CCF Transactions on High Performance Computing (2024) 6:646–664
https://doi.org/10.1007/s42514-024-00200-3

REGULAR PAPER

Optimizing depthwise separable convolution on DCU

Zheng Liu[1] · Meng Hao[1] · Weizhe Zhang[1,3] · Gangzhao Lu[2] · Xueyang Tian[1] · Siyu Yang[1] · Mingdong Xie[1] · Jie Dai[1] · Chenyu Yuan[1] · Desheng Wang[3] · Hongwei Yang[1]

Received: 24 July 2024 / Accepted: 19 October 2024 / Published online: 13 December 2024

Abstract
The integration of Large Language Models (LLMs) with Convolutional Neural Networks (CNNs) is significantly advancing the development of large models. However, the computational cost of large models is high, necessitating optimization for greater efficiency. One effective way to optimize the CNN is the use of depthwise separable convolution (DSC), which decouples spatial and channel convolutions to reduce the number of parameters and enhance efficiency. In this study, we focus on porting and optimizing DSC kernel functions from the GPU to the Deep Computing Unit (DCU), a computing accelerator developed in China. For depthwise convolution, we implement a row data reuse algorithm to minimize redundant data loading and memory access overhead. For pointwise convolution, we extend our dynamic tiling strategy to improve hardware utilization by balancing resource allocation among blocks and threads, and we enhance arithmetic intensity through a channel distribution algorithm. We implement depthwise and pointwise convolution kernel functions and integrate them into PyTorch as extension modules. Experiments demonstrate that our optimized kernel functions outperform the MIOpen library on the DCU, achieving up to a 3.59× speedup in depthwise convolution and up to a 3.54× speedup in pointwise convolution. These results highlight the effectiveness of our approach in leveraging the DCU's architecture to accelerate deep learning operations.

Keywords Depthwise separable convolution · Deep computing unit · Dynamic tiling · Channel distribution

1 Introduction

Since AlexNet (Krizhevsky et al. 2012) achieved a tremendous breakthrough in the ILSVRC (Russakovsky et al. 2015) in 2012, convolutional neural networks (CNNs) have demonstrated exceptional performance in a variety of tasks, such as image recognition, video processing and object detection (Szegedy et al. 2015; He et al. 2016; Redmon et al. 2016; Real et al. 2019; Bochkovskiy et al. 2020; GAO et al. 2022; LU and ZHENG 2023). In recent years, the great

✉ Weizhe Zhang
wzzhang@hit.edu.cn

Zheng Liu
zhengliu@stu.hit.edu.cn

Meng Hao
haomeng@hit.edu.cn

Gangzhao Lu
lugangzhao@cnaeit.com

Xueyang Tian
23s003122@stu.hit.edu.cn

Siyu Yang
yangsiyu1102@gmail.com

Mingdong Xie
goldenpotato137@hit.edu.cn

Jie Dai
daijiehit@foxmail.com

Chenyu Yuan
chenyuy001@gmail.com

Desheng Wang
wangdesheng@hit.edu.cn

Hongwei Yang
yanghongwei@hit.edu.cn

[1] Faculty of Computing, Harbin Institute of Technology, Harbin 150001, China

[2] China Nanhu Academy of Electronics and Information Technology, Jiaxing 314001, China

[3] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen 518055, China

success of transformer-based large language models (LLMs) (Vaswani et al. 2017; Devlin et al. 2018; Brown et al. 2020; Sun et al. 2021; JI et al. 2023; Achiam et al. 2023; Zhao et al. 2023) with hundreds of billions of parameters has significantly pushed the boundaries of natural language processing techniques, driving the growing scale of models. To combine the advantages of both powerful architectures, some studies (Dai et al. 2021; Yuan et al. 2021; Srinivas et al. 2021) integrate CNNs for local feature extraction with transformer-based models for establishing long-range dependencies, resulting in better generalization capability. These new applications indicate that CNNs remain highly relevant and useful in modern AI research.

Despite their success, large models require massive computing power and are computationally expensive (Thompson et al. 2020; Shoeybi et al. 2019) during the training and inference processes. As the size and complexity of models increase, the need for reducing computational cost becomes critical. Many researchers have focused on this field and proposed various methods targeting large models (Li et al. 2020; Yao et al. 2022; Zhu et al. 2023). One effective solution to reduce the computational cost for CNNs is the use of depthwise separable convolution (DSC) (Chollet 2017). This technique decomposes the standard convolution operation into two simpler operations: a depthwise convolution followed by a pointwise convolution. By doing so, it significantly reduces the number of computations and parameters required, leading to more efficient models. DSCs have been successfully implemented in various architectures, such as MobileNet (Howard et al. 2017; Sandler et al. 2018) and EfficientNet (Tan and Le 2019), demonstrating substantial improvements in computational efficiency without compromising performance.

Many works have focused on optimizing DSCs to further improve their efficiency and performance. Lu et al. (2021) designed a data reuse algorithm to reduce memory access latency and a dynamic tiling algorithm to improve hardware utilization. Wu and Huang (2019) presented methods to improve data reusability by managing the execution order of matrix multiplication and to reduce data transfer overhead by fusing layers. Qin et al. (2018) introduced a diagonalwise refactorization method to address low GPU utilization and accelerate depthwise convolution. Wei et al. (2022) proposed an optimized separable convolution, which features an optimal design for the number of groups and filter sizes compared to standard DSC. Beyond studies on NVIDIA GPUs, Bai et al. (2018) implemented optimization methods on FPGAs. Indeed, the landscape of hardware development has diversified, with numerous companies now developing their own computing accelerators, e.g. the Google Tensor Processing Unit (TPU) (Jouppi et al. 2017). In response to the escalating demands for developing domestic computing accelerators, the Deep Computing Unit (DCU) was developed by Hygon (Hygon 2023) for the Chinese supercomputer and AI market.

A DCU works as a hardware extension to the CPU host system via a PCI-E connection. The most critical component of a DCU is its computing units, as shown in Fig. 1. A significant hardware design difference from the GPU is that DCU groups 64 threads into a wavefront, which serves as the basic scheduling unit, whereas GPU organizes 32 threads into a warp. This can affect the hardware resource allocation and parallelism. On the software side, GPUs utilize the CUDA (Compute Unified Device Architecture) (Guide 2020) platform developed by NVIDIA, while DCUs are built upon AMD's open-source ROCm software stack and use the HIP (Heterogeneous-Compute Interface for Portability) programming model (AMD 2024). These differences present challenges when porting code from GPUs to DCUs, as we need to re-implement code in the new programming model and consider the hardware resources of the new device to optimize performance.

Our study aims to identify optimization opportunities and port our previous optimization methods (Lu et al. 2021) to the DCU, an area that has been under-explored in previous research, thereby contributing to the development of a robust ecosystem for this new device. For depthwise convolutions, we employ a row data reuse algorithm to minimize unnecessary memory access overhead. For pointwise convolutions, we extend and modify the existing dynamic tiling strategy and implement an automatic optimization pipeline.

We evaluate our methods by implementing HIP kernel functions for depthwise and pointwise convolutions and

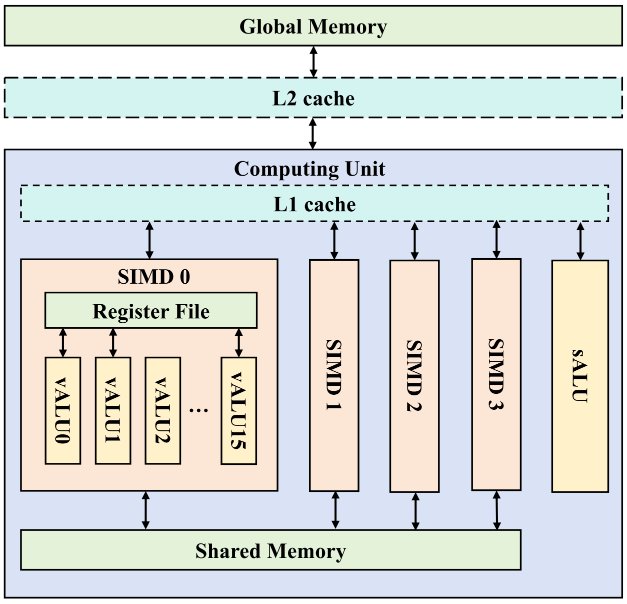


Fig. 1 Main architecture of a computing unit in DCU. Best viewed in color

comparing them to the MIOpen library (Khan et al. 2019), which provides high-performance machine learning primitives in the AMD ROCm stack. Based on these kernel functions, we implement extension modules for PyTorch (Paszke et al. 2019), a widely-used deep learning library in both industry and academia and use native PyTorch convolution modules as the benchmark. Experiments demonstrate that our depthwise convolution kernel functions achieve up to a 3.59× speedup, and pointwise convolution kernel functions achieve up to a 3.54× speed up compared to those in MIOpen. For the extension modules, the new depthwise convolution extensions achieve a speedup of up to 4.54×, while the new pointwise convolution extensions achieve a speedup of up to 1.78×. All experiments were conducted on a DCU provided by Sugon's cloud computing platform.

In this study, we make the following key contributions:

- We explore the optimization of DSC on DCU and implement additional HIP kernel functions to demonstrate the generality and applicability of our methods across different configurations.
- We develop PyTorch extension modules based on optimized kernel functions, facilitating their use in both academic research and industrial applications.
- We conduct experiments on a DCU, validating our approach and quantifying the performance gains.

In Sect. 2, we briefly discuss the DCU hardware architecture and depthwise separable convolution. In Sect. 3, we present an overview of our study. In Sects. 4 and 5, we elaborate on optimization strategies for depthwise and pointwise convolutions, respectively. We show experimental results in Sect. 6 and discuss future work in Sect. 7. Finally, in Sect. 8, we conclude the paper.

Online Material. The source code of this work is publicly available at https://github.com/HIT-HPC-Group/DSCOptimization.

2 Background

In this section, we briefly introduce the Deep Computing Unit and the Depthwise Separable Convolution.

