STen: An Interface for Efficient Sparsity in PyTorch

Andrei Ivanov
Department of Computer Science, ETH Zürich
{firstname.lastname}@inf.ethz.ch

Nikoli Dryden

Torsten Hoefler

Abstract

As deep learning models grow, sparsity is becoming an increasingly critical component of deep neural networks, enabling improved performance and reduced storage. However, existing frameworks offer poor support for sparsity. They primarily focus on sparse tensors in classical formats such as COO and CSR, which are not well suited to the sparsity regimes typical of deep learning, and neglect the broader sparsification pipeline necessary for using sparse models. To address this, we propose a new sparsity interface for PyTorch, STen, that incorporates sparsity layouts for tensors (including parameters and transients, e.g., activations), sparsity-aware operators, and sparsifiers, which define how a tensor is sparsified, and supports virtually all sparsification methods. STen can enable better sparse performance and simplify building sparse models, helping to make sparsity easily accessible.

1 Introduction

Deep learning models are growing voraciously, and require ever greater amounts of compute and memory [10, 15, 18, 20]. To address this, sparsity has emerged as a major research and engineering direction [6, 9]. Sparsity is widely used to reduce storage requirements and improve performance during inference. More recent work has begun to focus on sparsity during training, which can also improve performance while helping to break the memory wall for large models. Indeed, frameworks have begun to provide direct support for sparse tensors. The PyTorch [19] torch.sparse module includes COO and CSR tensors and a limited set of operations and the TensorFlow [1] tf.sparse module similarly supports COO tensors. However, these modules primarily support sparse tensors, rather than an entire sparsification pipeline, and lack native support for sparsification operators that can efficiently produce sparse tensors. They therefore offer limited productivity improvements and do not provide a clear path toward supporting broad usage of sparsity. Further, at sparsities common in machine learning (50–95%), although classical sparse matrix formats reduce storage, they perform worse than dense implementations. While blocked formats (e.g., ELL, BCSR) support efficient implementations by calling dense kernels for each block, they restrict where nonzeros can be placed and can limit the information preserved after sparsification. Another approach is to use masks, which emulate sparsity by zeroing out elements, but offer no storage reduction.

To address this, we first propose a new programming model for sparsity in PyTorch (§2), overviewed in Fig. 1. This model consists of three components: sparsity layouts for tensors; operators, which provide implementations for computations with any combination of sparsities for input and output tensors; and sparsifiers, which are applied to operator outputs to compute a new sparse tensor. Sparsifiers are further classified as streaming, blocking, or materializing, based on the number of output values they require. Our model supports the vast majority of sparsification approaches, and enables them to be implemented efficiently; for example, threshold pruning is a streaming sparsifier, and a high-performance implementation could be inlined into operators. We provide an initial implementation of this model, STen, in PyTorch (§3) for CPU inference, but aim to lay the foundation for sparse training on accelerators. We also provide performant operators and an $n:m$ sparsity format [16, 27], where each group of $m$ elements has $n$ nonzeros, which is a middle ground between

Appeared at the 2022 Sparsity in Neural Networks workshop.
unstructured and block sparsity that performs well for deep learning workloads and is practical on CPUs and GPUs. With STen, we show (§4) that sparse-dense matrix multiplication can outperform PyTorch’s built-in sparsity in the 50–90% sparsity regime: up to $54 \times$ faster than its COO format and $3 \times$ faster than its CSR format. We also match or outperform dense formats at 60% or greater sparsity. Our code and examples are available at https://github.com/spcl/sten.

2 Programming Model for Sparsity

We now introduce our programming model for sparsity. It consists of three concepts, sparsity layouts, operators, and sparsifiers, which we discuss in turn. We discuss implementation details in §3. While existing frameworks have some support for sparse tensors and operators, they lack explicit support for sparsifiers, which are manually incorporated by users in an ad hoc manner. Instead, we manage the entire sparsification pipeline, simplifying usage while enabling improved performance.

**Sparsity layouts** define the sparsity structure used by a tensor, extending the usual memory layout by annotating the sparsity format (e.g., CSR, COO, $n:m$) and associated parameters (e.g., $n$ and $m$).

