CCF Transactions on High Performance Computing (2024) 6:319–329
https://doi.org/10.1007/s42514-024-00181-3

REGULAR PAPER

oclCUB: an OpenCL parallel computing library for deep learning operators

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

Received: 6 October 2023 / Accepted: 5 January 2024 / Published online: 16 February 2024

Abstract
Deep learning (DL) mainly uses various parallel computing libraries to optimize the speed of model training. The underlying computations of the DL operators typically include essential functions such as reduction and prefix scan, the efficiency of which can be greatly improved using parallel acceleration devices. However, the acceleration of these computations is mainly supported by collective primitive libraries such as NVIDIA CUB and AMD hipCUB, which are only available on vendor-specific hardware accelerators due to the highly segregated computational ecology between different vendors. To address this issue, we propose an OpenCL parallel computing library called oclCUB that can run on different heterogeneous platforms. OclCUB abstracts the OpenCL execution environment, implements reusable common underlying computations of DL, and designs two types of interfaces targeting the operators' heterogeneous acceleration pattern, enabling users to design and optimize DL operators efficiently. We evaluate the oclCUB on various hardware accelerators across Nvidia Tesla V100s with OpenCL 1.2, AMD RADEON PRO V520 with OpenCL 2.0, MT-3000 with MOCL 3, and Kunpeng 920 with POCL 1.6. Our experiments show that the oclCUB-based operators achieve accurate computational results on various platforms. The results also demonstrate that oclCUB is able to maintain a smaller, acceptable performance gap with CUB, and comparable in performance to hipCUB.

Keywords Parallel computing · Deep learning · Heterogeneous computing · OpenCL · High Performance · Super Computer

1 Introduction

Deep learning (DL) is a complex task typically involving multiple stages, which can be simplified and optimized using DL frameworks. The DL frameworks integrate advanced algorithms, famous models, and rich tool interfaces to help users easily develop programs, allowing them to focus more on the algorithm and application design. As DL technology evolves, more and more application scenarios need to address more complex problems, requiring more data and finer model structures to improve model performance. As the model structure becomes more complex and the data size increases, the model training process becomes more time-consuming. To achieve more efficient computation, one approach of popular DL frameworks (Abadi et al. 2016; Paszke et al. 2019; Chen et al. 2015) is to implement DL operators using high-performance computing libraries, including cuDNN (Chetlur et al. 2014), cuBLAS (Cublas 2008), CUB (Merrill 2015), MIOpen (Khan et al. 2019), hipCUB (AMD ROCm 2019a, b), etc. These computing libraries use Compute Unified Device Architecture (CUDA) (Kirk 2007) or Heterogeous-compute Interface for Portability (HIP) (AMD ROCm 2019a, b) as the basis for heterogeneous programming, which means that popular DL frameworks must rely on vendor-specific GPGPUs to accelerate computations, and can't use other vendors' hardware accelerators.

Among the aforementioned computing libraries, deep neural network libraries and linear algebra libraries have received sufficient attention and developed many versions (Intel 2019; Nichols et al. 2019; Rupp et al. 2016; Cao et al. 2014) that can support a wide range of computational backends, while CUB and hipCUB are a class of computing libraries that are easily overlooked. In some of the most popular DL frameworks, typical DL operators that rely on CUB and hipCUB include Softmax, Embedding,

✉ Yufei Sun
yufei_sun@sina.com

[1] College of Software, Nankai University, Tianjin 300450, China

[2] ITAI, Haihe Lab, Tianjin 300350, China

Topk, Select, BiasAdd, Loss, Max, etc. These are all common operators in the underlying computations of DL models, and we should also pay attention to them.

However, CUB and hipCUB are based on proprietary heterogeneous programming models CUDA and HIP that can not run on other vendors' hardware accelerators. In contrast, OpenCL (Stone et al. 2010) can solve this problem very well. OpenCL is a cross-platform, parallel programming standard for heterogeneous systems. It provides programming interfaces, hardware abstractions, and a language for writing kernels. It targets a wide range of hardware accelerators, allowing users to accelerate computations in various heterogeneous environments. OpenCL has excellent portability and satisfactory performance, which can be a good alternative to CUDA (Fang et al. 2011; Komatsu et al. 2010).

In this paper, our main contributions can be summarised as follows:

We propose an OpenCL parallel computing library scheme that can be used in DL frameworks. By taking advantage of the unified standard of OpenCL, our work enables general hardware accelerators to accelerate the relevant DL operators.

We maintain an abstraction of the OpenCL heterogeneous programming model in oclCUB, addressing common problems when using OpenCL in large-scale projects.

We design two types of interfaces, one targeting the host side and the other targeting the device side, to support DL operators' heterogeneous programming flexibly. We also implement multiple reusable kernel computations to help users design and optimize DL operators.

We evaluate oclCUB on systems with different hardware and drivers. The results verify that the oclCUB-based operators compute correctly on various platforms. The results also illustrate that oclCUB can maintain a smaller, acceptable performance gap with CUB and is comparable in performance to hipCUB.

The rest of this paper is organized as follows. Section 2 analyzes the relevant computing libraries and current back ground. In Sect. 3, we present the research and implementation methodology of oclCUB. Section 4 presents the application of oclCUB. In Sect. 5, we evaluate oclCUB from several aspects. Section 6 concludes the work of this paper and plans future research.

