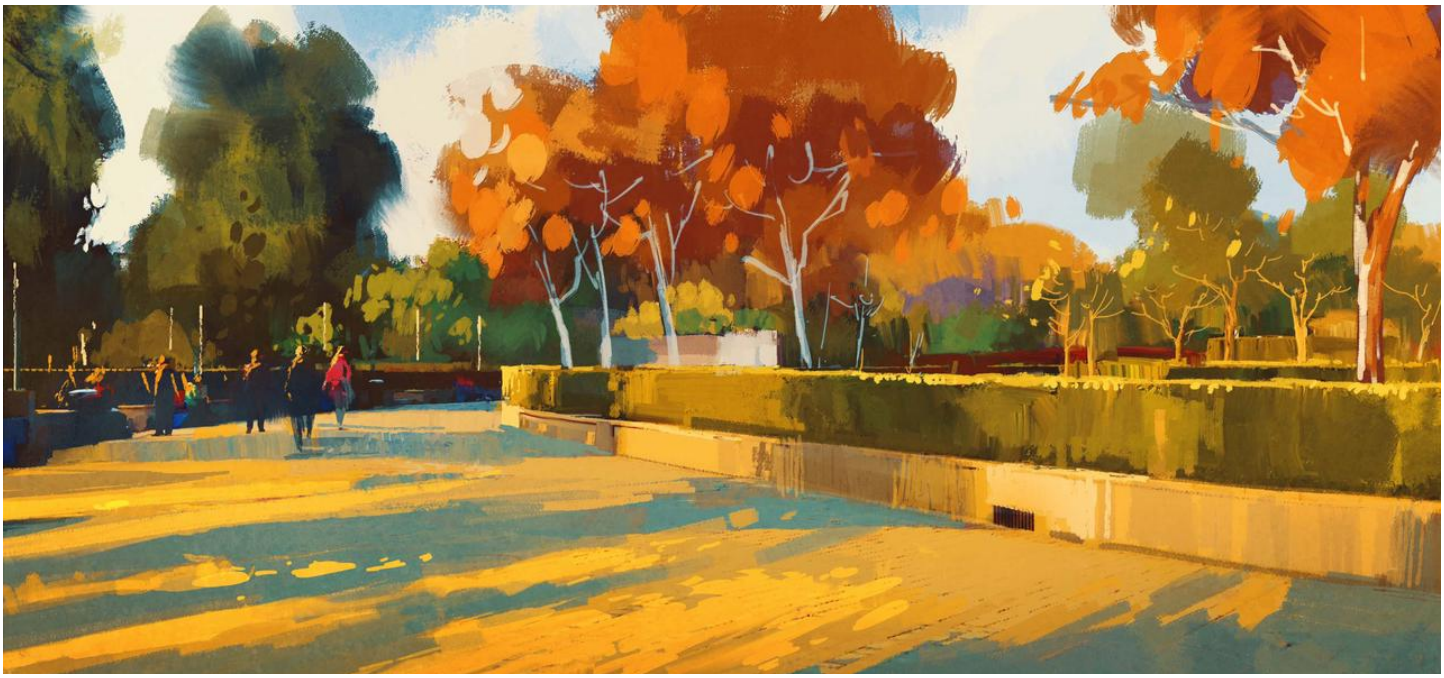


Llama模型中的MLP层实现

视频

上.mp4

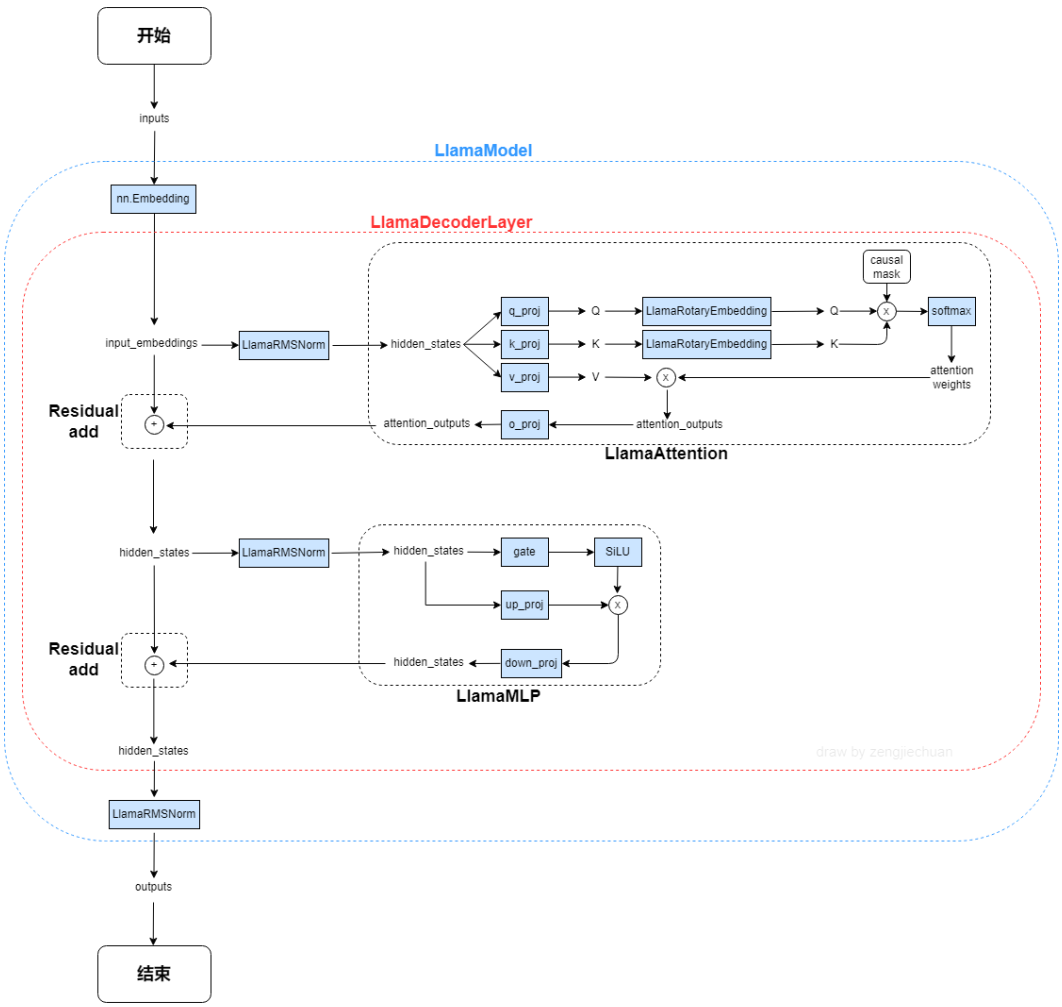
下.mp4

MLP模块的原理

MLP层又叫FeedForward（前馈神经网络）层是深度学习框架中的一个核心组件，其主要职责是对上一层传递来的特征进行非线性变换。一个典型的FFN包含两个线性变换层，这两个层之间通过一个激活函数（如ReLU或GELU）相连。在LLama的结构图中（见下图），这一部分对应于所谓的MLP层。在下图中，**蓝色矩阵代表的是算子**，例如RMSNorm、softmax等。

在MLP结构中，共有三个矩阵乘法操作，分别命名为gate、up_proj和down_proj。我们将hidden_states的维度记作seq×dim，其中seq代表输入序列中单词的数量，dim则是输入序列单词映射后的向量维度。**up_proj和gate这两个操作都是矩阵乘法，它们将hidden_states的维度映射到更高的维度，我们将维度记作hidden_dim。**也就是说，**经过映射后hidden_states的维度变为seq×hidden_dim，其中hidden_dim的值大于dim。**

