

第二阶段优化方法：CUDA Graph && 输入优化

- CUDA代码，数据传输肯定会存在的
- 每次调用 cudaMemcpy 都需要进行数据传输的准备和清理工作,包括在主机和设备之间建立通信、数据拷贝以及同步等操作。
 - 数据拷贝的瓶颈是带宽，这个就不谈了，硬件导致的
 - **减少数据传输的次数，减少这些通讯、同步的冗余**
- PCIE总线：CPU和GPU通过插槽，连接的，这个玩意的带宽，要看第几代的，比方说是30GB/S
 - 正常的CPU的内存的带宽和显存的带宽，都是几百G的
- NVlink是定制的连接方式，带宽较高
- **输入优化1:尽可能地减少数据传输和次数。**

如何做输入优化

问题1：什么是内存？

内存是程序与CPU进行沟通的桥梁，其作用是

存放CPU中的运算数据，以及存放与硬盘等外部存储设备交换的数据。

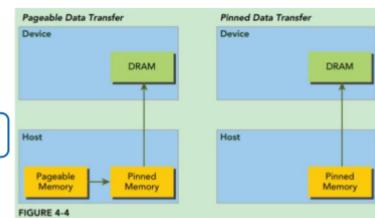


问题2：什么是可分页内存，什么是页锁定内存？

可分页内存是一种内存管理技术，它可以将物理内存划分为固定大小的页面，以便更有效地管理内存空间（提高利用率、支持多任务处理、内存保护等）。

页锁定内存的重要属性是主机的操作系统将不会对这块物理内存进行分页。

问题3：为什么从页锁定内存拷贝数据到 GPU，比从可分页内存拷贝要快得多？



注：为什么从GPU从可分页内存拷贝数据时，需要先将数据从可分页内存拷贝到页锁定内存？可以直接咨询文心一言

- **输入优化2:不得不进行数据传输的情况下,尽可能地加快主机到GPU的数据拷贝速度。**
 - 先直接说结论：
 - 主机端的内存可分为两种:可分页内存(Pageable Memory),页锁定内存(Page-locked Memory Or Pinned Memory)。
 - 从主机的页锁定内存拷贝数据到 GPU 要快得多！
- GPU每次都是需要从页锁定内存取的，是驱动层面的，至于为什么，去搜吧~

输入优化1：尽可能地减少数据传输和次数

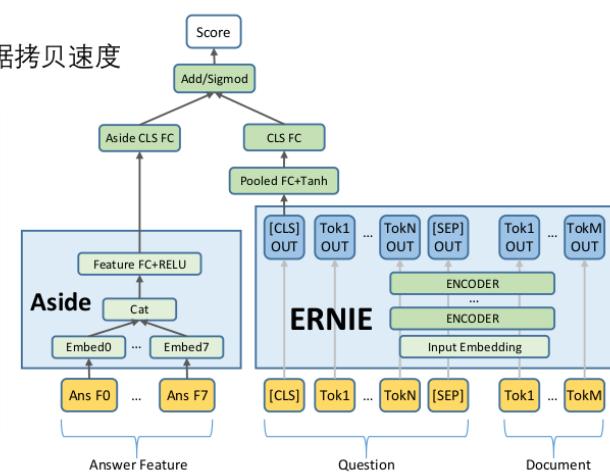
输入优化2：申请页锁定内存，加快主机到GPU的数据拷贝速度

```
// 计算输入的总最大值
int whole_input = 3 * max_batch_ * max_seq_len_ * sizeof(int) +
    (max_batch_ * sizeof(int) + 127) / 128 * 128 * 8;
int output_max_size = (max_batch_ * sizeof(int) + 127) / 128 * 128;
int whole_size = whole_input + output_max_size;

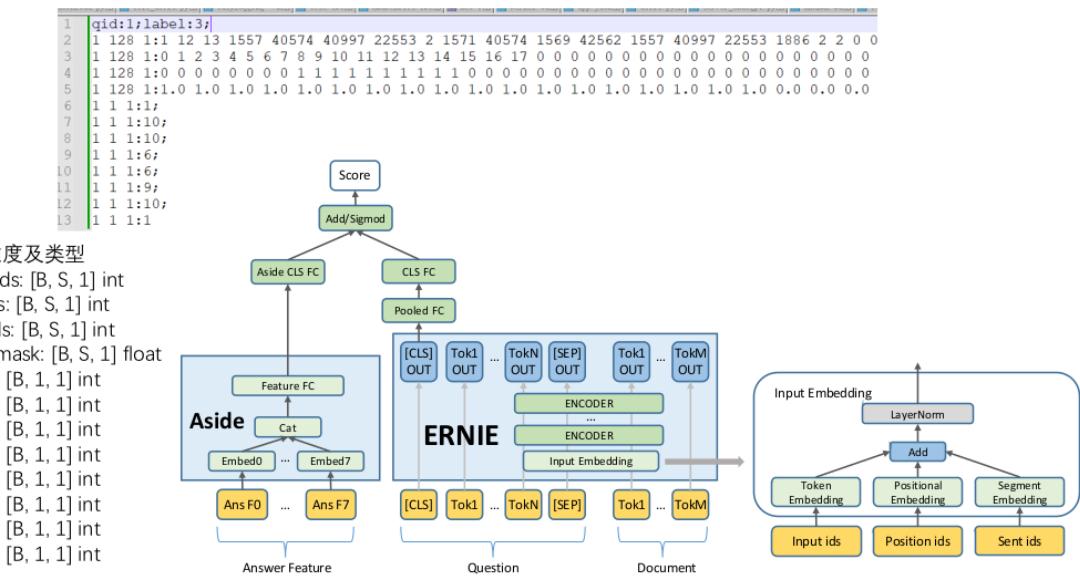
// 申请一次内存和显存
cudaMalloc(&d_buffer_, whole_size);
cudaMallocHost(&h_buffer_, whole_size);

// 将所有输入合并到 页锁定内存 中，host2host的拷贝
memcpy(h_buffer_ + s0_offset, s.i0.data(), input_bytes);
// ...
memcpy(h_buffer_ + s11_offset, s.i11.data(), aside_input_bytes);

// 一次性传输数据
cudaMemcpy(d_buffer_, h_buffer_, whole_input, cudaMemcpyHostToDevice);
```



- 中间讲了个模型的输入：



-
- 十二个输入，前面的是维度，后面是实际的数据，前四个长，后四个短
- 详细的见[第四章](#)
- 12个输入，正常写代码，就是分别12个输入，开辟12次（`malloc`可以提前申请，可以不考虑~），传输12次（必须的）
- 计算模型输入的总最大值
 - 12个输入和一个输出
 - mask的shape被修改了，变得和之前三个输入不一样了，但是没说具体的细节
- 我们下面要做的就是把所有的输入都先在CPU上拷贝到页锁定内存，再做一次的拷贝
- 这样子就尽可能的减少了传输的次数了
- 尽量把这个做成习惯！当你输入很多的时候！
 - **当我们用stream, `cudaMemcpyAsync`的时候，这个还有收益吗？**
- 收益是看情况的
 - 比如一个模型只有一个输入，或者输入比较小

