Optimizing Pointwise Convolutions on Multi-core DSPs

Yang Wang[1,2,3], Qinglin Wang[1,2](✉), Xiangdong Pei[1,2], Songzhu Mei[1], and Jie Liu[1,2]

[1] National Key Laboratory of Parallel and Distributed Computing, National University of Defense Technology, Changsha 410073, China

[2] Laboratory of Digitizing Software for Frontier Equipment, National University of Defence Technology, Changsha 410073, China

wangqinglin.thu@gmail.com

[3] Beijing Institute of Astronautical Systems Engineering, Beijing 100076, China

Abstract. Pointwise convolutions are widely used in various convolutional neural networks, due to low computation complexity and parameter requirements. However, pointwise convolutions are still time-consuming like regular convolutions. As a result of increasing power consumption, low-power embedded processors have been brought into high-performance computing field, such as multi-core digital signal processors (DSPs). In this paper, we propose a high-performance multi-level parallel direct implementation of pointwise convolutions on multi-core DSPs in FT-M7032, a CPU-DSP heterogeneous prototype processor. The main optimizations include on-chip memory blocking, loop ordering, vectorization, register blocking, and multi-core parallelization. The experimental results show that the proposed direct implementation achieves much better performance than GEMM-based ones on FT-M7032, and a speedup of up to 79.26 times is achieved.

Keywords: CNNs · Pointwise Convolution · Direct Convolution · DSPs · Parallel algorithm

1 Introduction and Related Work

Convolutional neural networks (CNNs) are extensively used in diverse fields such as computer vision and scientific computing [3,4,15,28]. As CNNs develop, more convolutional layers with small filters are applied in the models, such as pointwise convolutions in which the filter size is only 1×1. And this type of convolutional layer is commonly utilized in mainstream backbone networks, such as ResNet [6] and GoogleNet [21], and lightweight networks, such as MobileNetV1 [8] and MobileNetV2 [20]. Thus, it is very important to implement high-performance pointwise convolutions on targeted platforms.

supported by the National Natural Science Foundation of China under grant nos. 62002365 and 62025208.

The dominant methods for implementing convolutions are matrix multiplication-based, Winograd-based, Fast Fourier Transform (FFT)-based, and direct algorithms [2,5,7,9,10,12,22,23,26]. For the matrix multiplication-based method, the convolutions are converted into matrix multiplication operations in an explicit or implicit way. For example, Wang et al. [26] implemented two-dimensional convolutions using implicit matrix multiplication. Thus, the performance of convolutions largely relies on the performance of matrix multiplication on hardware platforms in this method. The fast methods including Winograd-based and FFT-based ones can effectively decrease the computational complexity of convolutions, while they are only applicable to convolutions with large filters. Since the direct method has no extra memory overhead and can gain high performance, numerous direct implementations for various types of convolutions have been proposed on different platforms, such as regular convolutions on Intel CPUs [7] and ARM Mali GPUs [16]. Lu et al. proposed two novel optimization techniques to improve the performance of pointwise convolutions by enhancing data reuse in row and column directions on NVIDIA mobile graphics processing units (GPUs) [13,14]. Wang et al. proposed a parallel direct algorithm for pointwise convolutions on ARMv8 multi-core CPUs [24]. However, there is little work on the direct implementation of pointwise convolutions on multi-core DSPs.

Multi-core digital signal processors (DSPs) have been brought into the high-performance computing field due to the low-power characteristic [11]. To diminish power consumption, DSPs usually adopt Very Long Instruction Word (VLIW) architecture, software-controlled on-chip memories, and Direct Memory Access (DMA) engines for data moving, which are unique and different from the architectures of modern CPUs and GPUs. There have been many parallel implementations of algorithms and applications on multi-core DSPs, such as matrix multiplications [19,27], matrix transpose [18], and GEMM-based convolutions [25], but the parallel direct optimization of pointwise convolutions targeting multi-core DSPs has not been found.

FT-M7032 is a CPU-DSP heterogeneous prototype processor which consists of one 16-core ARMv8 CPU for process management and four 8-core DSPs for offering major peak performance [27]. To improve the performance of pointwise convolutions on FT-M7032, this paper proposes a high-performance parallel direct implementation for pointwise convolutions targeting multi-core DSPs. In parallelization, many common optimization techniques are carried out, such as vectorization, register blocking, and multi-core parallelization. The experimental results demonstrate that the direct implementation gets the computation efficiency of 11.42% - 58.61% and outperforms the GEMM-based one with speedups of 1.43×–79.26× on multi-core DSPs in FT-M7032. Compared with the implementations in Pytorch [17] and ARM Computer Library [1] running on the ARMv8 CPU in FT-M7032, the proposed direct implementation gets a speedup of up to 35.84 times. To the best of our knowledge, this is the first work about the direct parallelization of pointwise convolutions on multi-core DSPs.

The structure of this paper is as follows. Section 2 outlines the definition of pointwise convolutions and the architecture of FT-M7032 processors. Section 3 describes our parallel direct implementation of pointwise convolutions on multi-core DSPs in FT-M7032 processors in detail. Section 4 shows the analyses of the performance results. Last, the conclusion and future work are given in Sect. 5.