seq×dim 经过gate和up_proj之后，它的维度是seq×hidden_dim

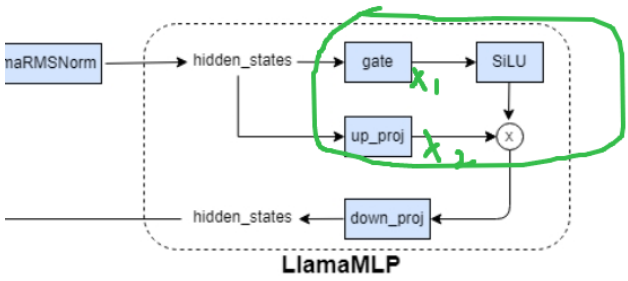


第一、二层将输入向量升维到一个更高的维度，第三层则将其降维回原始空间大小，另外激活函数引入了非线性复杂性，使得模型能够学习更复杂的输入数据之间的关系（在图中就是~~SiLU激活函数~~，此处多个模块组合就是Swiglu激活函数，它为MLP模块引入了非线性）。

hidden states = seq × dim

1. 使用Gate矩阵对MLP的输入向上映射之后，得到x1 = seq × hidden_dim，x2 = seq × hidden_dim，这两个输入再做一个激活，得到actout；

2. 随后再对 down_proj对上步的激活值输出actout做一个向下的映射，维度回到seq×dim，得到mlp模块的输出值。



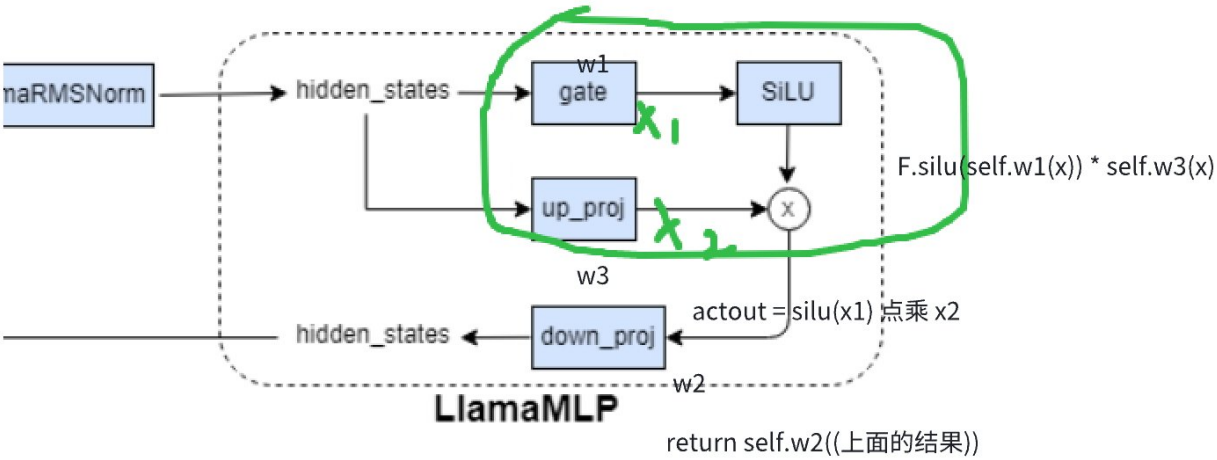
换句话说，如果没有非线性激活函数，整个网络将由一系列线性变换组成，无论多少层叠加，整个网络仍然只能表示线性关系。

Swiglu的计算方式如下：

$$function\ swiglu(x_1, x_2)$$
$$act、out\ =\ silu(x_1) \otimes x_2\ =\ (x_1\ \cdot\ sigmoid(x_1)) \otimes x_2$$

对于MLP层整体，有：

$$x_1\ =\ w_1 \cdot input\ embedding$$
$$x_2\ =\ w_3 \cdot input\ embedding$$
$$actout\ =\ swiglu(x_1, x_2)$$



定义了三个矩阵乘法算子，分别是w1\w3\w2，w2负责将维度压缩回seq×dim

首先我们准备了三个权重矩阵用于去线性变换输入的特征，这三个矩阵分别是w1、w2和w3，**权重矩阵w1的输入维度与输入特征维度相匹配**，而其输出维度则设为hidden dim，hidden_dim > dim。随后的第二个权重矩阵是w2，它的输入维度是hidden dim，输出维度是dim，**主要功能在于将先前提升至高维空间的特征值重新映射回原始空间大小，实现降维处理。**

$$out = w2 \cdot actout$$

Pytorch中的实现

以下代码节选自Meta公司LLama模型的FeedForward层实现。我们可以观察到，其中设定了 `hidden_dim = 1.5 × dim` 。针对输入 `input` ，首先通过两个矩阵乘法操作进行处理：一是线性层 `w1` 与输入 `input` 相乘，二是线性层 `w3` 同样与输入 `input` 相乘。