耗时(ms)	$B = 10, S = 128$	$B = 1000, S = 128$
multi cpy	0.085531	0.868582
pinned multi cpy	0.094667	0.272782
single cpy	0.017495	0.621185
pinned single cpy	0.018176	0.186403

- - T4显卡上的测试
 - 0.86ms，其实不低了，1ms都是很珍贵的
 - 用pinned memory在小数据上没什么变化，但是在大的数据上，收益就很大了
 - merge后，拷贝速度快了很多

- 实际的代码
- 输入优化3:Batching

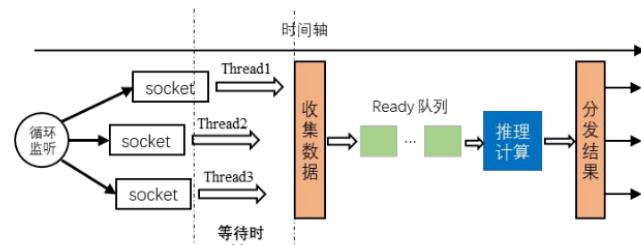
输入优化3: Batching

Tesla T4 GPU float 矩阵乘测速

$M = 10, K = 1024, N = 4096$
Execution Time: 0.142847 ms

$M = 100, K = 1024, N = 4096$
Execution Time: 0.468695 ms

计算数据量增加了10倍，但耗时只
增加了3.29倍



批量调度算法流程

- • 输入优化4:数据拷贝与计算 overlap 掩盖延迟

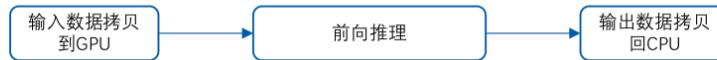
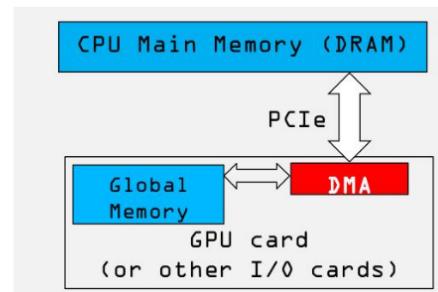
输入优化4: 数据拷贝与计算 overlap 掩盖延迟

为什么 overlap 能掩盖延迟?

数据拷贝占用PCIe总线的带宽资源;

计算过程占用GPU的算力资源;

两者互不干扰。



◦

- CPU里面，流水线，多线程什么的，都有掩盖延迟的概念

使用cuda-stream实现overlap是常见的方案，那如何设计 overlap?



◦

使用cuda-stream实现overlap是常见的方案，那如何设计 overlap?



◦



不建议以下设计方案



计算的overlap有可能会发生GPU资源竞争现象，导致两个前向推理都变慢。

除非一些特殊情况，比如GPU算力很强 并且 前向推理计算量很小。

- - 计算重叠不建议的
 - 当你计算资源占用小的时候可以这么做
 - 我突然想起来，之前做ros多个节点推理，节点越多就越慢，就是因为这个原因，导致每一个的前向推理都变慢了 但是你同时要跑这么多的模型，本就会在同一时间占用GPU的资源~
 - 所以应该是做成多batch的
 - 几个方案：
 - batch，一次推理
 - 多stream，多次推理，tensorrt这边怎么搞多stream的？
 - 多线程，ros里面不好搞，所以多节点
 - **假如你追求吞吐量，那么就是batching，合并输入**

小插曲cudaStream和cudaEvent

- CUDA stream是GPU上task 的执行队列,所有CUDA操作(kernel,内存拷贝等)都是在stream上执行的。
- CUDA stream有两种
 - 隐式流,又叫默认流,NULL流
 - 所有的CUDA操作默认运行在隐式流里。隐式流里的GPU task和CPU端计算是同步的。
 - 举例:n = 1这行代码,必须等上面三行都执行完,才会执行它。
 - 看PPT
 - 显式流:显式申请的流
 - 显式流里的GPU task和CPU端计算是异步的。不同显式流内的GPU task执行也是异步的。
- cudaStream的操作
 - 定义
 - `cudaStream_t stream;`
 - 创建
 - `cudaStreamCreate(&stream);`
 - 数据传输
 - `cudaMemcpyAsync(dst, src, size, type, stream)`
 - kernel在流中执行
 - `kernel_name<<<grid, block, sharedMemSize, stream>>>(argument list);`
 - 同步和查询
 - `cudaError_t cudaStreamSynchronize(cudaStream_t stream)`
 - `cudaError_t cudaStreamQuery(cudaStream_t stream);`
 - 销毁

- `cudaError_t cudaStreamDestroy(cudaStream_t stream);`
- 几种并行：
 - CPU计算和kernel计算并行
 - CPU计算和数据传输并行
 - 数据传输和kernel计算并行
 - kernel计算并行
- 强调知识点
 - 显式流里的GPU task与CPU端 task 的执行是异步的,使用stream一定要注意同步!
 - `cudaStreamSynchronize()` 同步一个流
 - `cudaDeviceSynchronize()` 同步该设备上的所有流
 - `cudaStreamQuery()` 查询一个流任务是否完成
- H2D 和 D2H 为什么没有重叠?它们已经在不同stream上了。
 - 因为CPU和GPU的数据传输是经过PCIe总线的,PCIe上的操作是顺序的。
 - 带有双工PCIe总线的设备可以重叠两个数据传输,但它们必须在不同的流和不同的方向上。
- CUDA Stream 优先级
 - ...
- CUDA Stream 为什么有效?
 - 一、PCIe总线传输速度慢,是瓶颈,会导致传输数据的时候GPU处于空闲等待状态。
 - 多流可以实现数据传输与kernel计算的并行。
 - 二、一个kernel往往用不了整个GPU的算力。多流可以让多个kernel同时计算,充分利用GPU算力。
 - 三、不是流越多越好。GPU内可同时并行执行的流数量是有限的。
 - CUDA加速,kernel合并,将小任务合并成大任务,更有效。
- 思考: GPU kernel耗时最大在哪里?
 - 计算密集型:耗时在计算,一次访存,数十次甚至上百次计算
 - 访存密集型:耗时在访存,一次访存,几次计算
 - GPU一般处理简单可并行计算,大部分kernel都是访存密集型
- 案例
 - 看PPT, 多流是怎么加速的
 - 以及合并成一个kernel
- CUDA Stream 默认流的表现
 - 单线程内,默认流的执行是同步的,显式流的执行是异步的
 - 单线程内,编译加上`--default-stream per-thread`后默认流的执行是异步的,显式流的执行是异步的
 - 多线程下,默认流的表现是什么呢?是一个默认流还是多个默认流?
 - **默认多线程共享一个默认流**
 - 每个线程都有一个默认流
 - <https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>