2 Backgound

2.1 Pointwise Convolution

For the forward propagation pass, pointwise convolutions work on input feature maps tensor $\boldsymbol{I}$ with filters tensor $\boldsymbol{F}$ to produce output feature maps tensor $\boldsymbol{O}$. The backward propagation and weight gradient update passes obtain the input feature maps gradient tensor $\boldsymbol{dI}$ and filter gradient tensor $\boldsymbol{dF}$ based on output feature maps gradient tensor $\boldsymbol{dO}$, respectively. With blocked data layout which is very beneficial to vectorization, the three passes above of pointwise convolutions are figured by Eqs. 1, 2, and 3.

$$\boldsymbol{O}_{n,k_d,h_o,w_o,k_l} \mathrel{+}= \boldsymbol{I}_{n,c_d,h_o\times S,w_o\times S,c_l} \times \boldsymbol{F}_{c_d\times L+c_l,k_d,0,0,k_l}, \tag{1}$$

$$\boldsymbol{dI}_{n,c_d,h_o\times S,w_o\times S,c_l} \mathrel{+}= \boldsymbol{dO}_{n,k_d,h_o,w_o,k_l} \times \boldsymbol{F}_{c_d\times L+c_l,k_d,0,0,k_l}, \tag{2}$$

$$\boldsymbol{dF}_{c_d\times L+c_l,k_d,0,0,k_l} \mathrel{+}= \boldsymbol{dO}_{n,k_d,h_o,w_o,k_l} \times \boldsymbol{I}_{n,c_d,h_o\times S,w_o\times S,c_l}, \tag{3}$$

where $n \in [0, N)$, $k_d \in [0, K_d)$, $h_o \in [0, H_o)$, $w_o \in [0, W_o)$, $k_l \in [0, L)$, $c_d \in [0, C_d)$, $c_l \in [0, L)$, N is the mini-batch size, C and K are the number of input and output channels, C_d and K_d represent the number of blocks in C and K dimensions, $C = C_d \times L$, $K = K_d \times L$, L is the number of lanes in vector units of DSPs, $H_{i/o}$ and $W_{i/o}$ denotes the spatial dimensions of different tensors, and S is the stride size. In this paper, only the unit-stride pointwise convolutions are involved so the stride size is 1 in the following.

2.2 Architecture of FT-M7032 Heterogeneous Processors

An FT-M7032 heterogeneous processor consists of a 16-core ARMv8 CPU and four GDPSP clusters, shown in Fig. 1. The 16-core CPU where the Linux operating system runs is mainly for process management and multi-node communication, and its single-precision peak performance is 281.6 GFlops with 2.2 GHz working frequency. Each GPDSP cluster, also called a multi-core DSP, includes eight DSP cores and global shared memory (GSM), which is connected by an on-chip crossbar network. Each core can offer 345.6 GFlops single-precision peak performance with 1.8 GHz working frequency so that the total peak performance of each GPDSP cluster can achieve up to 2764.8 GFlops. The 16-core CPU and

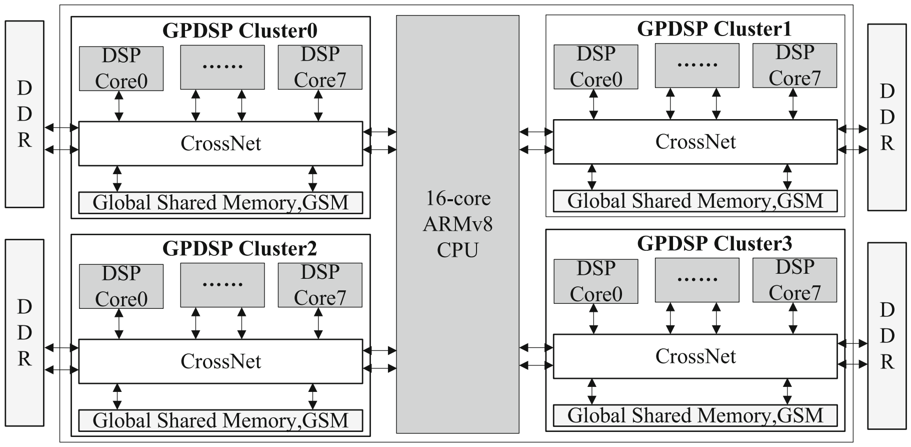

Fig. 1. Architecture of FT-M7032 Processors

four GPDSP clusters share the same memory space. Specifically, the CPU can access the whole main memory space in FT-M7032, while each GPDSP cluster can only access a specific part with 42.6 GBytes/s bandwidth. Therefore, four GPDSP clusters can communicate with each other via the CPU, and be mainly utilized by process-level parallelization.

The micro-architecture of each DSP core is shown in Fig. 2. Each core primarily includes a scalar processing unit (SPU), a vector processing unit (VPU), an instruction dispatch unit (IFU), and a DMA engine. SPU is used to support parallel execution of five scalar instructions, where the size of scalar memory (SM) is 64 KB. VPU is applied to carry out vector instructions, and the capacity of array memory (AM) is 768 KB. There are three 64-bit float-point fused multiply-add (FMAC) units in each of 16 vector processing elements (VPEs), so VPU can perform three vector 32-bit FMAC (VFMAC) operations with 32 lanes per cycle. VPU also has two parallel vector load-store units (VLoad/VStore), each of which can convey data of up to 2048 bytes per cycle between AM and vector registers. There are 64 1024-bit vector registers in total. SPU can directly transfer data to VPU through broadcast operations and shared registers. These DSP cores adopt VLIW architecture, and IFU can issue up to 11 instructions per cycle, including at most five scalar instructions and six vector instructions. The DMA engine is in charge of fast data transmission between different memories.