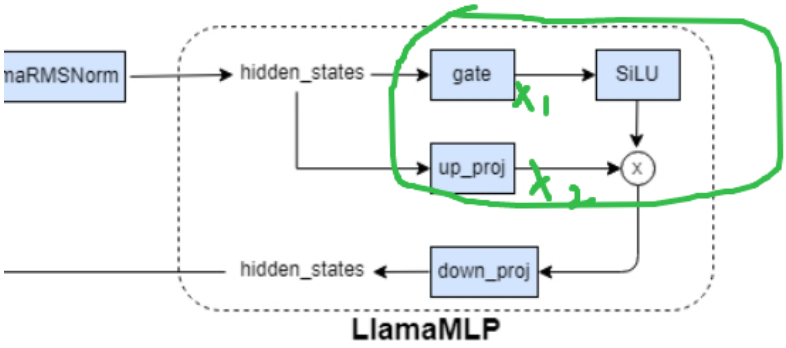
这两个线性层的输出维度均被设置为 `hidden_dim` 。接着，这两个矩阵乘法的结果被用作Swiglu函数的输入 `x1` 和 `x2` ，进而得到经过非线性激活处理的输出。随后，利用线性层 `w3` 对这个输出进行降维处理，将其维度降至 `dim` 并返回结果。

```
class FeedForward(nn.Module):
    def __init__(
        self,
        dim: int,
        hidden_dim: int,
        multiple_of: int,
        ffn_dim_multiplier: Optional[float],
    ):
        super().__init__()
        hidden_dim = int(2 * hidden_dim / 3)
        # custom dim factor multiplier
        if ffn_dim_multiplier is not None:
            hidden_dim = int(ffn_dim_multiplier * hidden_dim)
        hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)

        self.w1 = ColumnParallelLinear(
            dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
        ) # w1 和 w3是向上映射，也就是将seq×dim 映射为 seq×hidden dim
        self.w2 = RowParallelLinear(
            hidden_dim, dim, bias=False, input_is_parallel=True, init_method=lambda x: x
        )
        self.w3 = ColumnParallelLinear(
            dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
        )

```

```
26      def forward(self, x):
27
28          return self.w2(F.silu(self.w1(x)) * self.w3(x))
```

MLP层的总体流程实现

在先前的课程中，我们已经详细讲解了`matmul`和`rmsnorm`层的实现，包括CUDA和CPU后端两种的实现。因此，在今天的课程中，我们将不再赘述MLP中的线性层。我们的重点是分析整个模块的实现，以及其中Swiglu算子的具体实现。



从图中可以观察到，输入的`input embedding`首先会通过`rmsnorm`层进行归一化处理，这一点在代码的第1-5行得到了体现。这里的`ffn_rmsnorm`中的权重，也是按照相同的方法从权重模型文件中获取的（通过`mmap`打开，并移动到相应的位置以读取特定数量的权重），经过rmsnorm计算过的输出我们记作变量`ffn_norm_output`.

```
1
2     // ffn rmsnorm的输出为ffn_norm_output
3     tensor::Tensor ffn_norm_output = get_buffer(ModelBufferType::kFFNRMSNorm);
4     const auto& ffn_rmsnorm = llama_layers_->rmsnorm_layers_.at(layer_idx +
    config_->layer_num_);
5     ffn_rmsnorm->forward(input, ffn_norm_output);
6
7     // w1_layer.weight @ ffn_norm_output 结果存放于w1_output, 也就是x1
8     tensor::Tensor w1_output = get_buffer(ModelBufferType::kW1Output);
9     const auto& w1_layer = llama_layers_->w1_layers_.at(layer_idx);
10    w1_layer->forward(ffn_norm_output, w1_output);
11
12    // w3_layer.weight @ ffn_norm_output 结果存放于w3_output, 也就是x2
13    tensor::Tensor w3_output = get_buffer(ModelBufferType::kW3Output);
14    const auto& w3_layer = llama_layers_->w3_layers_.at(layer_idx);
15    w3_layer->forward(ffn_norm_output, w3_output);
16
```

```
17    // SwiGLU SWiglu(x1,x2) = SiLU(x1) 点乘 x2
18    llama_layers_->swiglu_layer_->forward(w1_output, w3_output, w1_output);
19
20    // w2_layer.weight @ w1_output 得到最终的结果。
21    tensor::Tensor w2_output = get_buffer(ModelBufferType::kW2Output);
22    const auto& w2_layer = llama_layers_->w2_layers_.at(layer_idx);
23    w2_layer->forward(w1_output, w2_output);
24
25    // residule add input = input+w2_output,w2_output是mlp整个的最终输出。
26    llama_layers_->add_layer_->forward(input, w2_output, input)
```