CUDA Event API

- CUDA Event,在stream中插入一个事件,类似于打一个标记位,用来记录stream是否执行到当前位置。Event有两个状态,已被执行和未被执行。
- 定义
 - `cudaEvent_t event`
- 创建
 - `cudaError_t cudaEventCreate(cudaEvent_t* event);`
- 插入流中

- cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0);
- 销毁
 - cudaError_t cudaEventDestroy(cudaEvent_t event);
- 同步和查询
 - cudaError_t cudaEventSynchronize(cudaEvent_t event);
 - cudaError_t cudaEventQuery(cudaEvent_t event);
- 进阶同步函数
 - cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event);
- 最常用的用法是测时间

CUDA 同步操作

- CUDA中的显式同步按粒度可以分为四类
 - device synchronize 影响很大
 - stream synchronize 影响单个流和CPU
 - event synchronize 影响CPU,更细粒度的同步
 - synchronizing across streams using an event
- `cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event);`
- 该函数会指定该stream等待特定的event,该event可以关联到相同或者不同的stream
- 这部分看的不仔细，有空再看了

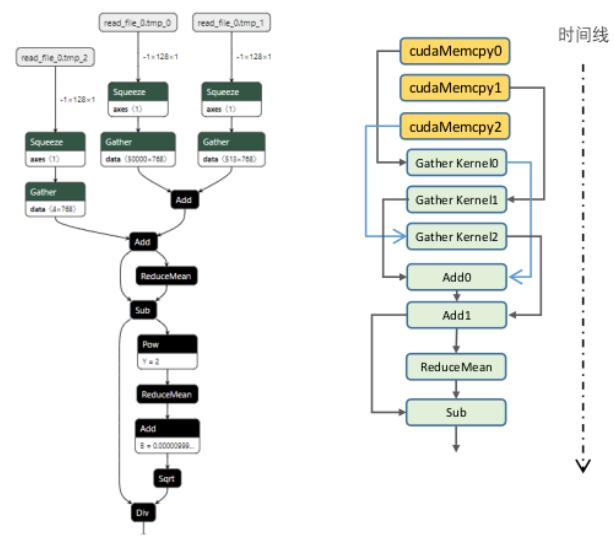
再往后就是一些nvvp的操作，暂时不需要看了~

回归正题，cuda graph

模型推理本质上是一个图结构，由一系列的GPU操作（Operation）组成。

GPU操作的种类：

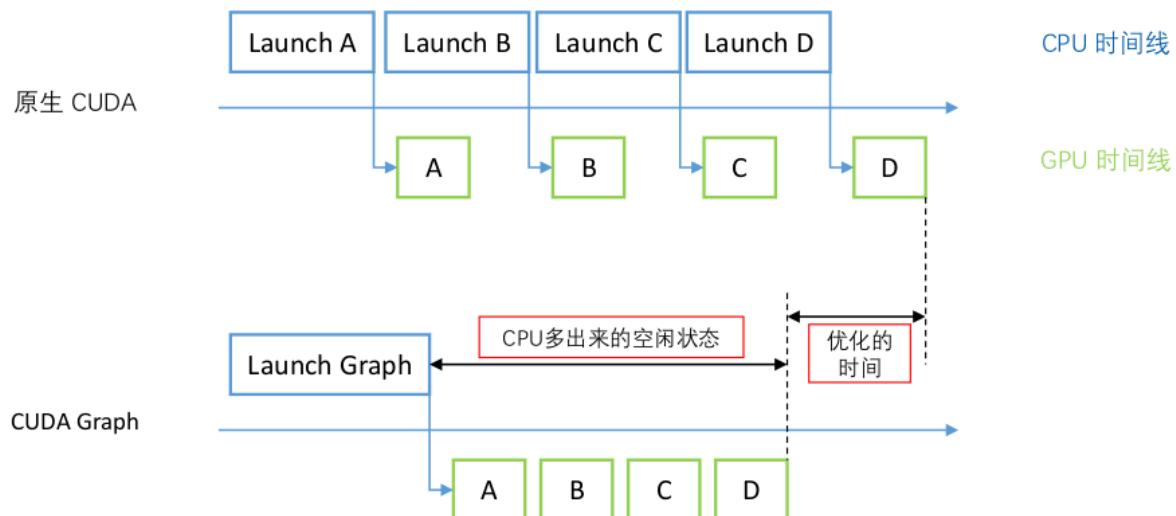
- CUDA Kernel
- 在CUDA端调用的函数，比如CUDASync函数
- Memcpy/Memset
- Memory Alloc/Free
- CUDA-Graph



- GPU操作不仅仅是只包括算子的，还有一些cudaAPI
- 上图是ernie的模型，没有mask，只有三个输入了，维度变换，gather就是embedding算子
- 做两个加法就是embedding那边的操作，需要看bert部分的代码
- 在下面是一个layerNorm

CUDA-Graph就是将一个子图内的所有GPU操作组装一个GPU操作。

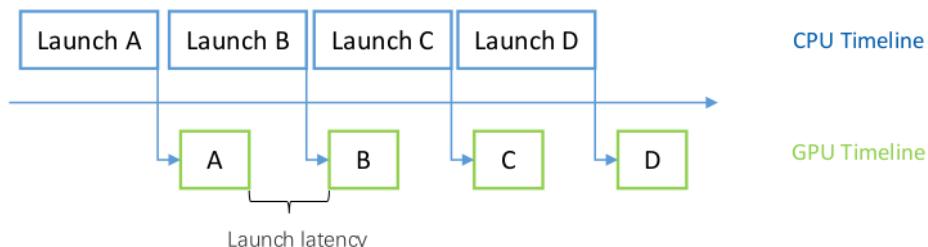
ABCD代表四个GPU操作。



- 你launch A了，A kernel也在执行，假如你launch B还没完成，那么AB两个kernel之间就存在一个空闲的状态了
- 假如你的A的kernel很大，那可能不会有这块空闲了，**大部分的GPU kernel而言，都没有那么大的~**
- 四个封装到一起，一次调用
- **CPU多出了空闲状态，GPU的空闲也干掉，GPU更为紧凑的执行**
- **为什么要做cuda graph**

原因：GPU操作的启动开销，相比于GPU操作（kernel运行或内存复制）而言，太耗时了！

- 现代GPU操作（kernel运行或内存复制）耗时以微秒为单位；
- GPU操作的启动开销也是微秒级别的。



复杂模型推理过程中的总算子数量基本以千为单位。以ERNIE模型为例，ONNX模型共有1932个算子，不考虑合并情况，假设每个算子的启动开销为1微秒，启动耗时近2毫秒（1毫秒=1000微秒）。