3 Parallel Direct Implementation

3.1 Overview of Our Implementation

Pointwise convolutions are computationally equivalent to matrix multiplication. Therefore, when directly mapping pointwise convolutions on multi-core CPUs and GPUs, the optimization methods for matrix multiplication are carried out

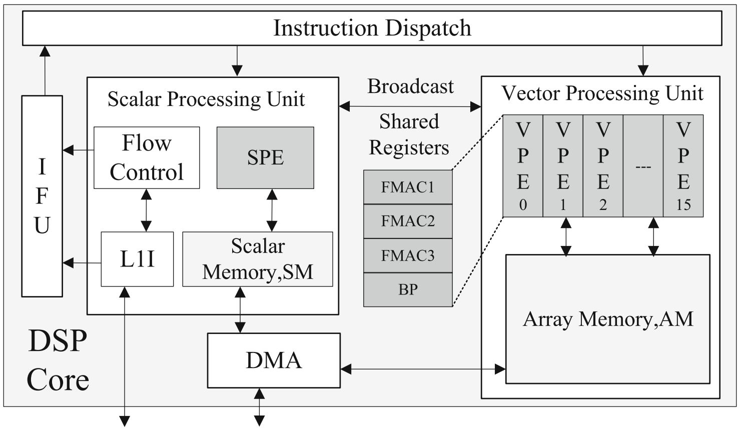

Fig. 2. Micro-architecture of each DSP core in FT-M7032 Processors

for high efficiency. This paper also follows this rule above and incorporates the architectural features of the GPDSP cluster in the FT-M7032 and the relatively small number of parameters in pointwise convolution for targeted algorithm design and optimization.

3.2 Multi-level Parallel Forward Propagation Algorithm

When the stride size is 1, the spatial dimensions H and W of feature maps can be merged into a single dimension denoted as $H \times W$. In this section, we propose a multi-level parallel direct algorithm named directConv1x1Fwd() for computing the forward propagation pass of pointwise convolutions in convolutional neural networks, shown in Algorithm 1. The implementation of the Conv1x1FwdAsm kernel function within directConv1x1Fwd() is presented in Algorithm 2. Since the storage cost of the filter tensor $\boldsymbol{F}$ in pointwise convolutions is typically low, directConv1x1Fwd() prioritizes loading $\boldsymbol{F}$ into the on-chip AM space or GSM space. To accommodate the unit-strided convolutions, directConv1x1Fwd() merges the dimensions H and W directly into one (Line 10). In the following, we primarily employ directConv1x1Fwd() as an exemplar to elucidate the meticulous design of a multi-level parallel forward propagation algorithm for realizing high-performance pointwise convolution.

On-Chip Memory Blocking and Loop Ordering. GPDSP clusters are equipped with on-chip storage spaces, namely SM, AM, and GSM spaces. In order to achieve high-performance computing objectives, algorithms commonly load relevant tensor data into these spaces in blocking format prior to performing calculations using the on-chip data. Furthermore, loop ordering is necessary to optimize the locality of the on-chip data within the storage space and reduce the overhead of accessing off-chip DDR storage.

Algorithm 1: Multi-level parallel forward propagation direct algorithm for unit-stride pointwise convolutions on multi-core DSPs

```
Input  : $\boldsymbol{I}[N][C_d][H_i][W_i][L]$, $\boldsymbol{F}[C][K_d][1][1][L]$
Output: $\boldsymbol{O}[N][K_d][H_o][W_o][L]$
1  Calculate the block size for each level
2  for $c_{gd} = 0: C_{dgb}: C_d$ do
3      for $k_{dg} = 0: K_{dgb}: K_d$ do
4          if $K_{dab} \times C_{dab}$ != $K_d \times C_d$ then
5              Load the $\boldsymbol{F}$ subblock into the GSM space $\boldsymbol{F_{gsm}}$ via DMA
6          else
7              Directly load the entire $\boldsymbol{F}$ into the AM space $\boldsymbol{F_{am}}$ via DMA
8          for $k_{da} = 0: K_{dab}: K_{dgb}$ do
9              for $n = 0: 1: N$ do in parallel
10                 for $hw = 0: HW_{ob}: H_o \times W_o$ do in parallel
11                     if $c_{gd}$ != 0 then
12                         Load the $\boldsymbol{O}$ subblock into the AM space $\boldsymbol{O_{am}}$ via DMA
13                     for $c_{da} = 0: C_{dab}: C_{dgb}$ do
14                         if $K_{dab} \times C_{dab}$ != $K_d \times C_d$ then
15                             Load the $\boldsymbol{F_{gsm}}$ subblock into the AM space $\boldsymbol{F_{am}}$ via
                                DMA
16                         for $c_{ds} = 0: C_{dsb}: C_{dab}$ do
17                             Load the $\boldsymbol{I}$ subblock into the SM space $\boldsymbol{I_{sm}}$ via
                                DMA
18                             Call Conv1x1FwdAsm()
19                     Store the $\boldsymbol{O_{am}}$ subblock in the DDR space $\boldsymbol{O}$ via DMA
```

Within the design of directConv1x1Fwd(), the SM space stores the blocking data of the input feature tensor $\boldsymbol{I}$, while the AM space accommodates the blocking data of both the output feature tensor $\boldsymbol{O}$ and the filter tensor $\boldsymbol{F}$. By prioritizing the loading of the filter tensor $\boldsymbol{F}$ as a whole, this design utilizes the GSM space to buffer the blocking data of $\boldsymbol{F}$. In this section, the subscripts $\boldsymbol{sm}$, $\boldsymbol{am}$, and $\boldsymbol{gsm}$ indicate the on-chip storage space positions of the tensors corresponding to the SM, AM, and GSM spaces, respectively. To load the relevant subblocks into their respective on-chip storage spaces, the corresponding dimensions of the filter tensor, input feature tensor, and output feature tensor must be divided for the GSM, SM, and AM spaces, labeled with the subscripts $\boldsymbol{gb}$, $\boldsymbol{sb}$, and $\boldsymbol{ab}$, respectively.

