Tensor Data Scattering and the Impossibility of Slicing Theorem[1]

Wuming Pan

College of Computer Science and College of Software Engineering, Sichuan University

Chengdu City, Sichuan Province, China
Panwuming@scu.edu.cn

Abstract

This paper establishes a broad theoretical framework for tensor data dissemination methods used in various deep learning frameworks. This paper gives a theorem that is very important for performance analysis and accelerator optimization for implementing data scattering. The theorem shows how the impossibility of slicing happens in tenser data scattering. This paper proposes an algorithm called ScatterX and its source code is provided.

Keywords: tensor, slice, scatter, deep learning

1. Introduction

Tensor data dissemination is a kind of operations provided by multiple deep learning frameworks. Such frameworks include TensorFlow and pyTorch. However, these operations are difficult to use the hardware features of machine learning accelerators. We will analyze the reasons for those difficulties in this paper.

Next in this paper we will define tensors and related operations mathematically. Then, in the section 3, we define the transformer and its provision tensor that are critical for tensor data distribution. In the fourth section we discuss the uncertainty brought about by the application of the transformer, which is the source of uncertainty in the scattering of tensor data.

---

[1] Written on November 8, 2020. Subject to change.

2. Tensor

To strictly discuss these problems, we need argue them mathematically. Other than in programming language, the syntax representations are not really able to make two mathematical objects different. All mathematical objects eventually should be embodied in the set theory. We need start from elementary mathematic objects, such as the integer set $\mathbb{Z}$, the real number set $\mathbb{R}$, products of sets, mappings and functions between sets, etc.

We use $\mathbb{N}_m$ to denote the set of m nonpositive integers from 0 to $m-1$, i.e. $\mathbb{N}_m = \{0,1,\cdots,m-1\}$. Specifically, we think that $\mathbb{N}_0 = \{\,\} = \emptyset$. We also use $\mathbb{N}_{h:k}$ to denote the set $\{h, h+1, \cdots, k-1\}$ where $h \leq k$ and both are integers, possibly negative.

2.1 Tuple

A tuple is an element in a Cartesian product of some sets. For a tuple t, $\ell(t)$ denotes its length, i.e., the number of sets in the product set which t is in. A tuple with length 1 is a number. A tuple with length 0 is not a number, just represented as $(\,)$. We treat indices as they can be concatenated with operator + as python tuples. For example:

$$(1,2,3) + 4 = (1,2,3,4)$$

$$(1,2,3) + (4,5) = (1,2,3,4,5)$$

$$(1,2,3) + (\,) = (1,2,3)$$

2.2 Tensor

A tensor is a multidimensional data array. Tensor's data elements can be accessed through their indices. An index of a data element is a tuple.

Definition 1: A **tensor** is a function

$$E: \prod_{i=0}^{k-1} \mathbb{N}_{m_i} \to \mathbb{R}$$

The tuple $(m_0, m_1, \cdots, m_{k-1})$ is called the shape of tensor E, denoted with $\mathbb{s}_E$. And the elements in $\prod_{i=0}^{k-1} \mathbb{N}_{m_i}$ is called **indices** of E. For any $(I_0, I_1, \cdots, I_{k-1}) \in \prod_{i=0}^{k-1} \mathbb{N}_{m_i}$, we simply write $E\big((I_0, I_1, \cdots, I_{k-1})\big)$ as $E[I_0, I_1, \cdots, I_{k-1}]$. The number of total elements in E is called its **size**. The notation $\mathbb{I}_S$ stands for the set of all indices of shape S.

To represent a tensor, we use [ ] to brackets tensor data. For example, a tensor $E0$ has the shape $(3,3,2)$, then $E0$ is represented as

$$E\_0 = \big[[[0 \quad 0][0 \quad 1][0 \quad 2]] \quad [[1 \quad 0][1 \quad 1][1 \quad 2]] \quad [[2 \quad 0][2 \quad 1][2 \quad 2]]\big]$$

or

$$E\_0 = \begin{bmatrix} [[0 \quad 0][0 \quad 1][0 \quad 2]] \\ [[1 \quad 0][1 \quad 1][1 \quad 2]] \\ [[2 \quad 0][2 \quad 1][2 \quad 2]] \end{bmatrix}$$

This is different than matrix data arrange format.