2.1 Deep computing unit

The Deep Computing Unit (DCU) is a computing accelerator developed and launched domestically in China by Hygon (Hygon 2023). Designed with a GPU-like architecture, it features low latency and high throughput and is suitable for highly parallel tasks, typically operating as a coprocessor to the CPU. In this setup, the host-side program runs on the CPU while the device-side kernel functions run on the DCU. Although the DCU is scheduled as a device by the host CPU system, it independently manages its computing units, memory system, and thread scheduling, maintaining relative autonomy during execution.

Computing units are the most crucial components in a DCU. The DCU in our experiment comprises 64 independent computing units. As illustrated in Fig. 1, each computing unit contains 4 SIMD units, and each SIMD unit has 16 ALUs. When the DCU is executing, threads are assigned to these ALUs, with 64 threads grouped into a wavefront as the basic execution unit. This structure enables all threads within a wavefront to execute the same instruction simultaneously, thereby enhancing computational efficiency.

Figure 1 also shows the memory hierarchy of the DCU, which is similar to that of the GPU (Mei and Chu 2016). The global memory is independent of the host system's memory and is used to store data for the computing units. To meet the demands for high throughput, the DCU supports advanced HBM2 memory (Jun et al. 2017), providing over 16 GB of space with bandwidth up to 1 TB/s. Although global memory offers the largest capacity on the DCU, it also has the highest access overhead, often becoming the main performance bottleneck in many programs. The next level of memory hierarchy is the shared memory within each computing unit. Each computing unit has 64 KB of shared memory accessible by the thread blocks living on it. Shared memory can be utilized as a fast cache controlled by the developer. By loading necessary data into shared memory, threads can avoid multiple accesses to the slower global memory (Xu et al. 2009). On the contrary, L1 and L2 caches cannot be programmed directly, but by carefully managing data access patterns to improve data locality, programmers can leverage the caching system to enhance program performance. Next, each SIMD unit provides registers for threads, which have the shortest access latency. Each thread can use up to 256 registers. For compute-intensive tasks, frequently used data can be stored in registers to further reduce memory access overhead (Iandola et al. 2013). Data in registers can also be transferred between threads using specific APIs. Both registers and shared memory are crucial hardware resources for threads, as these resources are limited, which in turn limits the number of active threads and thus the degree of parallelism. Our study leverages the feature of the memory hierarchy to optimize performance.

The DCU utilizes the AMD ROCm software stack, which includes the HIP (Heterogeneous Interface for Portability) C/C++ based programming model and runtime library (AMD 2024). A typical HIP program involves transferring data from the host to the device, launching kernel functions on the device, and copying the results back to the host for further processing. To launch a kernel function, developers need to configure the grid size and the block size. The device then determines the index for each thread based on these parameters. Because hardware resources are limited and arbitrary configuration can reduce hardware utilization, finding a balanced configuration is crucial for maximizing performance. This is especially important when optimizing pointwise convolution.

Recently, a growing number of studies have focused on developing optimization techniques for various applications running on the new DCU platform. For instance, Liu et al. (2024) introduced D-TADOC, a compressed data direct computing method for Chinese datasets on the DCU. Their approach accelerates the processing of Chinese text data by designing a parallel processing module specifically for the DCU architecture. Ma et al. (2022) optimized the Quantum Fourier Transform (QFT) algorithm by reducing communication overhead between the host and device while enhancing thread activity on the DCU. Our work aligns with these efforts, as we aim to reduce data movement overhead across the memory hierarchy and refine thread assignment strategies for improved efficiency. Furthermore, Zhou et al. (2023) present algorithmic improvements at the compiler level, incorporating DCU hardware characteristics to adjust thread allocation. Our tiling method also takes hardware resources into account to achieve better workload balance. In addition, Guo et al. (2024) focus on optimizing Sparse General Matrix-Matrix Multiplication (SpGEMM). Their solution improves load balancing, maximizes the utilization of registers and shared memory, and enhances global load distribution through fine-grained grouping and kernel configurations.

2.2 Depthwise separable convolution

Depthwise separable convolution can improve computational efficiency while maintaining inference accuracy. This technique was popularized by the MobileNet architecture, which demonstrated its effectiveness in reducing the number of parameters and computational cost (Howard et al. 2017).

In a conventional convolution operation, each filter is applied to all input channels, and the results are summed to produce the output feature map, as illustrated in Fig. 2. If there are M input channels and N filters, and assuming a stride of 1 to maintain the same spatial dimensions for the output as the input, the computational complexity of the operation is $O(M \times N \times K \times K \times H \times W)$, where K is the filter size, and H and W are the height and width of both the input and output feature maps, respectively. This process involves a substantial number of multiply-add operations, which makes it computationally expensive.

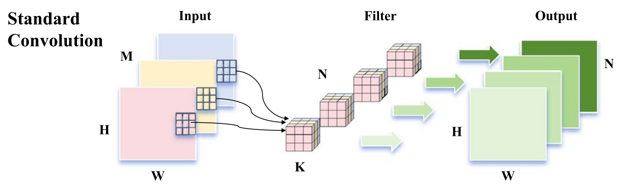


Fig. 2 Illustration of standard convolution. Best viewed in color

On the other hand, depthwise separable convolution decomposes the traditional convolution operation into two steps: depthwise convolution and pointwise convolution. This is presented in Fig. 3. In depthwise convolution, a single-channel filter is applied to each input channel separately. This means that if there are M input channels, there will be M separate spatial convolutions. The computational cost of depthwise convolution is $O(M \times K \times K \times H \times W)$. After depthwise convolution, a 1×1 pointwise convolution is applied. This operation squeezes and combines the depthwise convolution's output along channel dimension. If there are N filters, the computational complexity is $O(M \times N \times H \times W)$. In total, depthwise separable convolution significantly reduces the computational cost to $O(M \times K \times K \times H \times W + M \times N \times H \times W)$. This reduction leads to fewer parameters and operations, making depthwise separable convolution an efficient alternative to traditional convolution.

Depthwise separable convolutions have been successfully utilized in various networks, including MobileNet and EfficientNet, which are lightweight and suitable for deployment on resource-constrained devices, such as embedded systems and mobile phones. Our work further optimizes the performance of this efficient operation.

3 Overview

We optimize depthwise and pointwise convolution separately. The optimization framework is shown in Fig. 4.

For depthwise convolution, we adopt the row data reuse algorithm (Lu et al. 2021). The motivation is to maximize

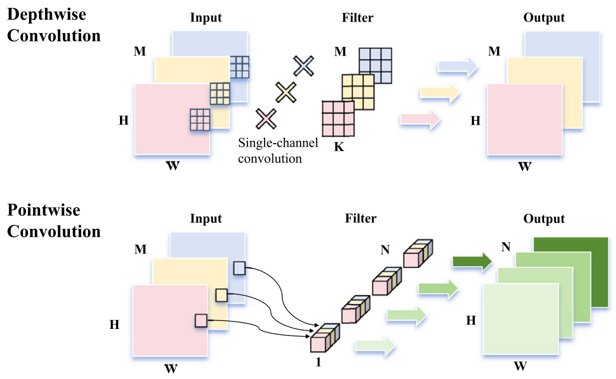


Fig. 3 Illustration of depthwise separable convolution. Best viewed in color

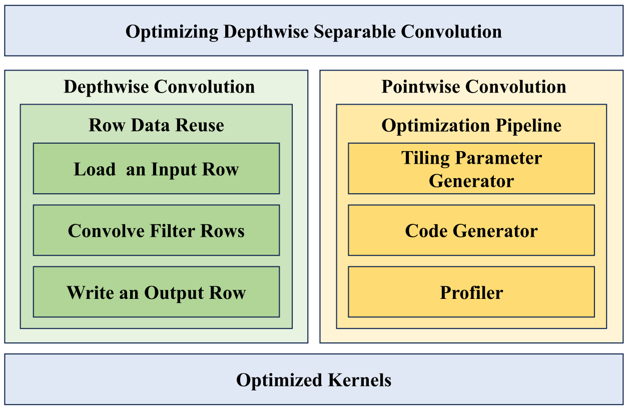


Fig. 4 Optimization Framework. Best viewed in color

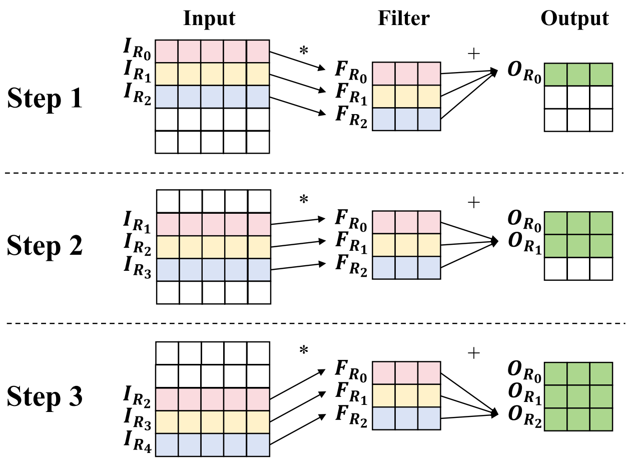

Input
Filter
Output
Step 1
Step 2
Step 3
I_{R_0} I_{R_1} I_{R_2} I_{R_3} I_{R_4}
F_{R_0} F_{R_1} F_{R_2}
O_{R_0} O_{R_1} O_{R_2}
*
+

Fig. 5 Illustration of naive depthwise convolution. Best viewed in color

the reuse of loaded data, thereby minimizing redundant memory access overhead. Such a strategy enhances data locality within a row and significantly improves the performance, as demonstrated in experiments.

For pointwise convolution, the objective is to increase hardware utilization and improve data arithmetic intensity. We adapt and extend our previous dynamic tiling strategy (Lu et al. 2021) to achieve balanced hardware allocation among threads and blocks and the channel distribution algorithm to enhance the reuse of loaded data in multiple operations. Moreover, we implement a three-stage optimization pipeline, integrating the model-and-profile approach to select the optimal configuration. The pipeline consists of three main components:

1 Tiling Parameter Generator: This component defines all relevant tiling parameters, with candidate values depending on both hardware limitations and problem size. Given the hardware resources, constraints are applied to discard invalid tiling configurations.
2 Code Generator: The code generator processes each viable configuration to produce the corresponding kernel function code. A key part of this stage is the channel distribution algorithm, which increases data arithmetic intensity by reusing loaded data, and the double buffering mechanism, which reduces data loading latency.
3 Profiler: The profiler measures the execution time of kernel functions generated from each tiling configuration and selects the fastest one.