In the directConv1x1Fwd() algorithm, the blocking data of $\boldsymbol{F}$ is stored in the GSM space, while the blocking data of tensors $\boldsymbol{I}$ and $\boldsymbol{O}$ need to be loaded from DDR to the SM space and AM space, respectively, during internal iterative calculations. To prevent simultaneous reading of both tensors from DDR during calculation, this section establishes the conditions $HW_{ab} = HW_{sb} = HW_{ob}$ and

$C_{dsb} \leqslant C_{dab}$ to balance the block parameters of the SM space and the AM space. In total, we derive the on-chip storage blocking limit conditions as presented in Eq. 4.

$$\begin{aligned}
sizeof(\boldsymbol{F_{gsm}}) &\leqslant sizeof(\text{GSM}) \\
sizeof(\boldsymbol{I_{sm}}) &\leqslant sizeof(\text{SM}) \\
sizeof(\text{AM}) &\geqslant sizeof(\boldsymbol{O_{am}}) + sizeof(\boldsymbol{F_{am}}) \\
sizeof(\boldsymbol{F_{gsm}}) &= C_{dgb} \times K_{dgb} \times L \times L \\
sizeof(\boldsymbol{I_{sm}}) &= C_{dsb} \times HW_{ob} \times L \\
sizeof(\boldsymbol{O_{am}}) &= K_{dab} \times HW_{ob} \times L \\
sizeof(\boldsymbol{F_{am}}) &= K_{dab} \times C_{dab} \times L \times L \\
C_{dsb} &\leqslant C_{dab} \leqslant C_{dgb} \leqslant C_d \\
K_{dab} &\leqslant K_{dgb} \leqslant K_d \\
HW_{ob} &\leqslant H_o \times W_o
\end{aligned} \tag{4}$$

To optimize the data locality in on-chip memories, the loop order in the original direct implementation of pointwise convolution was rearranged to achieve the loop order in directConv1x1Fwd(). The outermost two loops, c_{gd} and k_{dg}, are utilized to load the largest subblock of $\boldsymbol{F}$ into on-chip storage at once. If the size of $\boldsymbol{F}$ does not match the size of the allocated AM space for $\boldsymbol{F_{am}}$, i.e., $C_{kad} \times K_{dab} \neq K_d \times C_d$, then the $\boldsymbol{F}$ subblock will be cached in the GSM space $\boldsymbol{F_{gsm}}$ using DMA (Line 5). Otherwise, $\boldsymbol{F}$ will be directly loaded into the AM space $\boldsymbol{F_{am}}$ (Line 7). The three loops, k_{da}, n, and hw, are employed to load and store the output feature map tensor $\boldsymbol{O}$, followed by the loop c_{da} to determine the subblock of $\boldsymbol{F_{gsm}}$ that needs to be loaded into the AM space $\boldsymbol{F_{am}}$. The innermost loop, c_{ds}, is used to identify the subblock of the input feature map tensor $\boldsymbol{I}$ that must be loaded from DDR space to the SM space $\boldsymbol{I_{sm}}$. Within the c_{ds} loop, a subblock of $\boldsymbol{I}$ is loaded into the SM space using DMA, and then Conv1x1FwdAsm() is called once with the loaded data to perform the calculation.

Vectorization and Register Blocking. The employed second optimization technique is vectorization and register blocking. Once the relevant subblocks of tensors are loaded into the SM and AM spaces, effectively utilizing the execution units within a single DSP core to reduce computational costs becomes a critical concern. The objective of this approach is to minimize the runtime of Conv1x1FwdAsm() by maximizing the computational capacity of each DSP core. Specifically, it utilizes vectorization to harness the power of the 16 parallel VPEs in the VPU of each DSP core. Furthermore, register blocking techniques are utilized to conceal the pipeline latency of the VPU's execution units and take advantage of multiple vector floating-point multiply-add fusion units (VFMAC) within the VPU.

Vectorization is applied along the K dimension where the calculation associated with each element is independent, and there are L consecutive elements when accessing the K dimension of the related tensors ($\boldsymbol{O}$ and $\boldsymbol{F}$). To enhance data locality in registers, this method employs register blocking in the K_{dab}, HW_{ob}, and C dimensions, as described in Algorithm 2, and fully unrolls the cc, j, and i loops (Lines 8, 9, and 11) to conceal pipeline latency. The implementation of register blocking is subject to limitations imposed by the number of registers and the pipeline latency of the relevant functional units, as specified in Eq. 5, where $\text{Latency}_{\text{VFMAC}}$ and $\text{Latency}_{\text{FP32Bcast}}$ represent the latency time of the VFMAC units and FP32 Broadcasting, and $\text{Num}_{\text{VFMAC}}$ represents the number of the VFMAC units in VPUs of DSP cores.

Algorithm 2: Vectorized algorithm for the forward propagation of point-wise convolutions based on SPU and VPU in each DSP core

```
   Input  : $\boldsymbol{I}_{sm}[C_{dsb}][HW_b][L]$, $\boldsymbol{F}_{am}[C_{dsb} \times L][K_{dab}][L]$
   Output: $\boldsymbol{O}_{am}[K_{dab}][HW_b][L]$
1  for $k_d = 0: K_{drb}: K_{dab}$ do
2      for $hw = 0: HW_{rb}: HW_{ob}$ do
           // Load the $\boldsymbol{O}_{am,k_d,hw}$ subblock into the vector register
3          for $j = 0: 1: K_{drb}$ do
4              for $i = 0: 1: HW_{rb}$ do
5                  $\text{VR}_{j \times HW_{rb}+i} = \text{VLoad}(\boldsymbol{O}_{am,k_d+j,hw+i})$
6          for $cb = 0: 1: C_{dsb}$ do
7              for $cl = 0: C_{lrb}: L$ do
                   // The following loop will be fully unrolled in the
                      assembly implementation
8                  for $cc = cl: 1: cl + C_{lrb}$ do
9                      for $j = 0: 1: K_{drb}$ do
10                         $\text{VR}_f = \text{VLoad}(\boldsymbol{F}_{am,cb \times L+cc,k_d+i})$
11                         for $i = 0: 1: HW_{rb}$ do
12                             $\text{VR}_s = \text{SVBcast}(\text{FEXT}(\text{SLoad}((\boldsymbol{I}_{sm,cb,hw+j,cc}))))$
13                             $\text{VR}_{j \times HW_{rb}+i} = \text{VFMAC}(\text{VR}_f, \text{VR}_s, \text{VR}_{j \times HW_{rb}+i})$
           // Store the data in the vector register to $\boldsymbol{O}_{\text{am},k_d,hw}$
14         for $j = 0: 1: K_{drb}$ do
15             for $i = 0: 1: HW_{rb}$ do
16                 $\text{VStore}(\text{VR}_{j \times HW_{rb}+i}, \boldsymbol{O}_{am,k_d+j,hw+i})$
```