A tuple is not a one-dimension tensor. They are different, as they come from two different sets: the set of all tensors and the set of all tuples, and there are adopted different operations on the two sets.

Definition 2: The operator τ change a one-dimension tensor to tuple, and the operator $\mathbb{t}$ change a tuple to a one-dimension tensor.

2.3 Picks and Slices

Definition 2: A **pick** is a finite integer function $p: \mathbb{N}_n \to \mathbb{Z}$ for some nonnegative integer n. It can be used as an operation on indices. Let I be an index, $p(I)$ is an index J such that $J[i] = I[p(i)]$. A pick p is **smooth** if only if p monotonically maps consecutive numbers to consecutive numbers. A pick p operates on an integer sets A is just the set $A \cap \{p(i)\}$. We define picks $\mathbb{i}_n: \mathbb{N}_n \to \mathbb{N}_n$ as $\mathbb{i}_n(j) = j$ and call them **identity picks** with rank n. We define picks $\mathbb{i}_{n:m}: \mathbb{N}_{n:m} \to \mathbb{N}_{n:m}$ as $\mathbb{i}_{n:m}(j) = j$ and call them **identity picks** with rank n to m. We define that $\mathbb{c}(p) = p(\mathbb{N}_n)$. If $p(\mathbb{N}_n) = \mathbb{N}_n$, then we call p is a **shuffle**.

Sometimes a pick is like a projection into indices. For example

$$\begin{array}{ccc} 0 & 1 & 2 \\ 4 & 6 & 7 \end{array}$$

is a pick, we also write it as a one-dimensional tensor $[4 \quad 6 \quad 7]$.

The result of pick is deterministic.

Definition 3: For any k picks p_i, $i = 0,1,2,\cdots,k-1$, E restricted on $M = \prod_{i=0}^{k-1} p_i(\mathbb{N}_{m_i})$ is called a **embedded slice** of E. The tensor $G = E[p_0, p_1, \cdots, p_{k-1}]$ is defined as

$$G[I_0, I_1, \cdots, I_{k-1}] = E[p_0(I_0), p_1(I_1), \cdots, p_{k-1}(I_{k-1})]$$

is called a **slice** of E. If there is a j such that for $i = 0,1,2,\cdots,j-1$ the domain of p_i is $\{0\}$, and for $i = j, j+1, \cdots, k-1$, there is $p_i = \{\mathbb{i}_{m_i}\}$, and the tensor H with shape $(m_j, m_{j+1}, \cdots, m_{k-1})$ is defined as

$$H[I_j, I_{j+1}, \cdots, I_{k-1}] = E[p_0(I_0), p_1(I_1), \cdots, p_{k-1}(I_{k-1})]$$

then H is called a **subtensor** of E, and H is also denoted as $E[p_0(I_0), p_1(I_1), \cdots, p_{j-1}(I_{j-1})]$.

For example, let

$$E\_00 = [[0 \quad 0][0 \quad 1][0 \quad 2]]$$

$$E\_01 = [[1 \quad 0][1 \quad 1][1 \quad 2]]$$

$$E\_02 = [[2 \quad 0][2 \quad 1][2 \quad 2]]$$

then $E00$, $E01$ and $E02$ are subtensors of $E0$. And $E0$ can represented as

$$E\_0 = [E\_00 \quad E\_01 \quad E\_02]$$

$$= [E\_0[0] \quad E\_0[1] \quad E\_0[2]]$$

Definition 4: Let ${}^{\mathbb{I}_{S_1}}\!/_{p_1}$ be the set of subsets of $\mathbb{I}_{S_1}$, and for any $C \in {}^{\mathbb{I}_{S_1}}\!/_{p_1}$, there is a $I \in \mathbb{I}_{S_1}$, and

$$C = \{K | p_1(I) = p_1(K), K \in \mathbb{I}_{S_1}\}$$

we also denote $C = {}^{\mathbb{I}_{S_1}}\!/_{p_1}(I)$. We call C is a slice **picked** by p_1.

