CCF Transactions on High Performance Computing (2024) 6:287–300
https://doi.org/10.1007/s42514-024-00186-y

REGULAR PAPER

Opencl-pytorch: an OpenCL-based extension of PyTorch

Yicheng Sui[1] · Yufei Sun[1,2] · Changqing Shi[1] · Haotian Wang[1] · Zhiqiang Zhang[2] · Jiahao Wang[1] · Yuzhi Zhang[1,2]

Received: 24 October 2023 / Accepted: 11 March 2024 / Published online: 8 April 2024

Abstract
Currently, most Deep Learning (DL) frameworks support only CUDA and ROCm environments, limiting their use to NVIDIA and AMD GPUs. Since current High-Performance Computing (HPC) usually uses different types of heterogeneous devices to accelerate computing, some HPCs cannot utilize heterogeneous devices for computing based on the DL frameworks. To address this problem, we introduce OpenCL-PyTorch, a PyTorch extension based on OpenCL. This extension enables the deployment of DL models on a broader range of OpenCL devices, encompassing CPUs, GPUs, and other accelerators. A standout feature of OpenCL-PyTorch is our novel unified OpenCL device and memory management approach, which significantly enhances performance. We rigorously evaluated OpenCL-PyTorch with various DL models, confirming its accuracy and effectiveness. The validation of the management approach further underscores the importance of our unified device and memory management in optimizing operator performance.

Keywords Deep learning · Framework · PyTorch · OpenCL

1 Introduction

Deep learning (DL) frameworks like PyTorch (Paszke et al. 2019), TensorFlow (Abadi et al. 2016), MXNet (Chen et al. 2015), and Caffe2 are indispensable tools for constructing DL models. They provide building blocks for designing, training, and validating DL models through a high-level programming interface. Using these frameworks, developers can efficiently define a DL model's structure, configure the optimizer, and process raw data for training and deployment (Pouyanfar et al. 2018). As DL models have evolved in recent years, becoming deeper and encompassing more parameters, there has been a proportional surge in computational power requirements. To address this, DL frameworks have integrated support for heterogeneous parallel programming, leveraging the robust computational capabilities of devices such as GPUs (Nguyen et al. 2019).

✉ Yufei Sun
yufei_sun@sina.com

Yicheng Sui
suiyicheng@mail.nankai.edu.cn

[1] College of Software, Nankai University, Hongda 300457, Tianjin, China

[2] Haihe Lab of ITAI, Hi-Tech Area 300480, Tianjin, China

However, as shown in Table 1, the types of programming models they support are limited. Only a few manufacturers' devices can be utilized for computation. In recent years, with the development of computing devices such as graphics processing unit (GPU), field-programmable gate array (FPGA), digital signal processor (DSP), and application-specific integrated circuit (ASIC), some heterogeneous devices from other manufacturers can also provide excellent computing power and can be used for the computation of DL models (Reuther et al. 2020, 2021). Some HPC machines usually use these heterogeneous devices to accelerate calculations. If these heterogeneous devices can be supported in DL frameworks, it will help to use HPC machines to train DL models. Therefore, the heterogeneous programming models supported natively by the DL frameworks must be extended. Utilizing multiple types of devices by supporting a common programming model is feasible. In this way, DL frameworks can fully use the computing power of different types of devices.

As an open and portable heterogeneous programming standard, OpenCL can code based on a unified API and run on different heterogeneous computing devices, which has high versatility and strong portability. It creates a hardware-independent software development environment that supports different levels of parallelism and can be efficiently mapped to single or multi-device homogeneous or

Table 1 Heterogeneous programming models in DL frameworks

	TensorFlow	PyTorch	MXNet
CUDA	✓	✓	✓
HIP	✓	✓	✓
SYCL	✓	✗	✗
OpenCL	✗	✗	✗

heterogeneous systems consisting of CPUs, GPUs, FPGAs, and other devices. Adding support for OpenCL in the DL framework is an effective way to use the computing power of various types of devices. PyTorch strives to make writing models, data loaders, and optimizers as easy and productive as possible (Paszke et al. 2019). Thus, it is widely used in scientific research. Adding support for OpenCL in PyTorch is valuable so that various computing devices can be used to train and deploy DL models.

Therefore, in this paper, we implement OpenCL-PyTorch, an OpenCL extension of PyTorch, and propose a unified OpenCL management method for performance improvement. The main contributions of this paper are as follows:

1. We implemented OpenCL-PyTorch, an OpenCL-based extension of PyTorch, which could utilize multi-manufactured devices and support commonly used DL models. This implementation is extensible, and developers can add new operators based on the usage of different models.
2. We proposed a novel unified OpenCL device and memory management method, which has been used in OpenCL-PyTorch. OpenCL-PyTorch could implicitly manage OpenCL device and memory, avoiding unnecessary memory copying and creation overhead. Based on experiments, we demonstrate that the unified OpenCL management method improves performance significantly.
3. Based on the development process of OpenCL-PyTorch, we summarize some experience of developing OpenCL operators for specific DL models, including the relationship between common operators and OpenCL functions and the experience of using OpenCL-based acceleration libraries.

To evaluate the correctness of OpenCL-PyTorch we implemented, we have trained some typical DL models with it and compared them with the models trained with native PyTorch. The experiment results prove that the OpenCL-PyTorch can correctly train and deploy the DL models. To verify the effectiveness of the unified OpenCL management method, we selected some commonly used operators to compare the performance with or without the management. The experiment results prove that the unified OpenCL management method proposed in this paper improves performance significantly.

The remainder of this paper is organized as follows: In Sect. 2, we introduce the heterogeneous programming models, related works, and challenges. In Sect. 3, we elaborate on OpenCL-PyTorch's framework. In Sect. 4, we introduce the experience of OpenCL-PyTorch's development. Later in Sect. 5, we depict our experimental setup and results and conclude our paper in Sect. 6.

2 Background and challenges

2.1 Native support for heterogeneous programming models

DL frameworks make a performance profit from using some specialized devices present in accelerated computing environments. The current mainstream solution has been to use GPUs as general-purpose processors. Nowadays, popular alternatives to GPUs include FPGA and other dedicated devices for DL acceleration offered by some IT companies (Nguyen et al. 2019). To utilize these devices, DL frameworks usually provide native support for some heterogeneous programming models.

However, the support should be more comprehensive. Only a few programming models corresponding to the mainstream manufacturer can be supported by DL frameworks. We list the support of some mainstream DL frameworks for heterogeneous programming models in Table 1. Among them, CUDA and HIP are programming models provided by NVIDIA and AMD, respectively, which can utilize the GPU device of the corresponding manufacturer to perform efficient parallel computing. They are suitable for the characteristics of high parallelism of some operators in deep neural networks. SYCL (Keryell et al. 2015) is a free high-level abstract programming model that can utilize some manufacturers' FPGAs. It is based on different heterogeneous programming models, such as OpenCL. However, the types of heterogeneous programming models supported are also limited. If using OpenCL as the underlying implementation of SYCL, you need to provide additional extensions in the implementation of OpenCL, thus further limiting the use of SYCL.

