BladeDISC++: Memory Optimizations Based On Symbolic Shape

Xiulong Yuan* Alibaba Group
Xu Yan* Alibaba Group
Wenting Sheng Alibaba Group
Xiafei Qiu Alibaba Group
Ang Wang Alibaba Group

Jie Zhang Alibaba Group
Yong Li Alibaba Group
Wei Lin Alibaba Group

Abstract

Recent deep learning workloads exhibit dynamic characteristics, leading to the rising adoption of dynamic shape compilers. These compilers can generate efficient kernels for dynamic shape graphs characterized by a fixed graph topology and uncertain tensor shapes. However, memory optimization, although particularly crucial in this large model era, remains relatively underexplored for dynamic shape graphs. The fundamental challenge lies in the lack of precise tensor shapes which are essential in conventional methods such as operation scheduling(op scheduling) and rematerialization. To address this challenge, we propose op scheduling and rematerialization approaches based on symbolic shapes and developed BladeDISC++. Besides, since rematerialization decisions cannot be made solely at compile time when tensor shapes are unknown, BladeDISC++ employs a compilation-runtime combined strategy to optimally address shape dynamics. Evaluations indicate that BladeDISC++ effectively reduces memory usage for dynamic shape graphs, achieving memory consumption comparable to optimizations using precise shapes, thereby promoting the broader adoption of dynamic shape compilers.

1 Introduction

Dynamic shape compilers are becoming increasingly prevalent due to their ability to optimize deep learning workloads with dynamic characteristics. While systems like TorchInductor[14] and Modular[13] have made significant strides in kernel generation, memory optimization still remains underexplored. Conventional methods like op scheduling[9, 15, 2] and rematerialization[6, 3, 7, 10, 5] (recomputation and offloading included) rely on exact tensor shape to assess the memory impact of ops or rematerialization subgraphs, and make optimization decisions at compile time. However, in the absence of shape values, these methods become unfeasible.

BladeDISC++, built upon a dynamic shape compiler BladeDISC[16][17][11], leverages symbolic shapes to tackle the above challenges. With symbolic shapes, BladeDISC++ is able to derive comparative memory impacts of different op sequences and find the optimum scheduling order. For rematerialization, symbolic shapes are utilized to search for optimum recomputation subgraph at compile time and assist to conduct final rematerialization decisions at runtime.

Our evaluations demonstrate that BladeDISC++ can effectively reduce memory usage for training with dynamic shape graphs compared to BladeDISC. Additionally, BladeDISC++ achieves comparable memory consumption with static shape training while alleviating the overhead of recompilation and tensor padding.

*Equal Contribution.

Machine Learning for Systems Workshop at (NeurIPS 2024).

2 Memory optimizations based on symbolic shapes

As shown in Figure 1, given a dynamic shape computation graph, BladeDISC++ first performs symbolic shape analysis to create a global symbolic shape graph that describes the algebraic relationships between shape symbols(in section 2.1). Then, the symbolic shape graph, together with the computation graph, undergoes optimization passes including op fusion, op scheduling, and rematerialization for memory optimization.

As BladeDISC's prior work [16][17] has tackled the op fusion problem, this paper focuses on op scheduling (in section 2.2) and rematerialization (in section 2.3). In particular, with the symbolic shape graph instead of exact tensor shape, BladeDISC++ can still compare memory impacts of different op sequences, and determine whether a recomputation subgraph would benefit memory consumption. Additionally, because a dynamic shape graph might have varying memory footprints across different runs, it is impractical to make rematerialization decisions, such as how much memory to evict, solely at compile time. Therefore, BladeDISC++ explores all rematerialization candidates and searches their corresponding regeneration subgraphs and conduct final rematerialization decisions at runtime.