$$\begin{aligned} K_{drb} \times HW_{rb} &\geqslant \text{Latency}_{\text{VFMAC}} \times \text{Num}_{\text{VFMAC}} \\ K_{drb} \times HW_{rb} \times C_{lrb} &\geqslant \text{Latency}_{\text{FP32Bcast}} \times \text{Num}_{\text{VFMAC}} \end{aligned} \tag{5}$$

Multi-core Parallelization and Blocking Size Calculation. The third optimization method involves distributing tasks on multiple DSP cores and determining the appropriate block sizes for computation. In the algorithm for multi-level parallel implementation of pointwise convolution forward propagation, the calculation tasks are partitioned based on two loops: n and hw. A task pool is created, where each DSP core independently handles a task from the pool. The tasks from the task pool are processed in parallel by eight DSP cores until all tasks are completed.

In the previous parts, we have discussed the constraints that govern the blocking sizes of on-chip and register storage in this study. However, determining the appropriate block sizes remains an unresolved issue. The selected blocking sizes not only affect the efficiency of tensor access but also influence the overall data communication between off-chip and on-chip memories in the directConv1x1Fwd() algorithm. In the deep neural network library for the FT-M7032 heterogeneous general-purpose multi-core DSP, tensors are stored in the row-major format. After applying blocking, tensors require cross-stride reading. Larger blocking sizes in the tensor's inner dimensions facilitate more efficient access when using cross-stride reading. The Eq. 6 presents the calculation of the total amount of data transferred between off-chip and on-chip storage in directConv1x1Fwd(), where $sizeof(\boldsymbol{F})$, $sizeof(\boldsymbol{I})$, and $sizeof(\boldsymbol{O})$ denote the sizes of tensors $\boldsymbol{F}$, $\boldsymbol{I}$, and $\boldsymbol{O}$, respectively. Therefore, we calculate the blocking size in directConv1x1Fwd() while satisfying the conditions specified in Eqs. 4 and 5, guided by the following three principles. First, ensure that the larger blocking parameter is an integer multiple of the smaller blocking sizes (e.g., HW_{ob} must be an integer multiple of HW_{rb}). Second, minimize the value of $\text{Total}_{\text{conv1x1FwdS1}}$ as much as possible. Third, maximize the blocking size of the tensor's inner dimensions.

$$\text{Total}_{\text{conv1x1FwdS1}} = sizeof(\boldsymbol{F}) + \frac{K_d}{K_{dab}} \times sizeof(\boldsymbol{I}) + sizeof(\frac{C_d}{C_{dgb}}) \times \boldsymbol{O} \quad (6)$$

3.3 Multi-level Parallel Algorithms for Backward Propagation and Weight Gradient Update Propagation

The backward propagation pass of pointwise convolution involves taking the output feature map gradient $\boldsymbol{dO}$ and the convolution kernel $\boldsymbol{F}$ as input tensors and generating the input feature map gradient $\boldsymbol{dI}$ as the output tensor, shown in Eq. 2. The filter gradient $\boldsymbol{dF}$ is computed from the output feature maps gradient $\boldsymbol{dO}$ and the input feature maps $\boldsymbol{I}$ in the weight gradient update pass, shown in Eq. 3. The computational mode of the two passes above also is the matrix multiplication. Compared to the forward propagation pass, the main difference is that the matrix multiplications involve the matrix transposition in these two passes. Therefore, we get the multi-level parallel direct algorithms for the left two passes of pointwise convolutions, based on the parallel optimization approaches described in Sect. 3.2 and the vectorization matrix transpose kernel trnKernel-32 on multi-core DSPs proposed in [18].

4 Performance Evaluation

This section gives the test results of our direct implementation on multi-core DSPs and compares it with other implementations of pointwise convolutions on FT-M7032.

4.1 Experiment Setup

We chose ResNet50 [6] and MobileNetV1 [8] as representatives of widely-used backbone networks and lightweight networks, respectively. The performance of the pointwise convolution implementation is evaluated by employing the pointwise convolution layers from these models. The specific configurations are presented in Table 1. For the pointwise convolution tests, a batch size N of 64 is used for all tested network layers.

