Concepts, Constructs and the Design Rationale

August 24, 2020

Contents

In this document, "TensorShape", "Tensor", and "TensorArray" that begin with a capital letter particularly refer to a type, while "tensor shape", "tensor" and "tensor array" refer to the general concpet.

1 Basic concepts

1.1 TensorShape

TensorShape $S =< S_1, S_2, ..., S_n >$ is a tuple of n non-negative integers that specify the size of each dimension of a tensor.

- TensorShape is a collection of non-negative integers which are **ordered** and **immutable**.
- TensorShape is iterative. Its elements can be traversed through sequentially.
- TensorShape is written as two parentheses containing integers in it.

```
(3,5,3)
() # empty tuple ?
```

[TBD]: encode uncontrolled dimension and inform its range.

1.2 Tensor

A dense Tensor is a ***fixed-size multi-dimensional*** container of primary arithmetic type. A tensor x:Tensor is characterized by meta data:

1. elemental type T, including basic arithmetic type: real, bool, integer.
2. shape S(x): shape defines multi-dimensional indices, each index access decides the physical location of an item stored in the tensor.
3. layout: the order that each dimension is stored.

The n dimensions of a tensor canbe stored in any order. A n-dimensional tensor is a ***n-way collection of numbers with primary arithmetic type***. Data stored in a tensor can be accessed from n directions, but a high-dimensional tensor can only be accessed efficiently in one direction. Figure 1 shows the idea.

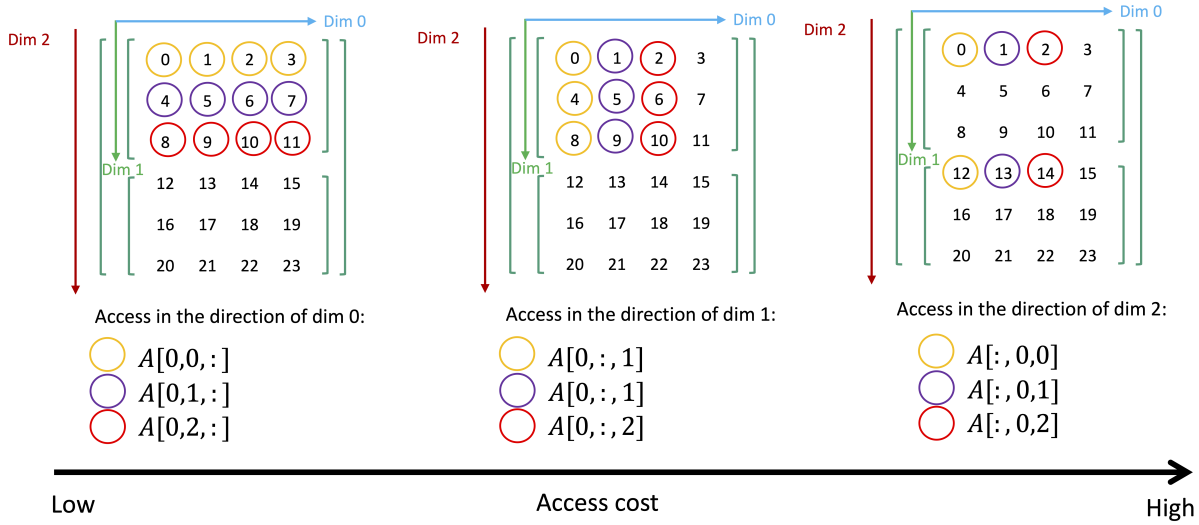


Figure 1: Access a 3-d tensor.

1.3 TensorArray

A TensorArray is a ***variable-size one-dimensional list-like*** container of fixed-size tensors or tensor arrays. Items in a TensorArray can either be Tensors or TensorArrays. A n-depth nested TensorArray is a ***n-way collection of fixed-size Tensors***. TensorArray can be analogous to std::vector<T> in C++, where T = Tensor or TensorArray.

