计算机科学
COMPUTER SCIENCE

Optimization and Deployment of Memory-Intensive Operations in Deep Learning Model on Edge

Peng XU, Jianxin ZHAO, Chi Harold LIU

引用本文

Peng XU, Jianxin ZHAO, Chi Harold LIUOptimization and Deployment of Memory-Intensive Operations in Deep Learning Model on Edge[J]. 计算机科学, 2023, 50(2): 3-12.

Peng XU, Jianxin ZHAO, Chi Harold LIU. Optimization and Deployment of Memory-Intensive Operations in Deep Learning Model on Edge [J]. Computer Science, 2023, 50(2): 3-12.

Optimization and Deployment of Memory-Intensive Operations in Deep Learning Model on Edge Peng

Peng XU, Jianxin ZHAO, Chi Harold LIU

Citation

Peng XU, Jianxin ZHAO, Chi Harold LIU. Optimization and Deployment of Memory-Intensive Operations in Deep Learning Model on Edge Peng[J]. Computer Science, 2023, 50(2): 3-12.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.20221100135

Optimization and Deployment of Memory-Intensive Operations in Deep Learning Model on Edge

Peng XU, Jianxin ZHAO and Chi Harold LIU
Department of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
(xupeng_mii@163.com)

Abstract As a large amount of data is increasingly generated from edge devices, such as smart homes, mobile phones, and wearable devices, it becomes crucial for many applications to deploy machine learning modes across edge devices. The execution speed of the deployed model is a key element to ensure service quality. Considering a highly heterogeneous edge deployment scenario, deep learning compiling is a novel approach that aims to solve this problem. It defines models using certain DSLs and generates efficient code implementations on different hardware devices. However, there are still two aspects that are not yet thoroughly investigated yet. The first is the optimization of memory-intensive operations, and the second problem is the heterogeneity of the deployment target. To that end, in this work, we propose a system solution that optimizes memory-intensive operation, optimizes the subgraph distribution, and enables the compiling and deployment of DNN models on multiple targets. The evaluation results show the performance of our proposed system.

Keywords Memory optimization, Deep compiler, Computation optimization, Model deployment, Edge computing
Chinese Library Classification TP311.5

1 Introduction

The fast development of machine learning models, especially deep learning (DL) models, has made a huge impact on a lot of fields, such as transportation[1], health[2], biology[3], etc. As a large amount of data is increasingly generated from edge devices, such as smart homes, mobile phone, and wearable devices, it becomes crucial for many applications to deploy machine learning modes across edge devices[4].

The execution speed of the deployed model is a key element to ensure service quality. There has already been a lot of technology developed for that purpose. Vendor libraries such as MKL and cuBLAS provide highly optimized computation performance for specific hardware architecture[5]. However, they lack support for specifically customized operators, and this approach is limited by specific hardware. In a highly heterogeneous edge deployment scenario, this approach is not viable.

Deep learning compiler is a new approach that aims to solve this problem[6]. The DL compilers take the model definitions described by certain DSLs, and generate efficient code implementations on different hardware devices. This field has attracted wide interest from academia and industry. However, there are still two aspects that are not yet thoroughly investigated yet.

The first is the optimization of memory-intensive operations. In deep learning compilers, a DNN is represented as a computation graph, where operators are the nodes in this graph. There are two types of operations. The first is computation-intensive, such as convolutions. They require a lot of computation resources. They are the major focus of current work. The other type is memory-intensive operations, e.g., copying a large matrix from source to destination. Such operations do not involve quite complex calculations but do a lot of memory operations. It has been shown that these operations have a non-trivial impact on the performance of computation.

Fig. 1 shows the ratio of memory-intensive computation for five representative models used in real-life production on GPU for tasks such as NLP, recommendation, speech, or image recognition. With an average ratio of 63% in execution time and 90% in total kernel numbers, memory-intensive computation has already become a dominating factor that significantly impacts the training and inference efficiency of many recent DNN workloads. Therefore, it is crucial to optimize memory-intensive operations.

A memory-intensive computation graph usually consists of tens or even hundreds of operators. There are two levels of dependencies. Operator-level dependency describes the operator connection represented in a subgraph. Element-level dependency indicates the dependency between elements within

Submission data:2022-11-15 Modification data:2023-01-19
This work was supported by the National Natural Science Foundation of China(U21A20519).
Corresponding author:Jianxin ZHAO(jianxin.zhao@bit.edu.cn)

tensors. The two-level dependencies combined with JIT demand make fusion optimization extremely challenging for modern memory-intensive ML models.

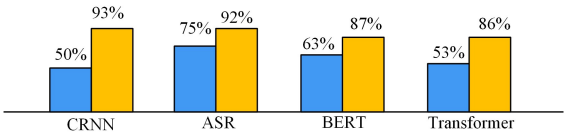


Fig. 1 Ratio of memory-intensive computations[7]

Faced with this problem on memory-heavy operations, the current deep learning compiler frameworks face the following dilemma when conducting fusion on these two patterns. The first is fusion leads to the redundant computation. When there are one-to-many element-level dependencies, as in repeat or reduction operations, where one element generated by the producer is required by multiple elements of the consumer(s), each thread of the consumer will independently compute this common element, causing significant computation redundancy.

The second problem is the heterogeneity of the deployment target. The compiled code can be deployed on a wide range of hardware accelerators. One popular choice is a graphics processing unit, or GPU. It provides a huge space for parallel execution. Recently there are also researches that focus on the Tensor Processing Unit (TPU). It is an accelerator developed by Google specifically for neural network machine learning[8]. Moreover, edge computing is fast growing. That leads to deployment on multiple lightweight target forms, including virtual machines and containers, etc.[9]. Due to the limited memory and computing resource, such deployment again requires optimization at the model compiling phase.