使用CUDA-Graph，只需要启动一次。

- 现在的GPU太强了，kernel执行的速度很快，启动的操作很耗时
- GPU启动的耗时，主要还是依赖于CPU那边指令发射的速度的，发展这么长时间，延时的降低没有GPU上延时降低的明显
- cuda-graph是CUDA10可能才出来的东西
- 使用cuda-graph，启动一次，可能就是10us，比2ms就优化很多了~
- 只是做个假设，tensorrt会融合算子，可能不一定那么多算子的，假设未必成立，自己做的话，速度提升，对于数据量很大的情况的话，提升就不止这么多了~

cuda-graph的语法

- 比较简单，且比之前的输入优化的性价比是要高一点的

CUDA-Graph是将一个子图内的所有CUDA操作组装一个CUDA操作。

一、数据类型定义

- `cudaGraph_t` 类型的对象定义了 cuda graph 的结构;
- `cudaGraphExec_t` 类型的对象是一个“可执行的 graph 实例”：它可以以类似于单个 kernel 的方式启动和执行。

```

bool graphCreated=false;
cudaGraph_t graph;
cudaGraphExec_t instance; 定义数据类型

for(int istep = 0; istep < NSTEP; istep ++){
    if(!graphCreated){

        cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal); 捕捉CUDA Graph 并实例化

        for( int ikrnl=0; ikrnl<NKERNEL; ikrnl++){
            shortKernel<<<blocks, threads, 0, stream>>>(out_d, in_d);
        }

        cudaStreamEndCapture(stream, &graph);
        cudaGraphInstantiate(&instance, graph, NULL, NULL, 0); 执行CUDA Graph

        graphCreated=true;
    }

    cudaGraphLaunch(instance, stream);
    cudaStreamSynchronize(stream);
}

```

- `cudaGraph_t` 是存储结构的，真正的执行的是 `cudaGraphExec_t`，可以类似单个 kernel 的方式启动和执行

二、捕捉CUDA Graph并实例化

- 通过 `cudaStreamBeginCapture` 和 `cudaStreamEndCapture` 函数来捕捉它们之间 `stream` 上
 - 一定是以流为单位捕捉的
 - 你所有的 GPU 操作都在这个流里面执行，通过这两个函数就可以把这个流上所有的操作捕捉，储存在 `graph` 上
 - 输入信息，参数信息，函数信息等都存储在 `cudaGraph_t` 里面
- 通过 `cudaGraphInstantiate` 生成 `graph` 实例

三、执行CUDA Graph

- 通过 `cudaGraphLaunch` 执行生成的实例。
- kernel graph 只需要捕获和实例化一次，并在所有后续循环中重复使用相同的实例。
- 意思就是不需要重复捕捉，直接执行 `cudaGraphLaunch` 即可
- 传入的 `stream` 可以是新的 `stream`，未必要是上面捕捉的 `stream`

CUDA Graph 缺点

```

cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);

for(int ikrnl=0; ikrnl<NKERNEL; ikrnl++){
    shortKernel<<<blocks, threads, 0, stream>>>(out_d, in_d);
}

cudaStreamEndCapture(stream, &graph);
cudaGraphInstantiate(&instance, graph, NULL, NULL, 0);

```

- CUDA Graph Capture 本质上是捕捉 `stream` 上的一段 GPU 操作并保存下来。即这段 GPU 操作内的 - 参数和内存指针等 固定下来了，不能变
 - 固定死了，那你的输入维度就是固定的了~，你只有输入维度是固定的，内存大小等，都是固定的
 - 会遇到要变的时候，大小是动态的
- 需要变的时候怎么办？CUDA Graph recapture
 - 不建议使用 CUDA Graph recapture
 - 因为 capture 是耗时的，假如你只变一次，我就重新捕捉一下

- 但Capture是耗时的,CUDA Graph适用于Capture一次,Launch多次的场景,不适用于常见的动态输入维度场景。
- 几乎每一次的推理的输入的维度都是变的，你是不知道你输入的维度是多大的
- 但是你还想用cuda graph，咋办

针对变长输入的CUDA Graph解决方案

- CV输入维度固定，但是NLP和ASR很多输入都是不固定的，这个解决方案可以记住了
- 变长输入:即输入长度是动态的。对应TensorRT dynamic shape模式,需要设置Profile shape。
 - tensorrt之前有两种模式的，static shape和dynamic shape，前者被舍弃了
- 变长是主流了
- (1)统计项目输入,将输入按一定的间隔分成N个固定大小;不够长就做padding~
- (2)然后,创建N个Context,并捕捉成N个CUDA-Graph。
- 项目Ernie中的输入统计结果:

	s[1, 32]	s[33, 64]	s[65, 96]	s[97, 128]	b_total
b[1, 5]	0	3	14	2	19
b[6, 10]	1	204	695	104	1004
s_total	1	207	709	106	1023

-
- 统计数据量
- 这个切割是自己设计的，看这个表可能看不出来~
- 因为你分成了N个固定的大小了，要创建N个context

TensorRT runtime 推理过程

1. 创建一个runtime对象
2. 反序列化生成engine : runtime ---> engine
3. 创建一个执行上下文ExecutionContext : engine ---> context
4. 填充数据
5. 执行推理 : context ---> enqueueV2
6. 释放资源 : delete

- 分成了[1, 32], [1, 64], [1, 96], [1, 128], [2, 32], [2, 64], [2, 96], [2, 128].....
- 分了1-10，即40个固定大小，开了40个context
- 40个context的大小就固定了
- 假设你把分块的区域设置为[1, 32]的时候，你的输入的数据大小是[1, 28]，你直接padding成32，这四个额外的计算量会快的
- 你要怕padding多浪费计算，你就以16个为区间分就行了
- 设置区间，维度，这个在tensorrt里面叫profile，tensorrt的基础知识

单profile的构建代码

```

profile = builder.create_optimization_profile()
min_shape = (1, 128, 1)
opt_shape = (5, 128, 1)
max_shape = (10, 128, 1)
profile.set_shape(src_ids_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(pos_ids_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(sent_ids_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(input_mask_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)

min_shape = (1, 1, 1)
opt_shape = (5, 1, 1)
max_shape = (10, 1, 1)
profile.set_shape(tmp6_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(tmp7_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(tmp8_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(tmp9_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(tmp10_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(tmp11_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(tmp12_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
profile.set_shape(tmp13_tensor.name, min=min_shape, opt=opt_shape, max=max_shape)
config.add_optimization_profile(profile)

```

Context的推理代码

```

void *device_bindings[13] = {rc_ids_tensor.get(), sent_ids_tensor.get(),
                           pos_ids_tensor.get(),
                           input_mask_tensor.get(),
                           tmp6_tensor.get(), tmp7_tensor.get(),
                           tmp8_tensor.get(), tmp9_tensor.get(), tmp10_tensor.get(),
                           tmp11_tensor.get(), tmp12_tensor.get(), tmp13_tensor.get(),
                           cuda_out_ptr.get()};
//printf("before enqueue\n");
bool ret = context_->enqueueV2(device_bindings, cuda_stream_, nullptr);

