```
#include <torch/extension.h>

template<typename T>
using accessor_2d = torch::PackedTensorAccessor32<T,2>;

template<typename T>
using accessor_1d = torch::PackedTensorAccessor32<T,1>;

__global__ void linear_function (accessor_2d<float> x,
                                 accessor_2d<float> w,
                                 accessor_2d<float> y,
                                 accessor_1d<float> b)
{
   auto n = blockDim.x * blockIdx.x + threadIdx.x;
   auto m = blockDim.y * blockIdx.y + threadIdx.y;

   float acc = 0;
   if (m < x.size(0) && n < w.size(0)) {
      for (int k = 0; k < x.size(1); k++) {
         acc += x[m][k] * w[n][k];
      }

      y[m][n] = acc + b[n];
   }
}

__global__ void arr_mult_2d (accessor_2d<float> a,
                             accessor_2d<float> b,
                             accessor_2d<float> c)
{
   auto n = blockDim.x * blockIdx.x + threadIdx.x;
   auto m = blockDim.y * blockIdx.y + threadIdx.y;
```

```
    float acc = 0;
    if (m < a.size(0) && n < b.size(1)) {
        for (int k = 0; k < a.size(1); k++) {
            acc += a[m][k] * b[k][n];
        }

        c[m][n] = acc;
    }
}

__global__ void arr_mult_2d_trans (accessor_2d<float> a,
                                   accessor_2d<float> b,
                                   accessor_2d<float> c)
{
    auto n = blockDim.x * blockIdx.x + threadIdx.x;
    auto m = blockDim.y * blockIdx.y + threadIdx.y;

    if (m < a.size(1) && n < b.size(1)) {
        float acc = 0;
        for (int k = 0; k < a.size(0); k++) {
            acc += a[k][m] * b[k][n];
        }

        c[m][n] = acc;
    }
}

__global__ void sum_for_bias (accessor_2d<float> x,
                              accessor_1d<float> y)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
```

```
    int n = x.size(0);
    int m = x.size(1);

    if (i < m) {
        float acc = 0;
        for (int j = 0; j < n; j++) {
            acc += x[j][i];
        }
        y[i] = acc;
    }
}

#define CHECK_CUDA(x) TORCH_CHECK(x.device().is_cuda(), #x " must be a CUDA tensor")
#define CHECK_CONTIGUOUS(x) TORCH_CHECK(x.is_contiguous(), #x " must be contiguous")
#define CHECK_INPUT(x) CHECK_CUDA(x); CHECK_CONTIGUOUS(x)

const int block_size = 16;

__forceinline__ int calc_grid_size(int m) {
    return (m + block_size - 1) / block_size;
}

torch::Tensor forward_linear(torch::Tensor x, torch::Tensor w, torch::Tensor b) {
    // проверки на введенные переменные
    CHECK_INPUT(x);
    CHECK_INPUT(w);
    CHECK_INPUT(b);

    // ввод x(m,k), w(n,k), b(n)
    int n = b.numel();
    int k = w.numel() / n;
```

```
    int m = x.numel() / k;

    // вывод y(m,n)
    auto options = torch::TensorOptions().dtype(torch::kF32).device(torch::kCUDA).requires_grad(true);
    torch::Tensor y = torch::zeros({m, n}, options);

    dim3 dimGrid(calc_grid_size(n), calc_grid_size(m));
    dim3 dimBlock(block_size, block_size);
    linear_function<<<dimGrid, dimBlock>>>(
        x.packed_accessor32<float, 2>(),
        w.packed_accessor32<float, 2>(),
        y.packed_accessor32<float, 2>(),
        b.packed_accessor32<float, 1>()
    );

    return y;
}

std::vector<torch::Tensor> backward_linear(torch::Tensor x, torch::Tensor w, torch::Tensor b,
torch::Tensor y) {
    // Проверка входных тензеров
    CHECK_INPUT(x);
    CHECK_INPUT(w);
    CHECK_INPUT(b);
    CHECK_INPUT(y);

    // Инициализируем переменные
    auto y_pa = y.packed_accessor32<float, 2>();
    auto x_pa = x.packed_accessor32<float, 2>();
    auto w_pa = w.packed_accessor32<float, 2>();
    auto b_pa = b.packed_accessor32<float, 1>();

    // int n = b.numel(); // что-то другое
```

```
// int k = w.numel() / n; // что-то другое
// int m = x.numel() / k; // что-то другое

int m = x_pa.size(0);
int n = y_pa.size(1);
int k = x_pa.size(1);

// вывод y(m,n)
auto options = torch::TensorOptions().dtype(torch::kF32).device(torch::kCUDA).requires_grad(true);
torch::Tensor grad_input = torch::zeros({m, k}, options);
torch::Tensor grad_weight = torch::zeros({n, k}, options);
torch::Tensor grad_bias = torch::zeros({n, }, options);

dim3 dimGrid(calc_grid_size(k), calc_grid_size(m));
dim3 dimBlock(block_size, block_size);
arr_mult_2d<<<dimGrid, dimBlock>>>(
   y_pa,
   w_pa,
   grad_input.packed_accessor32<float, 2>()
);

dim3 dimGrid2(calc_grid_size(k), calc_grid_size(n));