To that end, in this work, we propose a system solution-Owl that enables end-to-end deployment of optimized models on devices. Fig. 2 shows an overview of it. In Sec. 3, we present the optimization of two typical memory-intensive operations in DNN: repeat and reduction. In Sec. 4, we introduce a fair subgraph fusion scheme to address the challenge of computation redundancy-inefficiency tradeoff posed by memory-intensive operations in computation graphs. In Sec. 5, we explain the various aspects of the model deployment module, which lays the foundation of the previous two functionalities.

The novelty of this work is two-fold. The first is to propose a subgraph fusion scheme that fairly distributes the deployed subgraph resources in an edge computing environment, so as to utilize the model compiling optimization techniques, notably fusion, for computations with memory-intensive operations. The second is to propose a framework that incorporates the aforementioned scheme with optimizations on the performance of memory-intensive operators, supported by model deployment on multiple targets, including edge devices. For the rest of this work, we first briefly show related work and then discuss these aspects in detail.

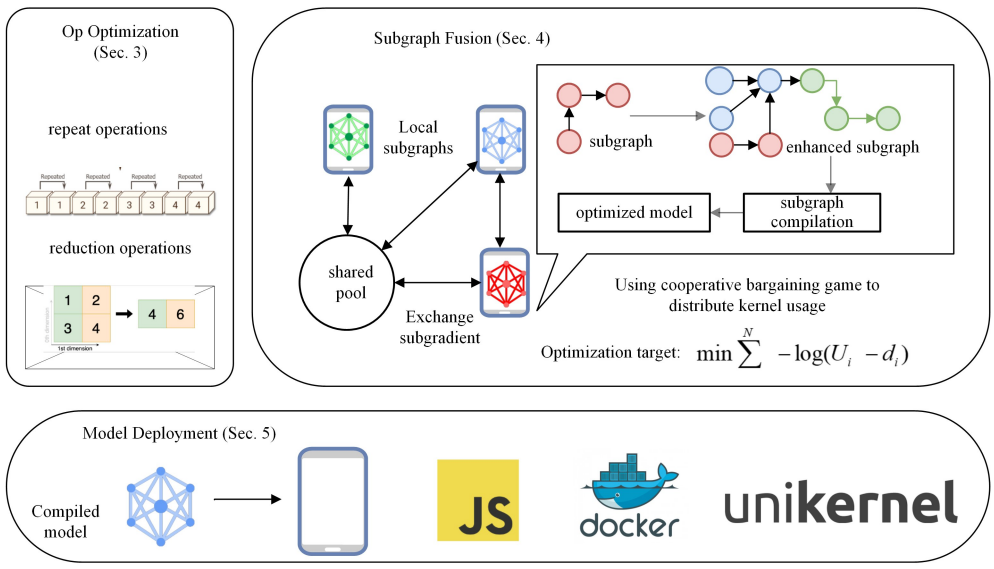


Fig. 2 Overview of the proposed Owl framework

2 Related Work

With the development of the Internet, more and more data are generated from devices such as mobile phones, smart homes, and self-driving cars. Deep Learning lays a foundation for using algorithms to parse and learn data. Numerous works have been proposed to address issues such as the unstable network environment of edge devices[10], reducing bandwidth usage and network data transmission delay[11], and ensuring user privacy security[12]. There are also new paradigms such as Federated Learning, which is a distributed deep learning training method that aims to build personalized models on edge devices[13-15].

2.1 Model Compilation and Optimization

Each neural network can be represented as a mathematical function. Due to their complexity, they are often expressed

as a graph instead of mathematical notations. Most research work and tools in the field of neural network are developed based on these graphs. Formally, they are called computation graphs and defined as directed graphs. Each node represents either input or computation, such as multiplication, summation, etc. The edges represent calling order among nodes. For example, Fig. 3 is a part of the computation graph of GoogleNet[16], a complex neural network architecture.

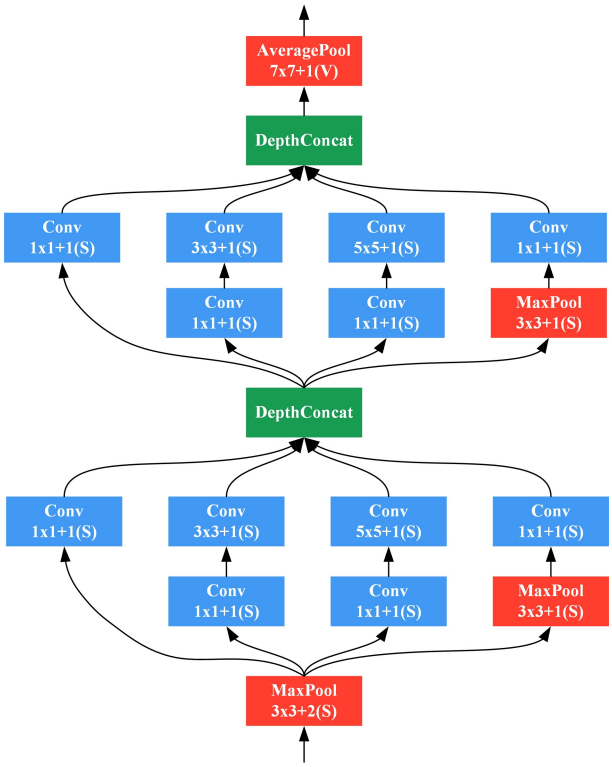


Fig. 3 An example of computation graph: part of the GoogleNet[16]

With the rapid development of deep learning, deep learning compilers are in the early stage of development. There are some popular deep learning compilers in the industry. Apache comes up with TVM, an end-to-end deep learning compiler. It is easily portable across hardware devices. Intel's open source nGraph[17] supports efficient memory management and data layout abstraction across several deep learning frameworks and hardware platforms. Along with the deep learning compiler I mentioned above, there are Meta Tensor[18], Google XLA[19], Glow[20], etc.