For a tensor array x:TensorArray[T], T is either:

1. T = Tensor called a primary TensorArray. For a primary TensorArray, all tensors stored in x ***should be homogenous***. That is, every tensor takes up the same size block of memory, and all blocks are interpreted in exactly the same way.
2. T = TensorArray indicates a nested tensor array. For a nested tensor array, tensors stored in all depths should have a same primary arithmetic type and a same shape. [TBD]: how flexible a TensorArray should be?

A tensor array x:TensorArray[T] is characterized by meta info:

1. elemental type T that is either Tensor or TensorArray.
2. length L(x): length defines one-dimensional indices, and maximum length is declared.

Nested tensor array provides a constraint way to encode sparsity on the basis of high-level abstraction tensor. The constrain requires indices in a single tensor array are continous. Future work that extends the internal storage format of TensorArray and optimizes accessing methods for such a format could make TensorArray supports general sparsity.

1.3.1 Motivations for TensorArray

Main purposes of introducing the TensorArray abstraction include:

1. Provides a flexible data structure to boundle data that are not as regular as tensors.
    - A nested tensor array forms a rectangular polyhedron in high-dimensional space, providing a way for users to encode structural data.
2. Raise the abstraction level of data unit manipulated in the computation.
    - Many optimized linear algebra routines are performed on blocks of memory and algorithm developers do not care more details about scalar-level computations. Based on these observations, nested tensor array provides a logic bundle of a set of large memory blocks. The deeper tensor arrays are nested, the larger memory is involved, and more opportunities to exploit parallelism and to reduce the overhead of item access are exposed.
    - Iterating and accessing tensor arrays derive its performance from the compile-time analysis.
    - TensorArray raises the abstraction level of Tensor. Interfaces to access items and iterate through Tensor and TensorArray share many similarities. They could be interconvertible through internal reasoning. High-level abstraction helps trade-off analysis complexity with global optimality.
3. Enable precise memory-based dependence analysis to improve program parallelism.
    - Parallel patterns become implicit for algorithms that iterate over ***nested*** tensor arrays using nested looping constructs. TensorArray abstracts away low-level details of underlying memory, and makes dependence analysis simplified with the help of high-level abstractions.
4. Separate the effects and interfaces of operations applied to fixed-size and variable-size memory blocks which also cause different storage formats and memory management internally.
    - Undecidable memory size at compile-time leads to dynamics. TensorArray operations will delegate to different implementations according to whether compile-time analysis can reason about strong static conditions of a tensor array.

1.3.2 Shape of a nested TensorArray

Tensors in tensor arrays and nested tensor arrays are stored in a continous memory. Shape of nested tensor array, a meta data, is in the form of a tree. Figure 2 shows the intuitioin.