2 Background and related work

CUB and hipCUB are frequently used computing libraries in frameworks, which implement common computations in DL operators, including reduction, prefix scan, parallel sort, etc. This section analyzes CUB, hipCUB, and other similar computing libraries and indicates the motivation of our work. The comparison of these computing libraries is listed in Table 1.

CUB is an essential computing library that supports reusable collective primitives for the CUDA programming model. CUB is part of the CUDA computing ecosystem and supports host-side and device-side programs. Although CUB is widely used to accelerate DL operators, it is restricted to Nvidia GPUs.

Corresponding to the CUB, AMD proposed hipCUB based on the Radeon Open Computing platform (ROCm) (AMD ROCm 2022a, 2022b). HipCUB is essentially an upper-level encapsulation of rocPRIM (AMD ROCm 2022a, 2022b) or CUB. HipCUB uses rocPRIM as its computational backend on the ROCm platform, but can also use CUB as an alternative backend by using the HIPify tool to port the CUB project to the HIP layer.

Thrust (Bell and Hoberock 2012) is a C++ template library proposed by Jared Hoberock et al., offering common algorithms and unique data structures. Thrust supports several backend technologies, including CUDA, Open Multi-Processing (OpenMP) (Dagum and Menon 1998) and Intel® Threading Building Blocks (TBB) (Pheatt 2008), but only supports acceleration based on NVIDIA GPUs and Intel multi-core CPUs. While Thrust uses GPUs to perform computational tasks, the interface it provides is primarily invoked from the host side. Thrust is unsuitable for designing complex algorithms when programming kernel functions because it is not as flexible as CUB.

CUB and hipCUB can't run on devices from other vendors. Other computing libraries like Thrust have special interfaces and data structures that are difficult to use effectively in DL frameworks. Users have limited options for accelerating their DL operators. Therefore, it is critical to implement a parallel computing library for the relevant computations that satisfy the following conditions: (1) The library should integrate well with DL frameworks and not be limited to vendor-specific hardware accelerators. (2) The

Table 1 The comparison of the relevant computing libraries

Items	CUB	hipCUB	Thrust	oclCUB
Portability and Generality	×	×	×	✓
High Performance Compared to CPU Calculations	✓	✓	✓	✓
Supports Reusable Kernel Computations	✓	✓	✓	✓
Provides Both Host-side and Device-side Interfaces	✓	✓	✓	✓

library should provide host-side and device-side interfaces to support DL operators more efficiently, while library-based operators should perform better than CPU-only operators. (3) The library should implement reusable kernel computations and maintain excellent extensibility to facilitate subsequent research in other areas.

3 Research and implementation

3.1 Overview of oclCUB

OclCUB is an OpenCL-based computing library that works on a wide range of hardware accelerators, including GPGPU, FPGA, DSP, etc. OclCUB supports users in designing and optimizing DL operators efficiently in frameworks.

OclCUB follows a modular and hierarchical design, which is convenient for later extensions and optimizations. Figure 1 shows the architecture of oclCUB. We abstract four layers in oclCUB: the OpenCL device layer, the kernel computation layer, the mapping layer, and the interface layer.

The OpenCL device layer is the basis for oclCUB to perform computations. This layer corresponds to various hardware accelerators, which are managed by the OpenCL platform APIs.

The kernel computation layer is decoupled from other modules, enabling better reuse of these computations. Writing and debugging kernel is a challenging and time-consuming task. To provide flexible interfaces and improve programming efficiency, we implement kernel codes related to parallel algorithms and abstract them as a kernel computation layer.

The mapping layer manages two types of interfaces according to two mechanisms. The implementation of operators in DL frameworks is generally heterogeneous, requiring host and device collaboration for computation. To this end, we abstract a mapping layer between the kernel computation layer and the interface layer, providing interfaces for the host and device sides, respectively.

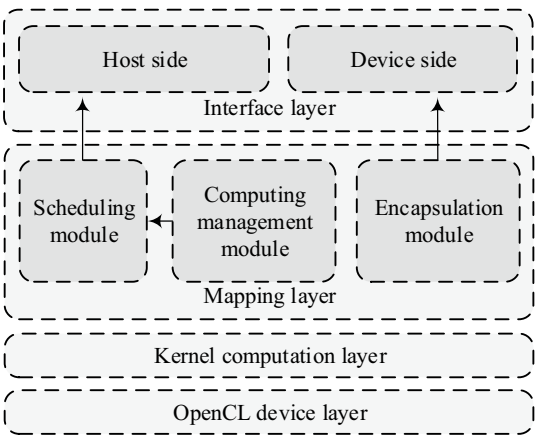

Fig. 1 Architecture of oclCUB

The interface layer contains a series of interfaces provided by oclCUB. These interfaces have two types corresponding to the two types of processing mechanisms in the mapping layer, one targeting host-side programs and the other targeting device-side programs.

The above describes the architecture of oclCUB. Next, we will analyze the common problems when using OpenCL in large-scale projects and present some specific designs to address corresponding challenges.

3.2 Abstracting the computing model

The APIs provided by OpenCL are com plex and not suitable for direct use in large-scale projects. From extensive practical experience, we observe the following problems: (1) The complex execution environment in OpenCL needs to be initialized before each computation. In computationally intensive scenarios, redundant initialization makes the application less efficient. (2) Excellent compatibility and portability make OpenCL more advantageous, but it requires more optimization and management for general hardware accelerators. Especially in the compilation stage of OpenCL kernel functions, the OpenCL runtime dynamically compiles the device-side codes into binary results at each computation. But in intensive computing, repetitive compilation comes with excessive performance loss. While some hardware vendors provide relevant ways to eliminate this impact, the implementation and principle are incompatible between different hardware vendors, making it difficult to apply across devices.