Compilers include front end, back end, and intermediate representation (IR), where IR is responsible for optimizing between the front end and back end. IR design is very important for compilers. IR needs to consider the integrity of compilation from source code to object code, ease of compilation optimization, and performance. Google's MLIR[21] provides an infrastructure and specification for multiple layers of IR to facilitate transformation between IRS and improve the possibility of reuse between different compiler optimizations. The TVM compiler was developed on top of the original NNVM IR, supplemented by the second-generation IR Relay[22]. In recent months, Relax has been proposed as the next generation of TVM graph-level IR to improve the efficiency and performance of TVM development.

The optimization of deep model inference and the improvement of neural network compilers are two hot research areas of deep learning compilers. AutoTVM[23] is an automatic optimization framework for the operator layer of compiler TVM. It uses template search space to find the best execution of operators on target hardware. Based on this, FlexTensor[24] and Ansor[25] reduce or even eliminate the dependence on templates in search and improve search results. TASO[26] is a computing layer optimizer that automatically generates good performance calculations by generating candidate replacement graphs and backtracking. The authors of [27] propose to use the polyhedron model to generate high-performance matrix multiplication vector code. In [28], the author designed an optimization framework PET, which first performs partial equivalence optimization at the tensor, operator and graph levels, and then automatically corrects the results to complete equivalence by looking for effective opportunities previously lost in partial equivalence transformation.

2.2 Model Deployment

Most existing machine learning frameworks, such as TensorFlow and Caffe, focus on training analysis models. Deployment of services is close to the idea of model services. The Clipper[29] service system is used for ML model-based prediction, selecting the model with the least delay from the models on multiple ML frameworks. It enables users to access models based on multiple machine learning frameworks. These models are implemented as containers.

TensorFlow Serving[30] is a much better model execution framework than Clipper. This model can be deployed as a container containing TensorFlow to provide predictive requests. Since our work on Zoo was published, several microservice deployment systems, such as Seldon, have been developed. It uses Docker to deploy the model. Seldon defines inference diagrams based on the model, and then deploys these diagrams in a deployment or production environment using the container orchestration system Kubernetes.

TensorFlow Serving[30] is more focused on using TensorFlow itself as the model execution framework than Clipper. The model can be deployed as a container containing TensorFlow to provide forecast requests. There are some micro-service deployment systems, such as Seldon[31]. Seldon defines inference diagrams based on the model, and then uses the container orchestration system Kubernetes to deploy these diagrams in a deployment or production environment.

Machine learning inference services are often latency critical, and the automatic scaling capabilities of serverless com-

puting can handle burst workloads well. Yang et al. presented a solution called INFLess[32], which reduces resource allocation for each serverless instance in order to achieve the best performance for the inference service.

3 Optimization of Memory-intensive Operations

In the computing graph, resource-intensive operations can be divided into computation-intensive operations and memory-intensive operations. The former has been themain topic of research in the past few years, such as better utilization of the parallel computing mechanism provided by hardware to improve computing efficiency and so on. But the importance of the latter is only beginning to be recognized. In this section, we present optimizations of two memory-intensive operations: reduction and repeat.

3.1 Reduction Operations

Reduction operations are an important group of operations in computing. Reduction operations such assum and max accumulate values along a particular axis by a particular function in an n-dimensional array (ndarray). For example, as shown in Fig. 4, a matrix can be reduced to a vector in the row dimension. If the sum operation is used, the result can be the sum of all the elements. If the max operation is used, the result can be the maximum value of those elements. Reduction operations are one of the key operations of advanced applications. For example, sum is used to implement the Batch Normalization neurons that are often used in DNN.

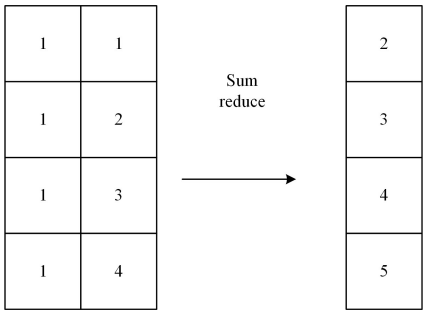


Fig. 4 Example of the sum reduction operations

Reduction operations follow similar patterns, which lead to similar design choices that can be summarized into several patterns. In most cases of these templates, we only need to define the summation function FN.

The reduction operation usually requires a specified axis. One of the challenges we faced is the multi-axis reduction. A simple implementation is to repeat the operation along one axis for each axis specified, and then repeat the process on the next axis. However, each uniaxial reduction requires additional temporary memory to store intermediate results. In applications that make heavy use of reduction operations, such as DNN, the inefficiency of reduction operations becomes a memory and performance bottleneck.

In a single-axis reduction algorithm, the source ndarray x needs to be compressed into a smaller target ndarray y. Suppose the dimension to be reduced is of size a, and the total number of elements in x is n. The basic idea is to iterate over their elements one by one, but the index in y keeps returning to 0 when it reaches $\frac{a}{n}-1$. We modified the procedure so that the index in y iterates repeatedly on a given axis, all using an intermediate memory.

An optimization step prior to this algorithm is to merge adjacent axes. For example, if a ndarray of shape (3,4,5,6) is to be reduced along the second and third axis, it can be simplified to the ndarray of the shape (3,20,6). The proposed algorithm is shown as below.