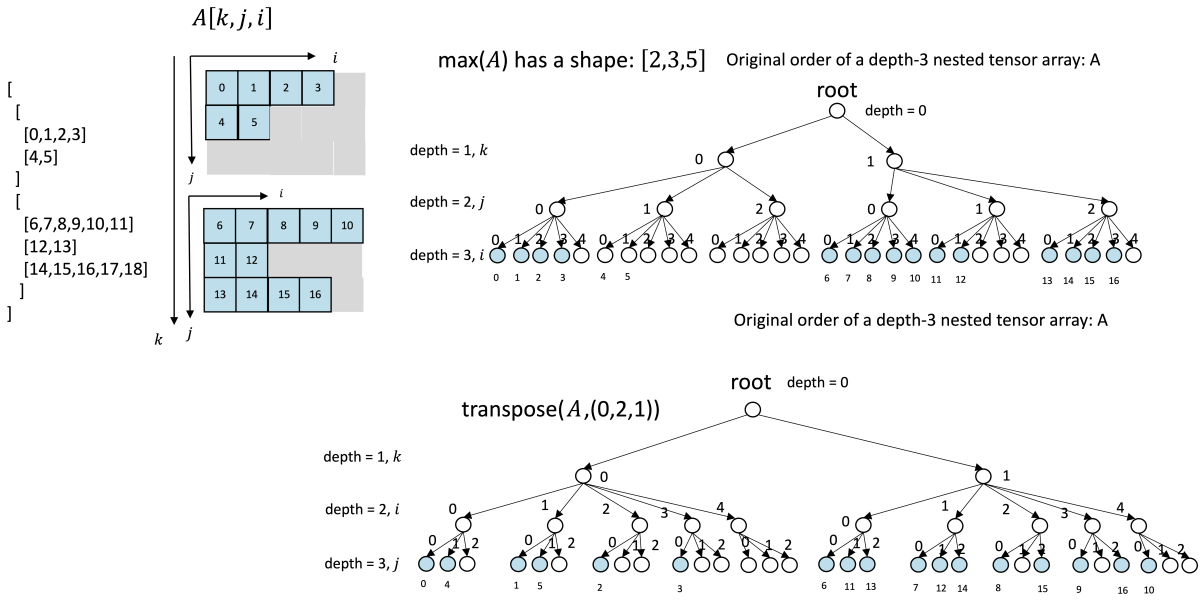


Figure 2: A conceptual diagram of the nested tensor array.

2 TensorShape operations

2.1 Basic query

2.1.1 *size*

size: returns a tuple of tensor A's size, or the size of a given dimension.

```
size(A:Tensor[T]) -> Tuple[int]
size(A:Tensor[T], dim:int) -> int
```

2.1.2 *numel*

numel: returns the cardinality of tensor A. It is equivalent to prod(size(A)).

```
numel(tup:Tuple[int]) -> int
```

2.1.3 *ndims*

ndims: returns number of dimensions of A.

```
ndims(tup:Tuple[int]) -> int
```

2.2 Element updating

- Tuples are immutable so its value cannot be updated or changed.
- A new tuple canbe created from a tuple.

2.2.1 *add*

```
add(tup1:Tuple[int], tup2:Tuple[int]) ->Tuple[int]
```

Examples:

```
tup1 = (1, 2, 3)
tup2 = (4, 5)
tup1 + tup2 = (1, 2, 3, 4, 5)
```

2.2.2 *insert*

insert an element into a tuple at a given position

```
insert(tup:Tuple[int], pos:int, value:int) -> Tuple[int]
```

Example:

```
tup = (1, 2, 3)
insert(tup, 1, 5)
ans = (1, 5, 2, 3)
```

2.2.3 *del*

remove tuple elements by index

```
del(tup:Tuple[int], pos:int) -> Tuple[int]
```

Example:

```
tup = (2, 3, 4)
del(tup, 1)
ans = (2, 4)
```

2.2.4 *replace*

replace the i-th tuple element with a given value. ***replace*** is a combination of ***del*** and ***insert***.

```
replace(tup:Tuple[int], pos:int, value:int) -> Tuple[int]
```

Example:

```
tup = (2, 3, 4)
replace(tup, 1, 5)
ans = (2, 5, 4)
```

2.3 Index generation

2.3.1 *cartesian_product*

```
cartesian_product(*tup:Tuple[int]) -> Tuple[Tuple[int]]
```

Examples:

```
tup1 = (2, 3, 4)
tup2 = (0, 1)
cartesian_product(tup1, tup2)
ans = ((2, 0), (3, 0), (4, 0), (2, 1), (3, 1), (4, 1))
```

2.3.2 *meshgrid*

2.3.3 *arange*

2.4 Shape function

Shape function recieves shapes of input operands and optional arguments, such as a specified dimension, stride, etc. and returns the shape of the output.

$$S(Y) = \Gamma(S(X_i), **\text{kwargs})$$

where $i = 0, ..., N$ and N is the number of operands.

3 TensorArray creation

3.1 Conversion from Tensor

3.1.1 *slices*

```
slices(X:Tensor, dim:int) -> TensorArray[Tensor]
```

3.1.2 *chunks*

3.2 Construction from Looping

4 Atomic item access and manipulation

Both Tensor and TensorArray are iterative. Their dimensions and elements can be efficiently traversed through using looping constructs defined in section 7 in some particularly order. It is sufficient to keep two rules in mind:

1. a tensor hold scalars while a tensor array holds fixed-size tensors.
2. tensor array provides constraint way to encode ***sparsity*** on the basis of the high-level abstraction tensor.