In summary, due to the limited support of the DL framework for the programming model, manufacturer-neutral devices cannot be supported. Even though some hardware from other manufacturers has a massively parallel architecture suitable for accelerating matrix-based operators, these devices cannot be used for DL model computations due to the lack of programming model support. Moreover, OpenCL can provide compatibility across heterogeneous hardware from any vendor. Therefore, when the deep learning

framework can support OpenCL, it can use a manufacturer-neutral device.

2.2 Related work

Some existing studies have proposed methods for supporting OpenCL in DL frameworks.

OpenCL Caffe: The Caffe framework (Jia et al. 2014) was initially written and developed in C++ and CUDA. OpenCL Caffe (Gu et al. 2016) targets in transforming the popular CUDA based framework caffe into OpenCL backend. The CUDA layer is responsible for optimizing the allocation and use of hardware resources, such as task scheduling between CPU and GPU, memory management, and task transmission. OpenCL Caffe migrates the three layers of the C++ machine learning interface, Wrapper, and GPU Kernel layer by layer. At the same time, it also analyzes performance bottlenecks through various analysis tools and proposes corresponding optimization techniques. Among them, according to the characteristics of OpenCL runtime compilation, it caches the compiled program, avoiding the overhead caused by repeated program compiling.

OpenCL-darknet: Darknet (Redmon 2013–2016) is a DL-based target detection framework known for its fast speed and simple structure, but it can only be based on NVIDIA GPU accelerated computing. In order to make Darknet available for general-purpose accelerator hardware, OpenCL-Darknet (Koo et al. 2021) converts CUDA-based Darknet to Darknet supporting OpenCL backend, and achieves performance similar to the original CUDA version. OpenCL-Darknet converts basic arithmetic functions, matrix multiplication and other operators into the OpenCL backend to minimize unnecessary data transmission between the host and device.

cltorch: cltorch is an OpenCL backend for the Torch scientific computing framework (Ronan et al. 2017), offering a high-performance matrix library that leverages GPU computational power. It includes features like standard operation support, profiling tools, point tensors to reduce pipeline stalls, custom user kernels, and compatibility with other libraries. However, cltorch cannot be applied to PyTorch because it is specifically designed for the Torch framework, which has a different architecture and API than PyTorch. PyTorch uses a different backend and is not compatible with the Torch-specific implementations and extensions provided by cltorch.

pytorch-dlprim: Pytorch_dlprim (Beilis 2023b), an OpenCL backend for PyTorch based on DLPrimitives (Beilis 2023a), facilitates the training of specific vision networks, such as AlexNet and ResNet, on OpenCL-compatible devices. This implementation has successfully validated these networks' forward and backward propagation, benchmarking against CPU performance for accuracy. It's important to note that this repository represents an early version, focusing on establishing foundational functionality and initial testing, and thus have limitations in terms of comprehensive functionality and optimized performance.

While previous efforts have converted the Caffe and Darknet frameworks to OpenCL, these adaptations were limited to kernel functions and overlooked crucial aspects like device abstraction. Moreover, while Cltorch provides an OpenCL extension for the Torch framework, it cannot be directly applied to PyTorch. Despite sharing similar design philosophies, Torch and PyTorch diverge significantly in code implementation and architectural design, rendering Cltorch incompatible with PyTorch. In contrast, Pytorch-dlprim focuses primarily on implementing specific visual networks. Our work, however, goes beyond these previous endeavors. We aim to fully extend PyTorch with OpenCL support, encompassing the implementation of OpenCL kernel functions and the comprehensive management of OpenCL devices and memory. This holistic approach addresses the gaps in existing adaptations and aligns with our goal of a complete OpenCL extension for the PyTorch framework.

2.3 Challenge

To add the extension for the OpenCL programming model in the PyTorch framework, we faced the following challenges:

1. PyTorch has a huge amount of code and complex hierarchical relationships. We need to find the native operator implementation function from the source code of PyTorch and implement the corresponding calculation function based on OpenCL. This makes us face significant challenges when implementing OpenCL-based operators.
2. The existing methods do not provide a unified OpenCL management. DL frameworks do not support OpenCL devices and memory. On the one hand, users can only use a single specific OpenCL device and not specify different OpenCL devices. On the other hand, repeated memory copying and creation are performed during execution, resulting in a large amount of unnecessary overhead and additional memory usage.
3. A large number of operators are usually provided in the DL framework, but the existing methods do not summarize the implementation of these operators. Therefore, when using OpenCL to develop these operators, we are faced with a huge development workload, and some

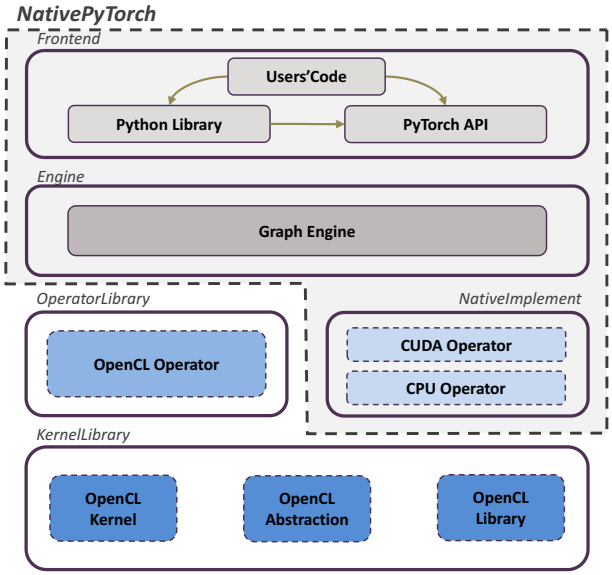


Fig. 1 The overall architecture of OpenCL-PyTorch

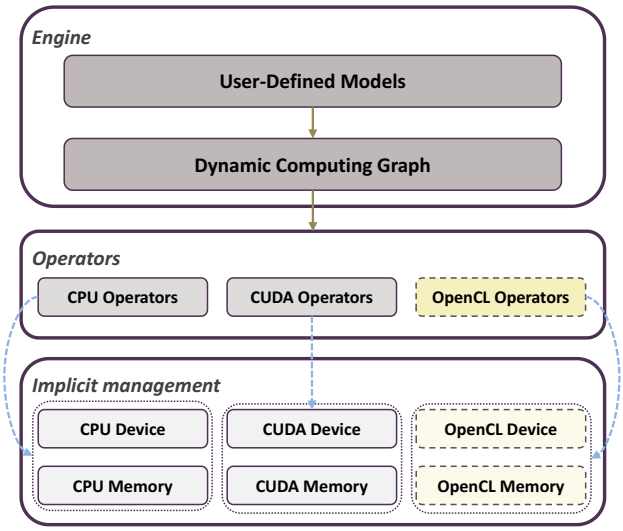


Fig. 2 Implicit management in PyTorch

OpenCL device-side codes cannot be reused, increasing the development difficulty.