Algorithm 1 Revised Multi-axes reduction on n-dimenional array

```
Input: source array x, shape of x
Output: target array y with reduction results
Initialize:
1. cnt=0; ndim=length(shape)
2. innersize=x_shape[ndim-1]
3. loopsize=x_shape[ndim-2]
4. compute strides according to shape
5. for ix=0 to N do
6.     for k=0 to innersize do
7.         accumulate x[ix+k] to y[k]
8.     end for
9.     ix += innersize; cnt++;
10.    if cnt == loopsize do
11.        iy=0; cnt=0;
12.        int iterindex=ix; int pre_iteridx=ix;
13.        for i=ndim -1 downto 0 do
14.            iterindex /= shape[i];
15.            residual=pre_iteridx - iterindex * x_shape[i];
16.            iy += residual * strides[i];
17.            pre_iteridx=iterindex;
18.        end for
19.    end if
20. end for
```

The complexity of the existing reduction algorithm is $O(HMN)$, where M is the number of slices, N is the source array x's slice size, and H is the number of axes. Compared to that, our proposed algorithm' complexity is $O(NI)$, where I is the number of inner slice sizes.

3.2 Repeat Operations

The repeat operation repeats elements of a ndarray along each axis for specified times. For example, a vector of shape (3,4) can be expanded to shape (6,4) if repeated along the first axis, or (3,8) along the second axis. Fig. 5 shows an example of this. It consists of internal repetitions and external repetitions (or "tile"). The former repeats the elements of the

input ndarray, while the latter constructs a ndarray by repeating the entire input ndarray a specified number of times along each axis.

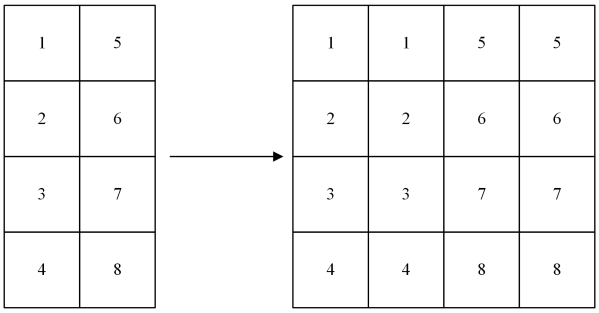


Fig. 5 Example of the repeat operation

Repeat is another operation that is often used in DNN, especially for implementing the UpSampling and BatchNormalisation neurons[33]. Reduction operations reduce the input ndarray, while repeat operations expand the input ndarray. Both operations require memory management rather than complex computation.

Each repeat operation along an axis requires the creation of additional memory space for the intermediate result. Thus, similar to the reduction function, to perform multi-axis repetition, using the existing operation only a few times will cause a memory bottleneck for the entire application. For this purpose, we implement the multi-axis repeat operation.

The optimizations we use in our algorithm follow two patterns. The first approach is to provide multiple implementations for different inputs. For example, if we only use one axis or only repeat the highest dimension, a specific implementation for that case would be much faster than a general solution.

The second approach is to reduce the creation of intermediate memory. A repeat algorithm is similar to a reverse of reduction: it requires expanding the source ndarray x into a larger destination ndarray y. Using elements that repeat as blocks, the repeat operation copies elements from x to y block by block. The indices in both ndarrays move by a step of block size, but with different periods.

In the modified implementation, the intermediate memory is created only once, and all iteration cycles along different axes are completed in the same memory. Specifically, we define h to be the highest non-one-repeat dimension, copy the HD dimension from source ndarray to target ndarray, and then copy the lower dimensions within target ndarray.

4 Fair Subgraph Fusion with Memory-Intensive Operations

Based on the operator optimization in the previous section, we propose a decentralized subgraph fusion framework that distributes proper fusion resources among different deployed subgraphs, while keeping the fairness of resource usage. We model this problem as a cooperative bargaining game (CBG) following that in [34]. In this section, we first present the design of this game and explain the utility function used in this game. The notation frequently used in this work is shown in Table 1.

Table 1 Notations

Notation	Meaning
N	Number of devices in system
K	Number of subgraphs
U_i	Utility function of device i
d_i	Disagreement value of devicei
r	Iterative update rounds
μ	Step size in sub-gradient method
$\boldsymbol{S}$	Subgraph distribution matrix
F	possible utility values

4.1 Game Design

The cooperative bargaining game is an important model in the cooperative game theory. Initially this game analysis two players. They cooperate in a certain task, get some reward such as money, and then each in turn demands a portion of it. If the proposals sum up to a value that is less than the total available reward, both players get what they demanded. Otherwise, both get only previously agreed disagreement value, which adds up to less than the total value, as return. Each player proposes according to certain utility.

In this work, we model the subgraph fusion problem in distributed model compilation as a cooperative game. In this framework, the participating edge nodes provide extra subgraph that they are willing to share with others for fusion operations in a subgraph. A node selects a certain amount of subgraph from each category according to its local preference. For example, a node is more willing to share the class of subgraph that is in a surplus locally. As a target, all participating nodes aim to enable as much fusion as possible, though the total available shareable subgraphs are limited. During the redistribution process, each participating node v_i tries to selfishly maximize its own utility U_i. By letting each participating node achieve its maximal utility, the system can get the most from the distribution process.

If the edge nodes cannot reach an agreement on how to distribute subgraphs, they fall back to the default disagreement value d_i, which is node i's utility before distribution. It means that all nodes need to achieve better utility or stay the same via participating in this game.

Formally, we define a bargaining game as a set (F, D). Here $F \in \mathbb{R}^N$ is a feasibility set that contains the possible utility values, and $D \in \mathbb{R}^N$ is the set of initially decided disagreement values. To solve the CBG, we use the Nash Bargaining Solution (NBS).

