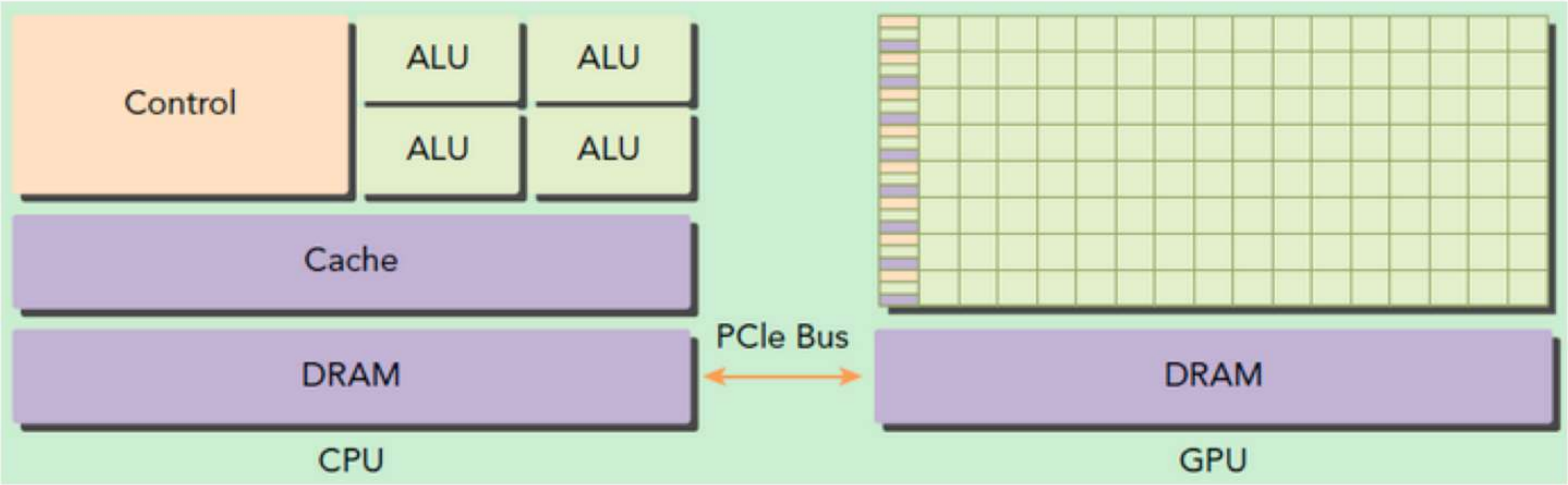
PyTorch-C++/CUDA编程

GPU-CUDA编程

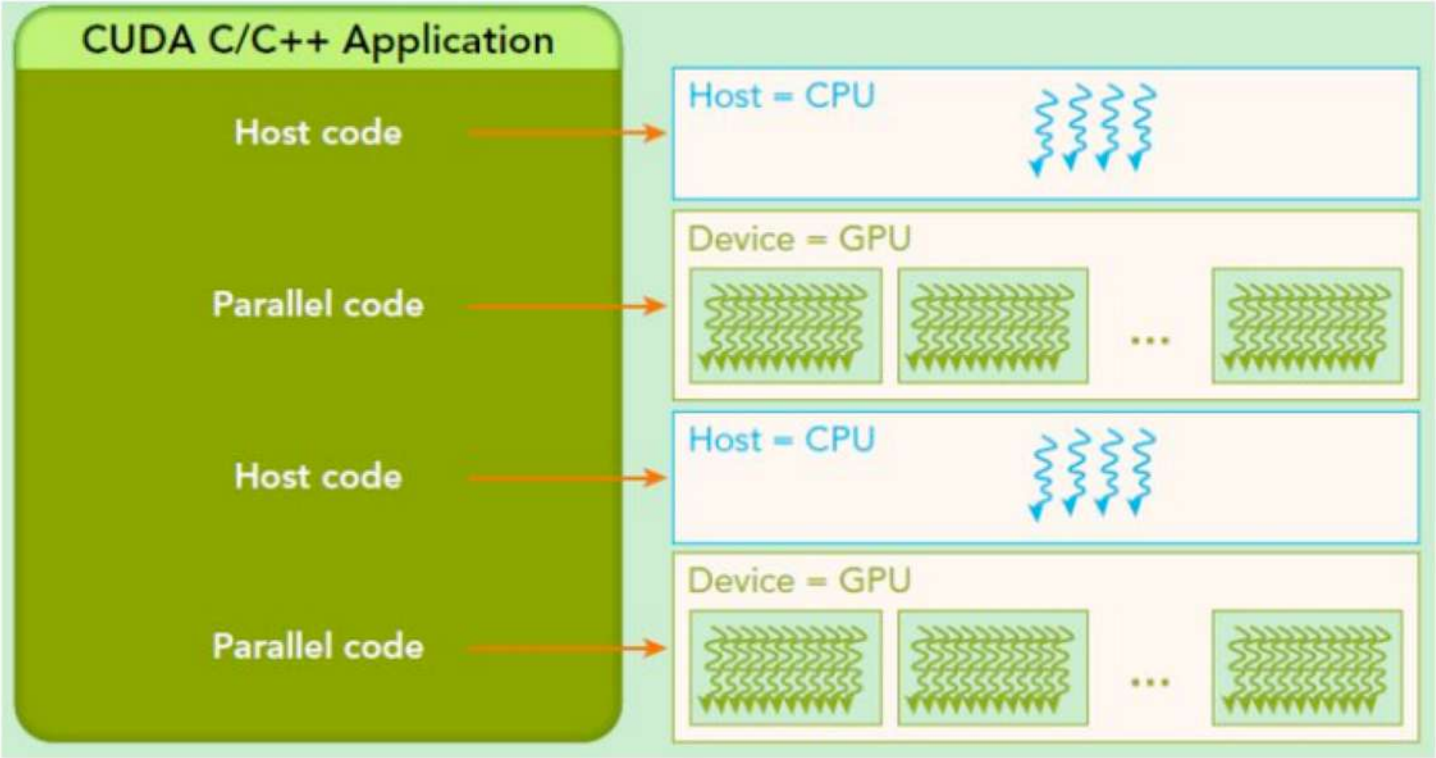
CUDA概览 | CPU / GPU

- GPU的计算单元远多于CPU，所以适合计算密集型任务
- 通过PCIe总线进行数据交换
- CPU所在位置称为主机端（host），GPU所在位置称为设备端（device）