4 Depthwise convolution optimization

In this section, we elaborate on the **row data reuse algorithm** adopted from our previous work for optimizing depthwise convolution. We begin with a simple example to illustrate the data reloading problem and its impact on memory efficiency. For simplicity, we assume that the input and output data are single-channel and only one thread block is used. We then explain how the data reuse algorithm mitigates memory access overhead and provide a detailed description of the algorithm.

4.1 Data reloading problem

Assume that an input data with size 5×5 (including the padding size of 1) is convolved with a 3×3 filter. If the stride is 1, then the output data will be 3×3. We use a 3-thread block to compute one output row at each step. Then one thread is assigned to one output element in the row, and they load required input and filter data to compute the output. This simple row by row computation process is shown in Fig. 5.

I, F and O denotes the input, filter and output data, respectively. R_i represents i-th row of the data. Initially, I_{R_0}, I_{R_1}, and I_{R_2} are loaded and perform convolution with F_{R_0}, F_{R_1}, and F_{R_2} to compute O_{R_0}. Then O_{R_1} and O_{R_2} are calculated in a similar manner by loading different input and filter rows. Mathematically, this process is expressed as:

$$O_{R_0} = I_{R_0} * F_{R_0} + I_{R_1} * F_{R_1} + I_{R_2} * F_{R_2}$$
$$O_{R_1} = I_{R_1} * F_{R_0} + I_{R_2} * F_{R_1} + I_{R_3} * F_{R_2}$$
$$O_{R_2} = I_{R_2} * F_{R_0} + I_{R_3} * F_{R_1} + I_{R_4} * F_{R_2}$$

Clearly, the central input rows are loaded multiple times. As the number of filter rows increases, the repetition also increases. These repeated loads lead to unnecessary memory access overhead, which negatively impacts performance. An intuitive mitigation strategy is to leverage the relatively faster shared memory to hold prefetched input data, allowing threads to load data from shared memory. However, the data reloading pattern still exists.

4.2 Row data reuse

To eliminate the reloading pattern and enhance efficiency, we shift from the output-centric approach to an input-centric method by reordering the computation process. This allows each loaded input row to be reused as many times as possible in multiple operations. Specifically, after loading an input row, it performs convolutions with multiple filter rows to generate intermediate results for multiple output rows that depend on this input row. Intermediate results are temporarily stored in registers to mitigate potential latency caused by frequent writing access to the output matrix in global memory. As this process continues, corresponding partial results for each output row are accumulated in the registers, and some rows complete their accumulation, then they can write the final result to global memory and release the registers. Then these registers can be used to store intermediate results for future output rows, ensuring a cyclic and limited register usage.

This process is illustrated in Fig. 6. Initially, upon loading I_{R_0}, it only performs an convolution with F_{R_0} because only O_{R_0} requires this partial result. However, after loading I_{R_2}, it is used to produce partial results for O_{R_0}, O_{R_1}, and O_{R_2} by performing convolutions with F_{R_2}, F_{R_1}, and F_{R_0}, respectively. At this point, O_{R_0} finishes accumulating, so the result can be written to the output matrix in global memory, and the registers are used to hold new partial results for O_{R_3} in future steps. The process continues, with each step efficiently using loaded data to compute necessary partial results until all are accumulated for each output row. The new computation process can be expressed as:

$$\begin{aligned}
&Load\ I_{R_0} : O_{R_0} = I_{R_0} * F_{R_0}\\
&Load\ I_{R_1} : O_{R_0} = O_{R_0} + I_{R_1} * F_{R_1}\\
&\qquad\qquad O_{R_1} = I_{R_1} * F_{R_0}\\
&Load\ I_{R_2} : O_{R_0} = O_{R_0} + I_{R_2} * F_{R_2} \rightarrow Write\ O_{R_0}\\
&\qquad\qquad O_{R_1} = O_{R_1} + I_{R_2} * F_{R_1}\\
&\qquad\qquad O_{R_2} = I_{R_2} * F_{R_0}\\
&Load\ I_{R_3} : O_{R_1} = O_{R_1} + I_{R_3} * F_{R_2} \rightarrow Write\ O_{R_1}\\
&\qquad\qquad O_{R_2} = O_{R_2} + I_{R_3} * F_{R_1}\\
&Load\ I_{R_4} : O_{R_2} = O_{R_2} + I_{R_4} * F_{R_2} \rightarrow Write\ O_{R_2}
\end{aligned}$$

Previously, in the simple implementation, input rows are loaded 9 times in total. In contrast, this optimized implementation requires only 5 loads. By decomposing the computation for each output row into multiple steps as input rows are loaded, this strategy significantly reduces the number of data loads, thereby minimizing memory access overhead and enhancing overall computation efficiency.

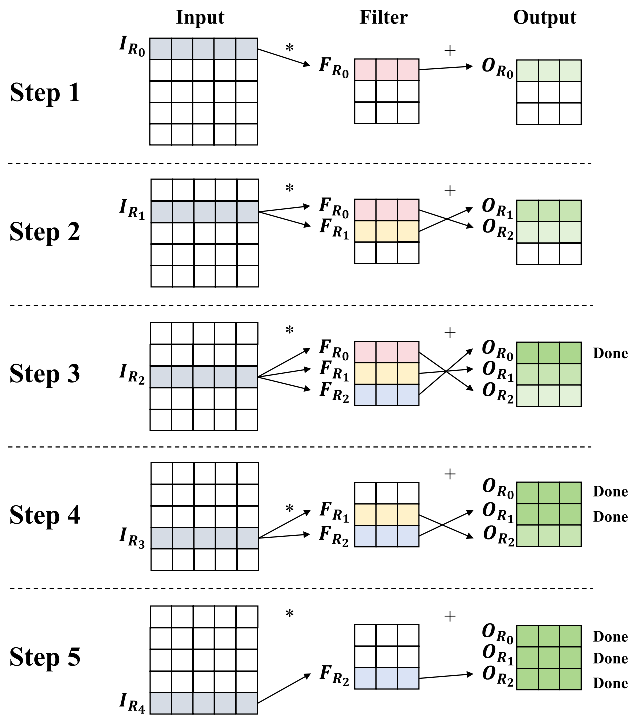


Fig. 6 Illustration of row reuse algorithm. Best viewed in color

Algorithm 1 RowDataReuse

```
Require: $I, F$
Ensure: $O = DepthwiseConv(I, F)$
1:  for $i \leftarrow 0$ to $I_H - 1$ do # each row
2:      Load Row $I[i]$;
3:      if $i < F_H - 1$ then
4:          for $j \leftarrow 0$ to $i + 1$ do
5:              $O[j] \leftarrow O[j] + I[i] \cdot F[i - j]$;
6:          end for
7:      else if $F_H - 1 \le i < I_H - F_H + 1$ then
8:          for $j \leftarrow 0$ to $F_H$ do
9:              $r \leftarrow i - F_H + j + 1$;
10:             $O[r] \leftarrow O[r] + I[i] \cdot F[F_H - 1 - j]$;
11:         end for
12:     else
13:         for $j \leftarrow F_H - 1$ to 0 do
14:             $r \leftarrow I_H - F_H + 1$;
15:             $O[r] \leftarrow O[r] + I[i] \cdot F[F_H - j]$;
16:         end for
17:     end if
18: end for
```

As described in Algorithm 1, it is important to note that for the input rows located at the edges, they are not convolved with all filter rows, unlike those positioned centrally. These edge cases require meticulous handling to prevent invalid memory accesses. In practice, when a data batch contains multiple samples and each sample has multiple channels, the DCU can launch a large number of thread blocks to process them in parallel. Within each block, a group of threads processes one output channel, as described in the algorithm. And each block can contain multiple groups to handle multiple output channels in parallel. Data is prefetched from global memory to shared memory by each thread block, ensuring efficient data loading and reuse by threads. Experimental results suggest that the row data reuse algorithm can substantially accelerate depthwise convolution, and we present the results in Sect. 6.

5 Pointwise convolution optimization

In this section, we introduce our optimization methods for pointwise convolution and implementation of model-and-profile three-stage optimization pipeline.

5.1 Tiling parameter generator

When launching a kernel function, developers need to configure the grid size and block size, which influence the amount of shared memory allocated to each block. Additionally, the way a thread block processes a data tile can also affect the hardware usage per thread. Given that shared memory and registers are crucial yet limited resources provided by a computing unit, arbitrary allocation and tiling can lead to low hardware utilization and poor performance. To address this, we employ the **dynamic tiling strategy** that defines some parameters to describe the workload for each block and wavefront. To find viable parameter combinations, we define resource constraints based on the available resources and problem size to eliminate unachievable ones.

5.1.1 Identify tiling parameters

Our dynamic tiling strategy is model-based. We describe the data tiling for thread blocks and wavefronts using a two-level tiling strategy. The first level tiling represents the part of the output data to be processed by each thread block. Within a block, the data is further partitioned to be processed by wavefronts, as the second level tiling. This approach extends our previous work by introducing more tiling parameters, which in turn affect the constraints of resource usage. We introduce the parameters in a bottom-up way.

To describe a wavefront's workload, we use $Wave_W$ and $Wave_C$ to represent the width and channel of the output data tile handled by a wavefront (i.e. second-level tiling), respectively. Therefore, the total number of elements in the output data tile for a wavefront is $Wave_W \times Wave_C$. Note that, in order to calculate the output tile, the wavefront is responsible for the corresponding input tile with the same $Wave_W$ width. To minimize control divergence problem (Xiang et al. 2014), which makes those allocated hardware resources wasted, we require candidate values for these parameters to be integer factors of the output data sizes. For example, if the output width is 16, then $Wave_W$ can be 1, 2, 4, 8, 16. This ensures that the data is always partitioned exactly and wavefronts receive balanced workloads.

