithmetical instructions in the innermost loop body.

There is a high FLOPs/IOPs deviation (Fig. 2-Fig. 4) for different layers for four main reasons. Regarding the 1st way of evaluation, 'big' layers (layers with a large number of tensor operations) give higher FLOPs/IOPs than 'small' layers for both methods, until we reach a critical point where the OpenMP overhead becomes low; the OpenMP overhead is lower for 'big' layers as the workload per thread is higher. Regarding oneDNN, the performance variation is even higher since the kernel generation cost is higher for small layers. Regarding the 2nd way of evaluation, performance is slightly affected by the layer size, as a) the OpenMP overhead is low, and b) the data are hot into the cache; the only case it is significantly affected is the HW1-int8 case where the OpenMP overhead is the highest among all four cases. Regarding the second reason, the layers with higher $(D \times K.Y \times K.X)$ values achieve higher FLOPs/IOPs, as the three innermost loops run for longer and the overhead of lines 16-21 (Algorithm 2) is lower. Third, the layers with stride 2 normally give lower performance than layers with stride 1 because data are loaded less efficiently. Last, layers with 3×3 kernels normally achieve higher FLOP values than 1×1 kernels, as the 3×3 case has a higher AI. For this reason, register blocking is more critical in the 1×1 case.

The proposed work achieves high speedup values in the 1st way of evaluation, and significant speedup values in the special case where a layer with the same input parameters runs many times (Table 2). Table 2 is calculated by accumulating the execution time values of all layers. The speedup values at the HW1-int8 case (1st way of evaluation) are the highest because the workload per thread is the lowest. We are not aware of the reason oneDNN does not generate efficient code in the HW2-FP32 case.

TABLE 2: Overall speedup over oneDNN library for all the convolution layers. The first three rows show the speedup of the dashed lines in Fig. 2- 4, while the last three show the speedup of the continuous lines.

	HW1-FP32	HW1-int8	HW2-FP32	HW2-int8
DenseNet-121 *	1.12	1.24	2.24	1.09
ResNet-50 *	1.1	1.15	1.92	1.1
SqueezeNet *	1.12	1.61	2.33	1.23
DenseNet-121	2.93	7.2	2.39	1.1
ResNet-50	1.72	3.18	1.98	1.17
SqueezeNet	2.92	6.47	2.38	1.41

Register blocking is by far the most performance critical optimization; on HW1, the register blocking factors used in most cases are $(Rm, Rx) = (4, 6)$. On HW2, the most used factors are $(2, 6)/(2, 5)$ for the FP32/int-8 case, respectively. The Rx value might be different when $Rx > X$ or $Rx < X < 2 \times Rx$ (y-loop might also blocked in this case). y-loop is parallelized in some cases on HW2 only.

Although B=40 in Fig. 2- 4, we have evaluated the two methods for $B = [20, 40, 80, 120, 1500, 4000]$. Regarding the 1st way of evaluation, the performance gain over oneDNN is higher for $B < 40$ and lower for $B > 40$, compared to B=40, for the reason explained above. Even in the extreme case where B=1500 or higher, the speedup values achieved are high (significantly higher than the speedup values shown in the first three rows of Table 2). Regarding the 2nd way of evaluation, the performance gains are similar.

5 Conclusions and Future Work

This research work provides the theoretical background to efficiently design and implement convolution layers on CPUs, based on the target layer parameters, quantization

level and HW architecture. We show that the optimization process is complex and includes a massive exploration space; therefore, peak performance cannot be achieved without building an analytical model.

Regarding oneDNN, although JIT provides the advantage of portability and generating optimized kernels tailored to the inputs provided, generating JIT kernels requires some time; thus, oneDNN performs well only when either the layer's workload is large enough, or when the exact same layer runs many times. This is why our method gives high speedup values for 'small' layers and small batch sizes. The proposed method achieves significant speedup values even when the kernel generation cost is amortized, because the optimization parameters are provided by an analytical model. We believe that the proposed method can be integrated to the oneDNN project to improve its performance. Furthermore, our analytical approach can be used to better guide auto-tuning systems (e.g., TVM auto-scheduler) and allow for higher quality solutions in lesser time.