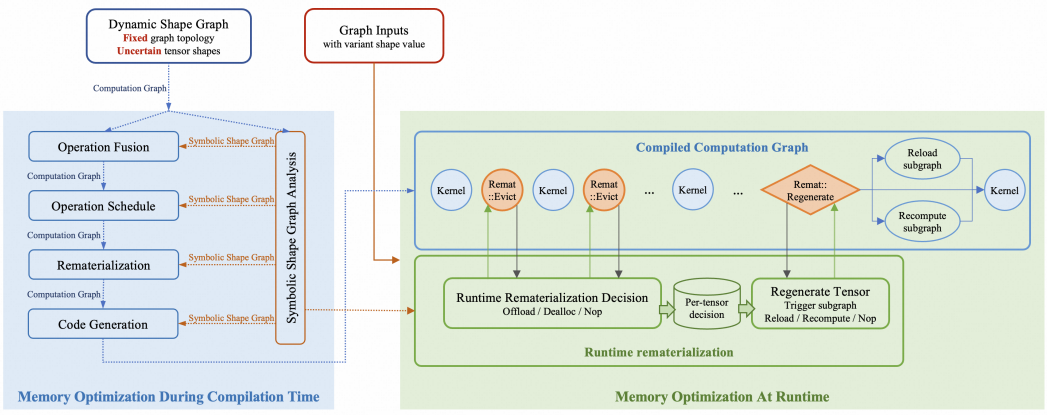

Figure 1: Memory optimizations based on symbolic shapes in BladeDISC++

2.1 Symbolic shape graph analysis

BladeDISC++ systematically analyzes and extracts shape information from the semantics of each op within the dynamic shape computation graph. It then constructs a global symbolic shape graph to represent the algebraic relationships between shape dimensions through shape value extraction and input-output shape inference.

```
func.func @main(%arg0: tensor<?,[@S0]>, %arg1: tensor<12x11008>) {
  %1 = broadcast(%arg1) -> tensor<4096x?, [@C4096, @S0]>
  %2 = dynamic_reshape(%arg0, %new_shape) -> tensor<?x12,[@S1, @C12]>
  // The last consumer of %2
  %3 = dot(%2, %arg1) -> tensor<?x11008, [@S1, @C11008]>
  // The last consumer of %3
  %4 = reduce(%3) -> tensor<?, [@S1]>
  %1084 = broadcast(%4) -> tensor<11008x?, [@C11008, @S1]>
  %1085 = broadcast(%arg0) -> tensor<1024x?, [@C1024, @S0]>
}
func.func @symbolic_shape_graph() {
  SymbolicDim @S0
  SymbolicDim @S1
  @S0 = Mul @C12, @S1
}
```

Listing 1: Example of a dynamic shape graph and its symbolic shape graph

As illustrated in Listing 1, BladeDISC++ introduces a *SymbolicDim* op to define a symbolic value, bond to a dimension of a tensor shape in the dynamic shape graph as op attributes, exemplified by *tensor<?x?, [@S0, @S1]>*. For instance, the equation *@S0 = 12 * @S1* stems from *DynamicReshapeOp* that its input and output tensor have the same number of elements.

Comparison between memory sizes of tensors is critical to op scheduling and rematerialization. BladeDISC++ introduces *SymbolicExpr* to express algebraic representations of symbolic dimensions, allowing for comparative evaluations with a best-effort strategy. For example, the element number of tensor *%1084* and *%1085* can be represented by *SymbolicExpr*s *expr1 = 11008 * @S1* and *expr2 = 1024 * @S0* respectively. As it's already derived from *DynamicReshapeOp* that *@S0 = 12 * @S1*, *exp1* can be simplified to *132096 * @S0*, thus BladeDISC++ can infer that *expr1* is less than *expr2*.

2.2 Operation scheduling

Op scheduling tries to find a memory-efficient op sequence from the original computation graph. Existing scheduling algorithms[8] often traverse the computation graph and select an op from a *ReadySet* (including ops whose predecessors have already been scheduled) at each step. The selection is mainly based on comparing different ops' memory impact, which is determined by the difference between bytes freed and allocated after scheduling a specific op. BladeDISC++ adopts a similar methodology, emphasizing the computation and comparison of memory impact among different ops with the absence of exact tensor shapes in dynamic shape graphs. Specifically, in BladeDISC++, the memory impact for each op is calculated using symbolic shapes and thus expressed as a *SymbolicExpr*. These *SymbolicExpr*s are then compared to each other with the help of symbolic shape graph.