Next, we use $Wave_N$ to denote the number of wavefronts contained in each thread block. Given that the wavefront size is fixed at 64 on current DCUs, the block size is $Wave_N \times 64$. These wavefronts can be arranged in various ways to process different data tiles. To describe this layout, we use the $(Repeat_W, Repeat_C)$ pair to represent the number of wavefronts arranged along the width and channel directions of the output data. This represents a significant modification to our previous work. Now, $Wave_N$ can have multiple candidate values, rather than being fixed at 4, and wavefronts can be arranged in various configurations, rather than just (2, 2). Since the maximum block size is 1024 in the current DCU design, the maximum candidate value for $Wave_N$ is $\frac{1024}{64} = 16$. Moreover, we require $Wave_N = Repeat_W \times Repeat_C$. For a given $Wave_N$, there can be multiple layout strategies, described by the $(Repeat_W, Repeat_C)$ pairs. Additionally, we ensure that the candidate values do not lead to control divergence. Figure 7 illustrates this with an example where $Wave_N = 4$, $(Repeat_W = 2, Repeat_C = 2)$, $Wave_W = 4$, and $Wave_C = 2$.

As the second-level tiling of each wavefront is described, the first level tiling is simply aggregate the workload of the wavefronts. The size of the data tile processed by a thread block (i.e. first-level tiling) is calculated as $Wave_W \times Wave_C \times Repeat_W \times Repeat_C$. All thread blocks are arranged along the height, width, and channel directions in the output data. Consequently, the grid size of the kernel function is calculated as $\frac{Total\ Output\ Elements}{Wave_W \times Wave_C \times Repeat_W \times Repeat_C}$.

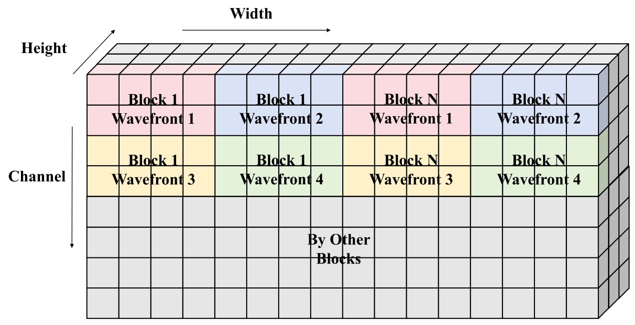


Fig. 7 Example of wavefronts logical layout on output. Best viewed in color

5.1.2 Identify resource parameters

In addition to the tiling parameters, we define two resource parameters that determine the resource allocation constraints of a computing unit and the resource usage of blocks and threads. These parameters, used together with the tiling parameters, help eliminate unachievable tiling cases. We discuss these constraints in Sect. 5.1.3.

The first resource parameter is $Block_N$, which describes the number of thread blocks scheduled onto one computing unit. Since the DCU explicitly limits the maximum number of wavefronts per computing unit to 40, the candidate values for $Block_N$ depend on $Wave_N$.

The second resource parameter is $Group_C$, which is used in the **channel distribution algorithm** (Lu et al. 2021) to describe the number of input channels grouped together. It affects the number of filter elements processed by a thread, denoted as $Thread_C$, which in turn affects the register usage. We use a concrete example to explain the channel distribution algorithm and how $Group_C$ changes register usage.

Similar to the row data reuse algorithm discussed in Sect. 4.2, the main motivation behind channel distribution is to improve arithmetic intensity, defined as $\frac{number\ of\ multiplications}{number\ of\ loaded\ elements}$. Higher arithmetic intensity helps to hide memory access overhead by overlapping computation with data loading. Specifically, we achieve this by distributing input and filter elements to the threads in a wavefront.

Assume that a wavefront handles 8×64 output data, so $Wave_W = 8$ and $Wave_C = 64$, and the input data has 56 channels, resulting in a filter of size 56×64, where there are 64 output channels and each output channel has 56 elements.

In a simple pointwise convolution process, each thread can be assigned to calculate an output channel with 8 output elements in a row. Accordingly, each thread is responsible for one filter channel with 56 elements. Each time, a thread loads 8 input data elements in a row and 1 filter element from its filter channel. By multiplying each input element with the filter element, a total of 8 partial output results are calculated. This process is repeated 56 times to accumulate the final result, as illustrated in Fig. 8. With this simple method, at each step, each thread loads 8 input elements and 1 filter element, resulting in a total of 9 elements. These 9 elements are used in 8 multiplication operations to produce 8 partial results. Thus, the arithmetic intensity is $\frac{8}{9}$, and the register usage is low.

The channel distribution method, as shown in Fig. 9, can improve arithmetic intensity. Assume that $Group_C = 8$, so every 8 input channels ($Wave_W \times Group_C = 8 \times 8 = 64$ input elements in total) are grouped to be processed by the wavefront at each time. The goal is to convolve them with

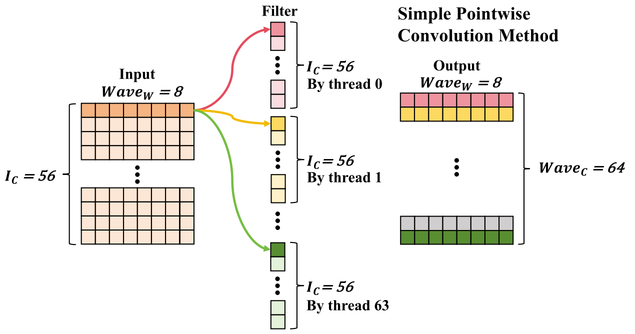


Fig. 8 Illustration of simple pointwise convolution method. Best viewed in color

corresponding filter elements in each channel ($Wave_C \times Group_C = 64 \times 8 = 512$ elements in total) to produce partial results. With many threads available, these filter elements are distributed among threads. If we regard every 8 consecutive elements in the filter channel as a group, then threads 0 to 7 take a group from filter channel 0, threads 8 to 15 take a group from filter channel 1, and so on. Each thead takes 1 element. If each thread only loads 1 filter element, then a wavefront can load $F_{num} = \frac{Wavefront\ Size}{Group_C} = \frac{64}{8} = 8$ groups, meaning the wavefront can produce partial results for 8 output channels with these elements. However, to calculate partial results for all 64 output channels simultaneously, each thread must load more elements into more registers.

In our example, each thread needs to load $Thread_C = \frac{Wave_C}{F_{num}} = \frac{64}{8} = 8$ filter elements. Specifically, threads 0 to 7 take groups from filter channels 0, 8, 16, 24, 32, 40, 48, 56, while threads 8 to 15 load from filter channels 1, 9, 17, 25, 33, 41, 49, 57, and so on. Each thread takes one filter element from each group. Grouped input channels ($Group_C = 8$) are also distributed to threads row by row, according to the filter elements, so each thread also loads $Wave_W = 8$ input elements. Each input element is multiplied with each filter element to produce a partial output result ($Wave_W \times Thread_C = 64$ in total).

By distributing the workload, the arithmetic intensity is $\frac{Wave_W \times Thread_C}{Wave_W + Thread_C} = \frac{8\times8}{8+8} = 4$. This approach improves arithmetic intensity by more than 4 times. This process repeats $\frac{Input\ Channels}{Group_C} = 7$ times to process all 56 input channels. In the end, threads use segmented parallel reduction to accumulate the partial results. The challenging part of this algorithm is mapping threads onto data correctly.

This example illustrates how $Group_C$ and $Thread_C$ affect register usage, which is useful when applying resource

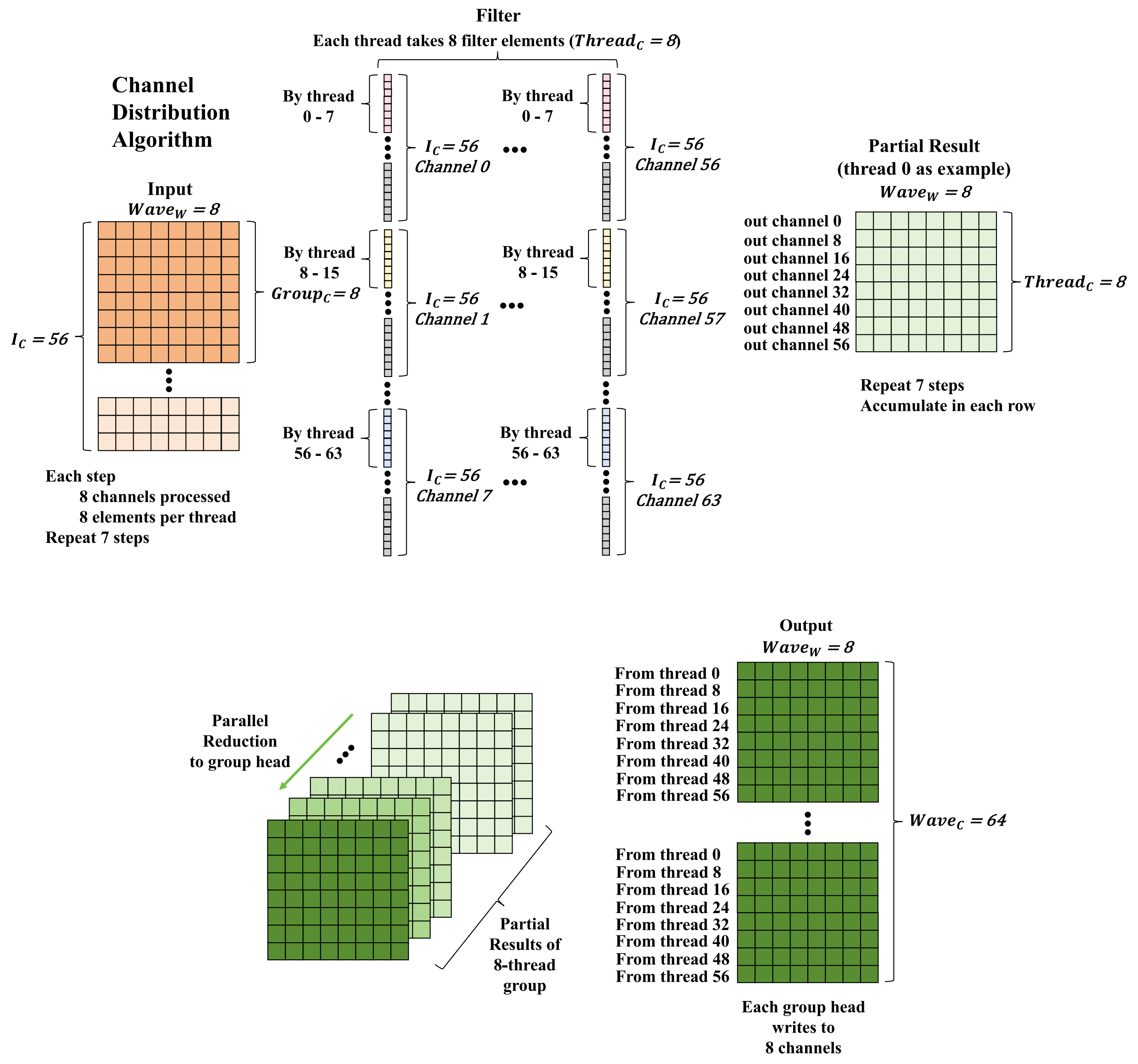


Fig. 9 Illustration of channel distribution algorithm. Top part shows how input and filter are distributed to threads, assuming that $Group_C = 8$. Bottom part shows how parallel reduction is performed to aggregate partial values, and how threads are assigned to write output data. Best viewed in color