For better execution of oclCUB in DL frameworks, we abstract the core OpenCL computing model and form it into a computing management module in the mapping layer that provides the necessary OpenCL execution environment and assists in implementing interface invocation.

The computing management module is a standardized set of abstractions for encapsulating and managing the OpenCL computing model. The coxputing management module consists of three core classes: the Manager class, the KernelManager class, and the MemManager class. The Manager class is the computing management module manager, responsible for collaborating with other core classes. We design the Manager class using the singleton pattern, initializing the OpenCL execution environment only once in a complete DL program to avoid redundant overhead. We design the MemManager class to receive device-side data and convert it to an OpenCL memory object which is the parameter needed for kernel computations. The MemManager class clears these memory objects according to whether the program has finished.

The KernelManager class handles the preparation of the OpenCL kernel before execution, including setting parameters, building program, etc. When building a program for the first time, we use a high-performance C++ container to cache the compilation results. Subsequently, identical kernels can use the compilation results directly, avoiding the overhead of repeated compilation. Figure 2 illustrates the computing management module.

3.3 Designing two types of programming interfaces

The operators in DL frameworks are typically in heterogeneous mode, including data processing on the host side and high-speed computation on the device side. On the host side, much data processing does not require writing additional kernel functions, which can be accelerated by reusable kernel computations. So we design two types of interfaces separately, which are more in line with the mechanism of heterogeneous acceleration.

The mapping layer's scheduling and encapsulation modules provide the host and device interfaces, respectively. Based on the computing management module, we implement the host-side interface invocation process in the scheduling module, including interface definitions, thread organization deployment, data processing, and kernel code scheduling. In the encapsulation module, we encapsulate complex kernel codes into reusable device functions, allowing kernel computations to be reused in the design of high-level kernels. We design the interface invocation to be procedure-oriented, effectively reducing programming difficulty and providing better compatibility with higher versions of OpenCL. Figure 3 illustrates the design scheme of the interface, targeting both host-side and kernel functions.

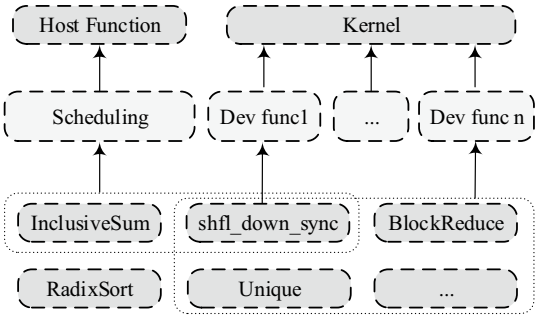

Fig. 3 The interface scheme with reusable kernel computations

3.4 Reusable kernel computations

The kernel computation is part of oclCUB that performs the actual computations. We decouple the kernel computation layer from other modules, allowing kernels implementing DL operators can be reused and recompiled. So oclCUB's kernel computations have become more scalable, making it easier to add new computing interfaces. Reusability is reflected in the kernel computations' availability in multiple host-side and kernel functions, as shown in Fig. 3.

Previous research has provided an in-depth analysis of parallel algorithms for related computations, such as (Martín et al. 2012; Adinets and Merrill 2022), etc. These works provide a practical guide for writing kernel codes in oclCUB. However, different programming models have distinctive characteristics. With portability and compatibility in mind, the implementation based on OpenCL is still complex and challenging.

We observe that the scheduling hardware of the OpenCL acceleration devices assigns tasks to individual kernels on a workgroup basis after the program starts. A key problem arising from this mechanism is that synchronizing data across multiple workgroups becomes difficult. Most

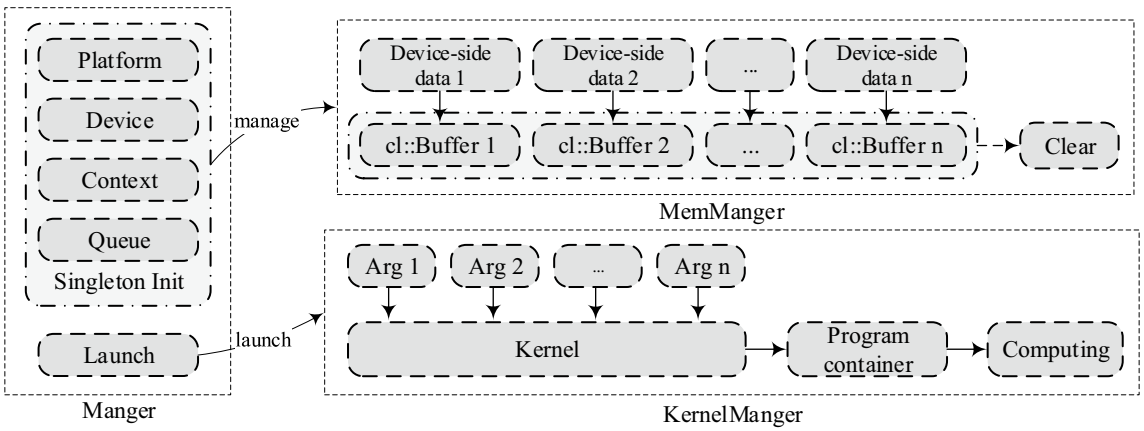

Fig. 2 The computing management module

acceleration devices also do not provide synchronization between workgroups. To this end, we adopt the strategy of algorithm splitting in implementing some parallel algorithms. That is, the specific implementation of an algorithm is appropriately split into multiple kernel functions while ensuring the overall idea of the algorithm. This strategy can effectively reduce data dependency. The computing management module can significantly reduce redundant time consumption when launching multiple kernel functions.