```

- 比如你这边，设置的动态的shape
- 具体的代码在：[/home/ros/Downloads/ernie/TensorRT-ERNIE/ONNX](#)
- 上面四个一个profile，下面8个一个profile，一共12个输入
- **单profile，dynamic shape的python部分的profile的代码**

```

def onnx2trt(args):
    onnx_file = args.model
    logger = trt.Logger(trt.Logger.VERBOSE)

    builder = trt.Builder(logger)
    config = builder.create_builder_config()
    profile = builder.create_optimization_profile()
    network =
builder.create_network(1<<int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BA
TCH))
    config.max_workspace_size = 3<<30

    if args.fp16:
        config.set_flag(trt.BuilderFlag.FP16)

    parser = trt.OnnxParser(network, logger)
    if not os.path.exists(onnx_file):
        raise RuntimeError("Failed finding onnx file!")

    print("Succeeded finding onnx file!")
    with open(onnx_file, 'rb') as model:
        if not parser.parse(model.read()):
            print("Failed parsing ONNX file!")
            for error in range(parser.num_errors):
                print(parser.get_error(error))
            exit()
    print("Succeeded parsing ONNX file!")

    # Create the network

```

```
src_ids_tensor = network.get_input(0)
pos_ids_tensor = network.get_input(1)
sent_ids_tensor = network.get_input(2)
input_mask_tensor = network.get_input(3)

tmp6_tensor = network.get_input(4)
tmp7_tensor = network.get_input(5)
tmp8_tensor = network.get_input(6)
tmp9_tensor = network.get_input(7)
tmp10_tensor = network.get_input(8)
tmp11_tensor = network.get_input(9)
tmp12_tensor = network.get_input(10)
tmp13_tensor = network.get_input(11)

profile = builder.create_optimization_profile()
min_shape = (1, 128, 1)
opt_shape = (5, 128, 1)
max_shape = (10, 128, 1)
profile.set_shape(src_ids_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(pos_ids_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(sent_ids_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(input_mask_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)

min_shape = (1, 1, 1)
opt_shape = (5, 1, 1)
max_shape = (10, 1, 1)
profile.set_shape(tmp6_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(tmp7_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(tmp8_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(tmp9_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(tmp10_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(tmp11_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(tmp12_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
profile.set_shape(tmp13_tensor.name, min=min_shape, opt=opt_shape,
max=max_shape)
config.add_optimization_profile(profile)

engine = builder.build_engine(network, config)
if not engine:
    raise RuntimeError("build_engine failed")
print("Succeeded building engine!")

print("Serializing Engine...")
```

```

    serialized_engine = engine.serialize()
    if serialized_engine is None:
        raise RuntimeError("serialize failed")

    with open(args.output, "wb") as fout:
        fout.write(serialized_engine)

```

- 推理的代码 : [code](#)

```

// set the input dim
void *device_bindings[13] = {rc_ids_tensor.get(), sent_ids_tensor.get(),
pos_ids_tensor.get(),
                           input_mask_tensor.get(),
                           tmp6_tensor.get(), tmp7_tensor.get(),
                           tmp8_tensor.get(), tmp9_tensor.get(),
tmp10_tensor.get(),
                           tmp11_tensor.get(), tmp12_tensor.get(),
tmp13_tensor.get(),
                           cuda_out_ptr.get()};
//printf("before enqueue\n");
bool ret = context_->enqueueV2(device_bindings, cuda_stream_, nullptr);
if (!ret) {
    std::cout << ("context_->enqueueV2 failed!") << std::endl;
    return -100;
}

cudaMemcpy(s.out_data.data(), cuda_out_ptr.get(), s.shape_info_0[0] *
sizeof(float), cudaMemcpyDeviceToHost);
cudaStreamSynchronize(cuda_stream_);

```

- 此时都是在显存上面的
- 正因为输入维度是动态的，那么这个bindings，每次指针都得变化

```

int TrtHepler::Forward(sample& s) {
    cudaSetDevice(_dev_id);
    auto rc_ids_tensor = CpuToDevice(s.shape_info_0, s.i0.data());
    auto sent_ids_tensor = CpuToDevice(s.shape_info_1, s.i1.data());
    auto pos_ids_tensor = CpuToDevice(s.shape_info_2, s.i2.data());
    auto input_mask_tensor = CpuToDevice(s.shape_info_3, s.i3.data());
    auto tmp6_tensor = CpuToDevice(s.shape_info_4, s.i4.data());
    auto tmp7_tensor = CpuToDevice(s.shape_info_5, s.i5.data());
    auto tmp8_tensor = CpuToDevice(s.shape_info_6, s.i6.data());
    auto tmp9_tensor = CpuToDevice(s.shape_info_7, s.i7.data());
    auto tmp10_tensor = CpuToDevice(s.shape_info_8, s.i8.data());
    auto tmp11_tensor = CpuToDevice(s.shape_info_9, s.i9.data());
    auto tmp12_tensor = CpuToDevice(s.shape_info_10, s.i10.data());
    auto tmp13_tensor = CpuToDevice(s.shape_info_11, s.i11.data());

    void* out_ptr;

```

```
auto ret_ = cudaMalloc(&out_ptr, s.shape_info_0[0] * sizeof(float)); //  
-1 * 1  
cuda_shared_ptr<void> cuda_out_ptr;  
make_cuda_shared(cuda_out_ptr, out_ptr);  
  
cudaEvent_t start, stop;  
float elapsed_time = 0.0;  
  
int binding_idx = 0;  
//std::vector<std::vector<int>> input_dims = {s.shape_info_0,  
s.shape_info_1, s.shape_info_2, s.shape_info_3,  
                                         //s.shape_info_4,  
s.shape_info_5, s.shape_info_6, s.shape_info_7,  
                                         //s.shape_info_8,  
s.shape_info_9, s.shape_info_10, s.shape_info_11};  
std::vector<std::vector<int>> input_dims = {s.shape_info_0,  
s.shape_info_1, s.shape_info_2, s.shape_info_3,  
                                         s.shape_info_4,  
s.shape_info_5, s.shape_info_6, s.shape_info_7,  
                                         s.shape_info_8,  
s.shape_info_9, s.shape_info_10, s.shape_info_11};  
// set device_bindings_ and setBindingDimensions  
for (size_t i = 0; i < input_dims.size(); i++) {  
    std::vector<int> dims_vec = input_dims[i];  
    nvinfer1::Dims trt_dims;  
    trt_dims.nbDims = static_cast<int>(dims_vec.size());  
    memcpy(trt_dims.d, dims_vec.data(), sizeof(int) * trt_dims.nbDims);  
    context_->setBindingDimensions(binding_idx, trt_dims);  
    binding_idx ++;  
}  
  
if (!context_->allInputDimensionsSpecified()) {  
    //gLogFatal << "context_->allInputDimensionsSpecified() error";  
    std::cout << ("context_->allInputDimensionsSpecified() error") <<  
std::endl;  
    assert(0);  
}  
  
// set the input dim
```