3 Methodology

In this section, we will elaborate on the architecture of OpenCL-PyTorch addressing the first challenge and the unified OpenCL management method addressing the second challenge. Moreover, we will detail the OpenCL-based operator development method, which is the most important work in implementing OpenCL-PyTorch.

3.1 Overall architecture

As an optimized tensor library, PyTorch has a large amount of source code with complex hierarchical relationships. Different modules correspond to different levels and functions. However, we only need to modify or extend some modules to support OpenCL in PyTorch. To reduce the coupling between the OpenCL extension and PyTorch, we designed the OpenCL-PyTorch architecture based on the adapter design pattern (McDonough and McDonough 2017) through the analysis of the PyTorch source code. This adapter architecture addresses the first challenge. Figure 1 shows the architecture of OpenCL-PyTorch. The OpenCL extension we implemented acts as a PyTorch adapter, which adapts the OpenCL kernel function library to the PyTorch framework at the operator layer.

Specifically, the region inside the dotted line shows the modules of the native PyTorch. The region outside the dotted line is OpenCL-PyTorch, which supports OpenCL by an extension. From a view of module hierarchy, PyTorch mainly has three parts: ***frontend***, ***engine***, and ***operators***. The frontend provides a series of Pythonic interfaces, which can parse the user's code, including DL model architecture, optimizer, scheduler, data loader, etc. The graph engine generates a dataflow graph based on the user-defined DL model architecture and gradually calls operators to compute according to the order in the dataflow graph. Operators are the functions that respond to computation, take data as input, and return results. In PyTorch, they perform all core computing functions. Among these three parts, the Pythonic frontend and graph engine have nothing to do with the programming model that executes the computation; only the operator execution process is related to the programming model. OpenCL can be used for computation as long as OpenCL-based operators are implemented. We designed an OpenCL-based operator module at the operator level as an adapter. It utilizes the OpenCL kernel library for computation and offers the same interfaces for PyTorch. This way, users do not need to modify their codes explicitly to use OpenCL.

3.2 Unified OpenCL device and memory management

To efficiently utilize OpenCL for computation, it is necessary to optimize the execution process of OpenCL operators. As shown in Fig. 2, PyTorch provides implicit memory and device management to perform computation efficiently, which is usually used during the execution of operators. It is necessary to manage memory and devices similarly, as

mentioned in Sect. 2.3. To address the second challenge, we propose a unified OpenCL device and memory management method based on analyzing the operator execution process and the characteristics of the OpenCL programming model. In this section, we will introduce the two parts of the method: OpenCL memory management based on PyTorch tensor mapping and OpenCL device management based on PyTorch device type support.

3.2.1 OpenCL device management

The execution of OpenCL device-side functions needs an initialized OpenCL device. Some existing methods for OpenCL device storage based on singletons are proposed to avoid repeated initialization. They can store the initialized OpenCL device in a singleton, ensuring it can only be initialized once. However, they do not support OpenCL devices in the DL frameworks. Therefore, they cannot use multiple devices simultaneously when multiple OpenCL devices can be used. Users can only use a specific OpenCL device set in the source code, cannot specify OpenCL devices on the frontend, and cannot utilize multiple OpenCL devices. Users should not be aware of the details of the device implementation. They need to use OpenCL devices for computation as efficiently as possible with CUDA devices, which requires full support for OpenCL device types in the PyTorch framework and implicit management of these devices. Therefore, it is necessary to manage OpenCL devices in PyTorch.

To manage OpenCL devices in PyTorch, we proposed a OpenCL device management method with two parts:

1. Device Type Support: PyTorch need to support OpenCL device types so users can use OpenCL devices like CUDA.
2. Device Information Storage: PyTorch need to store OpenCL device-related information implicitly to avoid repeated overhead.

Device Type Support: Based on the analysis of the code of PyTorch, we found that device type support and object management are implemented separately in PyTorch. It does not impose requirements on the implementation of device type. Switching different branches takes the device type as the judgment condition only. Figure 3 shows the process of calling the operator implementation function according to the device type in the execution of the matrix multiplication operator. The *mm* operator could call different functions to correspond to different device types. Among them, "*mm_cpu*" and "*mm_cuda*" are natively developed functions, and "*mm_opencl*" is a function we developed based on OpenCL. PyTorch will automatically choose the appropriate function when calling the operator.

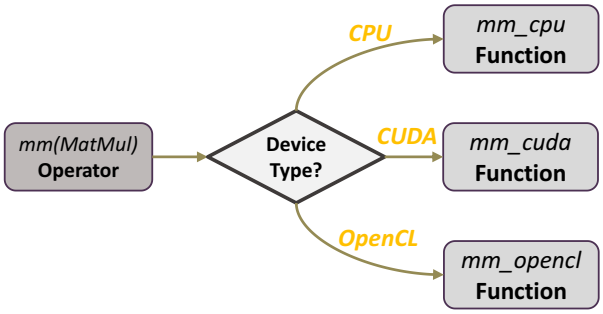


Fig. 3 Device type support of PyTorch

To support the OpenCL device type, we need to add an OpenCL branch to each switch that selects the execution path based on the device type so that the corresponding function can be called when the OpenCL device is used for computation. From a view of the source code, PyTorch switches branches in backend based on *Dispatch Key* and supports the device in frontend based on *Device Type*. For different device types, PyTorch will create different *Dispatch Key* collections in the context, which are used for switching to different branches. Corresponding to the *Device Type*, PyTorch provides an enumeration object to save the information of supported device types. In contrast, the *Dispatch Key* is implicitly stored in the context. It is automatically modified according during the execution, while *Device Type* is explicitly stored in *Tensor* properties and specified by the user. They correspond to frontend and backend, respectively, and cooperate to switch branches. Although PyTorch does not natively support OpenCL, it retains enumeration values for the device type of OpenCL. We map the OpenCL device to the unmapped *Dispatch Key* reserved by PyTorch to support the OpenCL device type.

This way, we can add support for OpenCL devices in PyTorch. Users can specify OpenCL devices on the front end just like using CUDA devices by adding the *Device Type* of OpenCL. Correspondingly, during the execution in the backend, PyTorch will automatically select the operator function corresponding to OpenCL according to the *Dispatch Key* mapping the *Device Type* of OpenCL.

Device Information Storage: To manage OpenCL devices, we designed an OpenCL device manager, which stores OpenCL device information and provides an interface to get OpenCL device objects based on the device name. Similar to the existing device storage method based on the singleton, we also use a singleton method to initialize it. That is, it can be only initialized once in the entire lifecycle of the PyTorch process. All available device objects are stored, which is also initialized only once. This way, users can use all the OpenCL devices in one process and do not need to be initialized repeatedly,

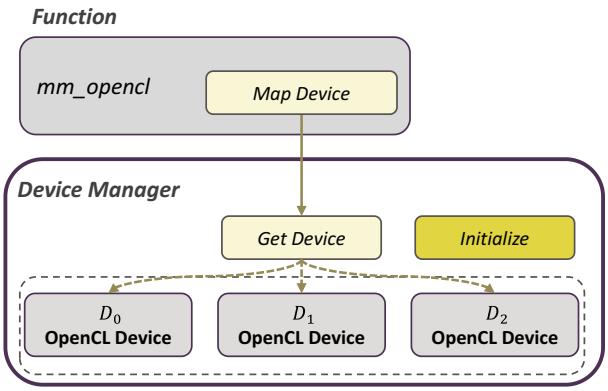


Fig. 4 Device manager use case

avoiding unnecessary performance overhead. The device manager acts as a singleton object that can be used during the execution of all OpenCL operators. It provides an interface for obtaining OpenCL device objects. In this way, the user can specify the device to be used on the front end, and PyTorch can obtain the specified OpenCL device object according to the index of the specific device on the back end to perform calculations.