3. Tensor Transformer and its Provision Tensor

Definition 5: A **tensor transformer** is a map $T: \mathbb{I}_{S_1} \to \mathbb{I}_{S_2}$, it can be used as an operation on tensors. Let A be a tensor with shape S_A, then $T(A)$ is a set of tensors. For any $B \in T(A)$ and $I \in \mathbb{I}_{S_A} \cap \mathbb{I}_{S_1}$, there exists a $J \in T^{-1}\big(T(I)\big)$, such that $B[T(J)] = A[J]$, and there is $\mathbb{I}_{S_B} = T\big(\mathbb{I}_{S_A} \cap \mathbb{I}_{S_1}\big)$.

A pick can act as a transformer, but it is not true vice versa.

Definition 6: A transformer can be defined by a tensor. Given a transformer $T: \mathbb{I}_{S_1} \to \mathbb{I}_{S_2}$, $\mathbb{e}(T)$ is a tensor E with shape $S_1 + \ell(S_2)$ and it is defined by

$$T(I) = \tau(E[I])$$

we call T is **provisioned** by E, and E is a **provisioner** of T.

For example, let $S_1 = (4,2)$ and $S_2 = (2,2,2,2)$, given tensor of shape $(4,2,4)$

$$E\_1 = \begin{bmatrix} \begin{bmatrix} [0 & 0 & 0 & 0] \\ [0 & 0 & 0 & 1] \end{bmatrix} \\ \begin{bmatrix} [0 & 0 & 1 & 0] \\ [0 & 0 & 1 & 1] \end{bmatrix} \\ \begin{bmatrix} [0 & 1 & 0 & 0] \\ [0 & 1 & 0 & 1] \end{bmatrix} \\ \begin{bmatrix} [0 & 1 & 1 & 0] \\ [0 & 1 & 1 & 1] \end{bmatrix} \end{bmatrix}$$

a transformer T can be defined as

$$T(I) = (E\_1[I(0), I(1), 0], E\_1[I(0), I(1), 1], E\_1[I(0), I(1), 2], \\ E\_1[I(0), I(1), 3])$$

If $I = (3,0)$ then $T(I) = (0,1,1,0)$.

4. Nondeterministic of Applying Transformer

Let A be a tensor with shape $\mathbb{s}_A$. Given a transformer $T: \mathbb{I}_{S_1} \to \mathbb{I}_{S_2}$, and a tensor $B \in T(A)$. For any $I \in \mathbb{I}_{S_1} \cap \mathbb{I}_{\mathbb{s}_A}$, and for any $J \in T^{-1}\big(T(I)\big) \cap \mathbb{I}_{S_1} \cap \mathbb{I}_{\mathbb{s}_A}$, providing there is $T(J) = T(I)$。 There is non-deterministic when applying a transformer on a tensor:

- For any $B \in T(A)$, that $B\big(T(I)\big) = A(I)$ or $B\big(T(I)\big) = A(J)$ holds is both possible but only one of them is true.

- Moreover, for some $K \in \mathbb{I}_{\mathbb{s}_B}$, possibly $K \notin T\left(\mathbb{I}_{\mathbb{s}_A}\right)$ is true.

Hence, we have:

Proposition 1: The result of applying a transformer on a tensor is nondeterministic.

Proof: Easy.

Now we can investigate the scattering algorithms in deep learning frameworks. Those algorithms can eliminate the second indeterminate problem above.

5. Scattering

5.1 Scatter APIs in Two Popular Deep Learning Frameworks

In TensorFlow, the typical scatter API looks like[1]:

```
tensor_scatter_nd_update(ts, indices, updates, name=None)
```

Use the notions in this paper, the *indices* argument in this API is a provisioner of a transformer $T_{indices}: \mathbb{I}_{S_1} \to \mathbb{I}_{S_2}$. The argument *ts* is a tensor, and

$$\ell(\mathbb{s}_{ts}) = \ell(S_2)$$

$$T_{indices}\left(\mathbb{I}_{S_1}\right) \subset \mathbb{I}_{\mathbb{s}_{ts}}$$

