A APPENDIX

A.1 FRAMEWORK INTEGRATE: TVM TENSOR EXPRESSION

To support TVM, we first need to specify that `tvm.nd.NDArray` is the data structure for Tensors in `metadist.platform.tvm`. Then, we must define and register basic operations such as `add`, `concatenate`, `chunk`, `allclose`, and others that are required to perform relevant calculations in the ShardCombine algorithm.

Here's an example of how we can use MetaDist to run the ShardCombine Algorithm on TVM:

```
import tvm, metadist

n = tvm.te.var("n")
A = tvm.te.placeholder((n, ), name="A")
B = tvm.te.placeholder((n, ), name="B")
C = tvm.te.compute(A.shape, lambda i: A[i] + B[i], name="C")

s = tvm.te.create_schedule(C.op)
tgt = tvm.target.Target(target="llvm", host="llvm")
fadd = tvm.build(s, [A, B, C], tgt, name="myadd")

def fadd_wrapped(a, b):
    c = metadist.platform.tvm.zeros_like(a)
    fadd(a, b, c)
    return c

meta_op_ = metadist.unifyshard.MetaOp(fadd_wrapped, ((a, b), {}))
meta_op_.sharding_annotation()
```

Note that in MetaDist, we need to shard and combine the input and output. However, TVM's kernel includes the output in the parameters of the input. Thus, we need to wrap the TVM kernel with the `fadd_wrapped` function before using it. Once the kernel is wrapped, we can apply the MetaSPMD annotation to the function via `sharding_annotation`. Adding distributed support for TVM is mainly missing communication features and distributed runtime. we will try to fully support TVM in future work.

A.2 DETAILED CONFIGURATION OF THE MODEL IN SECTION 4

For our weak-scaling experiments in Figure 8, we used three different models: GPT, WideResNet, and GAT. Regarding models for benchmarking, we carefully selected well-known models such as GPT and WResNet. These choices allowed us to conduct a scientific comparison with existing approaches effectively. Moreover, we intentionally included models like GAT, which lack designed parallelism and have not been extensively explored in the context of auto-parallelism.

The detailed configurations for each model are shown in Table 2, 3 and 4.

Table 2: Four sizes of GPT models for evaluation.

Number of GPUs	Number of parameters (billion)	Number of layers	Hidden size	Attention heads	Batch Size	TFLOPs
1	1.26	1	10240	40	8	27.75
2	2.52	2	10240	40	8	58.93
4	5.03	4	10240	40	8	121.29
8	10.07	4	14336	56	8	237.15

Table 3: Four sizes of WideResNet models for evaluation.

Number of GPUs	Number of parameters (billion)	Number of layers	Width	Batch Size	#FLOPs (tera)
1	0.63	50	448	64	37.08
2	0.63	50	448	128	74.17
4	1.23	50	320	128	145.22
8	1.23	50	320	256	290.44

Table 4: Four sizes of GAT models for evaluation.

Number of GPUs	Number of parameters (billion)	Number of nodes	Hidden size	Number of heads	#FLOPs (tera)
1	0.6	1024	24576	1	2.63
2	1.21	1024	24576	2	6.49
4	2.42	1024	24576	4	14.23
8	4.83	1024	24576	8	29.69

A.3 Details about Gather

What's challenging and interesting is the complexity of the different operation. MetaDist introduces two arguments in Gather, halo and block to support operators, such as convolution and concat. These two operators are the more frequently used operators in deep learning models. We found that the normal ShardCombine approach cannot describe these two operators. So we supported them by extending the Gather function. And because the complete exploration of the space is large and time consuming, we use some prior knowledge for efficiency.

A.3.1 HaloGather and Convolution

In Figure 9, the shard and gather is extended with a halo argument. In the left case, we . If it is positive, we add the data in the overlap region when we combine the results. If it is negative, we discard the data with width d and then perform the gather operation. In this case