This subsection introduces three metrics, namely computing time T_{conv}, computing performance P_{conv}, and computing efficiency E_{conv}, to evaluate the performance of convolution implementations. The relation among these metrics is outlined in Eq. 7. P_{peak} represents the peak performance of a given hardware platform, such as a single GPDSP cluster and a 16-core ARMv8 CPU. Additionally, $TotalOp_{conv}$ is the total floating-point operations involved in the convolution computation. For pointwise convolutions, the formula for $TotalOp_{conv}$ is given by $2 \times N \times K \times H_o \times W_o \times C \times 1 \times 1$.

$$\begin{aligned} P_{conv} &= \frac{TotalOp_{conv}}{T_{conv}}, \\ E_{conv} &= \frac{P_{conv}}{P_{peak}}. \end{aligned} \tag{7}$$

4.2 Performance

This section compares the direct implementation of pointwise convolutions with two GEMM-based implementations on FT-M7032. The first is a GEMM-based implementation method optimized for multi-core DSPs [25], in which matrix multiplication and all tensor transformations run on multi-core DSPs. The second is the GEMM-based implementation in Pytorch [17], which runs solely on the 16-core ARMv8 CPU of FT-M7032. These two GEMM-based implementations are referred to as ftmEconv and Pytorch-conv, respectively. Furthermore, we compare the performance of the forward propagation pass with ARM Computer Library (ACL), which does not implement the left two passes. The absolute performance of three passes in different implementations of pointwise convolutions running on the FT-M7032 processor is presented in Figs. 3, 4, and 5. We can find that our direct implementation outperforms all the other implementations on FT-M7032. In addition, ftmDconv-Dlt avoids all additional memory overhead in ftmEconv and Pytorch-Conv.

Table 1. The parameter configuration of the pointwise convolutional layers

Layer ID	Model	$C \times H_i \times W_i$	$H_f \times W_f$	S	P
1	Resnet50 [6]	$64 \times 56 \times 56$	$64 \times 1 \times 1$	1	0
2		$64 \times 56 \times 56$	$256 \times 1 \times 1$	1	0
3		$256 \times 56 \times 56$	$64 \times 1 \times 1$	1	0
4		$256 \times 56 \times 56$	$128 \times 1 \times 1$	1	0
5		$128 \times 28 \times 28$	$512 \times 1 \times 1$	1	0
6		$512 \times 28 \times 28$	$128 \times 1 \times 1$	1	0
7		$512 \times 28 \times 28$	$256 \times 1 \times 1$	1	0
8		$256 \times 14 \times 14$	$1024 \times 1 \times 1$	1	0
9		$1024 \times 14 \times 14$	$256 \times 1 \times 1$	1	0
10		$1024 \times 14 \times 14$	$512 \times 1 \times 1$	1	0
11		$512 \times 7 \times 7$	$2048 \times 1 \times 1$	1	0
12		$2048 \times 7 \times 7$	$512 \times 1 \times 1$	1	0
13	Mobilenetv1 [8]	$32 \times 112 \times 112$	$64 \times 1 \times 1$	1	0
14		$64 \times 56 \times 56$	$128 \times 1 \times 1$	1	0
15		$128 \times 56 \times 56$	$128 \times 1 \times 1$	1	0
16		$128 \times 28 \times 28$	$256 \times 1 \times 1$	1	0
17		$256 \times 28 \times 28$	$256 \times 1 \times 1$	1	0
18		$256 \times 14 \times 14$	$512 \times 1 \times 1$	1	0
19		$512 \times 14 \times 14$	$512 \times 1 \times 1$	1	0
20		$512 \times 7 \times 7$	$1024 \times 1 \times 1$	1	0
21		$1024 \times 7 \times 7$	$1024 \times 1 \times 1$	1	0

Figure 3 shows the computational performance of the forward propagation pass in four implementations, where the horizontal axis denotes the layer ID of different pointwise convolutional layers and the vertical axis represents the computational performance P_{conv} obtained by each implementation. The results indicate that ftmDconv-Pt achieves performance ranging from 336.57 GFlops to 1593.51 GFlops, resulting in a computational efficiency of 12.17% to 57.64%. Notably, ftmDconv-Pt has a significant speedup of 5.93 times to 35.84 times and 3.76 times to 24.07 times when compared with Pytorch-Conv and ACL algorithms, respectively. In the comparison with ftmEconv, the speedup is in the range of 1.55 times to 5.57 times, and the main reason for the observed performance speedup is that the direct implementation has no additional memory overhead and shows much better on-chip data locality.

For the backward propagation pass, we also compare the computational performance P_{conv} of ftmDconv-Pt with that of ftmEconv and Pytorch-Conv on all the tested network layers, as shown in Fig. 4. The ftmDconv-Pt implementation achieves performance ranging from 315.76 GFlops to 1620.33 GFlops, resulting in a computational efficiency of 11.42% to 58.61%. When compared with Pytorch-

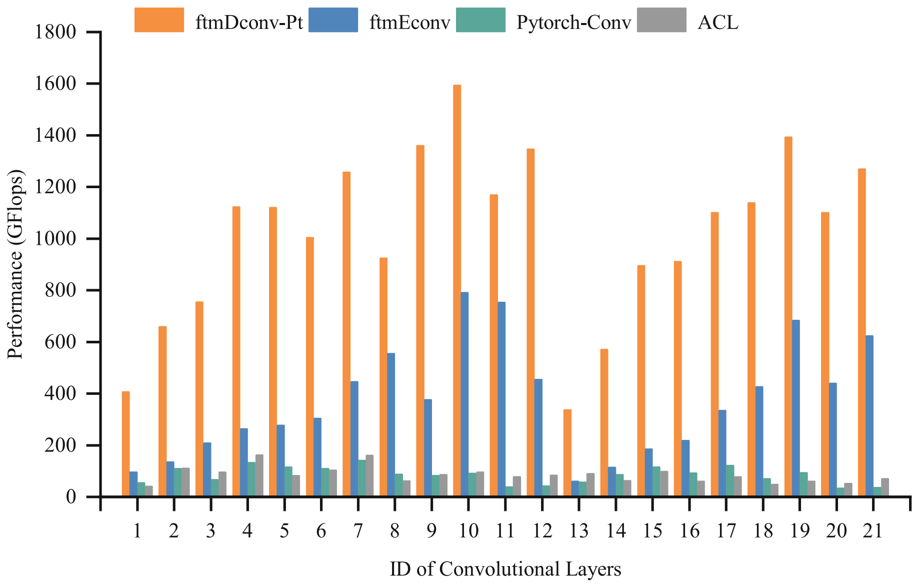

Fig. 3. Performance of various forward propagation algorithms for pointwise convolutions on FT-M7032 processors