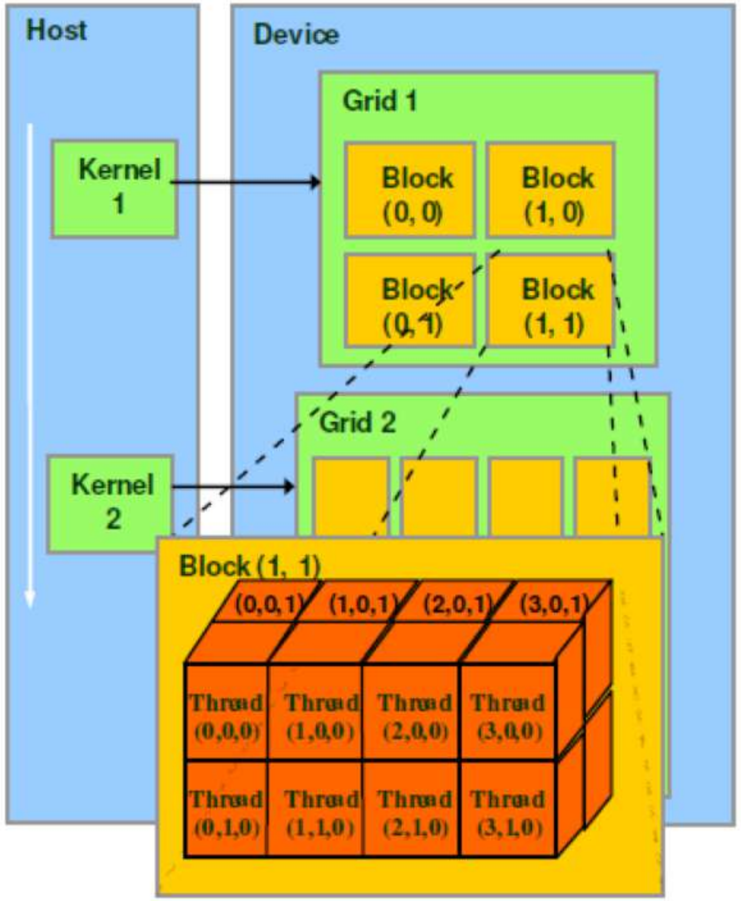
CUDA概览 | 执行逻辑

- CUDA的主程序是在CPU中运行的
- 其中可并行化的代码放到GPU中运行，实现CPU和GPU的交替运行



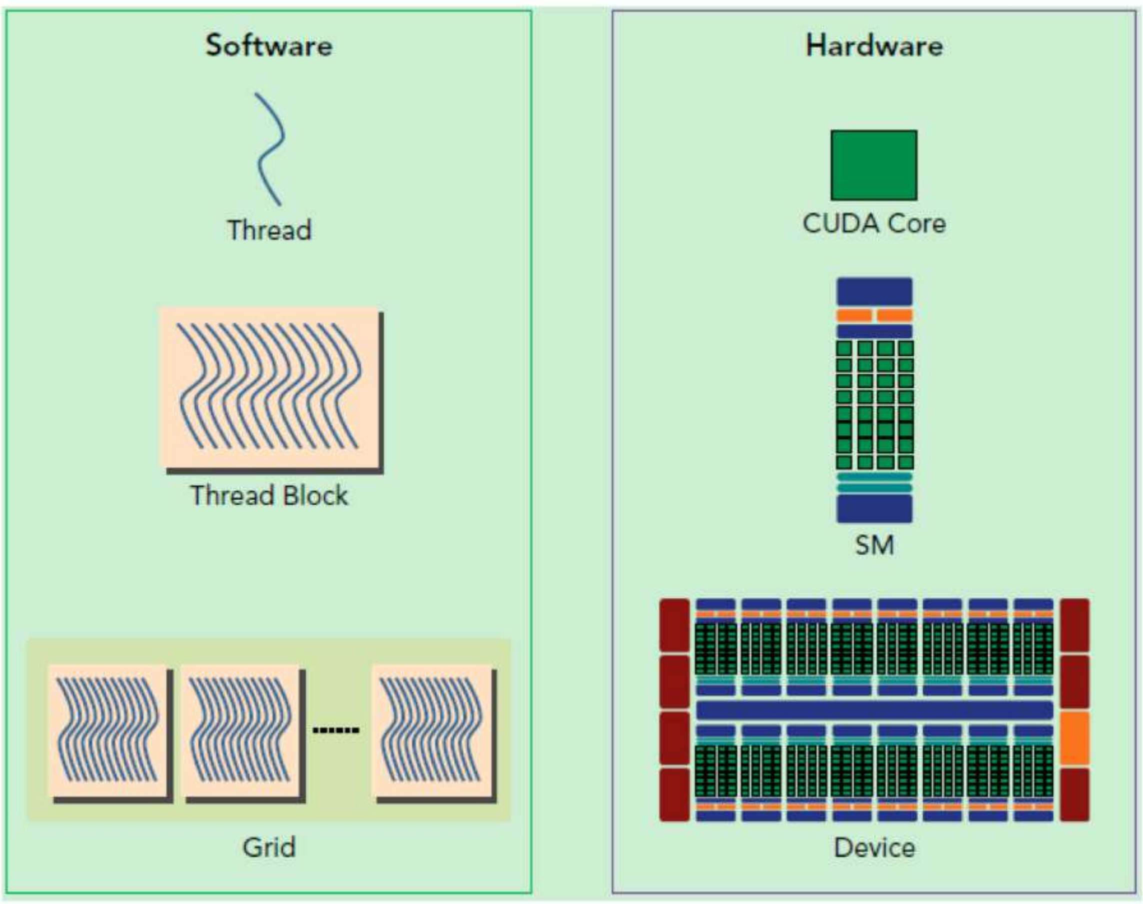
CUDA线程 | 线程层次结构

- CPU通过调用CUDA的kernel函数来在GPU上执行并行计算
- Kernel的3层结构
  - Grid -> Block -> (warp) -> Thread
- Grid(3-dim)：kernel启动的所有线程
- Block(3-dim)：线程块。线程数为32的倍数，至多含1024个线程
- (warp): 线程束，SM的基本执行单元。从block中划分，含32个线程
- Thread: GPU的1个轻量级线程