在代码的第8-9行，`w1`线性层会对`rmsnorm`层的输出进行映射，从而得到`w1_output`。接着，在第13-15行，`w3`线性层同样对`rmsnorm`层的输出进行映射，得到`w3_output`。正如之前所述，`w1_output`和`w3_output`的维度均为`hidden_dim`。

接下来，这两个张量将被送入Swiglu算子中进行运算，我们会稍后讨论Swiglu算子的具体实现，并将运算结果再次保存到`w1_output`中。然后，将`w1_output`通过`w2`线性层进行映射，得到输出`w2_output`。

紧接着，`w2_output`将与MLP模块的输入`input`相加（即图中的Residual add，残差连接）以得到本模块的最终输出，并将该输出保存回`input`张量中（此处实现了空间复用）。

CPU Swiglu算子的实现

接下来，让我们探讨一下`llama_layers_->swiglu_layer_->forward(w1_output, w3_output, w1_output);`这句代码中调用到的Swiglu算子如何在CPU上实现。

Swiglu的计算方式如下：

$$function\ swiglu(x_1, x_2)$$
$$out\ =\ silu(x_1) \otimes x_2\ =\ (x_1 \cdot sigmoid(x_1)) \otimes x_2$$

```
1   void swiglu_kernel_cpu(const tensor::Tensor& input1, const tensor::Tensor&
    input2,
2                          const tensor::Tensor& output, void* stream) {
3     // ...
4     // ...
5
6     arma::fvec input1_vec(const_cast<float*>(input1.ptr<float>()),
    input1.size(), false,
7                           true);
8     arma::fvec input2_vec(const_cast<float*>(input2.ptr<float>()),
    input2.size(), false,
9                           true);
```

```
10    arma::fvec output_vec(const_cast<float*>(output.ptr<float>()),
    output.size(), false,
11                          true);
12
13    input1_vec %= (1.0f / (1.0f + arma::exp(-input1_vec)));
14    output_vec = input1_vec % input2_vec;
15  }
```

首先是用两个输入input1和input2去构造armadillo数学库中的input1_vec和input2_vec向量，最后的两个布尔变量表示直接复用Tensor中的内存。在第13行中：

```
1   input1_vec %= (1.0f / (1.0f + arma::exp(-input1_vec))); // sigmoid(x)
2   // x 点乘 sigmoid(x) x = x * sigmoid(x)
3
4   // % 逐点相乘,并不是取余数的意思
5   // * 已经被矩阵乘占用了
```

