
✅ **Example 1 — Custom ReLU (C++ + CUDA)**

Step 1 — Write the CUDA kernel (relu_kernel.cu)

```
#include <cuda.h>
#include <cuda_runtime.h>

template <typename scalar_t>
__global__ void relu_forward_kernel(const scalar_t* __restrict__ input,
    scalar_t* __restrict__ output,     size_t numel) {

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < numel) {
      float x = input[idx];
      output[idx] = x > 0.0f ? x : 0.0f;
  }
}

extern "C" void launch_relu_forward(const float* input, float* output,
size_t numel, cudaStream_t stream) {
   int threads = 256;
   int blocks = (numel + threads - 1) / threads;
   relu_forward_kernel<<<blocks, threads, 0, stream>>>(input, output, numel);
}
```

Step 2 — Create a C++ wrapper (relu.cpp)

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

void launch_relu_forward(const float* input, float* output,
    size_t numel, cudaStream_t stream);

torch::Tensor relu_forward(torch::Tensor input) {
    auto output = torch::empty_like(input);
    launch_relu_forward( input.data_ptr<float>(), output.data_ptr<float>(),
input.numel(),
   at::cuda::getCurrentCUDAStream() );
   return output;
```

```
}

PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("relu_forward", &relu_forward, "Custom ReLU forward");
}
```

Step 3 — Python binding & build (build.py)

```
from torch.utils.cpp_extension import load

relu_ext = load(  name="relu_ext",  sources=["relu.cpp", "relu_kernel.cu"],  verbose=True)
```

✅ **Step 4 — Use it in PyTorch**

```
import torch
import relu_ext

x = torch.randn(10, device="cuda")
y = relu_ext.relu_forward(x)
print(y)
```

✅ **1. ReLU in Triton (Simple Example)**

✅ Triton Kernel: relu_kernel

```
import triton
import triton.language as tl
@triton.jitdef
relu_kernel(x_ptr, y_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)            # unique program ID
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)    y
    mask = offsets < n_elements  # avoid out-of-bounds
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.maximum(x, 0.0)             # ReLU
    tl.store(y_ptr + offsets, y, mask=mask)
```

✅ Python wrapper

```
 import torch
def relu_triton(x):
```

```
    y = torch.empty_like(x)
    BLOCK = 1024
    grid = lambda meta: ( triton.cdiv(x.numel(), BLOCK)), relu_kernelgrid,
BLOCK_SIZE=BLOCK )
    return y
```

✅ **Usage**

```
x = torch.randn(4096, device="cuda")
y = relu_triton(x)
print(y[:10])
```

✅ This ReLU is parallelized, runs entirely on GPU, and behaves like CUDA but in Python.