**Operators** are any standard operator, with any number of input and output tensors. Each tensor may have any memory and sparsity layout, and an operator may have different implementations for each combination of tensor layouts for maximum performance.

**Sparsifiers** define how to decide which output values to keep, and are associated with each output of an operator. A sparsifier can be thought of as a special kind of operator, and may include additional inputs that delay its application until they are ready (e.g., gradients for first-order sparsification). Note a sparsifier may produce output in a different sparsity layout than what its associated operator outputs. We use the term *sparse operator* to refer to the combination of an operator and sparsifier.

We further classify sparsifiers as one of *streaming*, *blocking*, or *materializing*. A streaming sparsifier is applied to each output value to decide whether to drop it, in a single pass, before writing to the output tensor. Blocking sparsifiers require a small set of output values to decide which ones to drop. Finally, materializing sparsifiers require the operator to fully store all values. Note that a sparsifier may be fused into an operator for performance, or used standalone to convert dense tensors to sparse.

Table 1 lists example sparsifiers and their characteristics. The trivial *keep-all* sparsifier preserves all produced values and is the default for dense tensors. It is not limited to dense tensors, however: the sum of two sparse tensors with a keep-all sparsifier may produce a new sparse tensor with nonzero values given by the union of the nonzeros of the inputs. A *random fraction* sparsifier drops values with a fixed probability, while a *scalar threshold* drops them if they are less than a fixed threshold. *Scalar fraction* drops the smallest portion of the values (i.e., magnitude pruning) and *block-wise fraction* drops entire blocks with the smallest combined absolute magnitude. *Per-block fraction* drops the smallest proportion of values within fixed blocks of elements. Finally, more advanced complex weight sparsifiers, which require additional information such as the loss or gradients, are also supported. These examples are not exhaustive, and our model can support nearly any sparsifier.

2.1 Constructing Sparse Models

We consider two cases for building sparse models: constructing a model from scratch or sparsifying an existing model. A sparse model is set up by providing a list of tensors and a desired sparsity layout...
Table 1: Sparsifier types and examples, the number of passes over a tensor made, their memory requirements (nnz total nonzeros, block size b when blocking), and sparsifier type. Some complex weight sparsifiers could be implemented more efficiently than with materialization.

<table>
<thead>
<tr>
<th>Sparsifier</th>
<th>Examples</th>
<th>Passes</th>
<th>Memory</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keep-all Sparse add</td>
<td></td>
<td>1</td>
<td>$O(1)$</td>
<td>streaming</td>
</tr>
<tr>
<td>Random fraction Dropout</td>
<td>[22]</td>
<td>1</td>
<td>$O(1)$</td>
<td>streaming</td>
</tr>
<tr>
<td>Scalar threshold ReLU</td>
<td>[14]</td>
<td>1</td>
<td>$O(1)$</td>
<td>streaming</td>
</tr>
<tr>
<td>Scalar fraction Magnitude</td>
<td>[5, 28]</td>
<td>2</td>
<td>$O(nnz)$</td>
<td>materializing</td>
</tr>
<tr>
<td>Block-wise fraction</td>
<td>Block magnitude [12]</td>
<td>2</td>
<td>$O(nnz)$</td>
<td>materializing</td>
</tr>
<tr>
<td>Per-block fraction</td>
<td>$n,m$ [16, 27]</td>
<td>2</td>
<td>$O(b)$</td>
<td>blocking</td>
</tr>
<tr>
<td>Complex weight sparsifiers</td>
<td>Movement [21], $\ell_0$</td>
<td>≥ 1</td>
<td>$O(nnz)$</td>
<td>materializing</td>
</tr>
</tbody>
</table>

for each. This provides enough information for a library to initialize tensors and dispatch operators to specific sparse formats. If an operator implementation is not available, the user can provide one or convert tensors to a supported sparsity layout. Note that all tensors are used as operator inputs, outputs, or both; for simplicity, we will ignore sparsity for model inputs and outputs. The remaining input-only tensors are typically weights. All other tensors are used as both inputs and outputs and occur within the computation graph (e.g., activations); we refer to them as intermediate tensors. Note that in practice, intermediate tensors do not exist until runtime in most frameworks, so their sparsity layout is instead defined by the operator that produces them.

