`xmm` : eXtended Matrix Multiplication

1. Formulation

`xmm` generalizes the matrix multiplication operation to a customizable form:

$$Z = \Omega_{\{\sigma,\mu\}}\left(A_1, ..., A_{N_a}, B_1, ..., B_{N_b}\right)$$

$$Z(i,j) \overset{\text{def}}{=} \underset{k}{\sigma}\left\{\mu\left(A_1(i,k), ..., A_{N_a}(i,k), B_1(k,j), B_{N_b}(k,j)\right)\right\}$$

where operation Ω is parameterized by two functions, μ the **combinator**, and σ the **reductor**.

2. Relations to Neural Networks

2.1. Linear Layer (Matrix Multiplication)

`matmul` is a special case of `xmm` where

- $\sigma = \sum$,
- $\mu(x, y) = xy$,
- $N_a = N_b = 1$.

2.2. Kolmogrov Arnold Layer

KAN-like operations is a special case of `xmm` where

- $\sigma = \sum$,
- $\mu(r, c_1, ..., c_n)$ is a customizable term.
  - Take polynomial KAN with `gridsize = 4` as an example.

$$\mu(r, c_1, ..., c_n) = c_1 + c_2 r + c_3 r^2 + c_4 r^3$$

- $N_a = 1,\ N_b =$ `gridsize`.

3. What Should `xmm` be Able To Do

`xmm` should be able to:

- Provide interface to customize μ. This *could* be done by:
  - parse an expression into μ.
  - other possible ways...

- Further, provide interface to customize σ by parsing a binary function $\mathbb{R}^2 \to \mathbb{R}$ into σ, resembling passing a functor to the `std::reduce` function in C++ STL.
- Just-in-time compile the operator for runtime efficiency.
- **Support pytorch autograd and computations on CUDA**, to provide gradient to **all** input tensors.
- **Provide optimized CUDA operator implementations**.

3.1. Side Notes

- I guess customizing μ is of higher priority than σ since I have not yet encountered scenario where σ should be customized to be other than $\sum$.
- B (column operands) should be transposed to comply with the PyTorch conventions, i.e. the input shapes are `(M, K)` for row operands and `(N, K)` for column operands.

4. Interface

4.1. Class `SumOperator`

- An operator with summation as its **reductor**, i.e. $\sigma = \sum$.

4.1.1. `SumOperator.__init__(self, nr: int, nc: int, expr: str)`

- Create **and parse** the operator, including:
  - Parse `expr`;
  - Compute (symbolic) derivatives;
  - Create kernels sources (codegen, forward and backward);
  - Save kernel identifier.

4.1.2. `SumOperator.compile(self, build_dir: Optional[str] = None, identifier: Optional[str] = None):`

- Compile the operator with `torch.utils.cpp_extensions`.
- Use an identifier (a `str` which has pattern `^[A-Za-z0-9_]+$`) to identify different operators. First If `identifier` is not specified, a `uuid` is generated for each new compilations.

- *Simply use the default parameters to recompile before every run.*

4.1.3. `SumOperator.forward(self, *args)`

- Execute the operation defined by the operator.
- `*args` should be row operands followed by column operands, in correct order.

4.1.4. `SumOperator.backward(self, grad_out: torch.Tensor, *args)`

- Compute derivatives w.r.t. all row operands and column operands.
- `grad_out` should be of shape `(M, N)`.
- `*args` should be row operands followed by column operands, in correct order.

4.1.5. Operator syntax

Take `nr = 1, nc = 3` as an example.

- All identifier you can use is `r1` for the row operand , and `c1, c2, c3` for the three column operands respectively, where `r1 = R1[i, k]`, `c1 = C1[j, k]`.
- Allow a fixed set of simple function, like `exp, sin, tanh, atan2`, etc.
- Constants should only contain its value, **not** followed by `f` or any other suffixes.

A legal expression is like:

```
"c1 * (c2 * c2 * (r1 + c3) * (r1 + c3) - 1.0) * exp(-0.5 * (r1 * r1 * c2 * c2))"
```

5. Modules

5.1. Module `xmm.preprocess`

5.1.1. `preprocess.expr2ast(expr: str)`

5.1.2. `preprocess.ast2sympy(ast_tree)`

5.1.3. `preprocess.sympy2ast(sympy_expr)`

5.1.4. `preprocess.ast2CUDAexpr(ast_rootnode)`

5.2. Module `xmm.codegen`

Generates the CUDA kernels to be compiled.

5.2.1. `codegen.generate_operator_source(nrow: int, ncol: int, expression) -> Tuple[str, str]`

- Returns both C++ wrapper and CUDA source code.

6. Example

We take the following polynomial expression as an example.

$$z(i,j) = \sum_k c_1(k,j) \cdot x(i,k) + c_2(k,j) \cdot x^2(i,k) + c_3(k,j) \cdot x^3(i,k) + b(k,j)$$

where $\underset{k}{\sigma}\{\cdot\} = \sum_k$.

```
from xmm.SumOperator import SumOperator
import torch
import torch.nn as nn

expression = "c1 * r1 + c2 * r1 * r1 + c3 * r1 * r1 * r1 + c4"
op = SumOperator(1, 3, expression)
op.compile(identifier="poly_op") # for identifying build files

class XmmFn(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, scale, bias, weight):
        ctx.save_for_backward(x, scale, bias, weight)
        return op.forward(x, scale, bias, weight)

    @staticmethod
    def backward(ctx, grad_output):
        x, scale, bias, weight = ctx.saved_tensors
        grad_x, grad_scale, grad_bias, grad_weight = op.backward(grad_output, x,
scale, bias, weight)
        return grad_x, grad_scale, grad_bias, grad_weight

class XmmLayer(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.scale = nn.Parameter(torch.ones(out_features, in_features))
```

```
        self.bias = nn.Parameter(torch.zeros(out_features, in_features))
        self.weight = nn.Parameter(torch.empty(out_features, in_features))
        nn.init.kaiming_uniform_(self.weight, a=(5 ** .5))
        self.bn = nn.BatchNorm1d(out_features)

    def forward(self, x):
        x = XmmFn.apply(x, self.scale, self.bias, self.weight)
        return self.bn(x)
```