Definition 1 a set of rewards $(U_1^*, U_2^*, \cdots, U_N^*)$ is a

NBS of the CBG problem if it is the solution to the optimization problem:

$$\text{maximise} \prod_{i=1}^{N}(U_i - d_i)$$

NBS satisfies a series of axioms. Most importantly, it guarantees pareto optimality, which means that the outcome of any player cannot be further improved without damaging the utilities of the others. Besides, it is invariant to equivalent utility representations. If we apply a transformation that keeps the same ordering preference, e.g., linear transformation, on the utility function, the outcome of the bargaining solution should not change. These properties provide good flexibility to the framework.

The optimization problem above, can be simplified by applying logarithm operation on both sides of the equation. Therefore, the final minimizing target becomes:

$$-\sum_{i=1}^{N}\log(U_i - d_i)$$

4.2 Centralized Solution

We start from a common centralized setting, where there is an edge server that has access to the information of all the participating nodes, and is fully in charge of solving the optimization problem.

For each node, the utility is defined as an array min $\boldsymbol{S}=(s_1, s_2, \cdots, s_N)$. There can be multiple possible definitions of $U(\boldsymbol{s}_i)$ (abbreviated as U_i). Combining the utility definition and the problem definition, the optimization problem is:

$$\min \sum_{i=1}^{N} -\log(U_i - d_i)$$

$$\text{s. t. } s_{i,k} \leqslant 0,$$

$$\sum_{i=1}^{N} s_{i,k} \leqslant T_k$$

Here the constraints are $\forall i \in [1, N]$, $\forall k \in [1, K]$. The first means that each element in the optimized distribution vector $\boldsymbol{s}$ should be non-negative, since negative value means a participant has to share the part of subgraph it does not intend to share. The second constraint means that the subgraph amount of one given category in the system T_k should remain the same after distribution.

This optimization is an Integer Programming problem. The solution requirement can be relaxed by allowing elements distribution matrix S to be non-negative float numbers. We only need to round the optimization results to integers later. The solver system has NK variables. For a practical edge deployment system, $|N|$ can be a very large number. This requires a lot of computation, which is why a decentralized solution is required.

4.3 Decentralized Solution

In this part, we derive a decentralized solution for the optimization problem due to the computation complexity and the fit for distributed edge deployment environment. Here each node solves an optimization sub-problem, without caring about the solution of other nodes. And together, they can somehow reach the overall optimization target.

Following this approach, we can make each node v_i to solve part of the problem of minimizing $-\log(U_i - d_i)$, where $s_{i,k} \geqslant 0$. One problem lies in that sub-problems are coupled by the second constraint, since it involves all $\boldsymbol{s}_i$, otherwise all the sub-problems can be separately solved by each node.

To relax this constraint, we apply the Lagrangian dual decomposition. Specifically, we form the partial Lagrangian $\mathscr{L}$: $\mathbb{R}^{NK} \rightarrow \mathbb{R}$:

$$\begin{aligned}\mathscr{L}(S, \boldsymbol{\lambda}) &= \sum_{i=1}^{N} -\log(U(\boldsymbol{s}_i) - d_i) + \sum_{k=i}^{K}\lambda_k\left(\sum_{i=1}^{N} s_{i,k} - T_k\right) \\ &= \sum_{i=1}^{N}\left(-\log(U_i - d_i) + \sum_{k=1}^{K}\lambda_k s_{i,k}\right) - \sum_{k=1}^{K}\lambda_k T_k\end{aligned}$$

Here $\lambda_k > 0$, $\forall k \in [1, K]$, and $\boldsymbol{\lambda}$ is the Lagrangian multiplier. In solving this problem, the extra item $\sum_{k=1}^{K}\lambda_k T_k$ at the end, which does not affect the optimization of $\boldsymbol{s}$, can be removed. Therefore, this equation can be separated into N parts, each independent from the others. It contains two components. The former part is the same as that in the equation in the previous section, and the latter one contains a variant of the second constraint, multiplied by the Lagrangian multiplier. We can see the dual function $\mathscr{L}(S, \lambda)$ as one that contains variable $\boldsymbol{\lambda}$, with Sfixed. Since $\boldsymbol{\lambda} \succeq 0$, and $\sum_{i=1}^{N} s_{i,k} - T_i \leqslant 0$, $\forall k$, the solution to this problem constitutes a lower bound to the original optimization problem. The target of solving this problem thus is to finding the proper $\boldsymbol{\lambda}$ parameters that minimizes the difference between the dual function and the original function.

To achieve that purpose, the sub-gradient optimization method is used. It is an iterative method for solving convex optimization problems using sub-gradients. A sub-gradient of a convex function f at x_0 is any v in the domain of f so that:

$$f(x) - f(x_0) \geqslant v^T(x - x_0), \forall x \in dom(f)$$

Here a sub-gradient of the dual function is calculated as $\sum_{i=1}^{N}\boldsymbol{s}_{i,k} - T_k$. Therefore, in solving the problem each node v_i finds $\boldsymbol{s}_i^*$ that minimize:

$$g_i(\boldsymbol{s}_i, \boldsymbol{\lambda}) = \sum_{i=1}^{N}\left(-\log(U_i - d_i) + \sum_{k=1}^{K}\lambda_k s_{i,k}\right)$$

5 Deployment of Compiled Model

Another challenge in conductingmachine-learning-based data analytics on edge devices is the deployment of compiled machine learning models. It is a key topic in edge computing[35]. In this section, we introduce how our system supports the deployment of compiled models on different backends, based on the work in [36]. It should be noted that deployment is not limited to edge devices, but can also be on cloud servers where the accelerators enable high performance.

5.1 Deployment Backends

With the fast development of edge technologies, there exist various types of deployment backends. One key principle is to support multiple deployment methods. A common type is hardware accelerators such as GPU and TPU. Compilers such as TVM target at these accelerators, and develop feature so that the implementation can benefit from the power of parallel computation etc. provided by these devices, such as SIMD vectorization, multi-core, cache.