Figure 4 shows a use case of the device manager. When a function needs to obtain the device, it can map the corresponding device according to the index. *Device Manager* has an initialization function called only once to initialize all OpenCL device objects and stores them in a *Device Manager* object. For example, when calling the "*mm_opencl*" function, it can obtain a specific OpenCL device by the device name stored in *Tensor* properties from the *Device Manager* for computation. If it is the first operator called in this PyTorch process, *Device Manager* will call the *initialize* function to initialize all OpenCL device objects.

3.2.2 OpenCL memory management

Data exchange is often the main factor affecting performance. Memory copying will cause a lot of overhead in the computation of DL model that frequently calls operator executions. As shown in Fig. 5, we compared the operator execution process with and without memory management. The tensor cannot be stored on the OpenCL device without memory management. Before the computation based on OpenCL, the OpenCL device must copy the tensors' data stored on the CPU to the OpenCL device. After the calculation, it must copy the data stored on the OpenCL device to the CPU side and assign it to the tensor corresponding to the output. Then it will release the useless memory. On the one hand, such an execution process increases unnecessary performance overhead (such as memory allocation, memory release, and memory copy). It makes the operator execution process more complicated. On the other hand, it also increases overall memory usage. The same data must be stored simultaneously on the CPU and the OpenCL device sides, reducing memory space utilization efficiency.

Correspondingly, as shown in the lower part of Fig. 5, when the tensor can be stored on the OpenCL device and flow in the dataflow graph, the operator execution process becomes much more straightforward. Therefore, to implement OpenCL memory management in the PyTorch framework, we need to bind the OpenCL device-side memory to PyTorch tensor so that the underlying OpenCL device-side memory can be managed through the Tensor management mechanism in the PyTorch framework.

To manage OpenCL memory within the PyTorch framework, enabling a tensor to store device-side memory in OpenCL is crucial. We began by analyzing the implementation of a tensor in PyTorch, which comprises two primary components: properties and data storage. The property section holds information like tensor shape, storage device, and

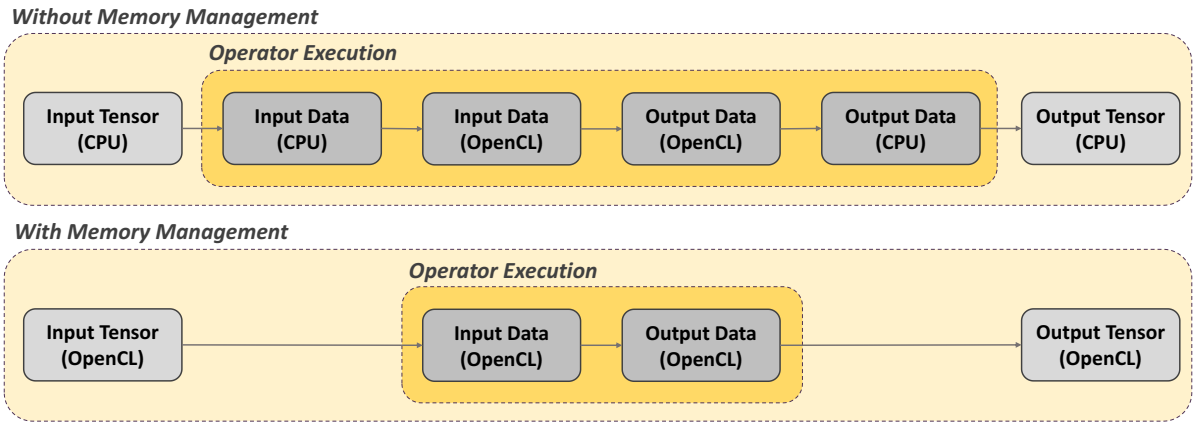


Fig. 5 Comparison of memory management

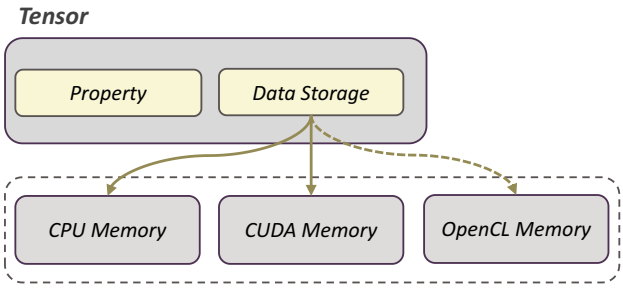


Fig. 6 The structure of tensor in PyTorch

memory access step size, all vital during operator execution. The data component typically corresponds to a continuous memory address on the host or device. When executing OpenCL or CUDA kernel functions, the computing device interacts with these memory addresses for reading or writing data.

We can set the corresponding OpenCL device in the tensor properties based on device management. However, it is not enough. The tensor's most essential data storage part must be transferred to the OpenCL device. As shown in Fig. 6, the solid line represents PyTorch's device storage method natively supported. The host-side or CUDA device-side memory address is stored in the tensor for use in the computation process. Similarly, we must also store the OpenCL device-side memory address in the tensor. Specifically, when tensors are created on an OpenCL device, memory needs to be allocated on the OpenCL device. Then, PyTorch can use this memory address to initialize the *Storage* object and use this object to initialize the tensor. This address will be stored in the tensor, and when PyTorch needs to read data, the OpenCL device-side memory address can be obtained from the *storage* property of the tensor.

Moreover, PyTorch provides tensor lifecycle management. When some tensors are not used, they will be automatically destroyed by PyTorch. We bind the OpenCL device-side memory to the tensor. The lifecycle of the PyTorch tensor and OpenCL device-side memory are synchronized. The lifecycle management of OpenCL memory can be realized based on that of the tensor. When a Tensor is created, it will allocate memory on the OpenCL device side; when a Tensor is destroyed, it will release the memory on the OpenCL device side at the same time. In this way, through the method mentioned above, the device-side memory on OpenCL can be automatically allocated and released according to the lifecycle of the tensor. More importantly, tensors stored on OpenCL devices can flow in the dataflow graph, reducing the overhead caused by unnecessary memory copies.

3.3 Operator development

Besides the unified OpenCL management method mentioned above, operator development based on OpenCL is critical to implement OpenCL-PyTorch. First, we will develop the functions for computation based on the OpenCL programming model. As mentioned in Sect. 3.2.1, PyTorch chooses the appropriate function based on the *Dispatch Key* when calling an operator. Then, we must bind the functions to the *Dispatch Key* corresponding to OpenCL to use these operators for computation.