arr_mult_2d_trans<<<dimGrid2, dimBlock>>>(
   y_pa,
   x_pa,
   grad_weight.packed_accessor32<float, 2>()
);

sum_for_bias<<<calc_grid_size(n), block_size>>>(
   y_pa,
   grad_bias.packed_accessor32<float, 1>()
);
```

```
    return std::vector<torch::Tensor>{grad_input, grad_weight, grad_bias};
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("my_forward_linear", &forward_linear, "Custom function forward linear layer");
    m.def("my_backward_linear", &backward_linear, "Custom function backward linear layer");
}
```

Main.py

```python
import unittest
import torch.nn
import torch.random
import numpy as np
import math
from torch.utils.cpp_extension import load

ext = load(
        name='linearlayer',
        sources=['linearlayer.cu'],
        extra_cuda_cflags=['-O4'],
        extra_cflags=['-O4'],
    )

class CudaLinearLayer(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weight, bias):
        ctx.save_for_backward(input, weight, bias)
        return ext.my_forward_linear(input, weight, bias)

    @staticmethod
    def backward(ctx, grad_output):
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        grad_input, grad_weight, grad_bias = ext.my_backward_linear(input,
weight, bias, grad_output)

        return grad_input, grad_weight, grad_bias


class PythonLinearLayer(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weight, bias):
        ctx.save_for_backward(input, weight, bias)
        return input @ weight.t() + bias

    @staticmethod
    def backward(ctx, grad_output):
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None
```

```python
        if ctx.needs_input_grad[0]:
            grad_input = grad_output @ weight
        if ctx.needs_input_grad[1]:
            grad_weight = torch.mm(grad_output.t(), input)
        if ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias


class LabTest(unittest.TestCase):
    @staticmethod
    def reset_parameters(w , b):
        torch.nn.init.kaiming_uniform_(w , a=math.sqrt(5))
        fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(w)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        torch.nn.init.uniform_(b, -bound, bound )

    def test_linear_layer(self):
        # Характеристики тензоров
        tensor_opt = {
            'device': 'cuda',
            'dtype': torch.float32,
            'requires_grad': True
        }

        # Переменные для своей реализации
        x = torch.ones((128, 9216), **tensor_opt)
        w1 = torch.empty((4096, 9216), **tensor_opt)
        b1 = torch.empty((4096, ), **tensor_opt)
        w2 = torch.empty((16, 4096), **tensor_opt)
        b2 = torch.empty((16, ), **tensor_opt)

        # Инициализация переменных
        LabTest.reset_parameters(w1, b1)
        LabTest.reset_parameters(w2, b2)

        # Реализация своей линейной свертки
        y = CudaLinearLayer.apply(x, w1, b1)
        z = CudaLinearLayer.apply(y, w2, b2)

        # Переменные для фреймворка
        x_ = x.detach().clone().requires_grad_()
        w1_ = w1.detach().clone().requires_grad_()
        b1_ = b1.detach().clone().requires_grad_()
        w2_ = w2.detach().clone().requires_grad_()
        b2_ = b2.detach().clone().requires_grad_()

        # Реализация линейной свертки фреймворка
        y_ = PythonLinearLayer.apply(x_, w1_, b1_)
        z_ = PythonLinearLayer.apply(y_, w2_, b2_)

        # Проверка результатов своей и фреймфорка
        self.assertTrue(torch.allclose(z, z_, atol=1e-4, rtol=1e-3))

        # Используем градиент
        z.backward(torch.ones_like(z))
        z_.backward(torch.ones_like(z_))

        # print(b1.grad, '\n', b1_.grad)

        # Проверка значений
        with torch.no_grad():
            self.assertTrue(
```

```python
                torch.allclose(x.grad, x_.grad, atol=1e-4, rtol=1e-3)
            )
            self.assertTrue(
                torch.allclose(w1.grad, w1_.grad, atol=1e-4, rtol=1e-3)
            )
            self.assertTrue(
                torch.allclose(w2.grad, w2_.grad, atol=1e-4, rtol=1e-3)
            )
            self.assertTrue(
                torch.allclose(b1.grad, b1_.grad, atol=1e-4, rtol=1e-3)
            )
            self.assertTrue(
                torch.allclose(b2.grad, b2_.grad, atol=1e-4, rtol=1e-3)
            )


if __name__ == '__main__':
    unittest.main()
```