However, despite the superb model inference performance, one drawback of this approach is that the users need to take a very long time, hours, sometimes days, to train a single model on a specific platform. Considering the huge number of edge devices often deployed in real world applications, this approach is often not practical.

Instead, we recognize the benefit of using portable technologies. Recently, containers such as Docker as a lightweight virtualization technology have gained wide application. It is used in deployment systems such as Kubernetes. Deploying models as Docker containers will effectively reduce the cost of re-training.

Besides container, virtual machine is another choice as targets. The common virtual machine tools such as VMWare are not suitable to be deployed on edge devices with limited resources. In such case, Unikernel is a better choice as VM manager. It builds small virtual machines with a specialized minimal operating system that hosts only one model and application. These specialized applications can then be deployed on edge devices. Specifically, we explore MirageOS, an unikernel system. Deploying to Unikernel has proved to be of low memory footprint, and thus is quite suitable for resource-limited edge devices.

Another execution target is JavaScript. Using JavaScript to perform data analytics at the web end has begun to attract interest from academia and industry. TensorFlow.js has demonstrated the possibility of defining neural network models on web end and performs inference on the web end. Facebook has developed Reason, a popular language that compiles its code into JavaScript. Techniques such as WebAssembly ensure good performance of such web-end models. By exporting machine learning models to JavaScript code, we can do complex computations on web browser, without relying on other dependencies such as third-party libraries. Furthermore, the JavaScript code can also be exported to edge devices as a lightweight backend.

5.2 Naming of Models

One important aspect of deployment of compiled ML models is version control. A common practice of the model developers is to keep modifying, compiling, and deploy their models on edges until it reaches a suitable status. As such, it is crucial for each version of a model to be assigned a unique ID so as to be well managed.

The naming scheme of a model is: id/[vid|latest]/pin. To access a model, users can either choose a specific version id, or to use the newest version ("latest") in the local model repository.

It is easy to see that using this option would lead to inconsistency among edge devices: the newest version of models on one device might not be so on the other; the developer might have already uploaded a revised model.

To get the newest version of model from the server, the download time of the current version of model on a local device is saved. If the latest symbol is set in the model ID, the newest model on the model server will be downloaded to the local repository after a certain amount of time. In a production edge deployment scenario, it is advised that each model contain a specific model version id using hash; the latest symbol is more suitable to be used in ML model development. Besides, the pin part in the model ID indicates if the model if finalized and its compiling and deployment configuration should be saved. Once the naming convention is fixed, users can contribute newly created models to the model repositories so that it can be accessed by others.

6 Evaluation

In this section we evaluated the proposed method, focusing on two aspects: the performance of memory-intensive computations in compiled models, and the performance of deployed models on various deployment targets.

6.1 Setup

For the performance measurements of memory-intensive operations, we use various input sizes and then measure the execution time and memory usage. We focus on edge devices for the evaluation and use the single-board computer Raspberry Pi 4 (rpi4) as the evaluation hardware. It has a 64-bit quad-core Cortex-A72 CPU of 1.5 GHz. It represents a typical edge device deployed in our application scenario. On the software side, we use version 1.16 of NumPy, and version 1.0 of Julia as a comparison. Both are state-of-the-art numerical computation tools.

Next, we compare the performance of different deployment backends we use. Specifically, we observe two types of typical operations: map and reduction operations on ndarray. Our compiler can produce two kinds of executables: bytecode and native. Native executables are compiled specifically for certain architectures and often run faster, while the advantage of bytecode executables is portable. A Docker container can adapt to both choices. Besides, we also compare the performance of the same computation deployed as JavaScript and virtual machine Mirage.

6.2 Memory-Intensive Evaluations

To measure the performance, we compare the copy operation in our system, NumPy, Julia, and C program. Here "Owl" indicates our proposed method. For the C program, we use memcpy. The compiling flags in C and Owl are set to the same level 3 optimization. The input is a vector of single-precision float numbers that gradually increase in size. The initial comparison results are shown in Fig. 6. The experimental results show that C program runs the fastest, and Owl and NumPy are quite close to C program. The copy time grows linearly with input size. The Julia implementation of copy though shows large variation. Copy operation, unlike the calculation-heavy sin operation, etc., is not limited by computation, and thus not very likely to be improved too much above C language.

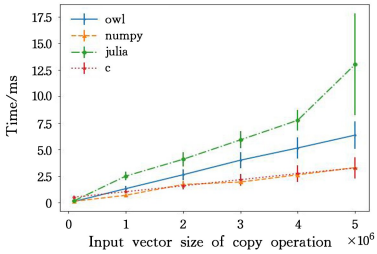
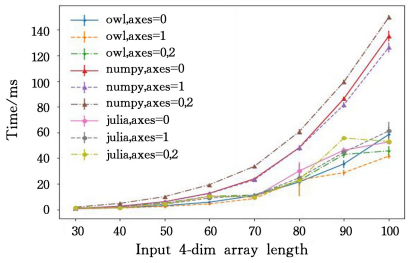
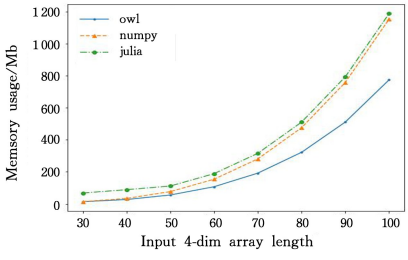


Fig. 6 Measure performance of copy operations on various platforms

Because there are multiple axes involved, we use a four-dimensional floating number ndarray as input to evaluate the reduction operation. All four dimensions have the same length. We measure the peak memory usage as the length increase, each for axis equals to 0, 1, and both 0 and 2 dimensions. The results of the evaluation compared with NumPy and Julia are shown in Fig. 7. Experimental results show that the memory usage of the proposed algorithm, measured on a single axis 0, is lower than that of NumPy and Julia.