The second rule helps to make primitives for item access and manipulation supported by TensorArray intelligible: primitives like slice and transpose naturally manipulate items having continous indices will not supported by TensorArray as a first design consideration.

Signatures of operations that access and manipulate tensor and tensor array elements share many similarities.

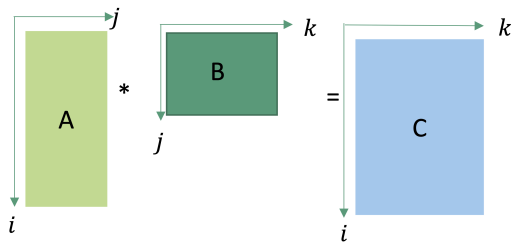
	Tensor	TensorArray
index	✓	✓
gather/scatter	✓	✓
slice	✓	X

Arithmetic computations defined on tensors and tensor arrays are indexed expressions, therefore iterating over dimensions and accessing items significantly affect performance. Tensor and tensor arrays cannot be efficiently accessed and iterated over by random orders, but in a specific order that will be optimized by program analysis and transformations internally. Figure 3 shows the intuitioin.

4.1 *index* and *index_**

```
Y:T = index(func::callable=unary_add, X:Tensor[T], index:Tuple[int])
Y:T = index(func::callable=unary_add, X:TensorArray[T], index:Tuple[int])
```

The efficiency of matrix multiplication $C = A * B$ is determined by the layout of three tensors: A, B and C.



- A is accessed along rows: row major has a lower access overhead.
- B is accessed along columns: column major has a lower access overhead.
- A lower access overhead of C is determined by the ordering of loop nests in the kernel:
  - iterating over i in the outer loop → row major
  - iterating over k in the outer loop → column major

Figure 3: Tensor layout and the order of loops in kernel computations are correlated.

1. y = index(x) creates a new Tensor/TensorArray y to hold the result of retrieving a ***single*** item from x. Type of the returned result is determined by the elemental type x holds which can be arithmetic type, a fixed-sized tensor, or a tensor array.
2. index_* is composite operation

4.1.1 Shape function

$S(Y) = S(T)$.

4.1.2 Forward computation

a pure access function that maps indices to a single data point.

4.1.3 Differentiation rule

```
dx:Tensor[T] = index_add(dX:Tensor[T], dY:T, index:Tuple[int])
```

index_add is a composite operation equivalent to:

```
index(func::callable=add, dX:Tensor[T], dY:T, index:Tuple[int])
```

- dX is the gradient variable corresponding to X. dX has exactly the same shape as X.
- dY is the gradient variable corresponding to Y. dY has exactly the same shape as Y.
- The backward computation is to compute dX and dY is known.

4.2 *gather_** and *scatter_**

1. gather and scatter are dual operations. gather is a multi-index selection method.
2. ***gather_**** and ***scatter_**** are enssentially composite functions. The "*" part is arithmetic operations, including "max","min","mean","sum".

```
Y:Tensor[T] = gather(func:callable, X:Tensor[T], dim:int, indices:Tuple[int])
Y:Tensor[T] = scatter(func:callable, X:Tensor[T], dim:int, indices:Tuple[int])
```

1. ***shape function***
2. ***computation***
3. ***differentiation rule***

4.3 *slice* and *slice_**

```
slice(X:Tensor[T], start:int, stride:int, end:int, dim:int) -> Tensor[T]
# or
X[:,idx,:]
```

1. shape function $S(\mathbf{Y})$

$$S(\mathbf{Y}) = \Gamma(S(\mathbf{X}), \text{dim}, \text{keep\_dim})$$

$$= \begin{cases} \text{insert}(\text{del}(S(\mathbf{Y}), \text{dim}), \text{dim}, 1), & \text{if} \quad \text{keep\_dim} \\ \text{del}(S(\mathbf{Y}), \text{dim}), & \text{otherwise} \end{cases}$$

2. computation
3. differentiation rule

5 Layout transformation

	Tensor	TensorArray
transpose	✓	X

5.1 *transpose*

6 Meta info modification: dimension construction

	Tensor	TensorArray
reshape	✓	X
squeeze	✓	X
unsqueeze	✓	X

6.1 *reshape*

6.2 *squeeze/unsqueeze*

7 Looping and conditional branching constructs

Looping constructs integrate legality preconditions in the construct's semantics. Consturcts that indicate parallel patterns do not need additionally dependence analysis.