Table 1 Summary of tiling and resource parameters

Parameter	Description
$Wave_W$	Output width handled by a wavefront
$Wave_C$	Output channel handled by a wavefront
$Wave_N$	No. of wavefronts in a block
$Repeat_W$	No. of wavefronts along width direction
$Repeat_C$	No. of wavefronts along channel direction
$Block_N$	No. of thread blocks per CU
$Group_C$	No. of grouped channels

constraints in Sect. 5.1.3. Table 1 lists all the tiling and resource parameters and their purposes as a short summary.

5.1.3 Apply resource constraints

To eliminate invalid tiling cases, we define two constraints based on the size of shared memory and the number of registers of a computing unit. We then calculate the resource usage of each tiling case, which must satisfy both constraints to be considered valid.

Firstly, we calculate the shared memory that can be allocated to each block, represented as *LimitS*, and the number of registers that can be allocated to each thread, denoted as *LimitR*, as follows:

$$LimitS = \frac{SMEM_{CU}}{Block_N} \tag{1}$$

$$LimitR = \frac{Reg_{CU}}{Block_N \times Wave_N \times 64} \tag{2}$$

where $SMEM_{CU} = 64$ KB is the size of shared memory and $Reg_{CU} = 65536$ is the number of registers on a computing unit. The value 64 in Eq. 2 is the wavefront size. Because wavefronts and blocks are work balanced, we also require the resources are partitioned equally.

Secondly, we calculate the size of shared memory allocated to a thread block, defined as:

$$\begin{aligned} UsedS = 4 \times (&Repeat_W \times Wave_W \times Group_C \\ &+ Repeat_C \times Wave_C \times Group_C) \times 2 \end{aligned} \tag{3}$$

where 4 represents the number of bytes in a floating-point number, and 2 accounts for the double buffering used. $Repeat_W \times Wave_W \times Group_C$ denotes the size of the loaded input data, and $Repeat_C \times Wave_C \times Group_C$ represents the size of the loaded filter data. Thus, the shared memory constraint is defined as:

$$UsedS \leq LimitS \tag{4}$$

Thirdly, we calculate the register usage. Each thread needs to compute partial results for $resultR = Wave_W \times Thread_C$ output elements, with the operands stored in $operandR = Wave_W + Thread_C$ registers. Moreover, because double buffering is used, a thread block uses registers as temporary fast cache by loading data into registers before moving them to shared memory. The number of these temporary registers used is calculated as follows:

$$\begin{aligned} tempR = &\left\lceil \frac{Repeat_W \times Wave_W \times Group_C}{Wave_N \times 64} \right\rceil \\ &+ \left\lceil \frac{Repeat_C \times Wave_C \times Group_C}{Wave_N \times 64} \right\rceil \end{aligned} \tag{5}$$

Equation 5 is similar to Eq. 3, because data is collaboratively loaded by threads in a block, and each thread contributes few registers for the block. Additionally, we leave 30 registers for the HIP compiler. The total number of registers used by a thread is defined as:

$$UsedR = resultR + operandR + tempR + 30 \tag{6}$$

To ensure that a tiling case satisfies the register usage, the second resource constraint is defined as:

$$UsedR \leq LimitR \tag{7}$$

By applying constraints defined in Eqs. 4 and 7, our tiling parameter generator efficiently produces potential tiling configurations that improve hardware utilization while adhering to the resource limitations of the DCU. To measure the performance of each configuration and identify the fastest one, we implement a code generator and a profiler.

5.2 Code generator and profiler

The code generator takes a potential tiling configuration as input and automatically generates kernel function code to be executed on the DCU. It calculates the grid size and block size for kernel function launch parameters based on the given configuration, as discussed in Sect. 5.1.1. Within the generator, it utilizes the channel distribution algorithm introduced in Sect. 5.1.2 to map the wavefronts to the correct output data tiles and allocate the necessary registers to hold operands, partial results, and temporary values. Additionally, it incorporates the double buffering mechanism to further exploit optimization opportunities. Data is transferred between global memory, shared memory, and registers to fully utilize the DCU's memory hierarchy, with the computation process proceeding as outlined in Algorithm 2.

Algorithm 2 Code Generator Workflow

```
Require: $Tiling\ Parameters, In, Filter$
Ensure: $Out = PointwiseConv(In, Filter)$
1: # Calculate all related parameters
2: $gridSize = \frac{Total\ output\ elements}{Wave_W \times Wave_C \times Repeat_W \times Repeat_C}$;
3: $blockSize = Wave_N \times 64$;
4: $InBuf1 = Repeat_W \times Wave_W \times Group_C$;
5: $InBuf2 = Repeat_W \times Wave_W \times Group_C$;
6: $FilterBuf1 = Repeat_C \times Wave_C \times Group_C$;
7: $FilterBuf2 = Repeat_C \times Wave_C \times Group_C$;
8: $opInR = Wave_W$;
9: $opFilterR = Thread_C$;
10: $tmpInR = \lceil \frac{Repeat_W \times Wave_W \times Group_C}{Wave_N \times 64} \rceil$;
11: $tmpFilterR = \lceil \frac{Repeat_C \times Wave_C \times Group_C}{Wave_N \times 64} \rceil$;
12: $resultR = Wave_W \times Thread_C$;
13: # Start Calculation
14: Map threads onto data;
15: Load $Group_C$ input to $InBuf1$;
16: Load $Group_C$ filter to $FilterBuf1$;
17: $\_\_syncthreads()$;
18: for $iter \leftarrow 0$ to $\frac{I_C}{2 \times Group_C}$ do
19:     Load $Group_C$ channels of input and filter into $tmpInR$ and $tmpFilterR$;
20:     Move $InBuf1 \rightarrow opInR$;
21:     Move $FilterBuf1 \rightarrow opFilterR$;
22:     $Each\ opInR \times opFilterR$;
23:     Accumulate into $resultR$;
24:     Move $tmpInR \rightarrow InBuf2$;
25:     Move $tmpFilterR \rightarrow FilterBuf2$;
26:     $\_\_syncthreads()$;
27:     # Double Buffer
28:     Repeat in this iteration,
29:     but swap $Buf1$ and $Buf2$;
30: end for
31: Use $\_\_shfl()$ to do segmented parallel reduction to accumulate the partial results;
32: Write to $Out$ by corresponding threads;
```

In the end, we implement a profiler to evaluate the performance of each kernel function, and select the fastest tiling configuration for a given problem as the final solution. With the automatic optimization pipeline, we significantly broaden the search space and speed up the whole development workflow.

6 Experiments

This section introduces the DCU experiment platform and experiments for evaluating our methods and results.

6.1 Platform

We conducted our experiments on Sugon cloud computing platform. The configuration of the computing node for experiments in this work is shown in Table 2. It has a Hygon C86 7285 32-core CPU. And we choose DTK–23.04 as our software development toolkit. The computing node supports a DCU Z100SM as the accelerator, which provides 16 GB of global memory and integrates 64 computing units.

6.2 Testing new kernel functions

6.2.1 Setup

In this experiment, we evaluate the performance of our approach against the MIOpen library (Khan et al. 2019) and calculate the speedup relative to the MIOpen kernel functions. MIOpen provides various convolution algorithms, including General Matrix Multiplication (GEMM) (Vasudevan et al. 2017; Li et al. 2019), Direct Convolution (Ferrari et al. 2023; Zhang et al. 2018), Fast Fourier Transform (FFT) indirect convolution (Li et al. 2019), Winograd indirect convolution (Yan et al. 2020), and Implicit GEMM (Wang et al. 2019), each with different performance depending on the problem size. To identify the fastest algorithm as a benchmark, we use the *miopenFindConvolutionForwardAlgorithm*() API function provided by MIOpen. We use the layer configurations from four popular depthwise separable networks, MobileNet V2, EfficientNet B0, MnasNet (Tan et al. 2019) and ShuffleNet V2 (Ma et al. 2018), which together include 30 different depthwise layers and 45 different pointwise layers. The batch sizes are set to 1, 8, 16, 32, and 64.

Tables 3 and 4 list the layer configurations used in this experiment. In the tables, I_C, I_H and I_W represent input channel, input height and input width, respectively. F_H and F_W denote the height and width of filter. In the end, O_C is the output channel.

6.2.2 Results

The performance comparison between kernel functions and the MIOpen library is shown in Figs. 10 and 11 for depthwise convolutions and pointwise convolutions, respectively.