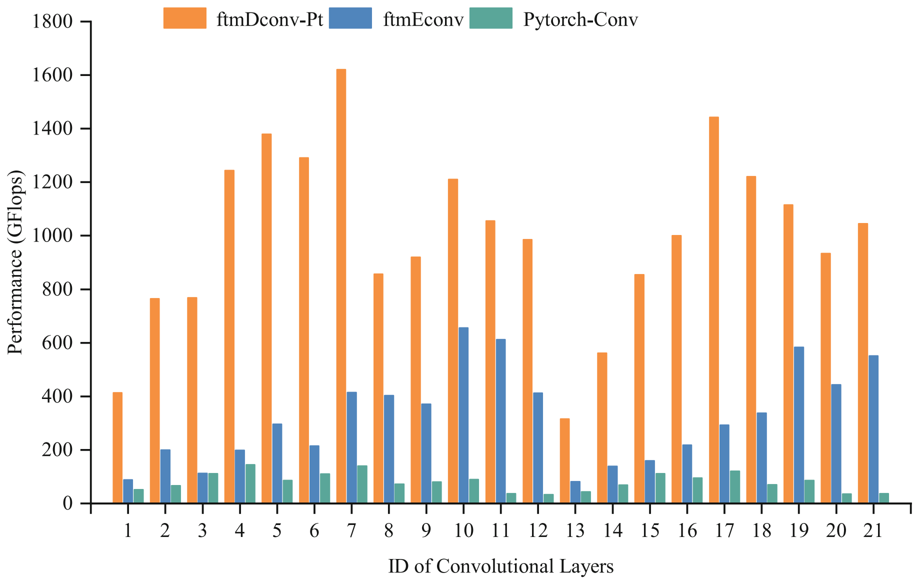

Fig. 4. Performance of various backward propagation algorithms for pointwise convolutions on FT-M7032 processors

Conv, ftmDconv-Pt achieves a significant speedup of 6.90 times to 29.14 times. In the comparison with ftmEconv, the maximum speedup is 6.80 times.

Figure 5 compares the computational performance P_{conv} of the direct implementation of the weight gradient update pass with that of ftmEconv and Pytorch-Conv on all the tested network layers. For all the tested network layers, ftmDconv-Pt achieves the performance of 366.216 GFlops - 1582.35 GFlops, resulting in a computational efficiency of 13.24% - 57.23%. When compared with Pytorch-Conv, ftmDconv-Pt obtains a speedup of 2.66 times to 13.27 times. In the comparison with ftmEconv, the maximum speedup is 79.26 times.

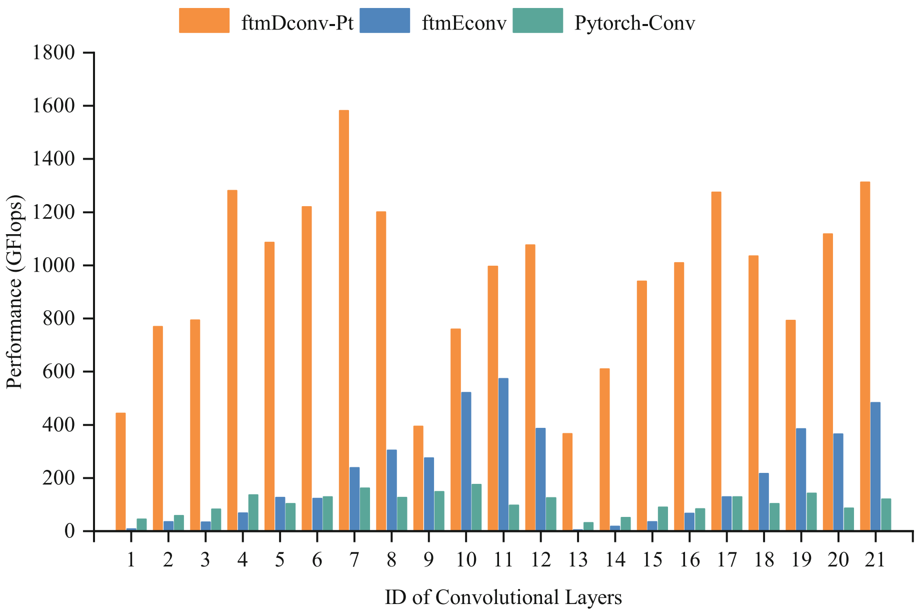

Fig. 5. Performance of various weight gradient update algorithms for pointwise convolutions on FT-M7032 processors

5 Conclusions and Future Work

This paper presents a high-performance parallel algorithm for the direct implementation of pointwise convolutions on multi-core DSPs in FT-M7032 heterogeneous processors. The parallel implementation can take full advantage of the parallel functional units and multi-level on-chip memories in multi-core DSPs. The primary optimizations involve multi-level memory blocking, loop ordering, vectorization, and multi-core parallelization. The experimental results on pointwise convolutional layers of popular networks show the proposed direct implementation outperforms other implementations on FT-M7032 heterogeneous processors, and get the maximum speedup of up to 79.26 times.

In the future, we will focus on the direct implementations for other types of convolutions on multi-core DSPs.