This API creates a tensor B which is a result of transformer $T_{indices}$ applying on tensor *updates*, for any $K \notin T_{indices}\left(\mathbb{I}_{S_1}\right)$, the identity $B(K) = ts(K)$ must holds

In pyTorch, the typical scatter API looks like[2]:

```
scatter(self, dim, index, src, reduce=None)
```

Still use the notions in this paper, the *index* argument in this API is a provisioner of a transformer $T_{index}: \mathbb{I}_{S_1} \to \mathbb{I}_{S_2}$. Then we can define a transformer $T_{scatter}: \mathbb{I}_{S_1} \to \mathbb{I}_{S_1}$ such that

$$T_{scatter}(I) = \mathbb{i}_{dim}(I) + T_{index}(I) + \mathbb{i}_{dim+1:\ell(I)}(I)$$

And then this API creates a tensor C which is a result of transformer $T_{scatter}$ applying on the tensor *src*, for any $K \notin T_{indices}\left(\mathbb{I}_{S_1}\right)$, the identity $C(K) = self(K)$ must holds.

To fuse these two algorithms is not as easy as it seems.

5.2 Defining Scattering

A scattering is a tensor transformer apply on a tensor A and the result tensor B is restricted by a tensor X.

Definition 7: A **scattering** is a triple $e = (T, A, X)$, where $T: \mathbb{I}_{S_1} \rightarrow \mathbb{I}_{S_2}$ is a tensor transformer, A and X are two tensors. A result of scattering e is a result B of T applying on A, and for any J, if $J \in \mathbb{I}_{S_B}$ and $J \notin \mathbb{I}_{S_A}$, then $B[T(J)] = X[J]$ holds.

Since a transformer can be represented by a tensor, a scattering is also defined by a triple (E, A, X) of tensors E, A and X. For an instance, using tensor E_1 in last section to provision a transformer, given

$$A\_1 = [[1 \quad 2] \quad [3 \quad 4] \quad [5 \quad 6] \quad [7 \quad 8]]$$

and

$$X\_1 = \begin{bmatrix} \begin{bmatrix} \begin{bmatrix} [0 & 0] \\ [0 & 0] \end{bmatrix} & \begin{bmatrix} [0 & 0] \\ [0 & 0] \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} \begin{bmatrix} [0 & 0] \\ [0 & 0] \end{bmatrix} & \begin{bmatrix} [0 & 0] \\ [0 & 0] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

then the result of applying E_1 on A_1 into X_1 is a tensor

$$B\_1 = \begin{bmatrix} \begin{bmatrix} \begin{bmatrix} [1 & 2] \\ [3 & 4] \end{bmatrix} \\ \begin{bmatrix} [5 & 6] \\ [7 & 8] \end{bmatrix} \end{bmatrix} & \begin{bmatrix} \begin{bmatrix} [0 & 0] \\ [0 & 0] \end{bmatrix} \\ \begin{bmatrix} [0 & 0] \\ [0 & 0] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The result of scattering is also indeterministic.

The counterpart scattering of the TensorFlow scatter API as in section 5.1 is $(T_{indices}, updates, ts)$, whereas the counterpart scattering of the pyTorch scatter API as in section 5.1 is $(T_{scatter}, src, self)$. The key difference of these two kinds of APIs is how the transformer in scattering is formed.

6. Sliceable Scattering

In some case, when we scatter data from a source tensor into a target tensor, we can replace a slice of target tensor with a slice of source tensor, and the two slices have the same shape. If both slices can specified with last few dimensions of shapes of source and target tensor, we call such scatter operation sliceable.

Definition 8: A tensor transformer $T: \mathbb{I}_{S_1} \to \mathbb{I}_{S_2}$ is called **sliceable**, if and only if there are two smooth picks p_1 and p_2, and the two conditions are satisfied:

1. for any $I \in \mathbb{I}_{S_1}$, there is $p_1(I) + p_2(I) = I$;
2. there is a transformer T' such that for any $I \in \mathbb{I}_{S_1}$, $T(I) = T'(p_1(I)) + p_2(I)$ holds.

p_2 is called a **sliceable end pick** of T. If there does not exist a nonempty sliceable pick for T, we call it **not sliceable**.