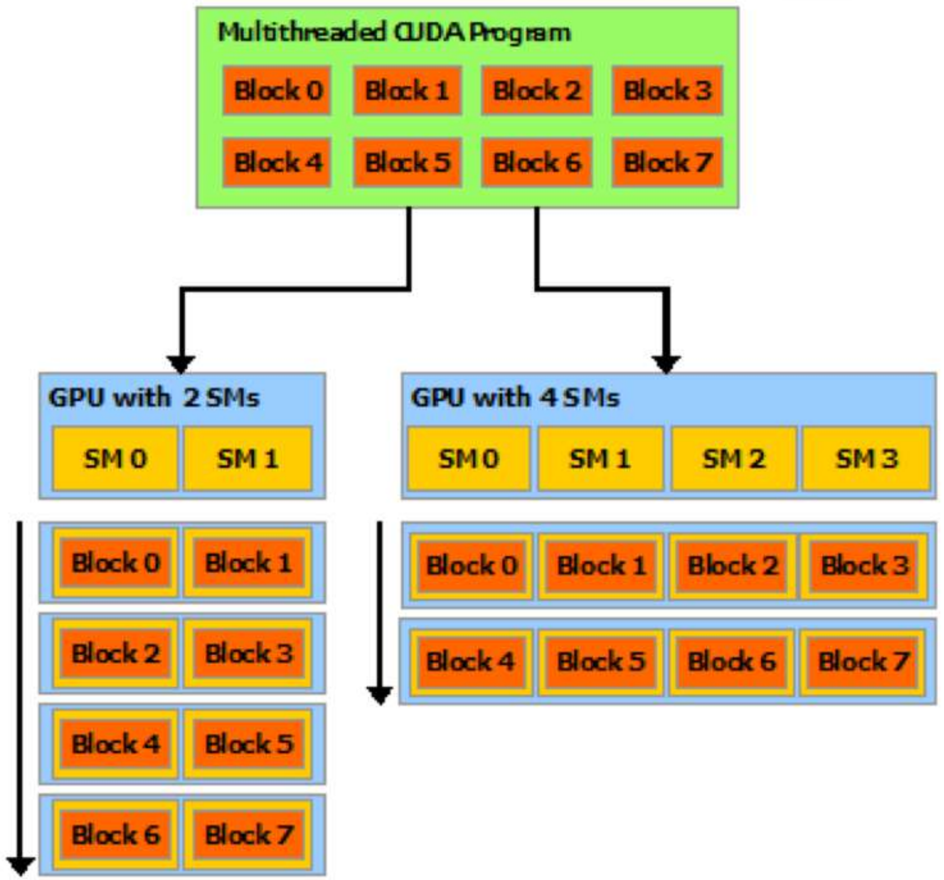
CUDA线程 | GPU物理组成

- GPU主要组成单元：SM
- SM（streaming multiprocessor）流式多处理器
  - SP (streaming processor) ，又称 CUDA Core，最基本处理单元
  - Shared Memory
  - Register File
  - warp Scheduler
- GPU可并发执行数百个线程



CUDA线程 | 逻辑与物理

- 将逻辑上的线程模型对应到物理模型
- 分配时，1个SM可以包含多个block，1个block只可以属于1个SM
- 运行时，1个SM被1个warp占用，多个warps轮流进入SM，由warp scheduler负责调度，以SIMT执行
- 1个SP执行1个thread
- 所以，GPU上resident threads最多只有 SM*warp个