As for the development of the device-side functions, we can develop OpenCL kernel functions. Some researchers (Martinez et al. 2011; Harvey and De Fabritiis 2011) proposed CUDA-to-OpenCL source-to-source translator. However, these translators cannot be directly applied to the translation of kernel functions in DL framework. On the one hand, due to the differences in programming languages between CUDA and OpenCL, these translators cannot translate codes with C++ characteristics, such as templates and classes used in CUDA kernel functions. On the other hand, the performance and availability of these automatically converted OpenCL kernel functions on different devices cannot be guaranteed, and we cannot optimize their performance easily. Therefore, we manually develop the OpenCL kernel function code. The method of developing kernel functions based on OpenCL is prevalent, and this is not our innovation, so we will not repeat it here. Finally, we developed over a hundred OpenCL kernel functions, which support different data types based on the compilation options.

```
TORCH_LIBRARY_IMPL(aten,
        PrivateUse1, m) {
      m.impl(''aten::mm'',&mm_opencl);
}
```

After developing OpenCL kernel functions, we bind them to the *Dispatch Key* corresponding to OpenCL. For example, the code above shows how to bind a function to a *Dispatch Key*. "*PrivateUse1*" is used as the *Dispatch Key* corresponding to OpenCL. The corresponding operator is "*aten::mm*", and the function's name is "*mm_opencl*". This way, when the *mm* operator is called, and the specified device is OpenCL, the *mm_opencl* function we developed will be called.

4 Development experience

Here we introduce some development experience of OpenCL-PyTorch. First, we classify commonly used operators based on the implementation according to our experience in the development process. Then we introduce some

Table 2 Correspondence of some important OpenCL functions and operators

Function	Operator for Example
Reduction	Softmax, LogSoftmax
	Normalize
	Sum, Maximum, Minimum
GEMM	Matrix Multiplication
	Batch Matrix Multiplication
	LSTM
	Convolution
Element-Wise	Add, Sub, Multiplication, Division
	Activation(Relu, Glu, Sigmoid, etc.)

acceleration libraries used in PyTorch and the experience of developing some common operators based on OpenCL acceleration libraries.

4.1 Commonly-used OpenCL functions

Table 2 shows the correspondence between operators commonly used in DL models and OpenCL kernel functions. We selected three commonly used functions: ***Reduction***, ***GEMM***, and ***Element-Wise***. Reduction is commonly used in data-parallel programming, which reduces a vector of data to a single value (Jin and Finkel 2018). The functions related to reduction are usually used as a part of an operator, such as the summation part in "*Softmax*". These operators are widely used in DL models and usually have a significant computational overhead. It is necessary to implement OpenCL-based reduction functions efficiently and in parallel. GEMM corresponds to the widely used matrix multiplication operator, which implements the matrix multiplication computation and is usually called by other operators, such as the GEMM-based convolution operator. After converting an image into a column-wise matrix, the GEMM function will be called. Besides, GEMM is also used in matrix multiplication operators directly, which is used to compose the linear layer in a DL model. GEMM is one of the most important operators in the OpenCL-based DL framework and usually accounts for the most significant computation overhead in a DL model. Element-wise functions refer to performing element-by-element operators on the data of one or more vectors. They are widely used in element-wise operators, such as adding or subtracting the corresponding elements in two or more tensors, calculating the maximum value, etc. Besides, they are also used in some activation functions, such as Relu, Sigmoid, etc.

4.2 OpenCL acceleration libraries

With the introduction of some typical DL models (such as CNN (Li et al. 2021), RNN (Tarwani and Edem 2017), Transformer (Kalyan et al. 2021), etc.), the main structures of deep neural network (DNN) have primarily converged, and a small number of operators occupy most of the computational overhead. For example, the convolution and pooling operators in the CNN usually take most of the computational overhead. In the Transformers model, the matrix multiplication operator consumes most of the computational overhead (Park et al. 2020). Therefore, optimizing these commonly used operators based on existing high-performance libraries can significantly improve the computational efficiency of DL models. As mentioned in section 2, to optimize DNN operators and matrix computing operators, PyTorch integrates the CUDA-based DNN library cuDNN and Basic Linear Algebra Subprograms (BLAS) library cuBLAS, respectively, which significantly improves the computing performance based on CUDA.

Therefore, to improve computing performance based on OpenCL, we also use the DNN library and BLAS library. We integrated the CLBlast (Nugteren 2018) and MIOpen (Khan et al. 2019) libraries to accelerate matrix multiplication and DNN-related operators.

5 Evaluation

5.1 Experiment design

5.1.1 Experimental setup

In this section, we aim to compare OpenCL-PyTorch and CUDA-PyTorch under identical hardware conditions, focusing on assessing the computational efficiency and correctness of OpenCL-PyTorch. We also conduct ablation studies to demonstrate the impact of our unified memory and device management approach on computational efficiency. Our experiments utilize PyTorch version 1.10, with CUDA-PyTorch installed and OpenCL-PyTorch deployed on the server. We chose NVIDIA GPUs for the experiments, leveraging their ability to support both CUDA and OpenCL environments. The detailed experimental settings are shown in Table 3.

5.1.2 Deep learning models for evaluation

In our experiments, we select four widely-used deep learning models from PyTorch's examples[1] for experimentation. The

[1] PyTorch Examples: https://github.com/pytorch/examples.

Table 3 Experiment settings

Name	Specification
CPU	Intel(R) Xeon(R) Gold 5218 CPU@2.30GHz
RAM	187GB DDR4 2933 MT/s
GPU	NVIDIA Tesla V100S
NVIDIA CUDA Toolkit	CUDA–10.2
OpenCL	CUDA 10.2 OpenCL 1.2
PyTorch	1.10

chosen models, which we detail in Table 4, include Convolutional Neural Networks (CNN), Long Short-Term Memory networks (LSTM), and Variational Autoencoders (VAE), each distinct in network layer structure, dataflow graph, operator types, and data scale. This variety is pivotal for assessing our OpenCL-PyTorch's ability to accurately train and test DNN models. For instance, the CNNs test convolution and pooling layers, while the LSTMs focus on LSTM layers, ensuring a thorough validation of the diverse operators used in OpenCL-PyTorch.

5.1.3 Evaluation metrics

As previously mentioned, with the addition of the OpenCL extension in PyTorch, the usage of OpenCL-PyTorch and CUDA-PyTorch is nearly identical, requiring only a modification in the device name. Therefore, in terms of correctness evaluation, we assess OpenCL-PyTorch from two aspects: operator and model training.

Specifically, for operators, we utilize OpenCL-PyTorch and CUDA-PyTorch to execute the same operators on the same data and then compare their results. If the results are sufficiently close, accounting for computational errors due to different programming models, we consider the OpenCL-PyTorch computation correct. For model training, we evaluate the correctness of OpenCL-PyTorch and CUDA-PyTorch by analyzing the loss reduction process when training the same models and by comparing the performance of the trained models on test datasets using both versions of PyTorch.

