# BladeDISC++: Enhancing Memory Usage Through Symbolic Shape Analysis


## Abstract

The increasing prevalence of dynamic characteristics in modern deep learning tasks has led to the growing importance of dynamic shape compilers. These compilers are designed to create effective kernels for dynamic shape graphs, which have a stable structure but uncertain tensor shapes. However, memory optimization, which is vital in the era of large models, has not been thoroughly investigated for dynamic shape graphs. The core issue lies in the absence of specific tensor shapes, which are generally required by existing methods like operation scheduling and rematerialization. To overcome this issue, we present operation scheduling and rematerialization strategies that utilize symbolic shapes, implemented in BladeDISC++. Furthermore, given that rematerialization decisions cannot be determined at compile time alone due to unknown tensor shapes, BladeDISC++ uses a hybrid approach combining compilation and runtime to address shape changes effectively. Our findings demonstrate that BladeDISC++ significantly reduces memory consumption for dynamic shape graphs, achieving levels similar to those of optimizations with precise shapes. This advancement facilitates the broader use of dynamic shape compilers.


## 1 Introduction

Dynamic shape compilers are becoming more and more necessary due to their ability to optimize deep learning tasks that have dynamic attributes. While advancements in kernel generation have been made by systems like TorchInductor and Modular, memory optimization remains a less-explored area. Traditional methods like operation scheduling and rematerialization, which encompass recomputation and offloading, depend on precise tensor shapes to evaluate the memory impact of operations or subgraphs, and consequently make optimization choices during compilation. However, these methods become impractical when shape values are not available.

BladeDISC++, which is based on the dynamic shape compiler BladeDISC, uses symbolic shapes to address these challenges. With symbolic shapes, BladeDISC++ is capable of comparing the memory effects of different operation sequences, and identifying the ideal scheduling order. For rematerialization, symbolic shapes are used to identify the optimal recomputation subgraph at compile time, and assist in making final rematerialization decisions during runtime.

Our experiments reveal that BladeDISC++ can efficiently reduce memory usage during training with dynamic shape graphs when compared to BladeDISC. Furthermore, BladeDISC++ achieves memory consumption similar to static shape training while eliminating the overhead associated with recompilation and tensor padding.

## 2 Memory optimizations based on symbolic shapes

As shown in Figure 1, BladeDISC++ starts with a dynamic shape computation graph, and proceeds by conducting a symbolic shape analysis to construct a global symbolic shape graph. This graph details the mathematical connections between the shape symbols, which will be discussed in section 2.1. Following this, the symbolic shape graph, along with the computation graph, is optimized through

.

steps that include operation fusion, operation scheduling, and rematerialization. These steps are aimed at memory usage reduction.

As previous work on BladeDISC has addressed operation fusion, this paper focuses on operation scheduling, which will be discussed in section 2.2, and rematerialization, which will be discussed in section 2.3. Using the symbolic shape graph instead of exact tensor shapes, BladeDISC++ can still compare the memory usage of different operation sequences and determine the benefit of recomputation subgraphs. Moreover, because the memory needs of a dynamic shape graph can fluctuate between different runs, it is not practical to base rematerialization decisions, such as how much memory to free, solely on compile time. Consequently, BladeDISC++ investigates all possible rematerialization options, searches for the corresponding regeneration subgraphs, and makes final rematerialization decisions during runtime.

[width=0.8]placeholder.png figureMemory optimizations based on symbolic shapes in BladeDISC++

### 2.1 Symbolic shape graph analysis

BladeDISC++ systematically analyzes and obtains shape information from the semantics of each operation within the dynamic shape computation graph. Following this, it establishes a global symbolic shape graph. This graph is designed to show the mathematical relationships between shape dimensions through shape value extraction and input-output shape inference.

```
func . func @main (% arg0 : tensor <? ,[ @S0 ] > , % arg1 : tensor <12 x11008 >) {
  %1 = broadcast (% arg1 ) -> tensor <4096 x ? , [ @C4096 , @S0 ] >
  %2 = d yna mi c_r eshape (% 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 <11008 x ? , [ @C11008 , @S1 ] >
  %1085 = broadcast (% arg0 ) -> tensor <1024 x ? , [ @C1024 , @S0 ] >
}

func . func @ s y m b o l i c _ s h a p e _ g r a p h () {
  SymbolicDim @S0
  SymbolicDim @S1
  @S0 = Mul @C12 , @S1
}
```

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

As shown in Listing 1, BladeDISC++ uses a SymbolicDim operation to represent a symbolic value. This value is linked to a dimension of a tensor shape in the dynamic shape graph as an attribute, for example, tensor<?x?, [@S0, @S1]>. The equation @S0 = 12 * @S1, for instance, is derived from a DynamicReshapeOp. It means the input and output tensors have an equivalent number of elements.