In our future work, we are planning to improve the way we select a solution in Subsection 3.4 and apply software prefetching. Furthermore, we are planning to extend this work to the backward propagation phase when the conditions in Subsection 3.5 are not met, as well as to the weights gradient update phase.

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*, 2018.

[2] A. Jayasimhan and P. Pabitha, "A comparison between cpu and gpu for image classification using convolutional neural networks," in *2022 International Conference on Communication, Computing and Internet of Things (IC3IoT)*, 2022, pp. 1–4.

[3] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, and J. Torrellas, "Sparsetrain: Leveraging dynamic sparsity in software for training dnns on general-purpose simd processors," in *PACT '20*, New York, USA, 2020, p. 279–292.

[4] N. Tollenaere, G. Iooss, S. Pouget, H. Brunie, C. Guillon, A. Cohen, P. Sadayappan, and F. Rastello, "Autotuning convolutions is easier than you think," *ACM Trans. Archit. Code Optim.*, Nov. 2022.

[5] Intel, "onednn v2.7.0 documentation, understanding memory formats," Oct. 2022. [Online]. [Online]. Available: https://oneapi-src.github.io/oneDNN/dev_guide_understanding_memory_formats.html

[6] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A Survey of Model Compression and Acceleration for Deep Neural Networks," *arXiv e-prints*, p. arXiv:1710.09282, Oct. 2017.

[7] 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 CVPRW Workshops*, 2019, pp. 1353–1360.

[8] 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.

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

[10] Z. Jia, A. Zlateski, F. Durand, and K. Li, "Optimizing n-dimensional, winograd-based convolution for manycore cpus," in *23rd ACM SIGPLAN*, ser. PPoPP '18, 2018, p. 109–123.

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

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

[13] V. Kelefouras and G. Keramidas, "Design and implementation of 2d convolution on x86/x64 processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3800–3815, 2022.

[14] 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.

[15] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, *Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs*, New York, NY, USA, 2018, p. 42–51.

[16] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Conference*, USA, 2018, p. 579–594.

[17] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Conf.*, ser. OSDI'20. USA: USENIX Assoc., 2020.

[18] R. Li, Y. Xu, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, "Analytical characterization and design space exploration for optimization of cnns," in *ASPLOS '21*. New York, NY, USA: Association for Computing Machinery, 2021, p. 928–942.

[19] T. R. Patabandi, A. Venkat, A. Kulkarni, P. Ratnalikar, M. Hall, and J. Gottschlich, "Predictive data locality optimization for higher-order tensor computations," in *5th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 43–52.

[20] 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.

[21] 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 IPDPSW*, 2020, pp. 548–556.

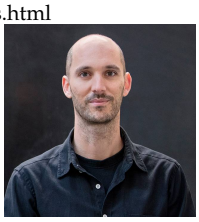
[22] S.-J. Lee, S.-S. Park, and K.-S. Chung, "Efficient simd implementation for accelerating convolutional neural network," in *ICCIP '18*, New York, NY, USA, 2018, p. 174–179.

[23] A. Tabuchi, K. Shirahata, M. Yamazaki, A. Kasagi, T. Honda, K. Kurihara, K. Kawakami, T. Tabaru, N. Fukumoto, A. Kuroda, T. Fukai, and K. Sato, "The 16,384-node parallelism of 3d-cnn training on an arm cpu based supercomputer," in *HiPC*, 2021.

[24] R. Hao, Q. Wang, S. Yin, T. Zhou, S. Shen, S. Mei, and J. Liu, "Towards effective depthwise convolutions on armv8 architecture," *CoRR*, vol. abs/2206.12124, 2022.

[25] V. Kelefouras, K. Djemame, G. Keramidas, and N. Voros, "A methodology for efficient tile size selection for affine loop kernels," *Int. J. Parallel Program.*, vol. 50, no. 3–4, p. 405–432, 2022.

[26] Intel, "Performance profiling example," Nov. 2022. [Online]. [Online]. Available: https://oneapi-src.github.io/oneDNN/page_performance_profiling_cpp.html



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 compiler optimization and High Performance Computing. 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 HW platforms. He has published more than 45 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, FPGA prototyping, and compiler optimizations techniques. He has published more than 70 papers.