我们来拆解这里的调用，(1.0f / (1.0f + arma::exp(-input1_vec)))就是在计算公式：

$$\frac{1}{1+exp(-input\_vec1)}$$

其中的百分号%表示逐点相乘，也就是说整体公式为：

$$\frac{input\_vec1}{1+exp(-input\_vec1)}$$

当前行计算的就是Swiglu公式的左边部分，随后我们再将这里的结果再次点乘input2_vec，得到最终的输出并返回。

$$out = input\_vec1 \otimes input\_vec2$$

这里的%其实是一种符号重载，实际上的实现是将输入的input1_vec和input2_vec进行遍历，并将两个数组中的每个元素进行逐点相乘，计算方式类似于：

```
1   #define arma_applier_1a(operatorA, operatorB) \
2     {\
3     for(uword i=0; i<n_elem; ++i)\
4       {\
5       out_mem[i] operatorA P1.at_alt(i) operatorB P2.at_alt(i);\
6       }\
7     }
8
9   x = x * sigmoid(x)
```

```
out_mem表示 x变量
P1表示x
P2表示sigmoid(x)

// 第一步
for i in range(0,n_elem):
    x[i] = P1[i] ×  P2[i] = x[i] × sigmoid(x)

// 第二步
output_vec = input1_vec % input2_vec;
for i in range(0,n_elem):
    output_vec[i] = P1[i] × P2[i] = input1_vec[i] × input2_vec[i];
```

此处，operatorB会被设置为×，在这段代码中，它将对数据进行遍历，并使用指定的运算符进行计算，而operatorA会被替换为=，用于将P1和P2逐点相乘的结果，再与output_vec中的相应值相乘。

CUDA Swiglu算子的实现

在Cuda的实现中，我们配置了M个Block，每个Block中配置了N个线程用于该非线性激活函数的计算，我们这里将M×N的大小设置为M×N大于输入向量的元素个数。

```
__global__ void swiglu_kernel_cu_fp32(int size, const float* in1, const float* in2, float* out) {
  int tid = threadIdx.x;
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  if (idx >= size) {
    return;
  }
  extern __shared__ float shared_mem[];
  float* smem1 = shared_mem;
  float* smem2 = shared_mem + blockDim.x;
  // block id = 1, idx 1 .. 8
  // tid 等于0或者 1
  // smem1[0] = 1 smem1[1] = 2
  // 当block id = 1的时候，smem = {1,2}
  // 当block id = 1的时候，smem = {3,4}
  smem1[tid] = in1[idx];
  smem2[tid] = in2[idx];
  __syncthreads();
  // 比如block等于1的时候，thread等于0或者1，smem1等于1或2，求出sigmoid
  // x * sigmoid = silu

  // 比如block等于2的时候，thread等于0或者1，smem1等于3或4，求出sigmoid
  // x * sigmoid = silu
  float value = 1.0f / (1.0f + exp(-smem1[tid])); // 共享内存当中取数据
```

```
    smem1[tid] = smem1[tid] * value;

    out[idx] = smem1[tid] * smem2[tid];
}

void swiglu_kernel_cu(const tensor::Tensor& input1, const tensor::Tensor&
input2,
                      const tensor::Tensor& output, void* stream) {

    int size = static_cast<int32_t>(input1.size());
    int threads = 128;
    int blocks = (size + threads - 1) / threads;
    const size_t shmem = threads * sizeof(float) * 2;
    if (!stream) {
     // 取启动核函数
      swiglu_kernel_cu_fp32<<<blocks, threads, shmem>>>(
          size, input1.ptr<float>(), input2.ptr<float>(), const_cast<float*>
(output.ptr<float>()));
    } else {
      cudaStream_t stream_ = static_cast<cudaStream_t>(stream);
      swiglu_kernel_cu_fp32<<<blocks, threads, shmem, stream_>>>(
          size, input1.ptr<float>(), input2.ptr<float>(), const_cast<float*>
(output.ptr<float>()));
    }
}
```

首先，我们准备了两个共享内存数组，用于将存放在全局内存中的数据预先加载到共享内存中。以以下存放在全局内存的数据为例：[1,2,3,4,5,⋯,7,8]，总共有8个数字。目前我们有4个block，每个block都拥有自己独立的共享内存区域。 block1: [1,2] block2: [3,4] block3: [5,6] ⋯ block4: [7,8]

假设我们当前处理的block id为1，smem1 = [1,2]，即正在处理第一个block。在此过程中，我们首先将全局内存中的数据1和2加载到该block的共享内存数组中。然后，我们将从共享内存数组中取出这些数据。在这里，我们使用__syncthreads()函数，其目的是为了同步同一个block内的所有线程，确保同一个block中的所有线程都执行到了这一步。完成同步后，我们便可以开始使用共享内存中的数据。

```
float value = 1.0f / (1.0f + exp(-smem1[tid]));
smem1[tid] = smem1[tid] * value;

out[idx] = smem1[tid] * smem2[tid];
```