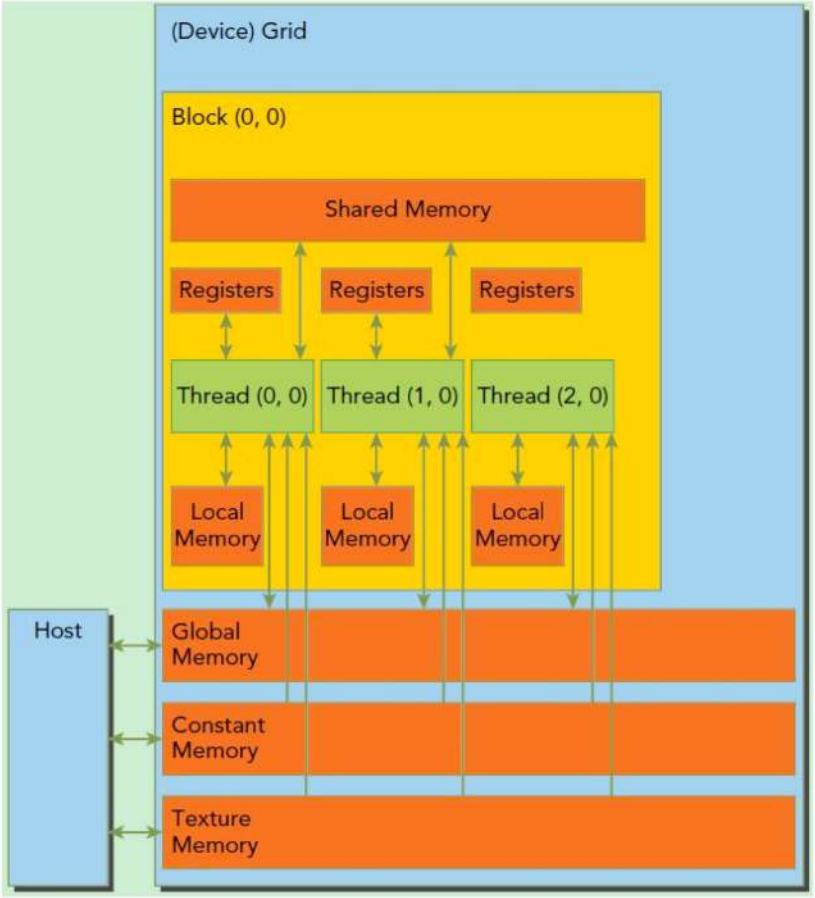


CUDA线程 | GPU架构概览

- 架构演进
  - Tesla -> Fermi -> Kepler -> Maxwell -> Pascal -> Volta -> Turing
- Titan XP的架构Pascal
  - 30个SM
  - 每个SM
    - 2个Warp Scheduler，4个Dispatch Unit
    - 64个CUDA Core（2 * 32）
    - 32个DP Unit（2 * 16）
    - 16个SFU和LD/ST Unit（2 * 8）

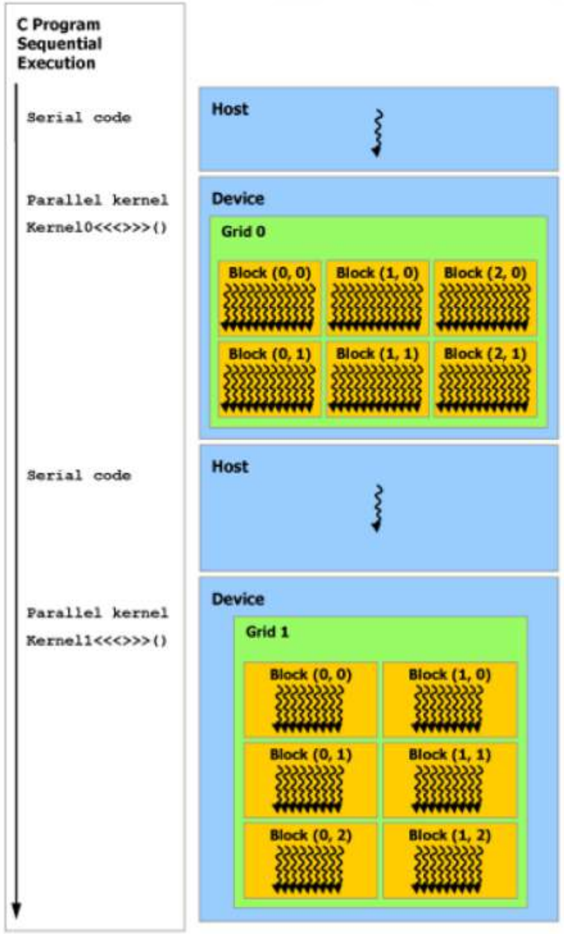
CUDA编程 | 内存模型

- Global Memory全局内存
  - cudaMemcpy函数发生位置
- Constant Memory常量内存
- Texture Memory纹理内存
- Shared Memory共享内存
  - 每个线程块独有
  - 利用它可以实现程序优化，减少片外IO交互
- Local Memory / Registers



CUDA编程 | 执行流程

- 两个概念：host和device
  - host指代CPU及其内存
  - device指代GPU及其内存
- 执行流程
  - 分配host内存，并进行数据初始化
  - 分配device内存，并将数据从host拷贝到device
  - 调用kernel函数在device上完成可并行的运算
  - 阻塞host，等待device完成运算（可选）
  - 将device上的运算结果拷贝回host
  - 使用运算结果，并释放host和device上分配的内存



CUDA编程 | 基础概念

- 执行流程中最重要的是kernel函数
  - 用__global__声明
  - 调用时用<<<grid, block>>>来指定kernel要执行的线程数量
- 每个线程都要执行kernel函数
  - blockIdx：线程所在block在grid中的索引 threadIdx：线程在所在block中的索引
  - gridDim：grid各个维度的大小 blockDim：block各个维度的大小
  - 利用blockIdx， threadIdx， blockDim可以获得当前线程的全局唯一标识
- CUDA的函数类型限定词
  - __global__：在device上执行，从host中调用，返回类型必须是void
  - __device__：在device上执行，从device中调用
  - __host__：在host上执行，从host中调用，一般省略不写
    - __host__可以和__device__同时用
    - __global__定义的kernel是异步的，这意味着host不会等待kernel执行完就执行下一步