**Constructing a sparse model** from scratch is similar to the typical process in PyTorch, but tensors and operators in its computational graph are annotated with specific sparsity layouts.

**Sparsifying an existing model** requires marking a subset of the model’s tensors as sparse. While this is straightforward for weights, making intermediate tensors sparse is more challenging. Unlike when building a model from scratch, we cannot mark operators as sparse: this would require modifying or rewriting the original model definition, a significant overhead for a user. Further, identifying all operators in a model is hard (not all have names or are registered, e.g., **nn.functional** operators). Instead, to identify intermediate tensors, we can run the model once to collect this information.

We now discuss sparse inference, then proceed to training. To sparsify existing dense weights or load sparse weights, we need only the desired sparsity layout and sparsifier. If a non-materializing sparsifier is given, we first trivially convert it to a materializing version. Then the weights are sparsified and subsequent operator calls will use the sparse version. Intermediate tensors are sparsified at runtime, as they do not exist in advance.

For training, we need to also consider error signals (sometimes called neural gradients) and gradients, which may have independent sparse layouts and sparsifiers from their associated forward pass tensors. These are treated identically to intermediate tensors in the forward pass. However, note that in this case, weight tensors are no longer input-only, as gradient updates are applied to them. This is not a significant change from the user perspective, and mainly implies that materializing sparsifiers may be less efficient and sparsifying on the fly with the gradient update operator may be faster.

### 3 STen Implementation

We now discuss the implementation of our sparsity programming model in PyTorch, STen. The interface is in Python and interoperates with standard PyTorch models to enable a user to construct a sparse model from scratch or sparsify an existing dense model.

**Model construction.** To build a sparse model from scratch, the user essentially constructs a model as usual in PyTorch, but uses sparse versions of operators and tensors. The STen API supports sparse tensor and parameter classes, which can store tensors using any sparsity layout, and can be extended by the user to support additional layouts. Operators (e.g., matrix multiplication) can infer the layout of their input tensors, but the desired output tensor layout must be explicitly specified. Finally, where desired, sparsifiers can be associated with an operator to select which output values to keep.

```python
import torch, sten
a, b = torch.randn(10, 20), torch.randn(10, 20)
```
Sparsifying existing models. Alternatively, one can construct a sparse model from an existing dense model. To do this, the user provides the original PyTorch model, a list of tensor identifiers belonging to the model, and the desired sparsity layout and sparsifier for each tensor. To identify weight tensors, we use the fully-qualified names provided by PyTorch. For intermediate tensors, we obtain names using the torch.fx tracing framework. Note that adding sparsity to an intermediate tensor is equivalent to using a sparse operator with that output layout. Sparsifiers are then applied to existing dense weight tensors, which are then replaced by a sparse tensor, and operators that output sparse intermediate tensors are replaced with the corresponding sparse operator.

Operator implementations. Additionally, providing operator implementations for all combinations of input and output sparsity layouts is unrealistic given the large number of combinations, especially when users may provide their own formats. To overcome this, we support an operator extension API that accepts the operator name, a matching pattern for supported input and output sparsity layout combinations, and an actual implementation. When such an operator is called with matching tensors, we dispatch to the given implementation, which can make use of existing high-performance implementations (e.g., [2, 4, 7, 8]). If an implementation does not exist, the interface can optionally convert input tensors to a supported format; however, this requires the destination format to support this transformation losslessly (e.g., CSR) and sacrifices many opportunities for performance with the original layout (e.g., optimizations for blocked layouts). Otherwise, we fall back to using a dense implementation and issue a warning. This enables incremental optimization by performance engineers, who can decide which operators it is most profitable to provide sparse implementations for. It also allows users to explore the effects of sparsity on models (albeit with a performance penalty).