- 你就得调用这个`context_->setBindingDimensions(binding_idx, trt_dims);`
- 你用cuda graph是没法用的，指针不能改变，同时`setBindingDimensions`这个接口也用不起来了
- 所以你怎么解决呢，就是多profile

多profile的构建代码

```

batchs = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
seq_lens = [1, 32, 64, 96, 128]

# batchs = [1]
# seq_lens = [64, 128]

for b in batchs:
    for s in seq_lens:
        profile = builder.create_optimization_profile()
        static_shape = (b, s, 1)
        profile.set_shape("src_ids", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("sent_ids", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("pos_ids", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("input_mask", min=static_shape, opt=static_shape, max=static_shape)

        static_shape = (b, 1, 1)
        profile.set_shape("tmp6", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("tmp7", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("tmp8", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("tmp9", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("tmp10", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("tmp11", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("tmp12", min=static_shape, opt=static_shape, max=static_shape)
        profile.set_shape("tmp13", min=static_shape, opt=static_shape, max=static_shape)
        builder_config.add_optimization_profile(profile)

```

- 上图里面的profile，很多的shape，比如
- $[1,1], [1,32], [1,64], [1,96], [1,128] \dots [10,1], [10,32], [10,64], [10,96], [10,128]$ 这样子一共50个profile
- 输入的时候，可以生成50个context，根据输入的维度走哪个context，就是分支，假如是 $[1,30]$ ，就走 $[1,32]$ 的， $[10,120]$ 就走 $[10,128]$ 的
- build的代码可以如下：

```

def onnx2trt(args):
    onnx_file = args.model
    logger = trt.Logger(trt.Logger.VERBOSE)

    builder = trt.Builder(logger)
    config = builder.create_builder_config()
    profile = builder.create_optimization_profile()
    network =
builder.create_network(1<<int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
    config.max_workspace_size = 3<<30

    if args.fp16:
        config.set_flag(trt.BuilderFlag.FP16)

    parser = trt.OnnxParser(network, logger)
    if not os.path.exists(onnx_file):
        raise RuntimeError("Failed finding onnx file!")

    print("Succeeded finding onnx file!")
    with open(onnx_file, 'rb') as model:
        if not parser.parse(model.read()):
            print("Failed parsing ONNX file!")
            for error in range(parser.num_errors):
                print(parser.get_error(error))

```

```
        exit()
print("Succeeded parsing ONNX file!")

# Create the network
src_ids_tensor = network.get_input(0)
pos_ids_tensor = network.get_input(1)
sent_ids_tensor = network.get_input(2)
input_mask_tensor = network.get_input(3)

tmp6_tensor = network.get_input(4)
tmp7_tensor = network.get_input(5)
tmp8_tensor = network.get_input(6)
tmp9_tensor = network.get_input(7)
tmp10_tensor = network.get_input(8)
tmp11_tensor = network.get_input(9)
tmp12_tensor = network.get_input(10)
tmp13_tensor = network.get_input(11)

batches = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
seq_lens = [1, 32, 64, 96, 128]
for b in batches:
    for s in seq_lens:
        profile = builder.create_optimization_profile()
        static_shape = (b, s, 1)
        profile.set_shape(src_ids_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(pos_ids_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(sent_ids_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(input_mask_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)

        static_shape = (b, 1, 1)
        profile.set_shape(tmp6_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(tmp7_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(tmp8_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(tmp9_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(tmp10_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(tmp11_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(tmp12_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        profile.set_shape(tmp13_tensor.name, min=static_shape,
opt=static_shape, max=static_shape)
        config.add_optimization_profile(profile)

engine = builder.build_engine(network, config)
if not engine:
```

```

    raise RuntimeError("build_engine failed")
    print("Succeeded building engine!")

    print("Serializing Engine...")
    serialized_engine = engine.serialize()
    if serialized_engine is None:
        raise RuntimeError("serialize failed")

    with open(args.output, "wb") as fout:
        fout.write(serialized_engine)

```

多个context的坑

- 多个 Context 问题—device_bindings的设置
- 多Context多profile的推理代码
- 单个profile的代码，直接把13个bindings的指针拿过来填进去就行了，直接enqueuev2执行即可
- 多context怎么设置呢？
- 每个context都是绑定一个profile的
- 直接调用传递就可以了？其实是不行的
- 结论如下：
多Context多profile的engine，
engine->getNbBindings() 函数返回的是 (input_num+output_num) * profile_num

```

    /**
     * \brief Get the number of binding indices.
     */
    /**
     * There are separate binding indices for each optimization profile.
     * This method returns the total over all profiles.
     * If the engine has been built for K profiles, the first getNbBindings() / K bindings are used by profile
     * number 0, the following getNbBindings() / K bindings are used by profile number 1 etc.
     */
    /**
     * \deprecated Deprecated in TensorRT 8.5. Superseded by getNbIOTensors.
     */
    /**
     * \see getBindingIndex()
     */
    TRT_DEPRECATED int32_t getNbBindings() const noexcept
    {
        return mImpl->getNbBindings();
    }

```

- 对于单个profile的话，那么这个值是13，但是对于多context的模型而言，就不是13了，而是 (12 + 1) * 50这么多，650~
- 所以你之前的

```

void *device_bindings[13] = {rc_ids_tensor.get(), sent_ids_tensor.get(),
pos_ids_tensor.get(),
                           input_mask_tensor.get(),
                           tmp6_tensor.get(), tmp7_tensor.get(),
                           tmp8_tensor.get(), tmp9_tensor.get(),
tmp10_tensor.get(),
                           tmp11_tensor.get(), tmp12_tensor.get(),
tmp13_tensor.get(),
                           cuda_out_ptr.get()};
```

- 不能开13个了，要开650个

```
device_bindings_.resize(engine_->getNbBindings());  
  
// set the input dim  
binding_idx = start_binding_idx_;  
device_bindings_[binding_idx++] = d_buffer_;  
device_bindings_[binding_idx++] = d_buffer_ + s2_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s3_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s4_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s5_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s6_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s7_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s8_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s9_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s10_offset;  
device_bindings_[binding_idx++] = d_buffer_ + s11_offset;  
device_bindings_[binding_idx++] = d_buffer_ + out_offset;  
  
// printf("before enqueue\n");  
ret = context_->enqueueV2((void **)device_bindings_.data(), cuda_stream_, nullptr);
```