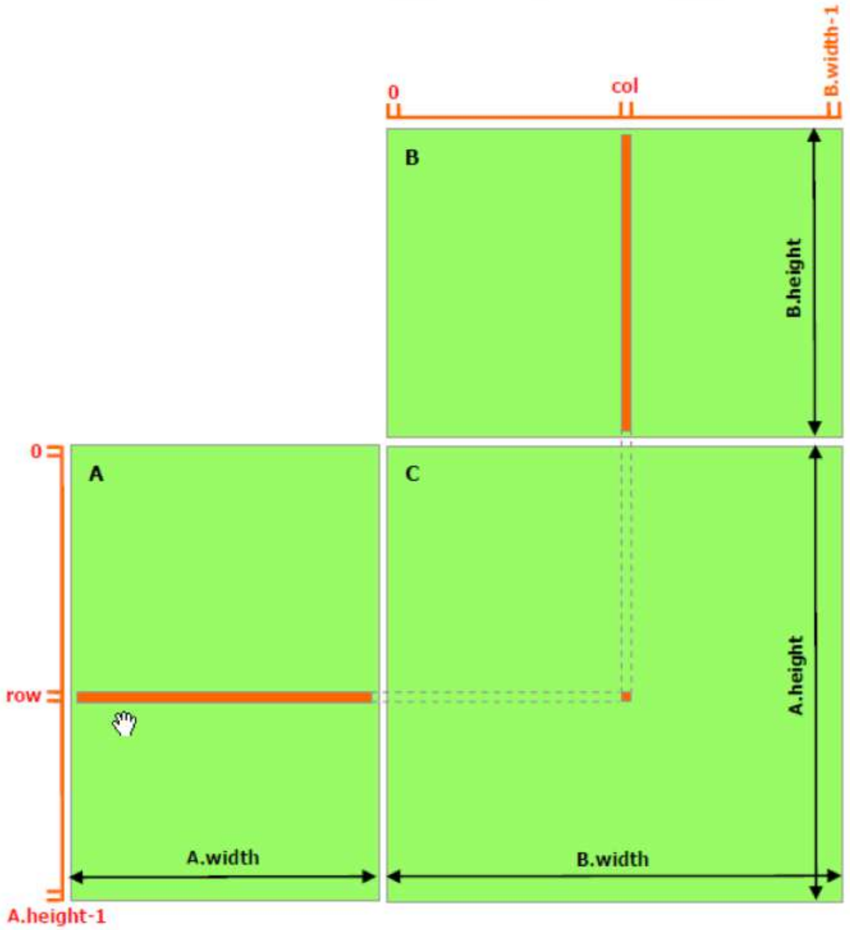
CUDA编程 | 矩阵加法

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

CUDA编程 | 矩阵乘法

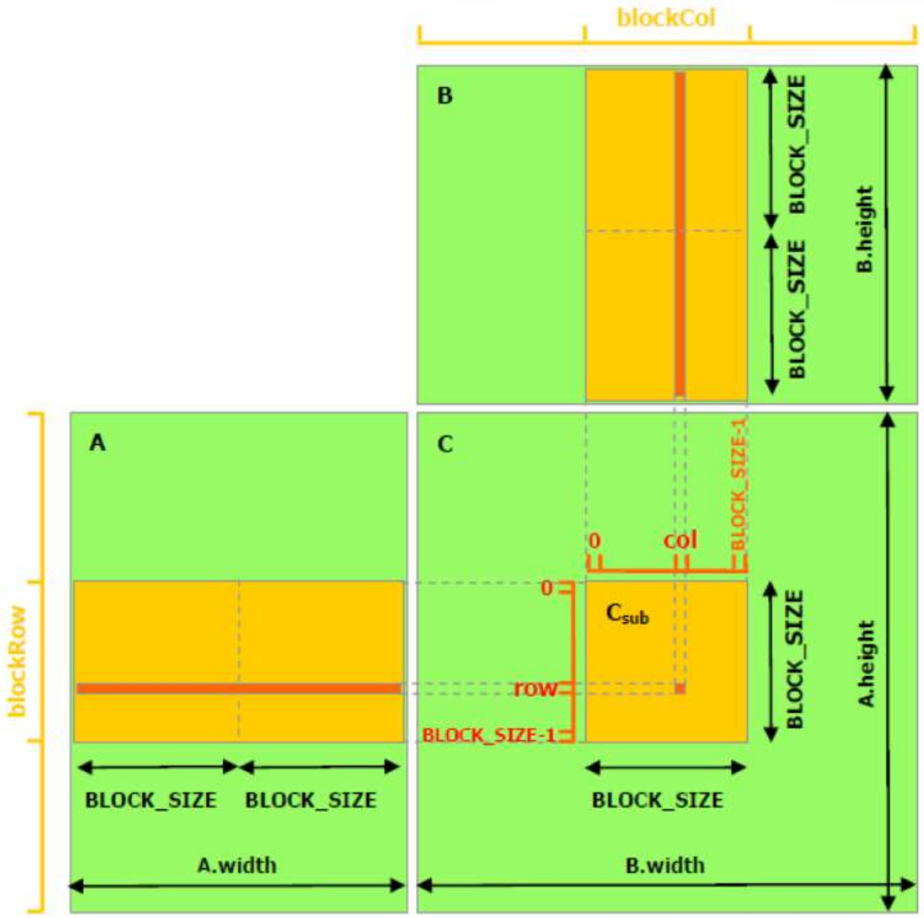
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```



- 可以优化的地方
  - 矩阵A/B中的每一个元素都被大量重复读取
  - A：B.width次 B：A.height次
  - 由于是从global memory中读取，就相当于存在大量的片外IO开销
  - GPU计算和片外存储器访问比例≈1:1

CUDA编程 | Shared Memory优化

- 减少片外存储器访问，将矩阵A/B的数据读取到block的shared memory
- 使用分片算法
  - 以block为单位进行运算
  - 将C_{sub}对应的A_{sub}和B_{sub}预先读取到shared memory
  - 矩阵A中的每一个元素的读取次数
    - B.width / BLOCK_SIZE
  - 矩阵B中的每一个元素的读取次数
    - A.height / BLOCK_SIZE
  - 此时GPU计算和片外存储器访问比例
    - BLOCK_SIZE : 1