Backpropagation. Our programming model also supports sparsity during backpropagation. However, supporting automatic differentiation and backpropagation on arbitrary sparsity layouts and operators is challenging in PyTorch, as its autograd engine makes assumptions that limit its flexibility. In particular, the autograd engine is implemented in C++ and expects instances of torch.Tensor, meaning we cannot provide a separate class or wrap existing tensors to support additional sparsity layouts, and also limits the usefulness of custom implementations via torch.autograd.Function. Further, while torch.Tensor can be subclassed, these subclasses will still inherit the dense memory of the base class. To address this, we take a hybrid approach, and inherit from the empty tensor, wrapping custom...
We have proposed, and conducted initial evaluations on, a new interface for sparsity in PyTorch, STen. By directly incorporating the notion of a sparsifier into the programming model, we allow the interface to both better optimize performance (e.g., by not materializing tensors before sparsification and fusing sparsification into operators) and provide a complete pipeline for sparsification. Our interface also makes it easy to sparsify existing dense models (e.g., fine-tuning for sparsity) and to add support for additional sparse tensor layouts, operators, or sparsifiers. With STen, we aim to make sparsity easily accessible to ML users and practitioners. As next steps, we plan to extend our implementation to fully support sparse training as well as GPUs and other accelerators.

Acknowledgements

This project received funding from the European Research Council under the European Union’s Horizon 2020 programme (Project PSAP, No. 101002047); and received EuroHPC-JU funding under grant EU-Pilot, No. 101034126, with support from the Horizon 2020 Programme.
References


A Extended Case Studies

Here we present full details of two case studies using the STen interface. In the first, we sparsify an existing BERT model. In the second, we demonstrate constructing a sparse MLP from scratch. For full details, including notebooks and additional examples, see https://github.com/spcl/sten.

A.1 Case study: BERT

modify_existing.ipynb

Here we demonstrate the workflow of adding sparsity into existing models. As an example we take a single encoder layer of BERT [3].

```python
import torch
input_shape = (8, 128, 768) # batch, sequence, features
model = torch.hub.load('huggingface/pytorch-transformers', 'model', 'bert-base-uncased').encoder.layer[0]
input = torch.rand(input_shape)
output = model(input)
print(output[0].shape) # -> torch.Size([8, 128, 768])
```

We target all linear layers in this model, including feedforward and attention projection layers. A linear layer computes $y = xA^T + b$ and is defined in `torch.nn.Linear` module. In particular, we are going to sparsify tensors $A$ by magnitude pruning of 90% of their values and storing them in the CSR format. In the following snippet we collect the six weight tensors from linear layers, and assign sparsifiers to them.

```python
import sten
weights_to_sparsify = []
sb = sten.SparsityBuilder(model)
for module_name, module in model.named_modules():
    if isinstance(module, torch.nn.modules.linear.Linear):
        weight = module_name + '.weight'
        weights_to_sparsify.append(weight)
sb.set_weight(
            name=weight,
            initial_sparsifier=sten.ScalarFractionSparsifier(0.9),
            inline_sparsifier=sten.KeepAll(),
            tmp_format=torch.Tensor,
            external_sparsifier=sten.KeepAll(),
            out_format=sten.CsrTensor,
        )
print(weights_to_sparsify)
```

This yields the fully qualified names assigned by PyTorch to each of these tensors.

```plaintext
['attention.self.query.weight', 'attention.self.key.weight',
 'attention.self.value.weight', 'attention.output.dense.weight',
 'intermediate.dense.weight', 'output.dense.weight']
```

Next, we repeat the same process for intermediate tensors. In this example, we target only the output of the GELU activation. However, it is challenging to refer to this intermediate tensor, as we treat the module as a black box that we do not modify, and internal operators may have varying or no name, depending on the implementation. Examining the layer modules (`print(model)`) shows the model structure:

```python
BertLayer(
    ... (intermediate): BertIntermediate(
        (dense): Linear(in_features=768, out_features=3072, bias=True)
    )
)```
From this we see that the `model.intermediate` submodule contains the GELU activation, but we still do not know the name of the output intermediate tensor. We use the `torch.fx` tracer to assign deterministic names to the intermediate tensors:

```python
torch.fx.symbolic_trace(model.intermediate).graph.print_tabular()
```

The result of running this command shows that the output of `torch.nn.functional.gelu` is assigned to the tensor with the name `gelu` inside the `model.intermediate` module:

<table>
<thead>
<tr>
<th>opcode</th>
<th>name</th>
<th>target</th>
<th>args</th>
<th>kwargs</th>
</tr>
</thead>
<tbody>
<tr>
<td>placeholder</td>
<td>hidden_states</td>
<td>hidden_states</td>
<td>()</td>
<td>{}</td>
</tr>
<tr>
<td>call_module</td>
<td>dense</td>
<td>dense</td>
<td>(hidden_states,)</td>
<td>{}</td>
</tr>
<tr>
<td>call_function</td>
<td>gelu</td>
<td><code>&lt;built-in function gelu&gt;</code></td>
<td>(dense,)</td>
<td>{}</td>
</tr>
<tr>
<td>output</td>
<td>output</td>
<td>output</td>
<td>(gelu,)</td>
<td>{}</td>
</tr>
</tbody>
</table>

We now assign a random fraction sparsifier with 90% zeroing probability to the GELU output intermediate tensor. The sparsifier stores the tensor in COO format.

```python
sb.set_interm(
    name="intermediate.gelu",
    inline_sparsifier=sten.RandomFractionSparsifier(0.9),
    tmp_format=sten.CooTensor,
    external_sparsifier=sten.KeepAll(),
    out_format=sten.CooTensor,
)
```

Finally, we create a new sparse model from the original dense model and run it with the same arguments as before:

```python
sparse_model = sb.get_sparse_model()
output = sparse_model(input)
print(output[0].shape)  # -> torch.Size([8, 128, 768])
```

### A.2 Case study: MLP

`build_from_scratch.ipynb`

In this example, we show how to build a sparse model from scratch using a simple MLP. As reference we use the following implementation of a dense MLP:

```python
import torch
class MLP(torch.nn.Module):
    def __init__(self, channel_sizes):
        super().__init__()
        self.layers = torch.nn.Sequential()
        in_out_pairs = list(zip(channel_sizes[:-1], channel_sizes[1:])))
        for idx, (in_channels, out_channels) in enumerate(in_out_pairs):
            if idx != 0:
                self.layers.append(torch.nn.ReLU())
            self.layers.append(torch.nn.Linear(in_channels, out_channels))
```

```
```python
def forward(self, input):
    return self.layers(input)

model = MLP([50, 40, 30, 20, 30, 10])
output = model(torch.randn(15, 50))
print(output.shape)
```

We are going to replace `torch.nn.Linear` with our custom `SparseLinear` module, which will call our sparse implementation of `torch.nn.functional.linear`.

```python
import sten

class SparseLinear(torch.nn.Module):
    def __init__(self, input_features, output_features, weight_sparsity):
        super().__init__()
        self.weight_sparsity = weight_sparsity
        dense_weight = sten.random_mask_sparsify(
            torch.randn(output_features, input_features),
            frac=weight_sparsity
        )
        self.weight = sten.SparseParameterWrapper(
            sten.CscTensor.from_dense(dense_weight)
        )
        self.weight.grad_fmt = (
            sten.KeepAll(),
            torch.Tensor,
            sten.RandomFractionSparsifier(self.weight_sparsity),
            sten.CscTensor,
        )
        self.bias = torch.nn.Parameter(torch.rand(output_features))
        self.bias.grad_fmt = (
            sten.KeepAll(),
            torch.Tensor,
            sten.KeepAll(),
            torch.Tensor,
        )

    def forward(self, input):
        sparse_op = sten.sparsified_op(
            orig_op=torch.nn.functional.linear,
            out_fmt=tuple(
                [(sten.KeepAll(), torch.Tensor, sten.KeepAll(), torch.Tensor)]
            ),
            grad_out_fmt=tuple(
                [(sten.KeepAll(), torch.Tensor, sten.KeepAll(), torch.Tensor)]
            ),
        )
        return sparse_op(input, self.weight, self.bias)
```