(a) execution speed



(b) memory usage

Fig. 7 Measure performance of sum reduction operations on various platforms

The evaluation ofrepeat operation is similar to that of reduction operations. We use a four-dimensional ndarray made up of floating-point numbers as input. All four dimensions have the same length. We measure the speed with increasing length, the repetition times is set to 2 on all dimensions. The evaluation results compared with NumPy and Julia on different devices are shown in Fig. 8.

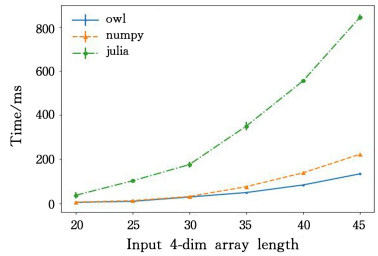


Fig. 8 Comparing the performance of repeat operation with NumPy and Julia, on desktop

The evaluations are performed on both desktop and edge device RaspberryPi. In both cases, the repetitive operation performance in the proposed system is superior to the other two libraries, NumPy and Julia. Julia's performance is much slower; the reason is that its implementation is not in native-C, and its multi-axes repeat operations are compounded by single-axis ones.

We also measure the peak memory usage shown in Fig. 9. As you can see, in NumPy, repeat operations are about half as effective in both execution speed and memory usage. In contrast to this implementation, the multi-axis repeats operation in NumPy is achieved by running multiple single-axis repeat, and thus is less efficient in both memory usage and execution speed.

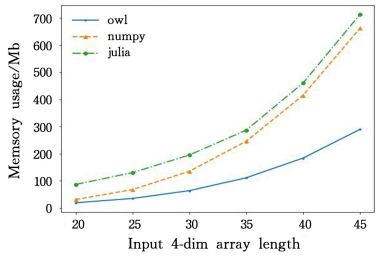


Fig. 9 Repeat operation memory usage comparison

The repeat operation in Julia is much slower than in the other two. One reason is that repeat operation is not a computation-intensive operation, so the optimization techniques such as static compilation and vectorization are of less importance than the other methods. Another reason is that this operation is implemented in pure Julia rather than the efficient C code.

6.3 Model Deployment

Next, we compare the performance of computation on different deployment backends; we use the widely used map and reduction (or "fold") operations on ndarray as the benchmark. Backends include native, bytecode, JavaScript, and vir-

tual machine. We use simple functions such as multiplication on 1-dimensional (size $<1\,000$) and 2-dimensional arrays. Fig. 10 shows the relationship between the total size of ndarray and the execution time; when projected to log scale, the relationship keeps linear.

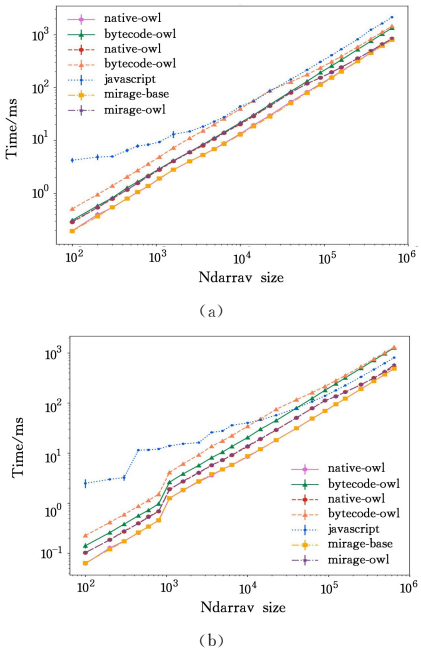


Fig. 10 Performance of (a) map and (b) fold operations on various deployment backends.

For both operations, our proposed deployment method is faster than the base version, and native executables outperform bytecode ones. The performance of Mirage executables is close to that of native code. Generally, JavaScript runs the slowest, but note how the performance gap between JavaScript and the others converges when the ndarray size grows. For the fold operation, JavaScript even runs faster than bytecode when the input size is sufficiently large. The same evaluation experiments are also conducted on edge devices. Despite that the performance is much slower than that on the laptop machine, the results are similar.

Conclusion In this work, we recognize two issues that are not well addressed yet in the previous work about the deep model compilation: optimization of memory-intensive operations and the deployment of models on various targets. We have proposed a framework solution to address them, including operation optimization, subgraph fusion distribution, and system deployment modules. The evaluation results show the good performance of our proposed solution.

References

[1] ZHAO J, CHANG X, FENG Y, et al, Participant Selection for Federated Learning with Heterogeneous Data in Intelligent Transport System[J]. IEEE Transactions on Intelligent Transportation Systems, 2023, 24(1): 1106-1115.

[2] CASTIGLIONI I, RUNDO L, CODARI M, et al. AI applications to medical images: From machine learning to deep learning[J]. Physica Medica, 2021, 83: 9-24.

[3] MAHMUD M, KAISER M S, MCGINNITY T M, et al. Deep learning in mining biological data[J]. Cognitive Computation, 2021, 13(1): 1-33.

[4] ZHAO J, HAN R, YANG Y, et al. Federated Learning with Heterogeneity-Aware Probabilistic Synchronous Parallel on Edge[J]. IEEE Transactions on Services Computing, 2021, 15(2): 614-626.

[5] BARRACHINA S, CASTILLO M, IGUAL F D, et al. Evaluation and tuning of the level 3 CUBLAS for graphics processors [C]// 2008 IEEE International Symposium on Parallel and Distributed Processing. IEEE, 2008: 1-8.

[6] LI M, LIU Y, LIU X, et al. The deep learning compiler: A comprehensive survey[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 32(3): 708-727.