- 你还要设置一个**start_binding_idx**，每一个**context**
- 这边是把输入都放在一块显存d_buffer里面，是输入优化那边提到的，通过指针的偏移取数据
- 假设你当前是第11个context，那么这个**start_binding_idx**开始就是130了
- 当你调用第11个context的时候，那么你就把你的数据填到device_bindings_[130~142]这一块区域的**
- 你不能对应错了，因为tensorrt就是从固定的位置取值计算的，**tensorrt8.6**那边为了简化，出了个**get_device_binding_name**
- 这样根据name找不会找错，而不是index，我们现在的容易错
- 一定是多profile的，多context**
- 假如你是一个profile，开了多个context的，那没事，直接从当前这个bindings里面，就是同一块指针指向的内存，就是13个数组即可
- 绑定了其他的profile的其他的context，就不是一块内存了~
- 多个Context问题二：显存冗余怎么办？**
- 还涉及到概念，**context**是执行的单位，会维持一块显存的
- context**内是可以使用API切换profile,但耗时较长，资源的申请和释放，耗时很长的
 - 这种应该不想多个context的，所以采用下面的策略
- 一个profile对应一个context是常用的策略。
- 多context共存的情况下，往往只有一个context在工作。**
 - 你去做overlap的时候，尽量不做计算的overlap，会导致都慢了
 - 所以只有一个在工作
- 每个context都持有一份显存，比较冗余。甚至可能显存放不下**
- 使用createExecutionContext通过engine生成context示例代码如下

```

TrtLogger trt_logger;
initLibNvInferPlugins(&trt_logger.getTRTLogger(), "");
auto runtime = MakeUnique(nvinfer1::createInferRuntime(trt_logger.getTRTLogger()));
auto e = runtime->deserializeCudaEngine((void*)contents.c_str(),
                                         contents.size(), nullptr);
engine_ = MakeShared(e);
context_ = MakeShared(engine_->createExecutionContext());
context_->setOptimizationProfile(0);

```

- - 这个就是正常的infer的过程~
 - 恩培的代码目前没有设置这个setOptimizationProfile(0)，即选择第0个profile，老杜也没写过~
 - 那看来这个是多profile的情况了
 - 如果你多个profile的话，你就是for循环了

```

engine_ = MakeShared(e);
context_ = MakeShared(engine_->createExecutionContext());
context_->setOptimizationProfile(0);

context_1 = MakeShared(engine_->createExecutionContext());
context_1->setOptimizationProfile(1);
context_2 = MakeShared(engine_->createExecutionContext());
context_2->setOptimizationProfile(2);

for (int i = 0; i < 50; ++i) {
    context_->setOptimizationProfile(i);
}

```

- 不晓得讲的是哪种设置方式~大概率是上面的多个context实例~
- 做了后，你申请的每一个context都是占用一份显存的
- 有解决办法的：
- 使用ICudaEngine::createExecutionContextWithoutDeviceMemory() API来节省显存,即创建Context的时候不申请Device Memory

```

///! \brief create an execution context without any device memory allocated
///!
///! The memory for execution of this device context must be supplied by the application.
///!
IExecutionContext* createExecutionContextWithoutDeviceMemory() noexcept
{
    return mImpl->createExecutionContextWithoutDeviceMemory();
}

```

- 就是申请了context，但是不申请显存~
- 手动申请一份Device Memory 传给context::setDeviceMemory(void* memory) 使用

```


//!
/// \brief Set the device memory for use by this execution context.
//!
/// The memory must be aligned with cuda memory alignment property (using cudaGetDeviceProperties()), and its size
/// must be at least that returned by getDeviceMemorySize(). Setting memory to nullptr is acceptable if
/// getDeviceMemorySize() returns 0. If using enqueueV2()/enqueueV3() to run the network, the memory is in use from
/// the invocation of enqueueV2()/enqueueV3() until network execution is complete. If using executeV2(), it is in
/// use until executeV2() returns. Releasing or otherwise using the memory for other purposes during this time will
/// result in undefined behavior.
//!
/// \see ICudaEngine::getDeviceMemorySize() ICudaEngine::createExecutionContextWithoutDeviceMemory()
//!
void setDeviceMemory(void* memory) noexcept
{
    mImpl->setDeviceMemory(memory);
}


```

- 自己申请一份，传入~
- **这个是context的接口~**
- 可以通过**getDeviceMemorySize()**
- 比如刚才申请的50个context。对应50个profile，每一个profile的维度都是不同的，那么需要维持的显存大小也是不同的
- 可以for循环50个context，获得最大的哪个context需要的显存，那你就申请最大的~
- **假如你有两个context工作，就申请两个最大的，就从50份显存降低到2份或者一份了**
- 建议自己写一遍，查一查API~

直播里面的讲解~

- 开源给出来的初始的版本的代码

```


class TrtHepler {
public:
    TrtHepler(std::string model_param, int dev_id);

    int Forward(sample& s);

    ~TrtHepler();

private:
    int _dev_id;
    // NS_PROTO::ModelParam *_model_param_ptr;
    std::string _model_param;
    std::shared_ptr<nvinfer1::ICudaEngine> engine_;
    std::shared_ptr<nvinfer1::IExecutionContext> context_;
    cudaStream_t cuda_stream_;

    // The all dims of all inputs.
    std::vector<nvinfer1::Dims> inputs_dims_;
    std::vector<void*> device_bindings_;
};



```

- 这个TrtHepler类里面就包含了一个engine_和一个context_
- 在直播课里面的版本是做了一个改进，就是把engine和context单独封装成了一个类

```
▽ class TrtEngine {
    public:
        TrtEngine(std::string model_param, int dev_id);

        ~TrtEngine() {};

        // private:
        int dev_id_;
        // NS_PROTO::ModelParam *_model_param_ptr;
        std::string _model_param;
        std::shared_ptr<nvinfer1::ICudaEngine> engine_;

        TrtLogger trt_logger;
   };

class TrtContext {
    public:
        TrtContext(TrtEngine *engine, int profile_idx);

        int Forward(sample &s);

        ~TrtContext();

        // private:
        int dev_id_;
        // NS_PROTO::ModelParam *_model_param_ptr;
        std::string _model_param;
        std::shared_ptr<nvinfer1::ICudaEngine> engine_;
        std::shared_ptr<nvinfer1::IExecutionContext> context_;
        cudaStream_t cuda_stream_;

        // The all dims of all inputs.
        std::vector<nvinfer1::Dims> inputs_dims_;

        std::vector<char *> device_bindings_;

        char *h_buffer_;
        char *d_buffer_;
};

• 为什么context里面套了个engine ?
int max_batch_;
int min_batch_;
int max_seq_len_;
int min_seq_len_;
int start_binding_idx_;

// TrtLogger trt_logger;
};

• 在cuda-graph的版本里面，又做了一些修改~对context做了一些修改~
```

```

int CaptureCudaGraph();

int Forward(sample &s);

~TrtContext();

// private:
int dev_id_;
// NS_PROTO::ModelParam *_model_param_ptr;
std::string _model_param;
std::shared_ptr<nvinfer1::ICudaEngine> engine_;
std::shared_ptr<nvinfer1::IExecutionContext> context_;
cudaStream_t cuda_stream_;
```