	TensorShape	Tensor	TensorArray
foreach	✓	X	✓
parallel_foreach	X	✓(elementwise)	✓
scan	X	✓	✓
zip	X	X	✓
reduce	X	✓	✓
filter	X	✓	✓
broadcast	X	✓	✓
apply_along_axis	X	✓	X

7.1 *foreach*

sequential looping whether its body can be executed in parallel needs dependence analysis.

7.2 *parallel_foreach*

explict parallel looping.

```
parallel_for(func:callable, X:Iterative, **kwargs) -> Iterative # parallel for
```

- iteration domain: $0 \le i \le \text{len}(X) - 1$
- access function: $i \to X[i]$
- shape function is defined when input X is a Tensor array, otherwise (a case is iterating over a index set) None.
  - Suppose shape function of func is $S(\text{func}) = \Gamma(S(X))$;
  - Then $S(\text{foreach}) = (\text{len}(X)) + S(\text{func})$.
- computation: **parallel_foreach** implies parallel execution of apply(func, $\mathbf{X}[i]$, **kwargs).

7.3 *zip*

7.4 *scan*

7.5 *reduce*

7.6 *apply_along_axis*:a combination of gather, scatter and parallel_foreach

```
apply_along_axis(func:callable, dim:int, X:Tensor, **kwargs) -> Tensor
```

1. shape function:
   - Suppose shape function of func is: $S(\text{func}) = \Gamma(x)$ which returns a tuple contains only 1 element;
   - Then, $S(\text{apply\_along\_axis}) = \text{replace}(S(\mathbf{X}), \text{dim}, S(\text{func}))$.

2. computation

$$\begin{aligned}\mathbf{Y}^{S(\text{apply\_along\_axis})} = \textbf{parallel\_foreach}(i, \text{index} \quad \textbf{in} \quad & \text{cartesian\_product}(\text{del}(S(\mathbf{Y}), \text{dim})) \\ & \text{indices} = \textbf{parallel\_foreach}(i \rightarrow \text{insert}(\text{indices}, \text{dim}, i), \text{arange}(\text{size}(\mathbf{Y}, \text{dim}))) \\ & \mathbf{Y}_0^{\text{size}(S(\mathbf{Y}),\text{dim})} = \textbf{gather}(\mathbf{Y}, \text{indices}) \\ & \mathbf{Y}_1^{\text{size}(S(\mathbf{Y}),\text{dim})} = \textbf{apply}(\textbf{func}, \text{args...}, \mathbf{Y}_0^{\text{size}(S(\mathbf{Y},\text{dim})}) \\ & \mathbf{Y}_2^{\text{size}(S(\mathbf{Y}),\text{dim})} = \textbf{scatter}(\mathbf{Y}, \text{indices})\end{aligned}$$

7.7 *broadcast*

```
broadcast(func:callable, X:Tensor, Y:Tensor) -> Tensor
```

X is the bigger Tensor, **Y** is the smaller tensor.

7.8 *filter*

7.9 *sortby*

8 Neural network specialized tensor, tensor array operations

8.1 *embedding*

Embedding is parallel slicing.

```
embedding(X:Vector[int], Y:Tensor[T], dim:int) -> Tensor[T]
```

1. shape function: $S(\mathbf{Z}) = \Gamma(S(\mathbf{X}), S(\mathbf{Y}), \text{dim})$

$$\begin{aligned}S(\mathbf{Z}) &= \Gamma(S(\mathbf{X}), S(\mathbf{Y}), \text{dim}) \\ &= (S(\mathbf{X})[0]) + \text{del}(S(\mathbf{Y}), \text{dim})\end{aligned}$$