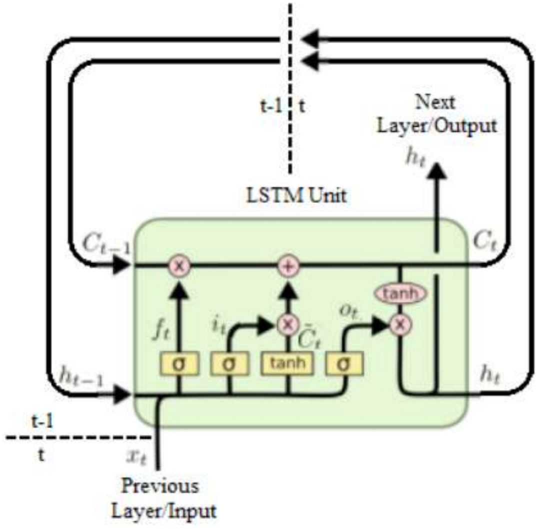
参考文献

- https://docs.nvidia.com/cuda/index.html
- https://zhuanlan.zhihu.com/p/34587739
- http://202.38.64.11/~xuyun/GPU_Computing.pdf
- CUDA Samples
  - Local /usr/local/cuda/samples/
  - Github https://github.com/NVIDIA/cuda-samples
- GPU架构
  - https://jcf94.com/2020/05/24/2020-05-24-nvidia-arch/

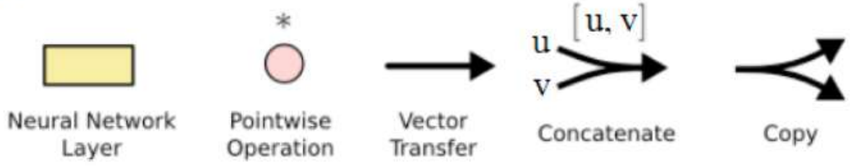
PyTorch-C++/CUDA拓展

LLTM | 模型结构

- 搬运自官方教程，以LLTM（long long-term unit）的CUDA优化为例
- LLTM: 比LSTM少一个遗忘门，tanh用elu代替



$$\cancel{f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)}$$
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \underset{\text{elu}}{\cancel{\tanh}}\left(W_C \cdot [h_{t-1}, x_t] + b_C\right)$$
$$C_t = \cancel{f_t *} C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$



LLTM | 原始代码

- 代码与图的对应关系
  - input: x_t
  - state: $[h_{t-1}, C_{t-1}]$
  - self.weights: $[W_i, W_o, W_C]$
  - self.bias: $[b_i, b_o, b_C]$
  - input_gate: i_t
  - output_gate: o_t
  - candidate_cell: $\widetilde{C}_t$
  - new_h: h_t
  - new_cell: C_t

```python
def forward(self, input, state):
    old_h, old_cell = state
    X = torch.cat([old_h, input], dim=1)

    # Compute the input, output and candidate cell gates with one MM.
    gate_weights = F.linear(X, self.weights, self.bias)
    # Split the combined gate weight matrix into its components.
    gates = gate_weights.chunk(3, dim=1)

    input_gate = torch.sigmoid(gates[0])
    output_gate = torch.sigmoid(gates[1])
    # Here we use an ELU instead of the usual tanh.
    candidate_cell = F.elu(gates[2])

    # Compute the new cell state.
    new_cell = old_cell + candidate_cell * input_gate
    # Compute the new hidden state and output.
    new_h = torch.tanh(new_cell) * output_gate

    return new_h, new_cell
```

先验知识丨文件结构

- PyTorch的C++/CUDA拓展有两种形式
  - 使用setuptools来实现“ahead of time”构建
  - 使用torch.utils.cpp_extension.load()来实现“just in time”构建
- 这里介绍“ahead of time”构建方式的文件结构
- \- LLTM
  - \- lltm.py // 拓展torch.autograd.Function，最终实现class LLTM(nn.Module)
  - \- lltm_cuda.cpp // 实现的C++函数将进行一些检查，然后调用CUDA代码中的函数
  - \- lltm_cuda_kernel.cu // 实际CUDA代码（可选）
  - \- setup.py // python打包工具setuptools所需编写的文件

先验知识丨setup.py文件

- CppExtension
  - setuptools.Extension的一个wrapper
  - 如果仅实现C++优化（不含CUDA），CUDAExtension替换为CppExtension
- BuildExtension
  - 管理C++/CUDA的混合编译

```python
from setuptools import setup
from torch.utils.cpp_extension import BuildExtension, CUDAExtension

setup(
    name='lltm_cuda',
    ext_modules=[
        CUDAExtension('lltm_cuda', [
            'lltm_cuda.cpp',
            'lltm_cuda_kernel.cu',
        ]),
    ],
    cmdclass={
        'build_ext': BuildExtension
    })
```

先验知识 | 相关头文件

- lltm_cuda.cpp
  - <torch/extension.h>，one-stop头文件，包括所有需要写Pytoch的C++拓展的API
    - ATen，主要的张量计算接口库
    - pybind11，为C++代码创建Python绑定的方法
    - 管理ATen和pybind11之间交互细节的头文件
  - 任何C++头文件，例如<iostream>, <vector>
- lltm_cuda_kernel.cu
  - <torch/extension.h>
  - 任何C++头文件
  - <cuda.h>，包括CUDA driver API
  - <cuda/runtime.h>，包括CUDA runtime API