For depthwise convolution kernel functions, we observe that as the batch size increases, the average time for our kernel functions remains significantly lower than that of the

Table 2 Experiment platform

Host System	
CPU	Hygon C86 7285 32-core
Memory	128 GB
Storage	480 GB
Operating System	CentOS release 7.6.1810
Development Toolkit	DTK−23.04
Device	
Type	DCU Z100SM
Computing Units	64
Shared Memory / CU	64 KB
Registers / CU	65,536
Global Memory	16 GB

Table 3 Configurations of depthwise convolution layers

	I_C	$I_H \times I_W$	$F_H \times F_W$	Stride
D1	32	112×112	3×3	1
D2	144	56×56	3×3	1
D3	240	28×28	5×5	1
D4	384	14×14	3×3	1
D5	960	7×7	3×3	1
D6	96	112×112	3×3	2
D7	240	28×28	3×3	2
D8, D9	480	14×14	$3 \times 3, 5 \times 5$	1
D10, D11	1152	7×7	$3 \times 3, 5 \times 5$	1
D12, D13	144	56×56	$3 \times 3, 5 \times 5$	2
D14, D15	192	28×28	3×3	1, 2
D16, D17	576	14×14	3×3	1, 2
D18, D19	672	14×14	5×5	1, 2
D20	72	56×56	3×3	1
D21	120	28×28	5×5	1
D22	24	28×28	3×3	1
D23	48	14×14	3×3	1
D24	96	7×7	3×3	1
D25	48	112×112	3×3	2
D26	72	56×56	5×5	2
D27	576	14×14	5×5	2
D28	24	56×56	3×3	2
D29	48	28×28	3×3	2
D30	96	14×14	3×3	2

MIOpen library. This demonstrates the effectiveness of our row data reuse algorithm in maintaining low memory access overhead, leading to substantial performance gains across all tested batch sizes. Table 5 shows the average speedup of the depthwise convolution kernel functions over MIOpen for each batch size. The average speedup is largest when the batch size is 16, indicating optimal utilization of the row data reuse strategy at this configuration.

Table 4 Configurations of pointwise convolution layers

	I_C	$I_H \times I_W$	O_C
P1	32	112×112	16
P2	16	112×112	96
P3	96	56×56	24
P4	24	56×56	144
P5	144	56×56	24
P6	144	28×28	32
P7	32	28×28	192
P8	192	28×28	32
P9	144	28×28	40
P10	40	28×28	240
P11	240	28×28	40
P12	192	14×14	64
P13	64	14×14	384
P14	384	14×14	64
P15	384	14×14	96
P16	96	14×14	576
P17	576	14×14	96
P18	240	14×14	80
P19	80	14×14	240
P20	480	14×14	80
P21	480	14×14	112
P22	112	14×14	672
P23	672	14×14	112
P24	576	7×7	160
P25	160	7×7	960
P26	960	7×7	160
P27	960	7×7	320
P28	320	7×7	1280
P29	672	7×7	192
P30	192	7×7	1152
P31	1152	7×7	192
P32	1152	7×7	320
P33	16	112×112	48
P34	48	56×56	24
P35	24	56×56	72
P36	72	56×56	24
P37	72	28×28	40
P38	40	28×28	120
P39	120	28×28	40
P40	480	14×14	96
P41	576	7×7	192
P42	24	28×28	24
P43	48	14×14	48
P44	96	7×7	96
P45	192	7×7	1024

For pointwise convolution kernel functions, MIOpen utilizes different algorithms for different batch sizes, resulting in a performance boost when the batch size is 16. However, the runtime of our kernel functions increases as the

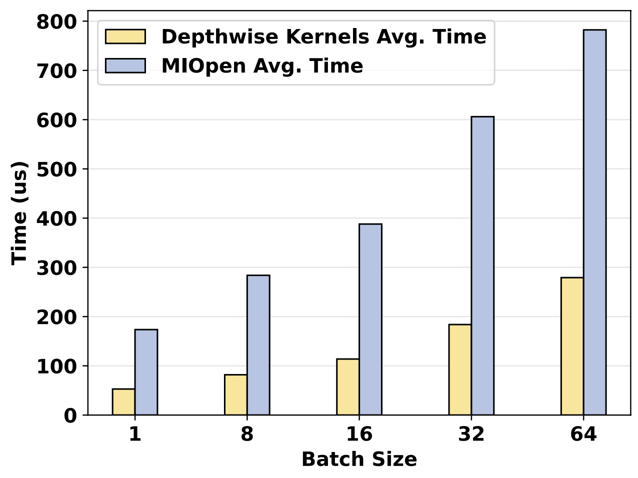


Fig. 10 Performance comparison between depthwise convolution kernel functions and MIOpen

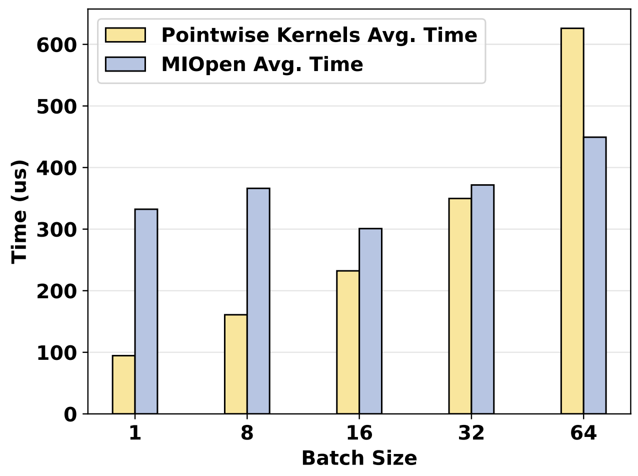


Fig. 11 Performance comparison between pointwise convolution kernel functions and MIOpen

Table 5 Average speed up of depthwise convolution kernel functions over MIOpen for different batch sizes

Batch Size	Ours	MIOpen	Speed Up
1	52.88 (μs)	173.59 (μs)	3.32
8	81.83 (μs)	283.82 (μs)	3.47
16	113.82 (μs)	387.94 (μs)	3.59
32	183.84 (μs)	606.12 (μs)	3.44
64	279.16 (μs)	782.29 (μs)	3.35

batch size grows. By improving hardware utilization, our optimized pointwise kernel functions run faster when the batch size is small, though the speedup decreases with larger batch sizes. Table 6 illustrates the average speedup

Table 6 Average speed up of pointwise convolution kernel functions over MIOpen for different batch sizes

Batch Size	Ours	MIOpen	Speed Up
1	94.55 (μs)	332.32 (μs)	3.54
8	160.96 (μs)	366.22 (μs)	2.48
16	232.26 (μs)	300.86 (μs)	1.49
32	349.77 (μs)	371.69 (μs)	1.31
64	626.09 (μs)	449.29 (μs)	0.9

of the pointwise convolution kernel functions compared to MIOpen for each batch size. The optimal performance is observed when the batch size is 1, with an average speedup of 3.54×.

In real-world inference scenarios, models typically process user requests as they arrive, forming small batches rather than the large, pre-processed batches used during training. This is especially important in real-time or interactive applications, where minimizing latency is critical. By optimizing for smaller batch sizes, we aim to ensure faster response times and enhance user experience by reducing delays.

6.3 Testing extension modules

6.3.1 Setup

We implement PyTorch extension modules based on the new kernel functions and evaluate the performance of them. PyTorch provides a C++ extension mechanism (Goldsborough 2024) that allows developers to create custom PyTorch operations separate from the PyTorch backend. Utilizing this mechanism, we implement backend C++ operations by wrapping our kernel functions as the forward pass functions. These backend operations are then bound to Python frontend using *pybind*11 (Jakob 2017). In the Python frontend, we further wrap the backend operations with *torch.autograd.Function* and *torch.nn.Module* to implement extension modules, making them callable as PyTorch modules. Our extension modules work in the same way as native PyTorch modules and take width, height and channel of input and output data as parameters.

To evaluate the performance, we feed random data with correct dimensions to the modules and set the batch sizes to 1, 8, 16, 32, and 64. Then we measure the forward pass execution time of our modules and compare them to those of PyTorch.

6.3.2 Results

Figures 12 and 13 show the performance comparison results between depthwise and pointwise convolution extension

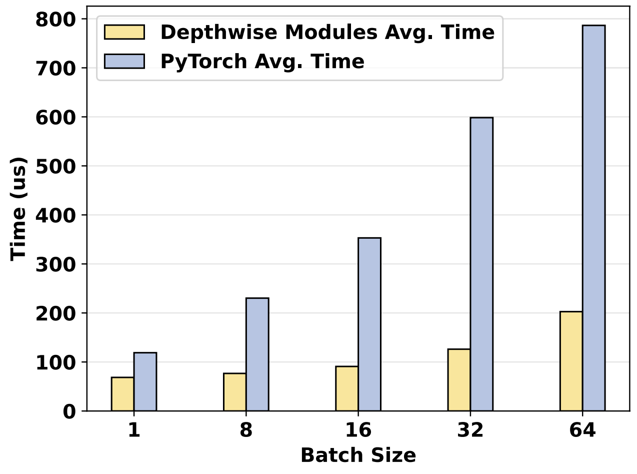


Fig. 12 Performance comparison between depthwise convolution extension modules and PyTorch

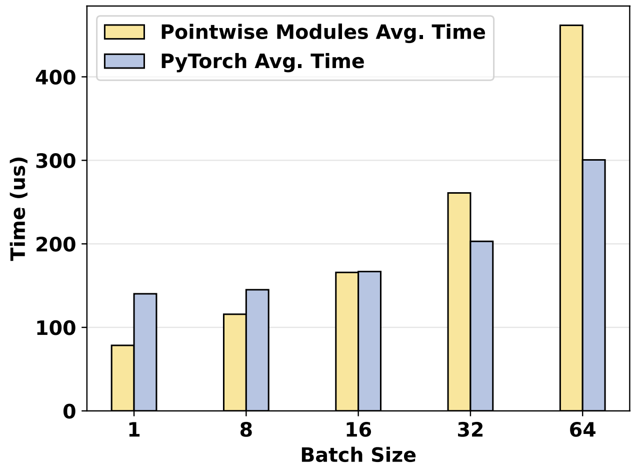


Fig. 13 Performance comparison between pointwise convolution extension modules and PyTorch

modules, respectively. Compared to the evaluation results for kernel functions, while our modules maintain similar performance, PyTorch runs much faster than MIOpen.