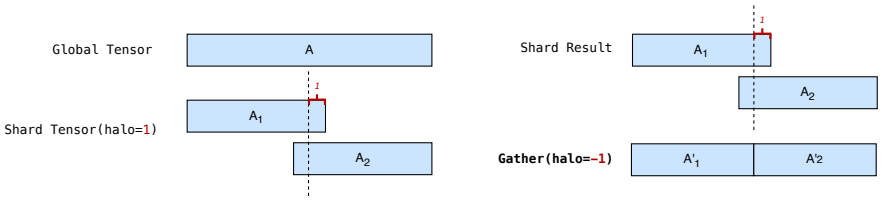

Figure 9: Meaning of the argument *halo* in Shard and Gather

Figure 10 illustrates the two most common convolution operators. The left panel displays $convolution(kernel = 3, pad = 0)$, while the right panel shows $convolution(kernel = 3, pad = 1)$. For simplicity and ease of understanding, we use 1D convolution. Similar methods can be used in 2D and 3D convolution. In both cases, we assume that the input Tensor is A, which has x elements. After shard, each of our two devices contains half of the elements of A. Then we perform a local convolution calculation. In convolution without padding, the halo argument can be inferred from the tensor size. In convolution with padding, since the last row of data is computed under ZERO padding and is not equivalent with the original computation, the size of the shard halo can be inferred from the `allclose_rows`. And when trying to GATHER, halo argument of GATHER can be inferred from the tensor size.

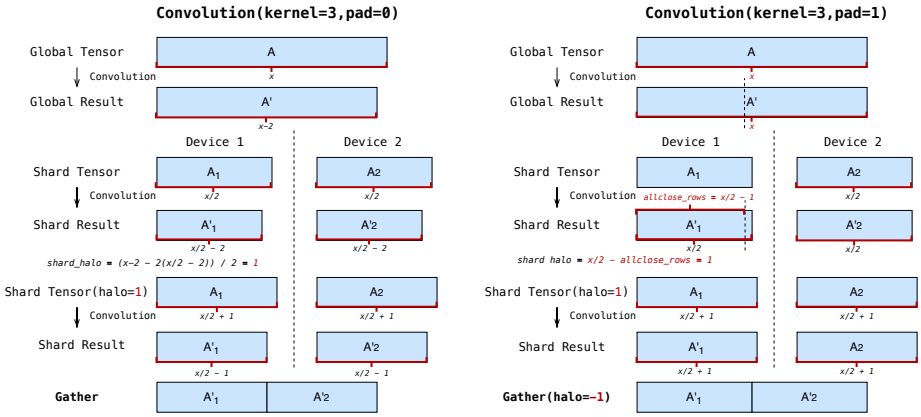

Figure 10: Two kinds of convolution (commonly used) are shown.

A.3.2 BLOCKGATHER AND CONCAT

The concat (or concatenate) operator accepts a set of tensors as input, and its output is a tensor created by splicing together the input tensors. We can observe that if we shard each input tensor and then concatenate them locally, the results do not align with the global result; it resembles a block-cyclic distribution. Therefore, we introduce the argument named `block`. The gather operation, with $block = n$, first divides the shard tensor into n parts, then performs `all-gather` on each part, and finally splices the results of the n `all-gather` parts.

Figure 11 shows depicts a concatenation of three tensors, A1, A2, B1, B2, C1, C2 represent the shards of these three tensors. After concatenating them locally, we observed that only the first `allclose_rows` elements could be aligned. Therefore, we can infer from this information that 'block' is set to 3, meaning that the combine function here is `Gather(block=3)`.'

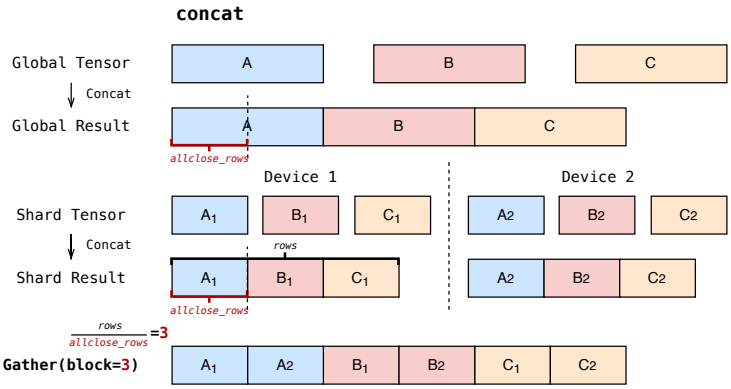

Figure 11: This example shows the concat operator concatenating three tensors. `rows` represents the number of rows in the first shard result. `allclose_rows` represents the number of rows in the first shard result that are allclose with the global result. Dividing the two yields the guessed argument of block, which is used to try and validate.