先验知识 | pybind11用法

- 写在lltm_cuda.cpp文件里
- pybind11可以将C++函数或类绑定到Python上
  - PYBIND11_MODULE: 宏，创建一个python中可以import的对象
  - TORCH_EXTENSION_NAME: 宏，同setup.py中的name变量
  - m: py::module_类型变量
  - m.def(): module_::def()，定义绑定代码
    - args1: 函数名
    - args2: C++实现的函数的地址
    - args3: 函数描述

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("forward", &lltm_forward, "LLTM forward (CUDA)");
  m.def("backward", &lltm_backward, "LLTM backward (CUDA)");
}
```

LLTM | C++版本

- 首先要介绍一下PyTorch的torch.autograd.Function的拓展
- 应用场景
  - pytorch可以自动求导，但有时候一些操作是不可导的，这时候需要自定义求导方式
- 以torch.nn.Linear的源码为例了解一下代码结构
  - backward()的输入和输出的个数对应forward()的输出和输入的个数
  - ctx从forward()中保存需要的信息 用于backword()
  - ctx.needs_input_grad用于判断是否需要回传梯度

```python
# Inherit from Function
class LinearFunction(Function):

    # Note that both forward and backward are @staticmethods
    @staticmethod
    # bias is an optional argument
    def forward(ctx, input, weight, bias=None):
        ctx.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias
```

LLTM | C++版本

- 类似的，就可以构建LLTM的PyTorch拓展
  - 主要是调用了用C++写的forward()和backward()函数

```python
class LLTMFunction(Function):
    @staticmethod
    def forward(ctx, input, weights, bias, old_h, old_cell):
        outputs = lltm_cuda.forward(input, weights, bias, old_h, old_cell)
        new_h, new_cell = outputs[:2]
        variables = outputs[1:] + [weights]
        ctx.save_for_backward(*variables)

        return new_h, new_cell

    @staticmethod
    def backward(ctx, grad_h, grad_cell):
        outputs = lltm_cuda.backward(
            grad_h.contiguous(), grad_cell.contiguous(), *ctx.saved_variables)
        d_old_h, d_input, d_weights, d_bias, d_old_cell, d_gates = outputs
        return d_input, d_weights, d_bias, d_old_h, d_old_cell
```

- 使用该自定义Function时，.apply()即可

```python
def forward(self, input, state):
    return LLTMFunction.apply(input, self.weights, self.bias, *state)
```

LLTM | C++版本

- 简单介绍一下Aten库的使用
  - 三个命名空间at::, c10::, torch::，最常用torch::
  - 张量数据类型 torch::Tensor 标量数据类型 torch::Scalar
  - sigmoid函数求导

```
// s'(z) = (1 - s(z)) * s(z)
torch::Tensor d_sigmoid(torch::Tensor z) {
  auto s = torch::sigmoid(z);
  return (1 - s) * s;
}
```

  - tanh函数求导

```
// tanh'(z) = 1 - tanh^2(z)
torch::Tensor d_tanh(torch::Tensor z) {
  return 1 - z.tanh().pow(2);
}
```

LLTM | C++版本

- 用Aten实现C++版本的LLTM的lltm_forward函数
  - 基本上就是用ATen对原始python版本的代码的翻译
  - 类似的有lltm_backward函数，类似之前的Linear的例子，涉及比较多的梯度求导，这里就不再介绍
- 这里要指出，PyTorch给出的C++的Aten库，底层实现了GPU加速

```cpp
std::vector<torch::Tensor> lltm_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);

  auto gate_weights = torch::addmm(bias, X, weights.transpose(0, 1));
  auto gates = gate_weights.chunk(3, /*dim=*/1);

  auto input_gate = torch::sigmoid(gates[0]);
  auto output_gate = torch::sigmoid(gates[1]);
  auto candidate_cell = torch::elu(gates[2], /*alpha=*/1.0);

  auto new_cell = old_cell + candidate_cell * input_gate;
  auto new_h = torch::tanh(new_cell) * output_gate;

  return {new_h,
          new_cell,
          input_gate,
          output_gate,
          candidate_cell,
          X,
          gate_weights};
}
```

LLTM | C++版本

- 对比原始版本和C++版本每次forward和backward的时间
- 原始版本

```
Forward: 187.719 us | Backward 410.815 us
```

- C++版本

```
Forward: 149.802 us | Backward 393.458 us
```

```python
import time

import torch

batch_size = 16
input_features = 32
state_size = 128

X = torch.randn(batch_size, input_features)
h = torch.randn(batch_size, state_size)
C = torch.randn(batch_size, state_size)

rnn = LLTM(input_features, state_size)

forward = 0
backward = 0
for _ in range(100000):
    start = time.time()
    new_h, new_C = rnn(X, (h, C))
    forward += time.time() - start

    start = time.time()
    (new_h.sum() + new_C.sum()).backward()
    backward += time.time() - start

print('Forward: {:.3f} us | Backward {:.3f} us'.format(forward * 1e6/1e5, backward * 1e6/1e5))
```

LLTM | CUDA版本

- 代码可并行之处
  - gates是一个三维数组(batch_size, which_gate, state_size)
  - torch::sigmoid之类的函数或运算均是作用于gates的每个元素
  - 所以可以使用CUDA的线程来并行

```
std::vector<torch::Tensor> lltm_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);

  auto gate_weights = torch::addmm(bias, X, weights.transpose(0, 1));
  auto gates = gate_weights.chunk(3, /*dim=*/1);

  auto input_gate = torch::sigmoid(gates[0]);
  auto output_gate = torch::sigmoid(gates[1]);
  auto candidate_cell = torch::elu(gates[2], /*alpha=*/1.0);

  auto new_cell = old_cell + candidate_cell * input_gate;
  auto new_h = torch::tanh(new_cell) * output_gate;

  return {new_h,
          new_cell,
          input_gate,
          output_gate,
          candidate_cell,
          X,
          gate_weights};
}
```

LLTM | CUDA版本

- lltm.cpp -> lltm_cuda.cpp
  - 核心部分的代码均放到lltm_cuda_kernel.cu中实现
  - 以lltm_forward函数为例，可以简化为