2. computation:

$$\begin{aligned}&\mathbf{Z} = \text{reshape}(\mathbf{Z}, (N, -1)) \\ &\textbf{foreach} \quad (i, x) \quad \textbf{in} \quad \mathbf{X}_N \\ &\quad \mathbf{Z}[i] = \text{slice}(\mathbf{Y}, \mathbf{X}[i], \text{dim}, \textit{keep\_dim} = \text{false}) \\ &\mathbf{Z} = \text{reshape}(\mathbf{Z}, (N) + \text{del}(S(\mathbf{Y}), \text{dim}))\end{aligned}$$

   - iteration domain: $0 \le i \le \text{size}(\mathbf{X}, 0) - 1$
   - access function:
     (a) $f_1 : \{i \rightarrow \mathbf{Z}[i]\}$
     (b) f_2 : [TBD]
     (c) $f_3 : \{i \rightarrow \mathbf{X}[i]\}$

3. differentiation rule:

9 Optimization capability as an explicit interface

9.1 Vectorized function: *vectorize*

10 Case Study: Transformer

Hyper parameters are **literal constants**:

1. $d = 512$: embedding dimension = hidden dimension
2. $n_{\text{head}} = 8$: number of heads
3. $d_{ff1} = 2048$
4. $d_{ff2} = d = 512$

Symbolic constants:

1. batch_size
2. L: sequence length

Notations:

- $\otimes$: matrix multiplication
- $\mathbf{Var}_n^{\text{tensor shape}}$: learnable parameter.
  - n is the index of learnable parameter.
  - tensor_shape is the shape of learnable parameter.
  - If tensor_shape is omitted and Var is not in bold, such a learnable parameter is a scalar.
- $\mathbf{Y}_n^{\text{tensor shape}}$: produced intermediate result which is immutable in forward computation.
  - n index of the variable.
  - tensor_shape: shape of the variable.
  - If tensor_shape is omitted and Y is not in bold, such a variable is a scalar.

10.1 Basic Building Blocks

10.1.1 Positional encoding

Can be fused into one element-wise kernel.

$$\mathbf{Y}_0^{d\times L} = \text{slice}(\mathbf{Token}^{L\times 1}, \mathbf{Var}_0^{d\times \text{vocab\_size}}, \text{dim} = 1). * \sqrt{d}. + \mathbf{PosEnc}^{d\times L}$$

10.1.2 Multi-head attention

Below is the equation for multi-head self-attention which suppress various details of computation process.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}})\mathbf{V} \tag{1}$$

1. $\mathbf{Y}_1^{d\times L} = \mathbf{Var}_1^{d\times d} \otimes \mathbf{Y}_0^{d\times L}$
2. $\mathbf{Y}_2^{d\times L} = \mathbf{Var}_2^{d\times d} \otimes \mathbf{Y}_0^{d\times L}$
3. $\mathbf{Y}_3^{d\times L} = \mathbf{Var}_3^{d\times d} \otimes \mathbf{Y}_0^{d\times L}$
4. $\mathbf{Y}_4^{n_1\times n_2\times L} = \mathbf{reshape}(\mathbf{Y}_1^{d\times L}, (n_1, n_2, L))$
   - $n_1 = n_{\text{head}}$, $n_2 = d//n_{\text{head}}$;

- the **Q** part in equation (1);