As for the models' performance, different models serve varied application scenarios and thus require distinct evaluation metrics. Accuracy is a primary indicator for classification tasks, as exemplified by the VGG16 network designed for classification, where Accuracy aptly reflects its performance. On the other hand, PSNR (Peak Signal-to-Noise Ratio) is crucial for super-resolution models. It quantifies the quality difference between original and processed images, making it an ideal metric for assessing these models' image and video resolution enhancement capabilities. By using PSNR, we can precisely measure the effectiveness of super-resolution models in improving image clarity and detail.

Similarly, to compare computational efficiency, we use OpenCL-PyTorch and CUDA-PyTorch to execute the same set of operators and training models. We then evaluate OpenCL-PyTorch's efficiency by precisely measuring and comparing their execution times, detailing specific performance metrics such as throughput and latency to ensure a comprehensive and fair assessment of the computational efficiency. Then, to verify the effectiveness of our unified memory and device management approach, we evaluate the computational efficiency based on the computation time of operators.

5.2 Evaluation of the correctness

In our correctness verification, we initiated by evaluating the precision of individual operators in OpenCL-PyTorch. Following this, we further assessed the training correctness by examining both the trend of loss reduction and the performance metrics of models on the test dataset. These comprehensive evaluations, focusing on detailed aspects of operator and training process, collectively validate the computational correctness of our OpenCL-PyTorch implementation.

5.2.1 Evaluation on operators

To validate the accuracy of operators within OpenCL-PyTorch, we compare their output against CUDA-PyTorch using specific built-in functions in PyTorch. Specially, we

Table 4 Deep learning projects for evaluation

Name	Main Architecture	Description
Image classification	CNN	Training of VGG (Simonyan and Zisserman 2014) on the Cifar10 dataset (Krizhevsky and Hinton 2009).
Time series prediction	LSTM	A toy example that uses LSTM (Yu et al. 2019) to learn Sine waves.
Auto-encoding variational bayes	VAE	An improved implementation of the paper "Auto-Encoding Variational Bayes (Kingma and Welling 2013)".
Super resolution	CNN	An example that uses the efficient sub-pixel convolution layer. Shi et al. (2016)

Table 5 Operator correctness

Operator	Correctness
Abs	✓
EmbeddingBag	✓
AdaptativeMaxPool2d	✓
AdaptativeAvgPool2d	✓
Softmax	✓
GridSampler2d	✓
GroupNormal	✓
Trace	✓
Dropout	✓

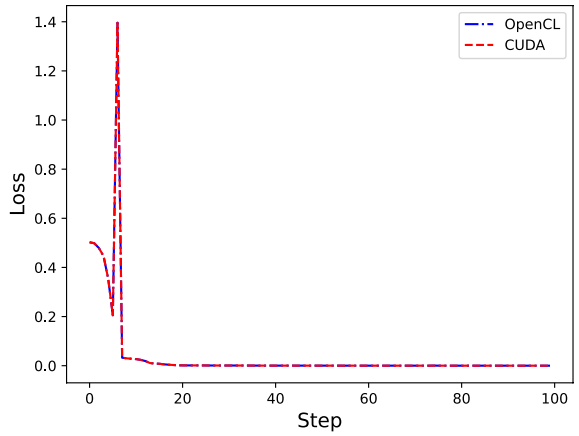
employed the *torch.testing.assert_close* method of PyTorch evaluates the closeness of two tensors by considering relative and absolute tolerances. It mandates that their difference must not exceed the combined threshold, determined by adding the absolute error threshold to the product of the relative error threshold and the compared value. In PyTorch, these thresholds are set explicitly at 1.3e−6 for relative error and 1e-5 for absolute error, providing a stringent yet practical criterion for numerical comparison in computational operations.

As listed in Table 5, our experiment involved ten different operators, and the findings indicate that OpenCL-PyTorch's results closely align with the expected outcomes. This consistency across all tested operators substantiates the correctness of OpenCL-PyTorch's operators. The operators we have chosen for evaluation represent a broad spectrum of functionalities, each finding its use in diverse models. By successfully implementing and verifying these varied operators in OpenCL-PyTorch, we demonstrate the feasibility of our approach and reinforce the assurance of operator correctness within the framework.

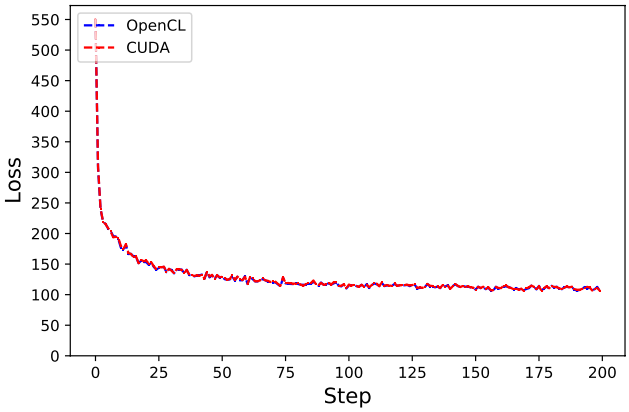
5.2.2 Evaluation on model training

First, we tracked and compared the loss reduction while training the Time Sequence Prediction and VAE models with both OpenCL-PyTorch and CUDA-PyTorch. The comparative results, as depicted in Fig. 7, indicate a synchronous loss reduction pattern across them, with each training step showing closely matched results.

Following the training phase, we tested the models to evaluate the performance of those trained with OpenCL-PyTorch. These tests, based on specific performance metrics, were designed to rigorously assess the fidelity of the models, ensuring that training with OpenCL-PyTorch yields results that meet established standards of model correctness. Table 6 shows that models trained through OpenCL-PyTorch and CUDA-PyTorch have similar performance.



(a) Time Sequence Prediction



(b) VAE

Fig. 7 The losses during the training

Table 6 Model performance comparison

Model	Indicator	OpenCL	CUDA
VAE	Loss	105.6090	105.4507
Time sequence prediction	Loss	6.4e–6	7.8e–6
VGG16	Accuracy (%)	52.57	50.80
Super resolution	PSNR(dB)	24.5237	24.3924

It is important to note that, we employed identical random number seeds for training the model with OpenCL-PyTorch and the CUDA-PyTorch. Nonetheless, inherent disparities in computation between OpenCL and CUDA can lead to slight result variations, which, over time, might slightly affect the outcomes. This accounts for the performance differences observed between OpenCL-PyTorch and the CUDA-PyTorch. However, considering the loss reduction trend and the trained model's performance, training with OpenCL-PyTorch yields results comparable to the

Table 7 Operator calculation time for comparison (microseconds)

Operator	OpenCL	CUDA
Abs	42.23	15.86
EmbeddingBag	132.2	37.44
AdaptativeMaxPool2d	41.03	15.68
AdaptativeAvgPool2d	73.84	23.60
Softmax	38.77	16.52
GridSampler2d	265.3	12.69
GroupNormal	77.91	22.62
Trace	104.2	13.27
Dropout	176.8	14.33

Table 8 Model training time for comparison (seconds)

Model	OpenCL	CUDA
VAE	136.03	69.95
Time sequence prediction	1072.48	203.01
VGG16	661.05	524.59
Super resolution	129.73	36.87

CUDA-PyTorch. This similarity confirms the correctness of OpenCL-PyTorch's training process.