Algorithm 1 Op Scheduler's Main Loop in BladeDISC++

```
1: initialize ReadySet with all operations that generate graph outputs
2: while ReadySet is not empty do
3:     best_op = ReadySet[0]
4:     for op in the ReadySet do
5:         best_op = CompareAndChoose(op, best_op) % Compare memory effect for each op
6:     end for
7:     ScheduleOp(best_op) % Revise the schedule state and include any input producer of best_op
   that has no remaining predecessors to ReadySet.
```

Figure 2: OpScheduler algorithm main loop

In Listing 1, for example, the *DynamicReshapeOp* and *DotOp* appear in the *ReadySet* at a specific step. *DotOp*, as the last consumer of %2 and producer of %3, has a memory impact of *10996 * @S1* due to the allocation for %3 and deallocation for %2. The *DynamicReshapeOp*'s memory impact, on the other hand, is *4096 * @S0* because scheduling it only involves allocation for %1. To compare two *SymbolicExpr*s containing different sets of symbols, we first simplify the *SymbolicExpr* of *DynamicReshapeOp*'s memory impact based on *@S1* to *49152 * @S1* with the same procedure described in 2.1, then it can be determined that the *DynamicReshapeOp* has a higher memory impact than the *DotOp*.

When it's unfeasible to compare two memory impact *SymbolicExpr*s, we resort to a commonly used strategy: selecting the op that results in smaller overall tensor lifetimes based on the graph topology.

2.3 Rematerialization

Conventional rematerialization methods[6, 10, 3]involve algorithms to determine which tensors to be evicted earlier to alleviate memory pressure, as well as how to perform subsequent regeneration, either through reloading or recomputation. These methods also include a search process to identify optimal recomputation subgraphs by evaluating their memory impacts. Notably, tensor rematerialization may negatively affect end-to-end performance, so it should only be employed when the graph's execution risks exceeding the memory limit. However, a dynamic shape graph, with undetermined tensor shapes, can exhibit varying peak memory usage across different runs. Some runs may not need rematerialization since they remain within memory limits, while others may need. It's impractical to make all decisions solely during compilation. Furthermore, the lack of exact shapes raises challenges in assessing the memory impacts of potential recomputation subgraphs.

To address these issues, BladeDISC++ utilizes a combined compilation-runtime strategy based on symbolic shapes to best manage shape dynamics across graph runs. During compilation, it explores all potential rematerialization candidates and identifies their corresponding regeneration subgraphs, which are then inserted into the original computation graph as different execution branches. Final decisions regarding which tensor to evict and the associated regeneration method are made at runtime.

During compile time, as illustrated in Figure 1, BladeDISC++ inserts a *Remat::EvictOp* after each op, checking if any active tensors at that point need to be evicted to alleviate memory pressure. For each candidate tensor, regeneration subgraphs, including those for reload and recomputation, are also generated. While reloading only involves a host-to-device (H2D) instruction and is memory-neutral, searching for recomputation subgraphs requires careful evaluation since sub-optimal choices may

even increase peak memory usage. BladeDISC++ uses a standard search process but assesses memory impact of potential subgraphs based on *SymbolicExpr*.

Taking recomputation subgraph searching for *%4* in Listing 1 as an example. Starting from the *ReduceOp*, BladeDISC++ determines the memory impacts: *-11007 * @S1* for just the *ReduceOp*, *-11 * @S1* with the addition of the *DotOp*, and *1 * @S1* when the *DynamicReshapeOp* is included. Although exact shape values are unknown, BladeDISC++ can still ascertain that the last recomputation subgraph is memory-efficient, whereas the others are not.

Then, BladeDISC++ inserts *Remat::RegenerateOp*s, along with the corresponding regeneration subgraphs (both reload and recompute), before each candidate tensor's subsequent consumers. The *Remat::RegenerateOp* checks whether a candidate tensor is evicted and its regeneration method,

At runtime, BladeDISC++ monitors memory usage throughout kernel execution. Each time an *EvictOp* is triggered, BladeDISC++ checks the current memory usage and performs an on-the-fly analysis of all candidate tensors provided by the *EvictOp* when the memory limit is about to be surpassed. The final decisions on which tensor from the above candidates needs to be evicted as well as the corresponding regeneration method are made by considering factors such as memory savings and end-to-end performance impact, following a similar approach as outlined in [10]. Subsequent *Remat::RegenerateOp*s then query these decisions and determine which regeneration subgraphs need to be triggered.

3 Evaluation

For our evaluation, we conducted experiments on the supervised fine-tuning of Llama-2-1b, a tailored model from the official Llama-2-7b[12] with the only change that decreasing *num_hidden_layers* from 32 to 4, on an Alibaba Cloud ecs.gn7-c12g1.3xlarge instance[4](with 40GB GPU RAM) using the CodeAlpaca-20K dataset [1]. CodeAlpaca-20K contains samples with text lengths ranging from approximately 100 to 3000 characters. In each training iteration, a fixed number of randomly selected samples are assembled into a batch, resulting in variable batch shapes across different iterations.