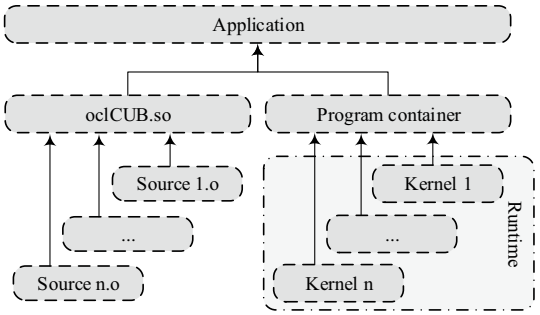

Fig. 4 The compilation of oclCUB

4 Application

In this section, we describe the compilation of oclCUB and how to use different types of interfaces in the DL operators.

4.1 The compilation of oclCUB and associated programs

Heterogeneous programs are based on the heterogeneous system consisting of CPU processors and hardware accelerators, which are compiled differently than homogeneous programs. Heterogeneous programs are written by heterogeneous programming models, and there is also a significant difference between the different programming models when compiling applications. CUDA is a single-source programming model whose compiler can simultaneously compile host-side and device-side codes. However, OpenCL is a multi-source programming model, and we should consider the features when compiling.

We compile oclCUB using the CMake tool, a process that requires few dependencies (OpenCL, G++, Python), and oclCUB can be compiled on a wide range of devices and systems. We let oclCUB first compile the host-side source codes while processing the kernels into the string parameters the OpenCL compilation interface needs. The device-side kernels are compiled at program runtime, and we cache the results in a high-performance C++container. Figure 4 illustrates these processes.

4.2 Utilizing the programming interfaces

The operator is the basic computing unit in DL frameworks. The application of computing libraries in DL frameworks generally revolves around implementing operators. For oclCUB, there are three elements for designing and optimizing an operator in DL frameworks: the execution environment provided by the computing management module, two types of interfaces provided by the mapping layer, and the kernel programming language of OpenCL. Next, we demonstrate with a simple example.

```
1. #include <oclcub/core/opencl-base.h>
2. #include
<oclcub/core/InclusiveSum.hpp>
3. template <typename T>
4. void operator(T* in, T* out, int
in_size, ...) {
5.     ...
6.     // Instance init
7.     Manager &manager =
Manager::instance();
8.     auto context =
manager.get_context();
9.     auto q = manager.get_queue();
10.    MemManager mManager;
11.    KernelManager kManager;
12.    cl::Buffer cl_in, cl_out, cl_tmp;
13.    // Convert device data to cl::Buffer
14.    if (SUCCESS !=
         mManager.addMem(cl_in, context,
         cl_mem_flags, sizeof(T) * in_size,
         const_cast<T*>(&in[0])) | ...) return;
15.    // Call kernel that use device-side
interface
16.    auto k = kManager.get_kernel(Foo);
17.    manager.launch(k, cl_in, cl_out,
cl_tmp, 0);
18.    // Call host-side interface
19.    oclCUB::core::InclusiveSum(q, cl_in,
                                  cl_out,
                                  in_size, ...);
20.    ...
21. }
```

Listing 1 The use of computing manage-ment module, host-side interface and kernel execution

In Listing 1, we include two header files in lines 1–2. One is the header file for the computing management module; the other is the header file that defines the corresponding host-side interface. Lines 6–14 show the usage of the singleton-based computing management module, lines 15–17 execute an OpenCL kernel that uses a device-side interface, and lines 18–19 call a host-side interface. One thing to note is that using oclCUB requires linking the oclCUB dynamic link library in DL frameworks.

```
1. #include "WarpReduce.h"
2. __kernel void Foo(__global Dtype* in,
__global Dtype* out, __global Dtype*
temp_storage, Dtype initVal, ...) {
3.     ...
4.     Dtype sum = initVal;
5.     // Call device-side interface
6.     sum = WarpReduceSum(1, sum,
temp_storage);
7.     ...
8. }
```

Listing 2 The use of device-side interface in OpenCL kernel

We show the usage of a device-side interface in Listing 2. Calling a device-side interface only requires including the corresponding header file in the OpenCL kernel. The first line is a header file containing the device-side interfaces, and line 6 is an invocation of the device-side interface.

5 Evaluations

In this section, a series of experiments are designed to evaluate the portability of oclCUB, the operator accuracy, the gains of the computing management module, and the operator performance.

5.1 Experimental subject and environment

Our experiments revolve around operators. In the subsequent experiments, we select six operators from PyTorch and TensorFlow frameworks that rely on CUB for acceleration. Then we perform a corresponding implementation of these operators based on oclCUB. We use CPU-based operators as benchmarks and oclCUB-based operators as experimental subjects to evaluate the portability and correctness of oclCUB. Subsequently, we use CUB-based operators and hipCUB-based operators as benchmarks, and oclCUB-based operators as experimental subjects to evaluate the performance of oclCUB.