5.3 Evaluation of computational efficiency

Our computational efficiency evaluation compared OpenCL-PyTorch with CUDA-PyTorch regarding operator calculation and overall model training times.

Based on ten selected operators, Table 7 reveals that OpenCL-PyTorch's single-operator execution time is generally about two or three times longer than CUDA-PyTorch's. This trend extends to model training times, where OpenCL-PyTorch consistently underperforms compared to CUDA-PyTorch. As illustrated in Table 8, the training times for OpenCL-PyTorch are consistently higher than those for CUDA-PyTorch. The training performance disparity between OpenCL-PyTorch and CUDA-PyTorch exhibits notable variation among various models. This variation is attributed to the diverse set of operators invoked during the training process of each model, where the performance disparity between these operators is not uniform. For instance, for the VGG16 model, the training times of OpenCL-PyTorch and CUDA-PyTorch are closely matched, owing to the similar performance of the convolution-related operators in both OpenCL and CUDA implementations. However, for Time Sequence Prediction, OpenCL-PyTorch's training time is over five times that of CUDA-PyTorch, due to the significant performance differences in the operators utilized during the model's training process. In summary, the discrepancies in model training efficiency are generally caused by the efficiency variations of the operators involved.

Based on the efficiency evaluation regarding operator calculation and overall model training times, we can observe a discrepancy in the computational efficiency of OpenCL-PyTorch and CUDA-PyTorch. The observed performance discrepancy between OpenCL-PyTorch and CUDA-PyTorch can be attributed to differences in inherent design and optimization capabilities. CUDA is NVIDIA's proprietary computing platform and programming model, which has been meticulously optimized for NVIDIA GPUs, offering advanced features like direct support for Tensor Cores. OpenCL is a framework for writing programs that execute across heterogeneous platforms and offers a more versatile solution. This broad compatibility ensures that applications written in OpenCL can run on diverse devices without being tied to a specific vendor's ecosystem. However, this versatility means that the performance of OpenCL applications may reach a different peak performance than CUDA on NVIDIA GPUs, primarily due to the lack of specific hardware optimizations and the generic nature of OpenCL that aims to accommodate a wide variety of computing devices. The generic approach of OpenCL must cater to a wide range of computing devices, making it challenging to optimize for any single type of hardware as effectively as CUDA does for NVIDIA GPUs. Despite these performance differences, OpenCL remains a valuable tool for developers seeking to write portable code that can leverage the computing power of various platforms, highlighting the trade-off between peak performance and broad hardware compatibility.

5.4 Evaluation of the management method

In our study, we conducted ablation experiments with six randomly chosen operators to assess the effectiveness of the unified device and memory management approach for OpenCL in OpenCL-PyTorch. In OpenCL-PyTorch, the integration of a unified OpenCL device and memory management system allows direct mapping of DL framework tensor storage to OpenCL memory, significantly reducing the memory copying overhead between CPU and OpenCL devices. To evaluate the benefits of this unified approach, we designed a specific ablation study. In this study, we removed the unified OpenCL management system from OpenCL-PyTorch, instead mapping tensor storage to the CPU memory. This meant that operators, even when utilizing OpenCL for computations, relied on tensors stored on the CPU side. This modification allowed us to compare the computation performance under different input storage scenarios. The "OpenCL-PyTorch w/o management" in Fig. 5 illustrates this comparison.

Figure 8 illustrates the efficacy of the unified memory and device management implemented in OpenCL-PyTorch.

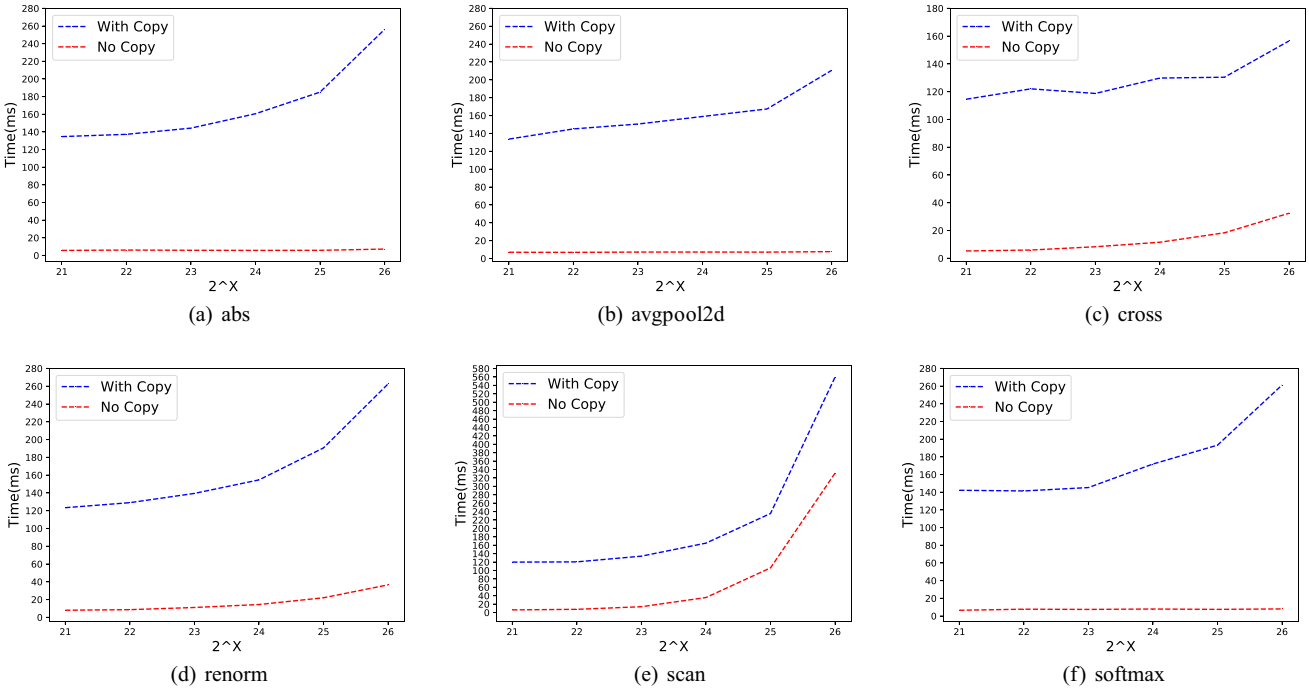


Fig. 8 The comparison of with or without unified OpenCL management

Our proposed unified management approach effectively eliminates unnecessary data copying during operator computations, as illustrated by the "No Copy" segment in red in the graph. In contrast, the computation times observed without this optimization, shown in blue as "With Copy", are markedly higher. The horizontal axis represents the data volume handled by the operators (ranging from 2^{21} to 2^{26}). As data volume increases, the difference in computation time becomes more pronounced. Notably, for operators like "abs", "avgpool2d", and "softmax", we observe a dramatic reduction in execution time by over 94%. Similarly, "renorm" and "cross" operators saw their invocation times decrease by over 80%, and the "scan" operator experienced a reduction of more than 40%. These results demonstrate that the time cost of data copying far exceeds the time cost of computations. This apparent contrast validates the significant efficiency enhancements achieved through our unified management method.