The important aspect is the use of `SparseParameterWrapper` to hold the data of sparse tensors. The code above shows the sparsity configuration of weight and intermediate tensors gradients that will appear in the backward pass, although they are dense in this example. The remaining piece is the implementation of `SparseMLP`:

```python
```
class SparseMLP(torch.nn.Module):
    def __init__(self, channel_sizes, weight_sparsity):
        super().__init__()
        self.layers = torch.nn.Sequential()
        in_out_pairs = list(zip(channel_sizes[:-1], channel_sizes[1:] + [1]))
        for idx, (in_channels, out_channels) in enumerate(in_out_pairs):
            if idx != 0:
                self.layers.append(torch.nn.ReLU())
                self.layers.append(SparseLinear(in_channels, out_channels, weight_sparsity))

    def forward(self, input):
        return self.layers(input)

Finally, after the replacement of `torch.nn.Linear` with the `SparseLinear` in the MLP implementation, we call it and observe the expected output.

```python
model = SparseMLP([50, 40, 30, 20, 30, 10], 0.8)
output = model(torch.randn(15, 50))
print(output.shape)
```

A.3 Case study: Customization
custom_implementations.ipynb

This example demonstrates the API to register custom operator implementations for specific input and output tensor formats. This example demonstrates customization API to define new sparse tensor formats and sparsifier. It shows how to register custom operator and sparsifier implementations for them.

```python
import torch
import sten
import scipy
```

Start from the dense implementation of $d = (a + b)c$.

```python
a = torch.randn(10, 20, requires_grad=True)
b = torch.randn(10, 20, requires_grad=True)
c = torch.randn(20, 30, requires_grad=True)
grad_d = torch.randn(10, 30)
d = torch.mm(torch.add(a, b), c)
d.backward(grad_d)
```

First we define a custom random fraction sparsifier functioning the same as `sten.RandomFractionSparsifier`. The sparsifier implementation is not defined here since it is characterized not only by the sparsifier itself but also by the input and output tensor formats. The sparsifier class only needs to declare its configurable parameters.

```python
class MyRandomFractionSparsifier:
    def __init__(self, fraction):
        self.fraction = fraction
```

Then declare a tensor in CSC format that will utilize scipy CSC implementation under the hood.

```python
class MyCscTensor:
    def __init__(self, data):
        self.data = data
```
@staticmethod  
def from_dense(tensor):  
    return MyCscTensor(scipy.sparse.csc_matrix(tensor))

def to_dense(self):  
    return torch.from_numpy(self.data.todense())

Then we make the result of addition $a + b$ sparse. To achieve this, we need to replace the addition operator with its sparse counterpart. For simplicity, we do not use an inline sparsifier, which is why the operator outputs a dense `torch.Tensor` after applying the `KeepAll` sparsifier. We use an external random fraction sparsifier with 0.5 dropout probability and output the tensor in the newly defined CSC format. The same specification is assigned to the gradient format, but nothing prevents us from applying a different sparsifier and using a different format for the gradient.

```python
sparse_add = sten.sparsified_op(
    orig_op=torch.add,
    out_fmt=(
        (sten.KeepAll(), torch.Tensor,  
         MyRandomFractionSparsifier(0.5), MyCscTensor),
    ),
    grad_out_fmt=(
        (sten.KeepAll(), torch.Tensor,  
         MyRandomFractionSparsifier(0.5), MyCscTensor),
    ),
)
```

Then we try to use the operator.

```python
d = torch.mm(sparse_add(a, b), c)
```