The depthwise convolution extension modules exhibit a consistent performance improvement over the PyTorch native modules. Table 7 highlights the speedup achieved, with the maximum observed at a batch size of 32, where our modules achieve a 4.54× average speedup.

For pointwise convolution extension modules, the performance varies with batch size. As shown in Fig. 13, our modules outperform PyTorch's native modules for smaller batch sizes, achieving an average speedup of 1.78× at a batch size of 1, as shown in Table 8. However, the performance improvement diminishes as the batch size increases, indicating that our methods can work well on small batch cases.

Table 7 Average speed up of depthwise convolution extension modules over PyTorch for different batch sizes

Batch Size	Module	PyTorch	Speed Up
1	68.52 (μs)	118.87 (μs)	1.73
8	76.66 (μs)	230.21 (μs)	2.92
16	90.88 (μs)	353.04 (μs)	3.7
32	126.09 (μs)	598.39 (μs)	4.54
64	202.65 (μs)	786.51 (μs)	4.49

Table 8 Average speed up of pointwise convolution extension modules over PyTorch for different batch sizes

Batch Size	Module	PyTorch	Speed Up
1	78.45 (μs)	140.21 (μs)	1.78
8	115.74 (μs)	145.06 (μs)	1.33
16	165.80 (μs)	166.84 (μs)	1.11
32	260.97 (μs)	202.99 (μs)	0.9
64	461.68 (μs)	300.53 (μs)	0.76

6.4 Ablation study

6.4.1 Setup

The primary goal of our ablation study is to assess how our depthwise and pointwise convolution optimization methods individually and jointly impact the performance of various networks. We select four representative networks for this study: EfficientNet B0, MnasNet, MobileNet V2 and ShuffleNet V2. Our setup is designed to minimize interference from non-convolution layers (such as batch normalization, pooling, and linear transformations) and reduce potential data transfer overhead, by isolating the depthwise and pointwise convolution layers and profiling their running time. To showcase the optimizing effect for the inference process with small batch size, we simply assume the batch is 1 here.

For each model, we run the following four cases:

- *Baseline*: We measure the performance of the original PyTorch depthwise and pointwise layers as a baseline.
- *Only Depthwise*: We replace the depthwise convolution layers with our optimized extension modules while retaining PyTorch's native pointwise layers.
- *Only Pointwise*: We replace the pointwise convolution layers with our optimized extension modules while retaining PyTorch's native depthwise layers.
- *Both Optimized*: We replace both depthwise and pointwise layers with our optimized extension modules.

This approach enables us to determine the impact of each optimization direction individually and in combination.

6.4.2 Results

Table 9 summarizes the results of our ablation study. For each model, we report the execution time of the baseline, the speedup obtained by optimizing only depthwise convolutions, the speedup from optimizing only pointwise convolutions, and the combined speedup from optimizing both.

From the results, we observe that optimizing depthwise layers alone yields a consistent performance improvement across all models, with a speedup around 1.20x, depending on the model. Similarly, optimizing only the pointwise layers also results in performance gains, with a speedup ranging from 1.47x to 1.51x. This can be attributed to the dynamic tiling and channel distribution strategies that improve hardware utilization for pointwise convolutions. When both depthwise and pointwise layers are optimized together, we observe the highest overall speedups, ranging from 1.71x to 1.98x. This demonstrates the cumulative benefits of reducing memory access and improving hardware efficiency across both types of convolutions. These results provide strong evidence that our proposed optimizations are effective in reducing the computational cost of depthwise separable convolutions in real-world deep learning models.

7 Discussion and future work

In this section, we discuss potential directions for future research.

Firstly, our optimization pipeline for pointwise convolution is model-and-profile-based. Utilizing a reinforcement learning-based method (Arulkumaran et al. 2017) can be a compelling alternative. One advantage of this solution is its flexibility to accommodate various layer configurations and hardware resource constraints. However, collecting the training data can be challenging and requires careful handling.

Secondly, we observe that the backpropagation steps cost longer time than the forward pass phase (Narayanan et al. 2019). This indicates that optimizing the backpropagation operations is also important for accelerating the overall training process, and this can be a focus of future work.

Thirdly, implementing and calling depthwise and pointwise convolutions separately in the model training and inference process can introduce additional context switch overhead. Using kernel fusion techniques (Wang et al. 2010) to combine depthwise and pointwise convolutions into a single kernel function can mitigate this overhead. This would reduce the number of kernel function calls, but it may couple the convolutions more tightly, sacrificing the modularity of the components.

Lastly, TensorRT (Jeong et al. 2021), specifically designed for NVIDIA GPUs, is a highly optimized deep learning library built on CUDA to accelerate inference. It employs various optimization techniques, including precision calibration, dynamic memory reuse, layer and tensor fusion, and kernel auto-tuning. While our work incorporates similar concepts, there are opportunities to enhance it further with additional strategies. More importantly, developing a high-performance inference library tailored for the DCU platform presents a promising research direction that could drive significant advancements.

Table 9 Ablation study result

Model	Baseline	Only Depthwise	Speed Up
EfficientNet	5.43 (ms)	4.54 (ms)	1.20
MNasNet	5.59 (ms)	4.71 (ms)	1.19
MobileNet	5.74 (ms)	4.87 (ms)	1.18
ShuffleNet	5.89 (ms)	4.99 (ms)	1.18
Model	Baseline	Only Pointwise	Speed Up
EfficientNet	5.43 (ms)	3.70 (ms)	1.47
MNasNet	5.59 (ms)	3.79 (ms)	1.48
MobileNet	5.74 (ms)	3.86 (ms)	1.49
ShuffleNet	5.89 (ms)	3.91 (ms)	1.51
Model	Baseline	Both Optimized	Speed Up
EfficientNet	5.43 (ms)	3.17 (ms)	**1.71**
MNasNet	5.59 (ms)	2.89 (ms)	**1.94**
MobileNet	5.74 (ms)	2.97 (ms)	**1.93**
ShuffleNet	5.89 (ms)	2.98 (ms)	**1.98**

8 Conclusion

We port our optimization methods for depthwise separable convolution from the GPU onto DCU, a computing accelerator developed in China. For depthwise convolution, we use the row data reuse algorithm to eliminate repeated data loading, thereby reducing memory latency and improving performance. For pointwise convolution, we modify the dynamic tiling strategy to enhance hardware utilization and utilized the channel distribution algorithm to increase arithmetic intensity for threads. Experimental results show that our optimization methods are effective for both depthwise and pointwise convolutions on DCU, especially when the batch size is small.

Acknowledgements This work is supported by the National Key Research and Development Plan (Grand No. 2023YFB4503205) and National Natural Science Foundation of China (Grant No. 62202123).

Data availability The data and code that support the findings of this study are openly available in repository DSCOptimization at https://github.com/HIT-HPC-Group/DSCOptimization.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no Conflict of interest.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv:2303.08774 (2023)

AMD: AMD ROCm^{TM} Documentation. https://rocm.docs.amd.com/en/latest Accessed 15 May 2024

Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: Deep reinforcement learning: A brief survey. IEEE Signal Process. Magaz. **34**(6), 26–38 (2017)

Bai, L., Zhao, Y., Huang, X.: A cnn accelerator on fpga using depthwise separable convolution. IEEE Trans. Circuits Syst. II: Express Briefs **65**(10), 1415–1419 (2018). https://doi.org/10.1109/TCSII.2018.2865896

Bochkovskiy, A., Wang, C.-Y., Liao, H.-Y.M.: Yolov4: Optimal speed and accuracy of object detection. arXiv:2004.10934 (2020)

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A.: Language models are few-shot learners. Adv. Neural Inf. Process. Syst. **33**, 1877–1901 (2020)

Chollet, F.: Xception: Deep learning with depthwise separable convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1251–1258 (2017)

Dai, Z., Liu, H., Le, Q.V., Tan, M.: Coatnet: Marrying convolution and attention for all data sizes. Adv. Neural Inf. Process. Syst. **34**, 3965–3977 (2021)

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805 (2018)

Ferrari, V., Sousa, R., Pereira, M., L. De Carvalho, J.a.P., Amaral, J.N., Moreira, J., Araujo, G.: Advancing direct convolution using convolution slicing optimization and isa extensions. ACM Trans. Architect. Code Opt. **20**(4), (2023) https://doi.org/10.1145/3625004

Gao, N., Yu, Y., Hua, X., Feng, F., Jiang, T.: A content-aware bitrate selection method using multi-step prediction for 360-degree video streaming. ZTE Commun. **20**(4), 96 (2022)

Goldsborough, P.: Custom C and cuda extensions. https://pytorch.org/tutorials/advanced/cpp_extension.html#custom-c-and-cuda-extensions Accessed 16 May 2024

Guide, D.: Cuda c++ programming guide. NVIDIA, July (2020)

Guo, H., Wang, H., Chen, W., Zhang, C., Han, Y., Zhu, S., Zhang, D., Guo, Y., Shang, J., Wan, T., et al.: Optimizing sparse general matrix–matrix multiplication for dcus. J. Supercomput. 1–25 (2024)

He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861 (2017)

Hygon: Deep Computing Unit. https://www.hygon.cn/product/accelerator Accessed 16 May 2024

Iandola, F.N., Sheffield, D., Anderson, M.J., Phothilimthana, P.M., Keutzer, K.: Communication-minimizing 2d convolution in gpu registers. In: 2013 IEEE International Conference on Image Processing, pp. 2116–2120 (2013). IEEE

Jakob, W.: Pybind11 Documentation. https://pybind11.readthedocs.io/en/stable/index.html Accessed 16 May 2024

Jeong, E., Kim, J., Tan, S., Lee, J., Ha, S.: Deep learning inference parallelization on heterogeneous processors with tensorrt. IEEE Embedded Syst. Lett. **14**(1), 15–18 (2021)

Ji, Y., Han, J., Zhao, Y., Zhang, S., Gong, Z.: Log anomaly detection through gpt-2 for large scale systems. ZTE Commun. **21**(3), 70 (2023)

Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T.V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C.R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J.: In-datacenter performance analysis of a tensor processing unit. (2017). arXiv: https://arxiv.org/pdf/1704.04760pdf