We compose four experimental environments using different vendors' devices and drivers (Zhang et al. 2018; Lu et al. 2022; Jääskeläinen et al. 2015). Table 2 illustrates the specific information of the three environments. We use four environments to verify the portability and correctness of oclCUB. Environment 1 ensures that CUB and oclCUB utilize the same acceleration device and thread resources. Environment 4 ensures that hipCUB and oclCUB utilize the same acceleration device and thread resources. Therefore, we use Environment 1 and Environment 4 for conducting performance evaluation experiments.

5.2 Portability and accuracy

A key point of oclCUB is its portability, which we will verify by testing the six operators mentioned above in three different environments. At the same time, we will also test the computational accuracy of the oclCUB-based operators.

We take the same random data as input and run these operators in three environments. We use the computational results of the CPU-based operators as a benchmark and then calculate the relative errors of the oclCUB-based operators. In most functions of the DL frameworks, we observe that the default relative error for determining data equality is 1e-5. So we set the standard of accuracy based on this value: errors less than or equal to 1e-5 is considered correct, and errors greater than 1e-5 is considered incorrect. For integer types, we consider it correct only if the computation results are exactly the same. The input data volume is set to 2^20. The experimental data types include 32-bit integer, 64-bit integer, 32-bit floating point, and 64-bit floating point.

Table 3 illustrates the test results of these operators in three environments. The oclCUB-based operators behave consistently in different data types. According to the standard we set, all operators pass the test in three experimental environments. The error magnitude 1e-5 is within an acceptable range in DL applications, verifying oclCUB's portability and accuracy.

5.3 Gains of computing management module

In Sect. 3, we present the oclCUB's architecture and the design of specific modules. The interfaces design and reusable kernel computations have been demonstrated in Sect. 4. Here we will test the gains from the computing management module.

The computing management module optimizes the application efficiency of OpenCL in large-scale projects. It is

Table 2 The experimental environments. "*" means that the parameter is not existing in this environment

Parameters	CPU	Clock	OpenCL Driver	Accelerator	Host Compiler
Environment1	Intel Xeon Gold 5218	2.30 GHz	OpenCL 1.2	Tesla V100s	GCC 7.5.0
Environment2	FT-2000	2.60 GHz	MOCL 3	MT-3000	GCC 9.3.0
Environment3	Kunpeng 920	2.60 GHz	POCL 1.6	*	GCC 7.3.0
Environment4	AMD EPYC 7R32	2.80 GHz	OpenCL 2.0	RADEON PRO V520	GCC 9.4.0

Table 3 The accuracy of the oclCUB-based operators. "✓" stands for correct. "*" means that the operator's input does not support this data type

Data Type	max	nllloss	scatter	softmax	biasadd	sum
int32(Environment1)	✓	*	✓	*	✓	✓
int32(Environment2)	✓	*	✓	*	✓	✓
int32(Environment3)	✓	*	✓	*	✓	✓
int32(Environment4)	✓	*	✓	*	✓	✓
int64(Environment1)	✓	*	✓	*	✓	✓
int64(Environment2)	✓	*	✓	*	✓	✓
int64(Environment3)	✓	*	✓	*	✓	✓
int64(Environment4)	✓	*	✓	*	✓	✓
float32(Environment1)	✓	✓	✓	✓	✓	✓
float32(Environment2)	✓	✓	✓	✓	✓	✓
float32(Environment3)	✓	✓	✓	✓	✓	✓
float32(Environment4)	✓	✓	✓	✓	✓	✓
float64(Environment1)	✓	✓	✓	✓	✓	✓
float64(Environment2)	✓	✓	✓	✓	✓	✓
float64(Environment3)	✓	✓	✓	✓	✓	✓
float64(Environment4)	✓	✓	✓	✓	✓	✓

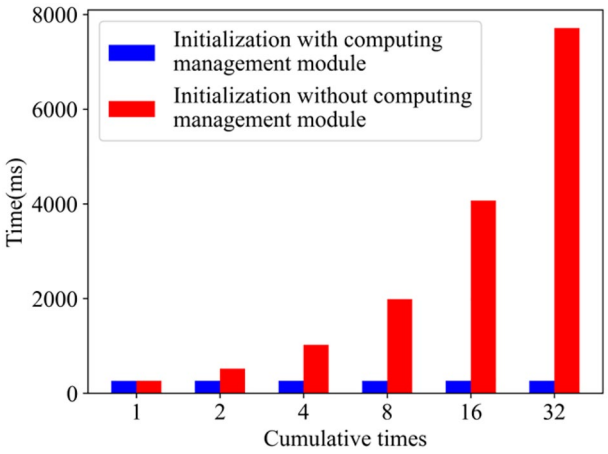

Fig. 5 The gains of computing management module in initializing the execution environment

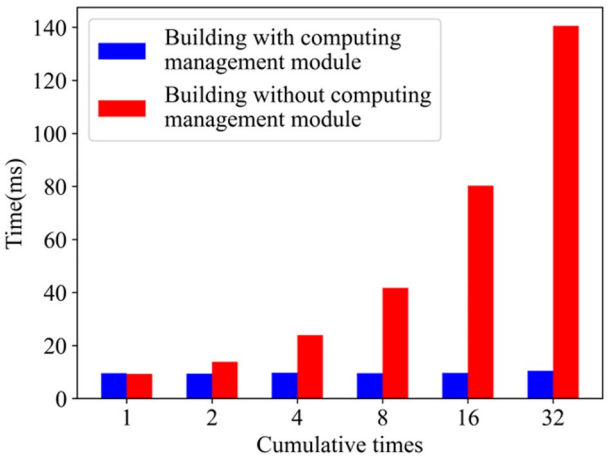

Fig. 6 The gains of computing management module in building the program

reflected in the two most time-consuming stages, the initialization of the execution environment and the program's building. We use the biasadd operator as an example and test it with the 2^20 data volume. DL generally has a large number of repeated computations, so we call an operator multiple times and accumulate the total time elapsed for each stage.