5. $\mathbf{Y}_5^{n_1\times n_2\times L} = \textbf{reshape}(\mathbf{Y}_2^{d\times L}, (n_1, n_2, L))$
   - $n_1 = n_{\text{head}}$, $n_2 = d//n_{\text{head}}$;
   - the **K** part in equation (1);
6. $$\mathbf{Y}_8^{n_1\times L\times L} = \textbf{parallel\_foreach}(\mathbf{Y}_{6i}^{n_2\times L}, \mathbf{Y}_{7i}^{n_2\times L} \quad \textbf{in} \quad \mathbf{Y}_4[i,:,:], \mathbf{Y}_5[i,:,:]), \quad i \in [0,1,...,n_1-1]$$
   $$\mathbf{Y}_{8i}^{L\times L} = (\mathbf{Y}_{6i}^{n_2\times L})^T \otimes \mathbf{Y}_{7i}^{n_2\times L}$$
7. $\mathbf{Y}_9^{n_1\times L\times L} = \mathbf{Y}_8^{n_1\times L\times L} .* \sqrt{d}$
8. $\mathbf{Y}_{10}^{n_1\times L\times L} = \textbf{apply\_along\_axis}(\text{softmax}, 1, \mathbf{Y}_9^{n_1\times L\times L})$
9. $\mathbf{Y}_{11}^{n_1\times n_2\times L} = \textbf{reshape}(\mathbf{Y}_4^{d\times L}, (n_1, n_2, L))$
   - $n_1 = n_{\text{head}}$, $n_2 = d//n_{\text{head}}$;
   - the **V** part in equation (1);
10. $$\mathbf{Y}_{12}^{n_1\times n_2\times L} = \textbf{parallel\_foreach}(\mathbf{Y}_{13i}^{n_2\times L}, \mathbf{Y}_{14i}^{n_2\times L} \quad \textbf{in} \quad \mathbf{Y}_{11}[i,:,:], \mathbf{Y}_{10}[i,:,:]), \quad i \in [0,1,...,n_1-1]$$
    $$\mathbf{Y}_{12i}^{n_2\times L} = \mathbf{Y}_{13i}^{n_2\times L} \otimes \mathbf{Y}_{14i}^{L\times L}$$
11. $\mathbf{Y}_{15}^{d\times L} = \textbf{reshape}(\mathbf{Y}_{12}^{d\times L}, (n_1 * n_2, L))$

Info:

$\mathbf{Y}^N = \text{softmax}(\mathbf{X}^N)$

1. $Y_0 = \textbf{reduce}(\max, \mathbf{X}^N)$
2. $\mathbf{Y}_1^N = \textbf{broadcast}(-, \mathbf{X}^N, Y_0)$
3. $\mathbf{Y}_2^N = \textbf{parallel\_foreach}(\exp, \mathbf{Y}_1^N)$
4. $Y_3 = \textbf{reduce}(+, \mathbf{Y}_2)$
5. $\mathbf{Y}_4^N = \textbf{broadcast}(/, \mathbf{Y}_2^N, Y_3)$

10.1.3 Add and norm

1. $\mathbf{Y}_{16}^{d\times L} = \textbf{parallel\_foreach}(+, \mathbf{Y}_{15}^{d\times L}, \textbf{PosEnc}^{d\times L})$
2. $\mathbf{Y}_{17}^{d\times L} = \textbf{apply\_along\_axis}(\text{norm}, 0, \mathbf{Y}_{16}^{d\times L})$
3. $\mathbf{Y}_{18}^{d\times L} = \textbf{broadcast}(*, \mathbf{Y}_{17}^{d\times L}, \text{Var}_4)$
4. $\mathbf{Y}_{19}^{d\times L} = \textbf{broadcast}(+, \mathbf{Y}_{18}^{d\times L}, \text{Var}_5)$

Info:

$\mathbf{Y}^N = \text{norm}(\mathbf{X}^N)$.

1. $Y_0 = \textbf{reduce}(+, \mathbf{X}^N)/N$
   - mean
2. $\mathbf{Y}_1^N = \textbf{broadcast}(-, \mathbf{X}^N, \mathbf{Y}_0)$
3. $\mathbf{Y}_2^N = \textbf{parallel\_foreach}(\textbf{pow}, \mathbf{Y}_1^N, 2)$
4. $Y_3 = \textbf{reduce}(+, \mathbf{Y}_2^N)$
5. $Y_4 = \text{sqrt}(Y_3/N)$
   - variance
6. $Y_5 = \textbf{parallel\_foreach}(x \to (x - Y_0)/(Y_5 + \epsilon), \mathbf{X}^N)$
   - $\epsilon = 1e^{-6}$ is a constant.