Jun, H., Cho, J., Lee, K., Son, H.-Y., Kim, K., Jin, H., Kim, K.: Hbm (high bandwidth memory) dram technology and architecture. In: 2017 IEEE International Memory Workshop (IMW), pp. 1–4 (2017). IEEE

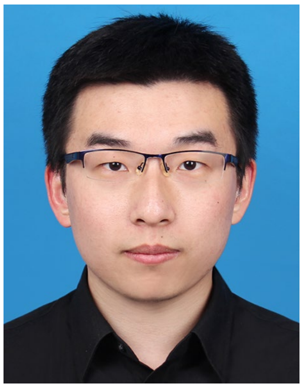
Khan, J., Fultz, P., Tamazov, A., Lowell, D., Liu, C., Melesse, M., Nandhimandalam, M., Nasyrov, K., Perminov, I., Shah, T., et al.:

Miopen: An open source library for deep learning primitives. arXiv:1910.00078 (2019)
Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. **25**, (2012)
Li, Z., Jia, H., Zhang, Y., Chen, T., Yuan, L., Cao, L., Wang, X.: Autofft: a template-based fft codes auto-generation framework for arm and x86 cpus. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2019)
Li, X., Liang, Y., Yan, S., Jia, L., Li, Y.: A coordinated tiling and batching framework for efficient gemm on gpus. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 229–241 (2019)
Li, Z., Wallace, E., Shen, S., Lin, K., Keutzer, K., Klein, D., Gonzalez, J.: Train big, then compress: Rethinking model size for efficient training and inference of transformers. In: International Conference on Machine Learning, pp. 5958–5968 (2020). PMLR
Liu, Y., Zhang, F., Pan, Z., Guo, X., Hu, Y., Zhang, X., Du, X.: Compressed data direct computing for chinese dataset on dcu. CCF Trans. High Perform. Comput. **6**(2), 206–220 (2024)
Lu, J., Zheng, Q.: Ultra-lightweight face animation method for ultra-low bitrate video conferencing. ZTE Commun. **21**(1), 64 (2023)
Lu, G., Zhang, W., Wang, Z.: Optimizing depthwise separable convolution operations on gpus. IEEE Trans. Parallel Distribut. Syst. **33**(1), 70–87 (2021)
Ma, N., Zhang, X., Zheng, H.-T., Sun, J.: Shufflenet v2: Practical guidelines for efficient cnn architecture design. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 116–131 (2018)
Ma, K., Han, L., Shang, J.-D., Xie, J.-M., Zhang, H.: Optimized realization of quantum fourier transform for domestic dcu accelerator. J Phys Conf Ser **2258**, 012065 (2022)
Mei, X., Chu, X.: Dissecting gpu memory hierarchy through microbenchmarking. IEEE Trans. Parallel Distribut. Syst. **28**(1), 72–86 (2016)
Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N.R., Ganger, G.R., Gibbons, P.B., Zaharia, M.: Pipedream: Generalized pipeline parallelism for dnn training. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 1–15 (2019)
Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Adv. Neural Inf. Process. Syst. **32**, (2019)
Qin, Z., Zhang, Z., Li, D., Zhang, Y., Peng, Y.: Diagonalwise refactorization: An efficient training method for depthwise convolutions. In: 2018 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2018). IEEE
Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. In: Proceedings of the Aaai Conference on Artificial Intelligence, vol. 33, pp. 4780–4789 (2019)
Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 779–788 (2016)
Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M.: Imagenet large scale visual recognition challenge. Int. J. Comput. Vision **115**, 211–252 (2015)
Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.-C.: Mobilenetv2: Inverted residuals and linear bottlenecks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4510–4520 (2018)
Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv:1909.08053 (2019)
Srinivas, A., Lin, T.-Y., Parmar, N., Shlens, J., Abbeel, P., Vaswani, A.: Bottleneck transformers for visual recognition. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 16519–16529 (2021)
Sun, Y., Wang, S., Feng, S., Ding, S., Pang, C., Shang, J., Liu, J., Chen, X., Zhao, Y., Lu, Y., et al.: Ernie 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation. arXiv:2107.02137 (2021)
Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., Le, Q.V.: Mnasnet: Platform-aware neural architecture search for mobile. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 2820–2828 (2019)
Tan, M., Le, Q.: Efficientnet: Rethinking model scaling for convolutional neural networks. In: International Conference on Machine Learning, pp. 6105–6114 (2019). PMLR
Thompson, N.C., Greenewald, K., Lee, K., Manso, G.F.: The computational limits of deep learning. arXiv:2007.05558**10** (2020)
Vasudevan, A., Anderson, A., Gregg, D.: Parallel multi channel convolution using general matrix multiplication. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 19–24 (2017). IEEE
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Adv. Neural Inf. Process. Syst. **30**, (2017)
Wang, G., Lin, Y., Yi, W.: Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In: 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing, pp. 344–350 (2010). IEEE
Wang, Q., Mei, S., 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). IEEE
Wei, T., Tian, Y., Wang, Y., Liang, Y., Chen, C.W.: Optimized separable convolution: Yet another efficient convolution operator. AI Open **3**, 162–171 (2022)
Wu, H.-N., Huang, C.-T.: Data locality optimization of depthwise separable convolutions for cnn inference accelerators. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 120–125 (2019). IEEE
Xiang, P., Yang, Y., Zhou, H.: Warp-level divergence in gpus: Characterization, impact, and mitigation. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 284–295 (2014). IEEE
Xu, C., Kirk, S.R., Jenkins, S.: Tiling for performance tuning on different models of gpus. In: 2009 Second International Symposium on Information Science and Engineering, pp. 500–504 (2009). IEEE
Yan, D., Wang, W., Chu, X.: Optimizing batched winograd convolution on gpus. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 32–44 (2020)
Yao, Z., Yazdani Aminabadi, R., Zhang, M., Wu, X., Li, C., He, Y.: Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. Adv. Neural Inf. Process. Syst. **35**, 27168–27183 (2022)
Yuan, K., Guo, S., Liu, Z., Zhou, A., Yu, F., Wu, W.: Incorporating convolution designs into visual transformers. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 579–588 (2021)
Zhang, J., Franchetti, F., Low, T.M.: High performance zero-memory overhead direct convolutions. In: International Conference on Machine Learning, pp. 5776–5785 (2018). PMLR

Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al.: A survey of large language models. arXiv:2303.18223 (2023)

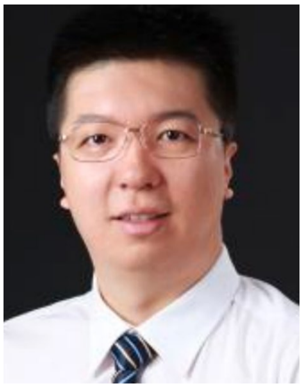
Zhou, Q.-W., Li, J.-N., Zhao, R.-C., Han, L., Wang, X.: Compilation optimization of dcu-oriented openmp thread scheduling. J Phys Conf Ser **2558**, 012003 (2023)

Zhu, X., Li, J., Liu, Y., Ma, C., Wang, W.: A survey on model compression for large language models. arXiv:2308.07633 (2023)



Zheng Liu earned his B.S. and M.Eng degree in Computer Engineering from the University of Illinois at Urbana-Champaign, USA, in 2020. He is currently pursuing his Ph.D. at the School of Computer Science and Technology at Harbin Institute of Technology, China. His research focuses on artificial intelligence, particularly on optimizing the training efficiency of distributed deep learning networks.

Meng Hao received the BS and Ph.D. degrees in computer science and engineering from Harbin Institute of Technology, China, in 2014 and 2021 respectively. He is currently an assistant professor in the School of Cyberspace Science, Harbin Institute of Technology. His research interests include high-performance computing, performance modeling, and parallel optimization.



Weizhe Zhang (Senior Member, IEEE) received B.Eng, M.Eng and Ph.D. degree of Engineering in computer science and technology in 1999, 2001 and 2006 respectively from Harbin Institute of Technology. He is currently a professor in the School of Cyberspace Science at Harbin Institute of Technology, China. His research interests are primarily in parallel computing, distributed computing, cloud and grid computing, and computer network.

Gangzhao Lu received the B.S. and Ph.D. degrees in computer science and engineering from Harbin Institute of Technology, China, in 2014 and 2022 respectively. His research interests include performance modeling, parallel optimization and auto-tuning.

Xueyang Tian received the bachelor degree in software engineering from Harbin Institute of Technology, China, in 2023. He is currently working toward a master degree in Harbin Institute of Technology. His research interests include high performance computing, and parallel optimization.

Siyu Yang received the bachelor degree in computer science and technology from Harbin Institute of Technology, China, in 2024. He is currently working toward a master degree in Harbin Institute of Technology. His research interests include high performance computing, and energy efficiency optimization.

Mingdong Xie earned his B.S. degree in Computational Mathematics from Harbin Institute of Technology, China. In 2024, he began pursuing his Ph.D. in Computer Science and Technology at Harbin Institute of Technology. His research focuses on the intersection of security and high-performance computing (HPC), particularly on accelerating security-related operators on heterogeneous systems.

Jie Dai obtained his Bachelor's degree in Information Security from Harbin Institute of Technology, China, in 2024. He is currently pursuing his Master's degree at the School of Computer Science and Technology at Harbin Institute of Technology. His research interest lies in machine learning systems.

Chenyu Yuan is a senior undergraduate student at the School of Computer Science and Technology, Harbin Institute of Technology, China. His research interests include high-performance computing, with a particular focus on optimizing GPU power consumption during deep learning inference tasks.

Desheng Wang received the Ph.D. degree in Cyberspace Security from Harbin Institute of Technology, Harbin, China, in 2022. He is currently an assistant professor in the School of Computer Science and Technology at Harbin Institute of Technology, Shenzhen, China. His research interests include cloud computing, edge computing, and cyberspace security.

Hongwei Yang is a research associate of the Network Security Center in the School of Cyberspace Science, Harbin Institute of Technology, China. He received the DEng degree in cyberspace science from the Harbin Institute of Technology. His research interests include deep learning optimization, graph mining.