The gains are shown in Figs. 5 and 6. The horizontal axis represents the number of operator cumulative executions, and the vertical axis represents this stage's accumulative total elapsed time. As the number of operator executions increases, both the initialization of the execution environment and the program's building, the unoptimized elapsed time shows an exponential increase due to the repeated overhead, while the optimized elapsed time is always kept at a low level. The results show that the gains of the computing management module for these two stages are highly significant.

6 Performance of oclCUB-based operators

Another key point in evaluating oclCUB is the performance of the oclCUB-based operators. The original intention of oclCUB is to enable general hardware accelerators to accelerate DL operators. This means that when the hardware accelerator has strong parallel computing capabilities, the performance of the oclCUB-based operators should achieve comparatively good expected results.

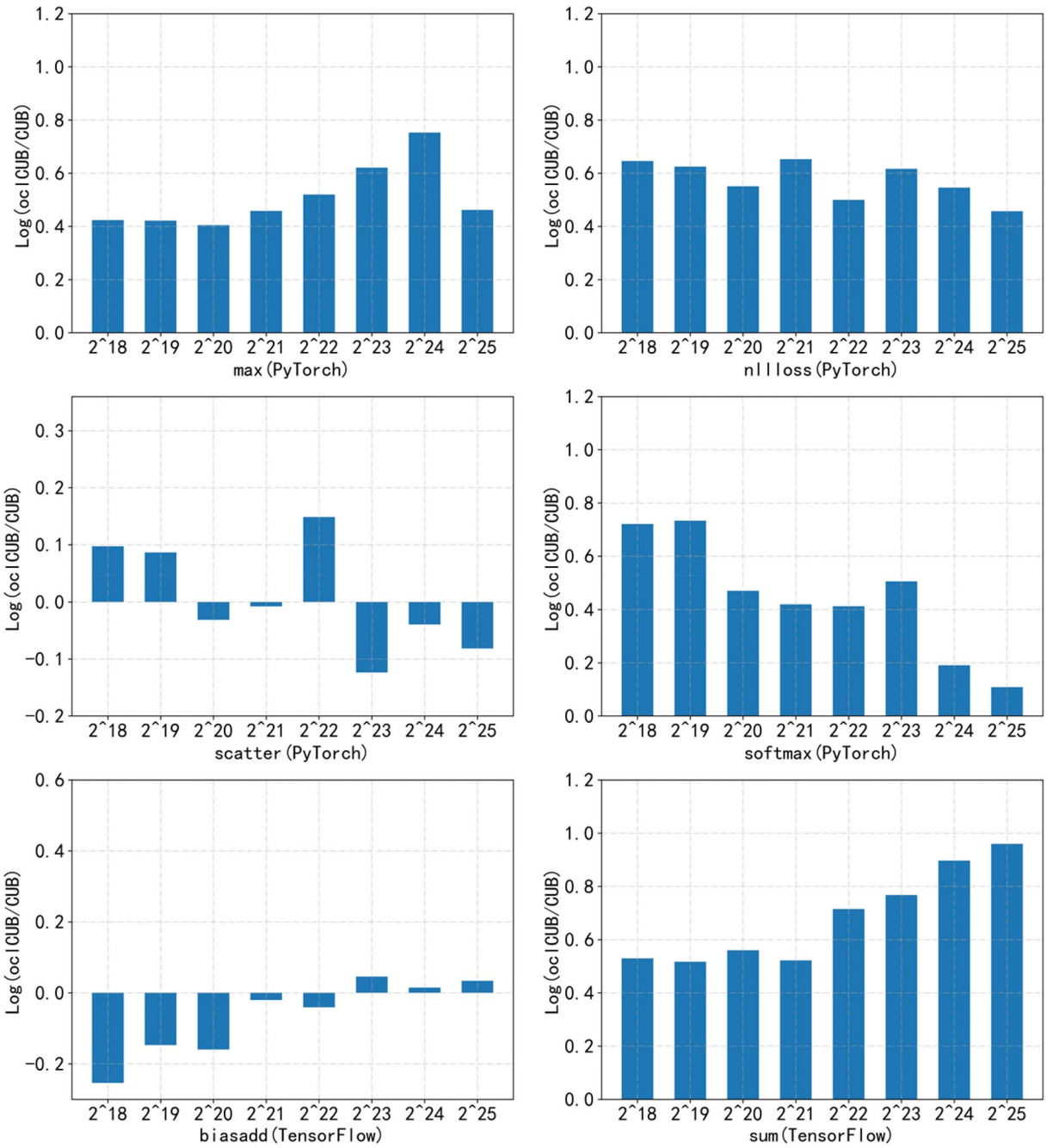

Fig. 7 Comparison of the performance between the oclCUB-based operators and the CUB-based operators

The oclCUB-based operators and CUB-based operators are run in Environment 1, using the same random data as input with float32 data type. The data volume of the input is incremental from 2^18 to 2^25. To eliminate the influence of other factors, we let operators execute multiple times and record the elapsed time each time. Then we average these data and evaluate the performance based on this value. For the convenience of presenting the results, we divide the execution time of the oclCUB-based operators by the execution time of the CUB-based or hipCUB-based operators, then take the logarithm of this result.