Output:

```
WARNING:root:Sparse operator implementation is not registered (fwd). op:  
    ~ <built-in method add of type object at 0x7f033a216ea0> inp: (<class  
    ~ 'torch.Tensor'>, <class 'torch.Tensor'>, None, None) out: (<class  
    ~ 'sten.KeepAll'>, <class 'torch.Tensor'>),.. Fallback to dense  
    ~ implementation.
WARNING:root:Sparsifier implementation is not registered. sparsifier:  
    ~ <class '__main__.MyRandomFractionSparsifier'> inp: <class  
    ~ 'torch.Tensor'> out: <class '__main__.MyCscTensor'>. Fallback to dense  
    ~ keep all implementation.
WARNING:root:Sparse operator implementation is not registered (fwd). op:  
    ~ <built-in method mm of type object at 0x7f033a216ea0> inp: (<class  
    ~ '__main__.MyCscTensor'>, <class 'torch.Tensor'>) out: (<class  
    ~ '__main__.MyCscTensor'>, <class 'torch.Tensor'>),.. Fallback to dense  
    ~ implementation.
```

The first error message indicates the operator implementation which is required is not registered. Here we register it and try calling the method again.

```python
@sten.register_fwd_op_impl(
    operator=torch.add,
    inp=(torch.Tensor, torch.Tensor, None, None),
    out=tuple([sten.KeepAll, torch.Tensor]),
)

def sparse_add_fwd_impl(ctx, inputs, output_sparsifiers):
    input, other, alpha, out = inputs
    return torch.add(input, other, alpha=alpha, out=out)
```

```python
d = torch.mm(sparse_add(a, b), c)
```
Here we see that sparsifier implementation is not registered. Let’s provide it.

```python
@sten.register_sparsifier_implementation(
    sparsifier=MyRandomFractionSparsifier, inp=torch.Tensor, out=MyCscTensor
)
def scalar_fraction_sparsifier_dense_coo(sparsifier, tensor):
    return sten.SparseTensorWrapper(
        MyCscTensor.from_dense(
            sten.random_mask_sparsify(tensor, frac=sparsifier.fraction)
        )
    )
```

As expected, it works. The next step is to call the backward pass and see what remains to be implemented there.

```python
d = torch.mm(sparse_add(a, b), c)
try:
    d.backward(grad_d)
except sten.DispatchError as e:
    print(str(e))
```
Sparse operator implementation is not registered (bwd). op: <built-in method mm of type object at 0x7f033a216ea0> grad_out: (torch.Tensor,), grad_inp: ((torch.Tensor), (torch.Tensor),) inp: (MyCscTensor, torch.Tensor).

Registering the backward implementation for `torch.mm`.

```python
@sten.register_bwd_op_impl(
    operator=torch.mm,
    grad_out=(torch.Tensor,),
    grad_inp=(
        (sten.KeepAll, torch.Tensor),
        (sten.KeepAll, torch.Tensor),
    ),
    inp=(MyCscTensor, torch.Tensor),
)
def torch_mm_bwd_impl(ctx, grad_outputs, input_sparsifiers):
    input1, input2 = ctx.saved_tensors
    [grad_output] = grad_outputs
    grad_input1 = torch.mm(grad_output, input2.T)
    grad_input2 = torch.from_numpy(input1.wrapped_tensor.data.transpose() @ grad_output)
    return grad_input1, grad_input2

d = torch.mm(sparse_add(a, b), c)
try:
    d.backward(grad_d)
except sten.DispatchError as e:
    print(str(e))
```

Output:

Sparse operator implementation is not registered (bwd). op: <built-in method add of type object at 0x7f033a216ea0> grad_out: (MyCscTensor,), grad_inp: ((torch.Tensor), (torch.Tensor), None, None) inp: (torch.Tensor, torch.Tensor, None, None).

Backward implementation for `torch.add`:

```python
@sten.register_bwd_op_impl(
    operator=torch.add,
    grad_out=(MyCscTensor,),
    grad_inp=(
        (sten.KeepAll, torch.Tensor),
        (sten.KeepAll, torch.Tensor),
        None,
        None,
    ),
    inp=(torch.Tensor, torch.Tensor, None, None),
)
def torch_add_bwd_impl(ctx, grad_outputs, input_sparsifiers):
    [grad_output] = grad_outputs
    dense_output = grad_output.wrapped_tensor.to_dense()
    return dense_output, dense_output, None, None
```
\[
    d = \text{torch.mm}(\text{sparse\_add}(a, b), c) \\
    d\text{.backward}(\text{grad}_d)
\]

Now backward pass is also fully functional.