Let’s see an example.

$$E\_2 = \begin{bmatrix} \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

is sliceable. Because there is a pick

$$p\_2\_1 = [0]$$

and a pick

$$p\_2\_2 = [1 \quad 2]$$

and a transformer T'_2 whose provision tensor is

$$\mathbb{e}(T'\_2) = \begin{bmatrix} [0 & 0] \\ [1 & 1] \end{bmatrix}$$

Definition 9: We say a tensor transformer $T: \mathbb{I}_{S_1} \to \mathbb{I}_{S_2}$ is **weak sliceable**, if and only if there are three picks p_1, p_2 and p, and there is a transformer T' such that for any $I \in \mathbb{I}_{S_1}$, $T(I) = p\left(T'(p_1(I)) + p_2(I)\right)$ holds. p_2 and q_2 are called **weak sliceable picks** of T.

Let’ s see another example. The transformer

$$E\_3 = \begin{bmatrix} \begin{bmatrix} \begin{bmatrix} [0 & 0 & 0 & 0] \\ [0 & 1 & 0 & 1] \end{bmatrix} \\ \begin{bmatrix} [1 & 0 & 1 & 0] \\ [1 & 1 & 1 & 1] \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} \begin{bmatrix} [2 & 0 & 0 & 0] \\ [2 & 1 & 0 & 1] \end{bmatrix} \\ \begin{bmatrix} [3 & 0 & 1 & 0] \\ [3 & 1 & 1 & 1] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

is weak sliceable. Because there are picks

$$p\_3\_1 = [0 \quad 1 \quad 2]$$

$$p\_3\_2 = [1 \quad 2]$$

and a pick

$$p\_3 = [0 \quad 1 \quad 2 \quad 3] = \mathbb{i}_4$$

and a transformer T'_3 whose provision tensor is

$$\text{ⓔ}(T'\_3) = \begin{bmatrix} \begin{bmatrix} \begin{bmatrix} [0 & 0] \\ [0 & 1] \end{bmatrix} \\ \begin{bmatrix} [1 & 0] \\ [1 & 1] \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} \begin{bmatrix} [2 & 0] \\ [2 & 1] \end{bmatrix} \\ \begin{bmatrix} [3 & 0] \\ [3 & 1] \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

7. X-Transformer

A x-transformer T is decorated with three picks and a function or a transformer.

Definition 10: A tensor transformer $T: \mathbb{I}_{S_1} \rightarrow \mathbb{I}_{S_2}$ is called a **x-transformer**, if and only if there are three picks p_1, p_2, p and a transformer f, and for any $I \in \mathbb{I}_{S_1}$ and

$J \in \mathbb{I}_{S_2}$, there is $J = p\left(f\left(p_1(I)\right) + p_2(I)\right)$. $p\left(f\left(p_1(I)\right) + p_2(I)\right)$ is called a **representation** of x-transformer T.

Corollary 1: A transformer is a x-transformer if and only if it is weak sliceable.

Proof: Easy.

Hence E_3 is a x-transformer, for it is weak sliceable.

The representation of a x-transformer is not unique.

Theorem 1: Any x-transformer T is not sliceable, if in all its representation $p\left(f\left(p_1(I)\right) + p_2(I)\right)$, there is $\mathbb{c}(p_1) \cap \mathbb{c}(p_2) \neq \emptyset$.

Proof: It is easy.

7.1 Disseminating slices

Given three picks p_1, p_2, p and a transformer f, a transformer $T: \mathbb{I}_{S_1} \to \mathbb{I}_{S_2}$ is defined as

$$T(I) = p\left(f\left(p_1(I)\right) + p_2(I)\right)$$

Definition 10: Notation being as above, if p_1 and p_2 do not chose any same dimension, and let C be any tensor such that $\mathbb{I}_C \subseteq {}^{\mathbb{I}_{S_1}}/_{p_2}$, we call $T(C)$ is a **disseminated slice**.

8. ScatterX Algorithm

Now we implement x-transformer using python code.

A tensor should be a multidimensional numpy array, its shape should is a tuple of nonnegative integers. A pick is denoted by a one-dimensional tensor with a shape n.