Figure 7 shows the performance of operators based on CUB and oclCUB. The horizontal axis represents operators at different data scales. The vertical axis represents the evaluation metric. When the evaluation metric is less than 0, it indicates that the performance of oclCUB-based operators is better

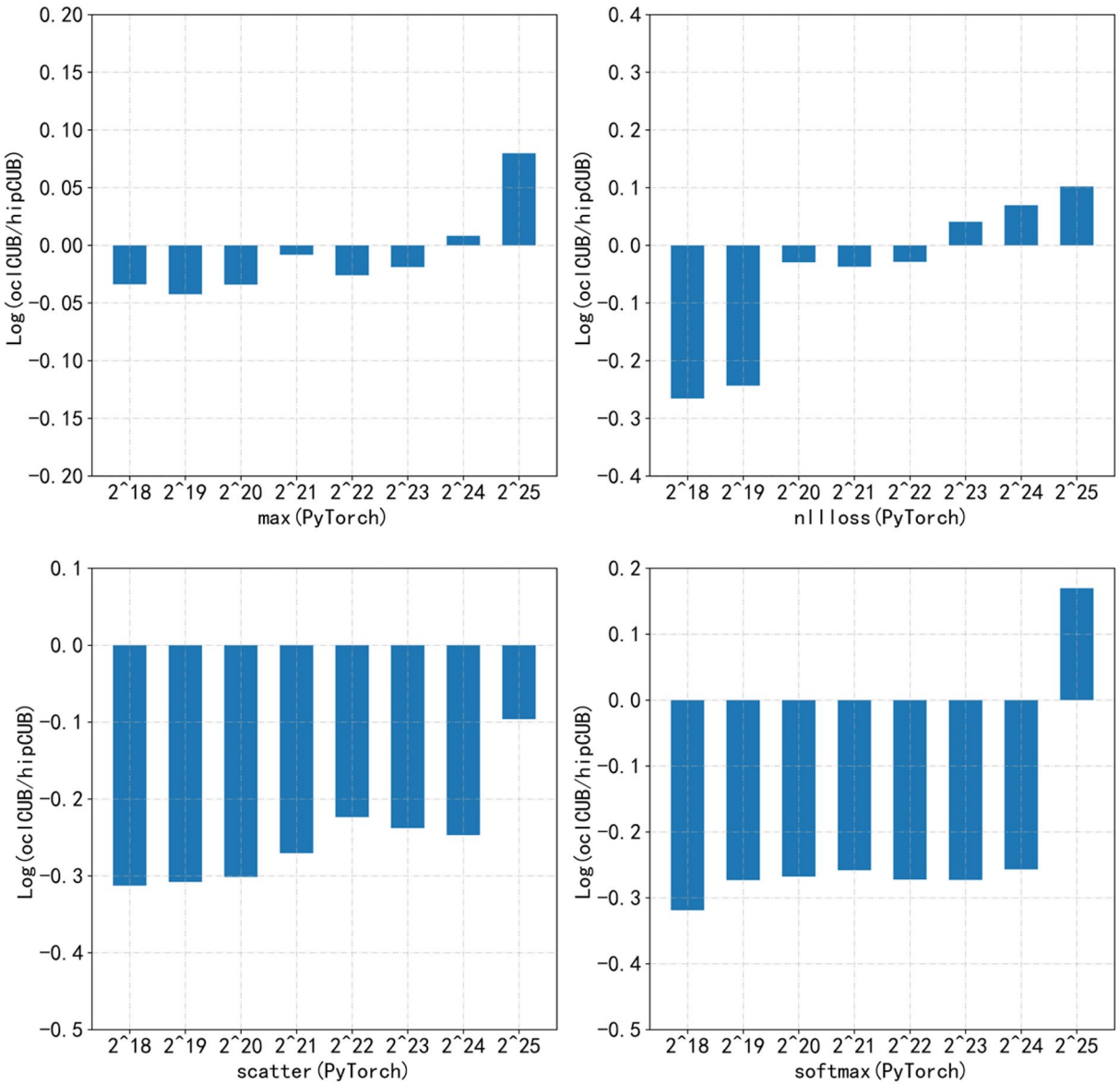

Fig. 8 Comparison of the performance between the oclCUB-based operators and the hipCUB-based operators

than CUB-based operators. Conversely, when the evaluation metric is greater than 0, it suggests that the performance of CUB-based operators is superior to oclCUB-based operators. In most cases, the performance of oclCUB-based operators is lower than CUB-based operators. However, there are also a small number of cases that the performance of oclCUBbase operators is slightly better than CUB-based operators. Overall, with the evaluation metric ranging between – 0.3 and 1, this implies that the performance difference between oclCUB and CUB is not significant. Additionally, operators in different frameworks exhibit similar performance. Attributable to the exclusive optimizations within the CUDA ecosystem, CUB is able to maximize its performance. In such millisecond-level computations, it is acceptable and expected that oclCUB maintains a smaller performance gap with CUB.