```cpp
// NOTE: AT_ASSERT has become AT_CHECK on master after 0.4.
#define CHECK_CUDA(x) AT_ASSERTM(x.type().is_cuda(), #x " must be a CUDA tensor")
#define CHECK_CONTIGUOUS(x) AT_ASSERTM(x.is_contiguous(), #x " must be contiguous")
#define CHECK_INPUT(x) CHECK_CUDA(x); CHECK_CONTIGUOUS(x)

std::vector<torch::Tensor> lltm_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  CHECK_INPUT(input);
  CHECK_INPUT(weights);
  CHECK_INPUT(bias);
  CHECK_INPUT(old_h);
  CHECK_INPUT(old_cell);

  return lltm_cuda_forward(input, weights, bias, old_h, old_cell);
}
```

LLTM|CUDA版本

- ll_cuda_forward的实现中主要还是标红的三块代码
  - 默认数据都在GPU上，所以不需要进行数据调度的操作
  - 块1：开辟数据存储空间
  - 块2：定义kernel函数需要的两个参数
  - 块3：调用kernel函数
- lltm_cuda_backward类似

```
std::vector<torch::Tensor> lltm_cuda_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);
  auto gate_weights = torch::addmm(bias, X, weights.transpose(0, 1));

  const auto batch_size = old_cell.size(0);
  const auto state_size = old_cell.size(1);

  auto gates = gate_weights.reshape({batch_size, 3, state_size});
  auto new_h = torch::zeros_like(old_cell);
  auto new_cell = torch::zeros_like(old_cell);
  auto input_gate = torch::zeros_like(old_cell);
  auto output_gate = torch::zeros_like(old_cell);
  auto candidate_cell = torch::zeros_like(old_cell);

  const int threads = 1024;
  const dim3 blocks((state_size + threads - 1) / threads, batch_size);

  AT_DISPATCH_FLOATING_TYPES(gates.type(), "lltm_forward_cuda", ([&] {
    lltm_cuda_forward_kernel<scalar_t><<<blocks, threads>>>(
        gates.packed_accessor<scalar_t,3,torch::RestrictPtrTraits,size_t>(),
        old_cell.packed_accessor<scalar_t,2,torch::RestrictPtrTraits,size_t>(),
        new_h.packed_accessor<scalar_t,2,torch::RestrictPtrTraits,size_t>(),
        new_cell.packed_accessor<scalar_t,2,torch::RestrictPtrTraits,size_t>(),
        input_gate.packed_accessor<scalar_t,2,torch::RestrictPtrTraits,size_t>(),
        output_gate.packed_accessor<scalar_t,2,torch::RestrictPtrTraits,size_t>(),
        candidate_cell.packed_accessor<scalar_t,2,torch::RestrictPtrTraits,size_t>());
  }));

  return {new_h, new_cell, input_gate, output_gate, candidate_cell, X, gates};
}
```

LLTM|CUDA版本

- 使用C++模板实现的kernel函数
- thread的全局索引的获得
  - 举个例子
  - state_size = 2048
  - batch_size = 4
  - block_size =1024
  - 则实现了4*2048/1024=8个block
  - gates
    - dim1: 索引batch_size
    - dim2: which_gate
    - dim3: which_state_size

```
template <typename scalar_t>
__global__ void lltm_cuda_forward_kernel(
    const torch::PackedTensorAccessor<scalar_t,3,torch::RestrictPtrTraits,size_t> gates,
    const torch::PackedTensorAccessor<scalar_t,2,torch::RestrictPtrTraits,size_t> old_cell,
    torch::PackedTensorAccessor<scalar_t,2,torch::RestrictPtrTraits,size_t> new_h,
    torch::PackedTensorAccessor<scalar_t,2,torch::RestrictPtrTraits,size_t> new_cell,
    torch::PackedTensorAccessor<scalar_t,2,torch::RestrictPtrTraits,size_t> input_gate,
    torch::PackedTensorAccessor<scalar_t,2,torch::RestrictPtrTraits,size_t> output_gate,
    torch::PackedTensorAccessor<scalar_t,2,torch::RestrictPtrTraits,size_t> candidate_cell) {
  //batch index
  const int n = blockIdx.y;
  // column index
  const int c = blockIdx.x * blockDim.x + threadIdx.x;
  if (c < gates.size(2)){
    input_gate[n][c] = sigmoid(gates[n][0][c]);
    output_gate[n][c] = sigmoid(gates[n][1][c]);
    candidate_cell[n][c] = elu(gates[n][2][c]);
    new_cell[n][c] =
        old_cell[n][c] + candidate_cell[n][c] * input_gate[n][c];
    new_h[n][c] = tanh(new_cell[n][c]) * output_gate[n][c];
  }
}
```

LLTM | CUDA版本

- 对比原始版本和C++版本每次forward和backward的时间
- 原始版本

```
Forward: 187.719 us | Backward 410.815 us
```

- C++版本

```
Forward: 149.802 us | Backward 393.458 us
```

- CUDA版本

```
Forward: 129.431 us | Backward 304.641 us
```

参考文献

- Extenting cpp/cuda
  - Github https://github.com/pytorch/extension-cpp
  - Doc https://pytorch.org/tutorials/advanced/cpp_extension.html
  - Api https://pytorch.org/cppdocs
- Extending pytorch
  - Doc https://pytorch.org/docs/stable/notes/extending.html
  - Translate https://blog.csdn.net/tsq292978891/article/details/79364140

Thank You!