10.1.4 Pointwise feed forward

$$\text{FFN}(\mathbf{X}) = \max(0, \mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

1. $\mathbf{Y}_{20}^{L \times d_{ff1}} = (\mathbf{Y}_{19}^{d \times L})^T \otimes \mathbf{Var}_6^{d \times d_{ff1}}$
2. $\mathbf{Y}_{21}^{L \times d_{ff1}} = \textbf{broadcast}(+, \mathbf{Y}_{20}^{L \times d_{ff1}}, \mathbf{Var}_7^{d_{ff1}})$
3. $\mathbf{Y}_{22}^{L \times d_{ff1}} = \textbf{parallel\_foreach}(\text{relu}, \mathbf{Y}_{21}^{L \times d_{ff1}})$
   - relu is a scalar function:

$$\text{relu}(x) = \begin{cases} x & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

4. $\mathbf{Y}_{23}^{L \times d_{ff1}} = \textbf{parallel\_foreach}(\text{dropout}, \mathbf{Y}_{22}^{L \times d_{ff1}}, \text{drop\_rate=0.5})$
   - dropout is a scalar function:

$$\text{dropout}(x, \text{drop\_rate}) = \begin{cases} x & \text{random}() \leq \text{drop\_rate} \\ 0 & \text{otherwise} \end{cases}$$

5. $\mathbf{Y}_{24}^{L \times d} = \mathbf{Y}_{23}^{L \times d_{dff1}} \otimes \mathbf{Var}_8^{d_{ff1} \times d}$
6. $\mathbf{Y}_{25}^{L \times d} = \textbf{broadcast}(+, \mathbf{Y}_{24}^{L \times d}, \mathbf{Var}_9^{d})$

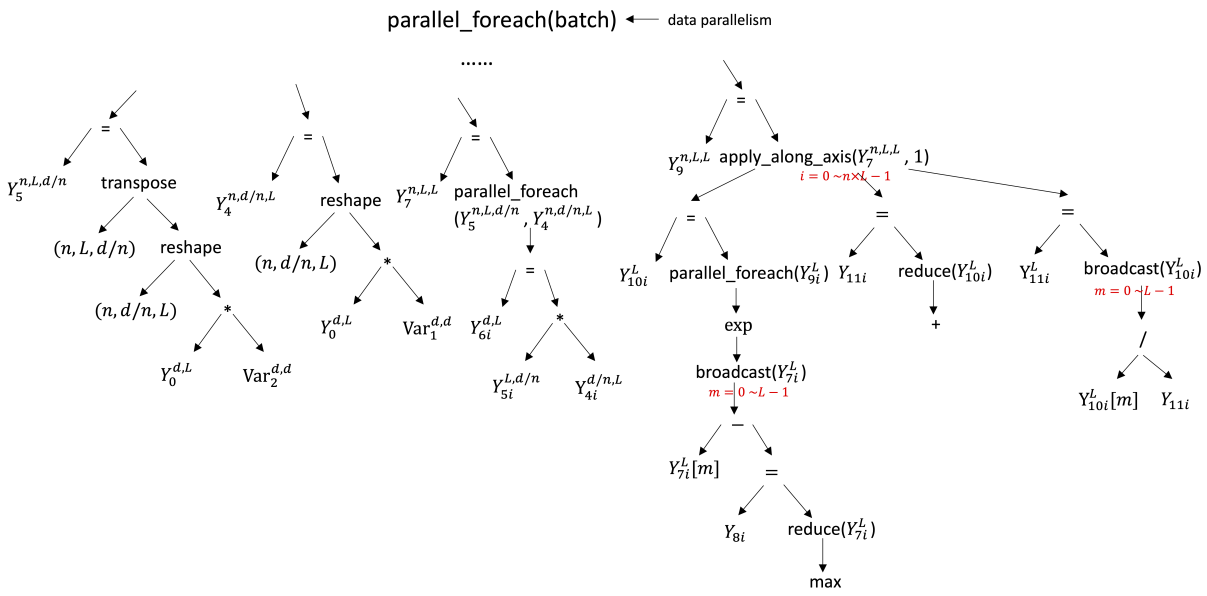


Figure 4: AST for softmax($K^T Q$)

10.2 Optimizations

10.2.1 Loop fusion

10.2.2 Loop distribution

10.2.3 Data movement removal (layout optimization)

10.2.4 CSE

10.2.5 Auto-batching of variable-length sequences