Figure 8 shows the performance comparison between oclCUB and hipCUB. Overall, the evaluation metrics range from – 0.4 to 0.3, indicating that the performance of oclCUB-based operators is slightly better than that of hipCUB-based operators. Only in cases of large data scales does the performance of hipCUB-based operators surpass that of oclCUB-based operators. As the data scale increases, the performance gap between the two tends to decrease, but the difference remains consistently small. Although the AMD RADEON PRO V520 supports ROCm and HIP for executing deep learning tasks, hipCUB does not receive extensive optimization on this device. In this scenario, oclCUB is able to achieve

performance comparable to hipCUB, and even slightly better. In summary, although the portability of oclCUB may slightly weaken its exclusive optimization for specific devices, oclCUB still achieves relatively high efficiency in performing computational tasks.

7 Conclusion

In this paper, we propose and implement a scheme for the OpenCL computing library with relevant essential functions. This computing library is designed to accelerate DL operators using general hardware accelerators. We maintain an abstraction of the OpenCL execution environment in oclCUB to enable effective integration with DL frameworks. We design two types of interfaces to support DL operators in heterogeneous acceleration mode better. We implement a range of reusable kernel computations based on OpenCL to help users design and optimize DL operators. In our evaluation experiment, we demonstrate that oclCUB possesses a good portability, and its architectural design yields significant gains. The experimental results also indicate that the performance of oclCUB has a small, acceptable gap compared to CUB and is comparable in performance to hipCUB. In future work, we will consider expanding the computational content of the computing library, running it efficiently in DL models, and further validate performance portability on outstanding devices.

Acknowledgements This research is supported by National Key R&D Program of China grant 2021YFB0300104, as well as by Tianjin Research Innovation Project for Postgraduate Students grant 2022BKY023.

Declarations

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

References

Abadi, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)

Adinets, A., Merrill, D.: Onesweep: a faster least significant digit radix sort for GPUs. arXiv preprint arXiv:2206.01784 (2022)

AMD ROCm: A thin wrapper library on top of rocPRIM or CUB. https://github.com/ROCmSoftwarePlatform/hipCUB (2019a)

AMD ROCm: A C++ Runtime API and Kernel Language. https://github.com/ROCm-Developer-Tools/HIP (2019b)

AMD ROCm: AMD ROCm Platform Documentation. https://rocmdocs.amd.com/ (2022a)

AMD ROCm. A header-only library providing HIP parallel primitives. https://github.com/ROCmSoftwarePlatform/rocPRIM (2022b)

Bell, N., Hoberock, J.: "Thrust: A Productivity-Oriented Library for CUDA." GPU Computing Gems, Jade, pp. 359–371. Morgan Kaufmann (2012)

Cao, C., et al.: clMAGMA: high performance dense linear algebra with OpenCL. In: Proceedings of the International Workshop on OpenCL 2013 & 2014 (2014)

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

Chetlur, S., et al. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)

Cublas, N.C.: Library. NVIDIA Corporation, Santa Clara (2008)

Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)

Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: 2011 International Conference on Parallel Processing. IEEE (2011)

Intel: oneAPI Deep Neural Network Library. https://github.com/oneapi-src/oneDNN (2019)

Jääskeläinen, P., de La Lama, C.S., Schnetter, E., et al.: pocl: A performance-portable OpenCL implementation. Int. J. Parallel Prog. **43**, 752–785 (2015)

Khan, J., et al.: Miopen: an open source library for deep learning primitives. arXiv preprint arXiv:1910.00078 (2019)

Kirk, D.: NVIDIA CUDA software and GPU parallel computing architecture. In: ISMM. Vol. 7 (2007)

Komatsu, K., et al.: Evaluating performance and portability of OpenCL programs. In: The Fifth International Workshop on Automatic Performance Tuning. Vol. 66 (2010)

Lu, K., Wang, Y., Guo, Y., et al.: MT-3000: a heterogeneous multi-zone processor for HPC. CCF Trans. High Perform. Comput. **4**(2), 150–164 (2022)

Martín, P.J., Ayuso, L.F., Torres, R., et al.: Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In: 2012 International Conference on High Performance Computing & Simulation (HPCS). IEEE, pp. 511–519 (2012)

Merrill, D. CUB v1. 5.3: CUDA Unbound, a library of warp-wide, blockwide, and device-wide GPU parallel primitives. NVIDIA Res. (2015)

Nichols, D., et al.: MagmaDNN: accelerated deep learning using MAGMA. In: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) (2019)

Paszke, A., et al.: Pytorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32 (2019)

Pheatt, C.: Intel® threading building blocks. J. Comput. Sci. Coll. **23**(4), 298–298 (2008)

Rupp, K., et al.: ViennaCL–-linear algebra library for multi-and many-core architectures. SIAM J. Sci. Comput. **38**(5), S412–S439 (2016)

Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66 (2010)

Zhang, P., Fang, J., Yang, C., et al.: Mocl: an efficient OpenCL implementation for the matrix-2000 architecture. In: Proceedings of the 15th ACM International Conference on Computing Frontiers, pp. 26–35 (2018)

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

Yuqiao Chen is a graduate student in College of Software at Nankai University. His research interests include heterogeneous computing and computer architecture.

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

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

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

Yuzhi Zhang is the chair professor and the Dean of software college in Nankai University. His research interests focus on artificial intelligence, etc.