[7] ZHENG Z, YANG X, ZHAO P, et al. A Stitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures[C]// Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 359-373.

[8] KLJUCARIC L, JOHNSON A, GEORGEA D. Architectural analysis of deep learning on edge accelerators[C]// 2020 IEEE High Performance Extreme Computing Conference(HPEC). IEEE, 2020: 1-7.

[9] BARHAM P, DRAGOVIC B, FRASER K, et al. Xen and the art of virtualization[J]. ACM SIGOPS Operating Systems Review, 2003, 37(5): 164-177.

[10] MAO Y, YOU C, ZHANG J, et al. A survey on mobile edge computing: The communication perspective[J]. IEEE Communications Surveys & Tutorials, 2017, 19(4): 2322-2358.

[11] SHI W, CAO J, ZHANG Q, et al. Edge computing: Vision and challenges[J]. IEEE Internet of Things Journal, 2016, 3(5): 637-646.

[12] CHEN M X, ZHANG J B, LI T R. Survey on Attacks and Defenses in Federated Learning [J]. Computer Science, 2022, 49(7): 310-323.

[13] LI Q, WEN Z, WU Z, et al. A survey on federated learning systems: vision, hype and reality for data privacy and protection[J/OL]. IEEE Transactions on Knowledge and Data Engineering, 2021. https://ieeexplore.ieee.org/document/9599369/.

[14] LI T, SAHU A K, TALWALKAR A, et al. Federated learning: Challenges, methods, and future directions[J]. IEEE Signal Processing Magazine, 2020, 37(3): 50-60.

[15] MCMAHAN B, MOORE E, RAMAGE D, et al. Communication-efficient learning of deep networks from decentralized data[C]// Artificial Intelligence and Statistics. PMLR, 2017: 1273-1282.

[16] SZEGEDY C, LIU W, JIA Y, et al. Going deeper with convolutions[C]// Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015: 1-9.

[17] CYPHERS S, BANSAL A K, BHIWANDIWALLA A, et al. Intel ngraph: An intermediate representation, compiler, and execu-

tor for deep learning[J]. arXiv:1801.08058,2018.
[18] VASILACHE N,ZINENKO O,THEODORIDIST,et al. Tensor comprehensions:Framework-agnostic high-performance machine learning abstractions[J]. arXiv:1802.04730,2018.
[19] LEARY C,WANG T. XLA - TensorFlow,compiled [EB/OL]. TensorFlow Dev Summit. https://www.tensorflow.org/xia.
[20] ROTEM N,FIX J,ABDULRASOOL S,et al. Glow:Graph lowering compiler techniques for neural networks[J]. arXiv:1805.00907,2018.
[21] LATTNER C,AMINI M,BONDHUGULA U,et al. MLIR:A compiler infrastructure for the end of Moore's law[J]. arXiv:2002.11054,2020.
[22] ROESCH J,LYUBOMIRSKY S,KIRISAME M,et al. Relay:A high-level compiler for deep learning[J]. arXiv:1904.08368,2019.
[23] CHEN T,ZHENG L,YAN E,et al. Learning to optimize tensor programs[C]// Proceedings of the 32nd International Conference on Neural Information Processing Systems(NIPS'18). 2018:3393-3404.
[24] ZHENG S,LIANG Y,WANG S,et al. Flextensor:An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system[C]// Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020:859-873.
[25] ZHENG L,JIA C,SUN M,et al. Ansor:Generating High-Performance Tensor Programs for Deep Learning[C]// 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 2020:863-879.
[26] JIA Z,PADON O,THOMAS J,et al. TASO:optimizing deep learning computation with automatic generation of graph substitutions[C]// Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019:47-62.
[27] WANG B Y,PANG J M,XU J L,et al. Matrix Multiplication Vector Code Generation Based on Polyhedron Model[J]. Computer Science,2022,49(10):44-51.
[28] WANG H,ZHAI J,GAOM,et al. {PET}:Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections[C]// 15th USENIX Symposium on Operating Systems Design and Implementation(OSDI 21). 2021:37-54.
[29] CRANKSHAW D,WANG X,ZHOUG,et al. Clipper:A Low-Latency Online Prediction Serving System[C]// 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). 2017:613-627.
[30] Olston 2017:TensorFlow Serving[EB/OL]. TensorFlow Serving. https://www.tensorflow.org/tfx/guide/serving.
[31] KLAISE J,VAN LOOVEREN A,COX C,et al. Monitoring and explainability of models in production[J]. arXiv:2007.06299,2020.
[32] YANG Y,ZHAO L,LI Y,et al. INFless:a native serverless system for low-latency, high-throughput inference[C]// Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022:768-781.
[33] IOFFE S,SZEGEDY C. Batch normalization:Accelerating deep network training by reducing internal covariate shift[C]// International Conference on Machine Learning. PMLR,2015:448-456.
[34] ZHAO J,FENG Y,CHANG X,et al. Energy-Efficient and Fair IoT Data Distribution in Decentralised Federated Learning [J/OL]. IEEE Transactions on Network Science and Engineering,2022,23. https://ieeexplore.ieee.org/document/9804872.
[35] LIU Z H,ZHENG H Q,ZHANG J S,et al. Computation Offloading and Deployment Optimization in Multi-UAV-Enabled Mobile Edge Computing Systems[J]. Computer Science,2022,49(6A):619-627.
[36] ZHAO J,TIPLEA T,MORTIER R,et al. Data analytics service composition and deployment on edge devices[C]// Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks. 2018:27-32.

Peng XU, born in 1978, Ph.D. His mian research interests include edge computing and deep learning.

Jianxin ZHAO, born in 1990, Ph.D, professor. His mian research interests include numerical computation, high performance computing, machine learning, and their application in the real world.

(Responsible editor:Yahui LI)