References

1. Arm Corporation: Arm computer library: A software library for machine learning. https://www.arm.com/technologies/compute-library (2023). Accessed 3 Jan 2023
2. Chaudhary, N., et al.: Efficient and generic 1d dilated convolution layer for deep learning. arXiv preprint arXiv:2104.08002 (2021)
3. Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K., Yuille, A.L.: DeepLab: semantic image segmentation with deep convolutional nets, Atrous convolution, and fully connected CRFs. IEEE Trans. Pattern Anal. Mach. Intell. **40**(4), 834–848 (2017)
4. Chen, X., Liu, J., Pang, Y., Chen, J., Chi, L., Gong, C.: Developing a new mesh quality evaluation method based on convolutional neural network. Eng. Appl. Comput. Fluid Mech. **14**(1), 391–400 (2020)
5. Chetlur, S., et al.: CUDNN: efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
6. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90
7. Heinecke, A., et al.: Understanding the performance of small convolution operations for CNN on intel architecture. In: Poster in the International Conference for High Performance Computing, Networking, Storage, and Analysis (2017)
8. Howard, A.G., et al.: MobileNets: efficient convolutional neural networks for mobile vision applications. CoRR (2017)
9. Huang, X., Wang, Q., Lu, S., Hao, R., Mei, S., Liu, J.: Evaluating FFT-based algorithms for strided convolutions on ARMv8 architectures. Perform. Eval. **49**, 102248 (2021). https://doi.org/10.1016/j.peva.2021.102248
10. Huang, X., Wang, Q., Lu, S., Hao, R., Mei, S., Liu, J.: NUMA-aware FFT-based convolution on armv8 many-core CPUs. In: 2021 IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), pp. 1019–1026 (2021). https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00142
11. Igual, F.D., Ali, M., Friedmann, A., Stotzer, E., Wentz, T., van de Geijn, R.A.: Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC. In: SC 2012: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE (2012)
12. Kim, M., Park, C., Kim, S., Hong, T., Ro, W.W.: Efficient dilated-winograd convolutional neural networks. In: 2019 IEEE International Conference on Image Processing (ICIP), pp. 2711–2715. IEEE (2019)
13. Lu, G., Zhang, W., Wang, Z.: Optimizing GPU memory transactions for convolution operations. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER), pp. 399–403. IEEE (2020)
14. Lu, G., Zhang, W., Wang, Z.: Optimizing Depthwise separable convolution operations on GPUs. IEEE Trans. Parallel Distrib. Syst. **33**(1), 70–87 (2021)

15. Mehta, S., Rastegari, M., Caspi, A., Shapiro, L., Hajishirzi, H.: ESPNet: efficient spatial pyramid of dilated convolutions for semantic segmentation. In: Ferrari, V., Hebert, M., Sminchisescu, C., Weiss, Y. (eds.) ECCV 2018. LNCS, vol. 11214, pp. 561–580. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01249-6_34
16. Mogers, N., Radu, V., Li, L., Turner, J., O'Boyle, M., Dubach, C.: Automatic generation of specialized direct convolutions for mobile GPUs. In: Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit, pp. 41–50 (2020)
17. Paszke, A., et al.: Pytorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, vol. 32 (2019)
18. Pei, X., et al.: Optimizing parallel matrix transpose algorithm on multi-core digital signal processors (in Chinese). J. Natl. Univ. Defense Technol. **45**(1), 57–66 (2023)
19. Safonov, I., Kornilov, A., Makienko, D.: An approach for matrix multiplication of 32-bit fixed point numbers by means of 16-bit SIMD instructions on DSP. Electronics **12**, 78 (2022)
20. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv 2: Inverted residuals and linear bottlenecks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4510–4520 (2018)
21. Szegedy, C., et al.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
22. Wang, Q., Li, D., Huang, X., Shen, S., Mei, S., Liu, J.: Optimizing FFT-based convolution on ARMv8 multi-core CPUs. In: Malawski, M., Rzadca, K. (eds.) Euro-Par 2020. LNCS, vol. 12247, pp. 248–262. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57675-2_16
23. Wang, Q., Li, D., Mei, S., Lai, Z., Dou, Y.: Optimizing Winograd-based fast convolution algorithm on Pythium multi-core CPUs (in Chinese). J. Comput. Res. Dev. **57**(6), 1140–1151 (2020). https://doi.org/10.7544/issn1000-1239.2020.20200107
24. Wang, Q., Li, D., Mei, S., Shen, S., Huang, X.: Optimizing one by one direct convolution on ARMV8 multi-core CPUs. In: 2020 IEEE International Conference on Joint Cloud Computing, pp. 43–47. IEEE (2020). https://doi.org/10.1109/JCC49151.2020.00016
25. Wang, Q., et al.: Evaluating matrix multiplication-based convolution algorithm on multi-core digital signal processors (in Chinese). J. Natl. Univ. Defense Technol. **45**(1), 86–94 (2023). https://doi.org/10.11887/j.cn.202301009
26. Wang, Q., Songzhu, M., Liu, J., Gong, C.: Parallel convolution algorithm using implicit matrix multiplication on multi-core CPUs. In: 2019 International Joint Conference on Neural Networks (IJCNN), pp. 1–7 (2019). https://doi.org/10.1109/IJCNN.2019.8852012
27. Yin, S., Wang, Q., Hao, R., Zhou, T., Mei, S., Liu, J.: Optimizing irregular-shaped matrix-matrix multiplication on multi-core DSPs. In: 2022 IEEE International Conference on Cluster Computing (CLUSTER), pp. 451–461 (2022). https://doi.org/10.1109/CLUSTER51413.2022.00055
28. Yu, F., Koltun, V.: Multi-scale context aggregation by dilated convolutions. arXiv preprint arXiv:1511.07122 (2015)