首先，我们从input_vec1中取出元素，这些元素实际上是存储在共享内存smem1中的。接着，我们再从input_vec2中取出元素，这些元素则存储在另一个共享内存smem2中。完成元素的提取后，我们按

照既定的公式进行计算。

算子的分别注册和调用

```
SwigluKernel get_swiglu_kernel(base::DeviceType device_type, void* stream) {
  if (device_type == base::DeviceType::kDeviceCPU) {
    return swiglu_kernel_cpu;
  } else if (device_type == base::DeviceType::kDeviceCUDA) {
    return swiglu_kernel_cu;
  } else {
    LOG(FATAL) << "Unknown device type for get a swiglu kernel.";
    return nullptr;
  }
}
```

我们在swiglu函数中完成对swiglu算子的注册，如果用户传入的参数device_type为CPU类型的时候，我们就返回swiglu_kernel_cpu，如果device_type是GPU类型的时候，我们就返回swiglu_kernel_cu。随后我们再创建一个SwiGLULayer类对该过程完成调用封装：

```
base::Status SwiGLULayer::forward() {
  auto status = check();
  if (!status) {
    return status;
  }
  auto input1 = this->get_input(0);
  auto input2 = this->get_input(1);
  auto output = this->get_output(0);
  if (device_type_ == base::DeviceType::kDeviceCUDA) {
    CHECK(cuda_config_ != nullptr);
  }
  kernel::get_swiglu_kernel(device_type_)(input1, input2, output,
                                          cuda_config_ ? cuda_config_->stream
: nullptr);
  return base::error::Success();
}
```

当我们在模型的推理过程中调用llama_layers_->swiglu_layer_->forward(w1_output, w3_output, w1_output)时流程就会到SwiGLULayer::forward，因为swiglu_layer_就是一个SwiGLULayer的实例，在这里我们再对参数的个数和类型进行校验，如果没有问题后再调用get_swiglu_kernel获取对应设备类型的算子实现并进行调用。

单元测试

```
TEST(test_swiglu_cu, swiglu_nostream) {
  auto alloc_cu = base::CUDADeviceAllocatorFactory::get_instance();
  auto alloc_cpu = base::CPUDeviceAllocatorFactory::get_instance();

  int32_t size = 32 * 151;

  tensor::Tensor in_cpu(base::DataType::kDataTypeFp32, size, true, alloc_cpu);
  tensor::Tensor wei_cpu(base::DataType::kDataTypeFp32, size, true,
alloc_cpu);
  tensor::Tensor out_cpu(base::DataType::kDataTypeFp32, size, true,
alloc_cpu);

  std::random_device rd;
  std::mt19937 mt(rd());
  std::uniform_real_distribution<float> dist(0.f, 1.f);
  for (int i = 0; i < size; ++i) {
    in_cpu.index<float>(i) = dist(mt);
    wei_cpu.index<float>(i) = dist(mt);
  }

  tensor::Tensor in_cu = in_cpu.clone();
  tensor::Tensor wei_cu = wei_cpu.clone();
  tensor::Tensor out_cu = out_cpu.clone();
  in_cu.to_cuda(nullptr);
  wei_cu.to_cuda(nullptr);
  out_cu.to_cuda(nullptr);

  kernel::get_swiglu_kernel(base::DeviceType::kDeviceCUDA)(in_cu, wei_cu,
out_cu,
                                                           nullptr);
  out_cu.to_cpu();

  kernel::get_swiglu_kernel(base::DeviceType::kDeviceCPU)(in_cpu, wei_cpu,
out_cpu,
                                                          nullptr);

  for (int i = 0; i < size; ++i) {
    ASSERT_NEAR(out_cu.index<float>(i), out_cpu.index<float>(i), 1e-5f);
  }
}
```

我们在单元测试中准备了同一份数据并存放在张量中，根据需要张量分别存放在内存和显存里，我们调用了CPU和CUDA上的Swiglu函数实现再比较它们的结果。