Import the package:

```
import numpy as np
```

Define pick:

```
def getPick(p):
    if p.ndim != 1:
        raise Exception("ndim does not equal 1 ! ");
    lp = p.shape[0]

    def pick(i):
        if lp == 0:
            return np.array([], dtype='int16')
        mx = p.max();
        mn = p.min();
        if i.ndim != 1:
            raise Exception("ndim does not equal 1 ! ");

        if mx >= i.shape[0]:
            raise Exception("Argument Invalid ! ");
        j = np.zeros(lp, dtype = 'int16');
        for k in range(lp):
            j[k]= i[p[k]];
        return j
    return pick
```

Define traversal:

```
def traversal(traversalShape,action):
    ilen = len(traversalShape)
    if ilen == 0:
        raise Exception("traversalShape's length == 0 ! ");
    elif traversalShape[0] == 0:
        raise Exception("traversalShape's length == 0 ! ");
    traversalInd = np.array(traversalShape);
    def traversalForInner(i):
        n = traversalShape[i];
```

```
        if i < ilen-1:
            for j in range(n):
                traversalInd[i] = j;
                traversalForInner(i+1);
        elif i == ilen-1:
            for j in range(n):
                traversalInd[i] = j;
                action(traversalInd);
        else:
            raise Exception("Exception raised！");
    traversalForInner(0)
```

Define getProvisionXTransformer:

```
def getProvisionXTransformer(shape0, shape1, T0, p0, p1, p):
    if p0.dtype != 'int16' or p0.dtype != 'int16' or p.dtype != 'int16':
        raise Exception("Data type error！");
    elif p0.ndim != 1 or p0.ndim != 1 or p.ndim != 1:
        raise Exception("ndim to form pick does not equal 1！");
    E = np.zeros(shape0 + (len(shape1),), dtype = 'int16');
    I = np.zeros(len(shape0), dtype = 'int16');
    J = np.zeros(len(shape1), dtype = 'int16');
    pick0 = getPick(p0);
    pick1 = getPick(p1);
    pick = getPick(p);
    def combine(IC):
        J0 = pick0(IC);

        Jp = T0[tuple(J0)]
        J1 = pick1(IC)
        Jl = list(tuple(Jp) + tuple(J1))
        Ja = np.array(Jl, dtype = 'int16')
```

```
        J = pick(Ja)
        E[tuple(IC)] = J;
    traversal(shape0,combine)

    return E
Define Scattter:
def scatter(src, X, XTransformer):
    shape = XTransformer.shape[:-1];
    I = np.zeros(len(shape), dtype = 'int16');
    def action(Ia):
        src[tuple(XTransformer[tuple(Ia)])] = X[tuple(Ia)]
    traversal(shape,action)
    return src

def scatterX(src, X, shape1, T0, p0, p1, p):
    shape0 = X.shape;

    XTransformer = getProvisionXTransformer(shape0, shape1, T0, p0, p1, p);

    scatter(src, X, XTransformer)
    return src
def tensorflowScatter(tensor, indices, updates):
    l = list(range(len(indices.shape) - 1))

    tl = indices.shape[-1]
    ln = list(range(len(tensor.shape)))

    t1 = ln[tl:]

    p0 = np.array(l, dtype='int16')
    p1 = np.array(t1, dtype='int16')
    p = np.array(ln, dtype='int16')
    scatterX(tensor, updates, tensor.shape, indices, p0, p1, p)
```

More test case please see GitHub project [3].

9. Conclusion

Tensor data dissemination is a task that is difficult to use the hardware features of machine learning accelerators. This article theoretically analyses the reasons for this difficulty. And a general theory and algorithm of tensor data dissemination is established in this article. Based on the theories and algorithms in this article, we will be able to design algorithm implementations that can make better use of accelerator features.

References

[1] TensorFlow API: tf.tensor_scatter_nd_update. https://www.tensorflow.org/api_docs/python/tf/tensor_scatter_nd_update.

[2] PyTorch Docs: torch.Tensor.scatter_: https://pytorch.org/docs/stable/tensors.html?highlight=scatter#torch.Tensor.scatter.

[3] ScatterX Project. https://github.com/wmpan/ScatterX.

Appendix

Python code.