6 Conclusion

In this paper, we implement OpenCL-PyTorch, an OpenCL-based extension of PyTorch, it can utilize manufacturer-neutral OpenCL devices for the computation of DL models. Moreover, we propose a unified OpenCL device and memory management method based on our analysis of the OpenCL operator, which exhibits a nice performance improvement. Experiments using some typical DL models and randomly selected operators have demonstrated the correctness of OpenCL-PyTorch and the effectiveness of our proposed unified OpenCL management method. We take the first step to implement an OpenCL-based extension of PyTorch. In the future, we will optimize the performance and verify the correctness of OpenCL-PyTorch in more DL models.

Acknowledgements This research is supported by National Key R &D Program of China grant 2021YFB0300104.

Data availability This paper proposed an extension of PyTorch and the optimization solutions therein. In this paper, we did not explicitly use a dataset, so no data needs to be publicly available.

Declarations

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

References

Abadi, M., Barham, P., Chen, J., et al.: {TensorFlow}: a system for {Large-Scale} machine learning. In: 12th USENIX symposium on operating systems design and implementation (OSDI 16), pp 265–283 (2016)

Beilis, A.: dlprimitives: Deep learning primitives and mini-framework for opencl (2023a). https://github.com/artyom-beilis/dlprimitives

Beilis, A.: pytorch_dlprim: Dlprimitives/opencl out of tree backend for pytorch (2023b). https://github.com/artyom-beilis/pytorch_dlprim

Chen, T., Li, M., Li, Y., et al.: Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems (2015). arXiv preprint arXiv:1512.01274

Gu, J., Liu, Y., Gao, Y., et al.: Opencl caffe: accelerating and enabling a cross platform machine learning framework. In: Proceedings of the 4th International Workshop on OpenCL, pp 1–5 (2016)
Harvey, M.J., De Fabritiis, G.: Swan: a tool for porting cuda programs to opencl. Comput. Phys. Commun. **182**(4), 1093–1099 (2011)
Jia, Y., Shelhamer, E., Donahue, J., et al.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia, pp 675–678 (2014)
Jin, Z., Finkel, H.: Optimizing an atomics-based reduction kernel on opencl fpga platform. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp 532–539 (2018)
Kalyan, K.S., Rajasekharan, A., Sangeetha, S.: Ammus: A survey of transformer-based pretrained models in natural language processing (2021). arXiv preprint arXiv:2108.05542
Keryell, R., Reyes, R., Howes, L.: Khronos sycl for opencl: a tutorial. In: Proceedings of the 3rd International Workshop on OpenCL, pp 1–1 (2015)
Khan, J., Fultz, P., Tamazov, A., et al.: Miopen: An open source library for deep learning primitives (2019). arXiv preprint arXiv:1910.00078
Kingma, D.P., Welling, M.: Auto-encoding variational bayes (2013). arXiv preprint arXiv:1312.6114
Koo, Y., Kim, S., Yg, H.: Opencl-darknet: implementation and optimization of opencl-based deep learning object detection framework. World Wide Web **24**, 1299–1319 (2021)
Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. Handb. Syst. Autoim. Dis. **1**(4) (2009)
Li, Z., Liu, F., Yang, W., et al.: A survey of convolutional neural networks: analysis, applications, and prospects. IEEE Trans. Neural Netw. Learn. Syst. (2021)
Martinez, G., Gardner, M., Feng, Wc.: Cu2cl: A cuda-to-opencl translator for multi-and many-core architectures. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems, IEEE, pp 300–307 (2011)
McDonough, J.E., McDonough, J.E.: Adapter design pattern. In: A Practical Approach, Object-Oriented Design with ABAP, pp. 191–205 (2017)
Nguyen, G., Dlugolinsky, S., Bobák, M., et al.: Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. Artif. Intell. Rev. **52**, 77–124 (2019)
Nugteren, C.: Clblast: a tuned opencl blas library. In: Proceedings of the International Workshop on OpenCL. Association for Computing Machinery, New York, NY, USA, IWOCL '18 (2018). https://doi.org/10.1145/3204919.3204924
Park, J., Yoon, H., Ahn, D., et al.: Optimus: optimized matrix multiplication structure for transformer neural network accelerator. Proc. Mach. Learn. Syst. **2**, 363–378 (2020)
Paszke, A., Gross, S., Massa, F., et al.: Pytorch: an imperative style, high-performance deep learning library. Adv. Neural Inform. Process. Syst. 32 (2019)
Pouyanfar, S., Sadiq, S., Yan, Y., et al.: A survey on deep learning: algorithms, techniques, and applications. ACM Comput. Surv. 51(5) (2018).https://doi.org/10.1145/3234150
Redmon, J.: Darknet: open source neural networks in c (2013–2016). http://pjreddie.com/darknet/
Reuther, A., Michaleas, P., Jones, M., et al.: Survey of machine learning accelerators. In: 2020 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–1 (2020). https://doi.org/10.1109/HPEC43674.2020.9286149
Reuther, A., Michaleas, P., Jones, M., et al.: Ai accelerator survey and trends. In: 2021 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–9 (2021). https://doi.org/10.1109/HPEC49654.2021.9622867
Ronan, C., Clement, F., Koray, K., et al.: Torch: a scientific computing framework for luajit. In: A Scientific Computing Framework for Luajit, Torch (2017)
Shi, W., Caballero, J., Huszár, F., et al.: Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network (2016). arXiv:1609.05158
Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014). arXiv preprint arXiv:1409.1556
Tarwani, K.M., Edem, S.: Survey on recurrent neural network in natural language processing. Int. J. Eng. Trends Technol. **48**(6), 301–304 (2017)
Yu, Y., Si, X., Hu, C., et al.: A review of recurrent neural networks: Lstm cells and network architectures. Neural Computat. **31**(7), 1235–1270 (2019)

Yicheng Sui is a Ph.D. student in the College of Software at Nankai University. His research interests include artificial intelligence and deep learning.

Yufei Sun is a professor at the College of Software, Nankai University. Her research interests include heterogeneous computing and artificial intelligence.

Changqing Shi is a Ph.D. student in the College of Software at Nankai University. His research interests include high-performance computing, machine learning, and computer architecture.

Haotian Wang is a Ph.D. student in the College of Software at Nankai University. His research interests include natural language processing and deep learning.

Zhiqiang Zhang is a graduate student in the College of Software at Nankai University. His research interests include heterogeneous computing and artificial intelligence.

Jiahao Wang is an M.S. student in the College of Software at Nankai University. His research interests include artificial intelligence and high-performance computing.

Yuzhi Zhang is the chair professor and the dean of the College of Software at Nankai University. His research interests focus on artificial intelligence, etc.