The comparison of tensor memory sizes is vital for both operation scheduling and rematerialization. BladeDISC++ uses SymbolicExpr to show mathematical expressions of symbolic dimensions. This allows for comparisons using a best-effort approach. For example, the element count of tensors

### 2.2 Operation scheduling

Operation scheduling aims to discover a memory-efficient sequence of operations from the initial computation graph. Existing scheduling algorithms typically traverse the graph and select an operation from a ReadySet, which includes operations whose predecessors have been scheduled, at each step. The selection is mainly based on a comparison of the memory impact of the different operations, which is determined by calculating the difference between the memory freed and the memory allocated after scheduling a particular operation. BladeDISC++ employs a similar strategy, emphasizing the calculation and comparison of memory impact among different operations when exact tensor shapes are unavailable in dynamic shape graphs. In BladeDISC++, the memory impact of each operation

is calculated using symbolic shapes, resulting in a SymbolicExpr. These SymbolicExprs are then compared using the symbolic shape graph.

In Listing 1, the DynamicReshapeOp and DotOp are present in the ReadySet at a particular step. DotOp, being the last consumer of

When comparing memory impact SymbolicExprs is not possible, we use a standard approach: selecting the operation that results in shorter overall tensor lifespans based on the graph's structure.

### 2.3 Rematerialization

Traditional rematerialization methods use algorithms to decide which tensors to release early to reduce memory pressure, and how to conduct the following regeneration via reloading or recomputation. These methods also search for optimal recomputation subgraphs, evaluating their memory effects. Tensor rematerialization can negatively impact end-to-end performance, so it should only be used when the graph's execution could exceed memory limits. However, dynamic shape graphs, with uncertain tensor shapes, may show varied peak memory use between different runs. Some runs may not need rematerialization as they remain within memory limits, whereas others may. Therefore, it is impractical to make decisions solely at compilation time. Also, the absence of exact shapes presents challenges in evaluating the memory effects of potential recomputation subgraphs.

To address these challenges, BladeDISC++ uses a combined compilation-runtime approach based on symbolic shapes to better manage shape variations during graph runs. At compile time, it explores all possible rematerialization candidates and identifies the regeneration subgraphs associated with them. These subgraphs are incorporated into the original computation graph as separate execution paths. Final choices regarding which tensor to release and the related regeneration method are made during runtime.

During compilation, as shown in Figure 1, BladeDISC++ adds a Remat::EvictOp after each operation. This checks if active tensors at that point need to be released to lower memory pressure. Regeneration subgraphs, including reload and recomputation, are created for each potential tensor. While reloading only involves a host-to-device instruction and has no impact on memory, finding recomputation subgraphs needs thorough evaluation as poor choices can increase peak memory consumption. BladeDISC++ uses a standard search approach, but assesses the memory impact of subgraphs using SymbolicExpr.

Taking the recomputation subgraph searching for

Following this, BladeDISC++ inserts Remat::RegenerateOps, with corresponding regeneration subgraphs for both reload and recompute. These are inserted before each potential tensor's subsequent consumers. The Remat::RegenerateOp checks if a tensor has been released, and which regeneration method is being used.

During runtime, BladeDISC++ monitors memory usage throughout kernel execution. Whenever an EvictOp is triggered, BladeDISC++ checks the present memory usage. When the memory limit is about to be exceeded, it performs a real-time analysis of all potential tensors offered by the EvictOp. Final decisions about which tensor needs to be released, and the regeneration method, are determined by taking memory savings and end-to-end performance into account, following a similar approach as detailed in. Subsequent Remat::RegenerateOps then check these choices to decide which regeneration subgraphs to trigger.

## 3 Evaluation

For our evaluation, we performed experiments on the supervised fine-tuning of Llama-2-1b, which is a customized model from the official Llama-2-7b with only the number of hidden layers decreased from 32 to 4. This was done on an Alibaba Cloud instance, with 40GB of GPU RAM. We used the CodeAlpaca-20K dataset, which contains text samples with lengths from about 100 to 3000 characters. During each training cycle, a fixed amount of randomly selected samples are put into a batch. This leads to variations in batch shapes between cycles.

To evaluate the effectiveness of BladeDISC++, we compared memory usage and end-to-end performance of dynamic shape training with BladeDISC++ against both dynamic and static shape

training with BladeDISC. For static shape training, following common methods, input sequences are padded to the closest power of 2 in length. This balances redundant computation and compilation overhead. Additionally, we set the largest bucket size to be equal to the longest sequence length in the dataset. This was done to investigate whether comparable memory optimization can be achieved using symbolic shapes instead of exact shapes.

The experimental results show that BladeDISC++ is able to reduce peak memory consumption during dynamic shape training. BladeDISC++ also demonstrated memory consumption similar to static shape training, while improving end-to-end performance by eliminating the overheads 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 study presents our practical experience in optimizing memory for dynamic shape graphs. We have introduced operation scheduling and rematerialization strategies that use symbolic shapes, implemented in BladeDISC++. Evaluations demonstrate that BladeDISC++ effectively decreases memory usage for dynamic shape training and can match the memory optimization results of static shape training. To the best of our knowledge, this work is the first attempt in this area. We hope it will support the compiler community in handling dynamic shape tasks, and increase the use of dynamic shape compilers.