To assess the effectiveness of BladeDISC++, we compared memory usage and end-to-end performance in dynamic shape training using BladeDISC++ against both dynamic and static shape training using BladeDISC. For static shape training, following common practice, input sequences are padded to nearest power of 2 in length to balance redundant computation and compilation overhead. Besides, in our experiments, we deliberately set the largest bucket size equal to the longest sequence length in the dataset to investigate whether we can achieve comparable memory optimization results using symbolic shapes instead of exact shapes.

The experimental results indicate that BladeDISC++ can effectively reduce peak memory consumption for dynamic shape training. Furthermore, BladeDISC++ demonstrates comparable memory consumption to static shape training while also improving end-to-end performance by alleviating the overhead of recompilation and input bucketing.

Table 1: Training throughput of Llama-2-1b on CodeAlpaca-20K(tokens/second)

Batchsize	14	16	18
BladeDISC(dynamic shape training)	5662.34(38.20 GiB)	OOM	OOM
BladeDISC(static shape training)	5242.02(35.75 GiB)	5429.38(37.71 GiB)	5103.31(38.92 GiB)
BladeDISC++	5749.20(35.76 GiB)	6078.71(37.89 GiB)	5738.79(39.18 GiB)

4 Conclusion

This paper shares our industry experience in optimizing memory for dynamic shape graphs . We proposed op scheduling and rematerialization based on symbolic shapes and developed BladeDISC++. Evaluations show that BladeDISC++ can effectively reduce memory usage for dynamic shape training and can achieve comparable memory optimization results to static shape training. As far as we know, this is a pioneering effort in this area, and we aspire that it will support the compiler community in managing dynamic shape workloads and promote wider use of dynamic shape compilers.

References

[1] Codealpaca-20k dataset, 2024. Accessed: October 30, 2024.

[2] Renze Chen, Zijian Ding, Size Zheng, Chengrui Zhang, Jingwen Leng, Xuanzhe Liu, and Yun Liang. Magis: Memory optimization via coordinated graph transformation and scheduling for dnn. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 607–621, New York, NY, USA, 2024. Association for Computing Machinery.

[3] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[4] Alibaba Cloud. Alibaba cloud ecs.gn7-c12g1.3xlarge instance, 2024. Accessed: October 30, 2024.

[5] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization, 2020.

[7] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. *CoRR*, abs/2006.09616, 2020.

[8] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[9] Benoit Steiner, Mostafa Elhoushi, Jacob Kahn, and James Hegarty. Olla: Optimizing the lifetime and location of arrays to reduce the memory usage of neural networks, 2022.

[10] Yu Tang, Chenyu Wang, Yufan Zhang, Yuliang Liu, Xingcheng Zhang, Linbo Qiao, Zhiquan Lai, and Dongsheng Li. Delta: Dynamically optimizing gpu memory beyond tensor recomputation, 2022.

[11] BladeDISC Team. Bladedisc github repository, 2021. Accessed: October 30, 2024.

[12] Meta Llama Team. meta-llama/llama-2-7b, 2024. Accessed: October 30, 2024.

[13] Modular Team. Dynamic shape in modular, 2024. Accessed: October 30, 2024.

[14] Pytorch Team. Dynamic shape in pytorch, 2023. Accessed: October 30, 2024.

[15] Zihan Wang, Chengcheng Wan, Yuting Chen, Ziyi Lin, He Jiang, and Lei Qiao. Hierarchical memory-constrained operator scheduling of neural architecture search networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 493–498, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Zhen Zheng, Zaifeng Pan, Dalin Wang, Kai Zhu, Wenyi Zhao, Tianyou Guo, Xiafei Qiu, Minmin Sun, Junjie Bai, Feng Zhang, et al. Bladedisc: Optimizing dynamic shape machine learning workloads via compiler approach. *Proceedings of the ACM on Management of Data*, 1(3):1–29, 2023.

[17] Kai Zhu, WY Zhao, Zhen Zheng, TY Guo, PZ Zhao, JJ Bai, Jun Yang, XY Liu, LS Diao, and Wei Lin. Disc: A dynamic shape compiler for machine learning workloads. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 89–95, 2021.