// The all dims of all inputs.

```

std::vector<nvinfer1::Dims> inputs_dims_;

std::vector<char*> device_bindings_; I
std::vector<char*> host_bindings_;
```

static std::vector<char*> s_device_bindings_;

```

char *h_buffer_;
char *d_buffer_;

int max_batch_;
int max_seq_len;
int start_binding_idx_;
```

- 合并输入的代码，就是_和d_buffer_，上来就申请很大的显存，
s_device_bindings_就是根据指针偏移量去这些buffer选取数据

```

int profile_idx_;

int align_input_bytes_;
int align_aside_input_bytes_;
int whole_bytes_;

cudaGraph_t graph_;
cudaGraphExec_t instance_;
bool graph_created_ = false;

// TrtLogger trt_logger;
};
```

- 还有cudaGraph相关的变量，还有一些辅助的变量
- **profile_idx_、start_binding_idx_**这两个需要注意，PPT上面也是重点的
 - 就是多context的坑，要记录开始的binding index，根据context的index

详细的代码

- engine类的构造函数

```

TrtEngine::TrtEngine(std::string model_param, int dev_id_)
    : dev_id_(dev_id_), _model_param(model_param) {
    std::ifstream t(_model_param); // string pth
    std::stringstream buffer;
    buffer << t.rdbuf();
    std::string contents(buffer.str());
}

CUDA_CHECK(cudaSetDevice(dev_id_));

initLibNvInferPlugins(&trt_logger.getTRTLogger(), "");

auto runtime =
    MakeUnique(nvinfer1::createInferRuntime(trt_logger.getTRTLogger()));
auto e = runtime->deserializeCudaEngine((void *)contents.c_str(),
                                         contents.size(), nullptr);
engine_ = MakeShared(e);

cout << "getNbIOTensors:" << engine_->getNbIOTensors() << endl;
}

```

- 前面部分，load plan文件然后创建runtime之类的是没有什么大变化的
- 原始版本的代码：

```

TrtHepler::TrtHepler(std::string model_param, int dev_id)
    : _dev_id(dev_id), _model_param(model_param) {
{ // read model, deserializeCudaEngine and createExecutionContext
    std::ifstream t(_model_param); // string pth
    std::stringstream buffer;
    buffer << t.rdbuf();
    std::string contents(buffer.str());

    CUDA_CHECK(cudaSetDevice(_dev_id));
    CUDA_CHECK(cudaStreamCreate(&cuda_stream_));

    TrtLogger trt_logger;
    initLibNvInferPlugins(&trt_logger.getTRTLogger(), "");
    auto runtime =
        MakeUnique(nvinfer1::createInferRuntime(trt_logger.getTRTLogger()));
    auto e = runtime->deserializeCudaEngine((void*)contents.c_str(),
                                             contents.size(), nullptr);
    engine_ = MakeShared(e);
    context_ = MakeShared(engine_->createExecutionContext());
    context_->setOptimizationProfile(0);
}
}

```

- context部分的代码是有很多的变化的

```

constexpr size_t kAlignment = 128;
constexpr int ceildiv(int a, int b) {
    return (a + b - 1) / b;
}
constexpr int AlignTo(int a, int b = kAlignment) {
    return ceildiv(a, b) * b;
}

```

```
std::vector<char*> TrtContext::s_device_bindings_;
```

- 一些小函数

```

TrtContext::TrtContext(TrtEngine *trt_engine, int profile_idx) {
    profile_idx_ = profile_idx;
    engine_ = trt_engine->engine_;
    dev_id_ = trt_engine->dev_id_;
    CUDA_CHECK(cudaSetDevice(dev_id_));
    CUDA_CHECK(cudaStreamCreate(&cuda_stream_));

    context_ = MakeShared(engine_->createExecutionContext());
    // context_->setOptimizationProfileAsync(profile_idx, cuda_stream_);
    context_->setOptimizationProfile(profile_idx);

    start_binding_idx_ = profile_idx * engine_->getNbBindings() /
        engine_->getNbOptimizationProfiles();
    auto max_profile = engine_->getProfileDimensions(
        start_binding_idx_, profile_idx, nvinfer1::OptProfileSelector::kMAX);

    max_batch_ = max_profile.d[0];
    max_seq_len_ = max_profile.d[1];

    // 4 inputs: [B, S]
    align_input_bytes_ = AlignTo(max_batch_ * max_seq_len_ * sizeof(int));
    // 8 aside inputs and 1 output: [B]
    align_aside_input_bytes_ = AlignTo(max_batch_ * sizeof(int));

```

- 因为是一个context对应一个profile，由于`getNbBindings()`得到的是全部的bindings。在我们这边就是650个
- 然后我们50个profile，所以我们得到一个profile（context）就是13个bindings
- 计算出开始的binding index
- 拿到max_profile，因为我们在build的时候的profile的大小都是固定的，直接拿了max
- 然后计算所有的输入和输出的总大小

```

whole_bytes_ = align_input_bytes_ * 4 + align_aside_input_bytes_ * 9;

CUDA_CHECK(cudaMalloc(&d_buffer_, whole_bytes_));
CUDA_CHECK(cudaMallocHost(&h_buffer_, whole_bytes_));

```

- 一次申请显存和pinned memory

```

auto h_buffer_ptr = h_buffer_;
auto d_buffer_ptr = d_buffer_;

device_bindings_.resize(engine_->getNbBindings());
for (size_t i = 0; i < device_bindings_.size(); i++) {
    device_bindings_[i] = d_buffer_ptr;
}

```

- `device_bindings_`这个变量很大，每一个profile都是占用13块，假如10个profile那么这个 `device_bindings_` 的大小就是130
- 初始化的时候，这么650个指针都得是有效的值，不能是空，空值直接报错，trt8.5下面是报错的，你可以不用但是不能是空
- 这边就是随便赋值了一个起始值

```

// 4 inputs: [B, S]
int b_i = 0;

while (b_i < 4) {
    device_bindings_[start_binding_idx_ + b_i] = d_buffer_ptr;
    host_bindings_.push_back(h_buffer_ptr);

    h_buffer_ptr += align_input_bytes_;
    d_buffer_ptr += align_input_bytes_;
    I
    b_i++;
}

```

- 假如是第一个profile，那么`start_binding_idx_`就是13开始的，那就是650个指针，只用第13-25个指针

```

while (b_i < 13) {
    device_bindings_[start_binding_idx_ + b_i] = d_buffer_ptr;
    host_bindings_.push_back(h_buffer_ptr);

    h_buffer_ptr += align_aside_input_bytes_;
    d_buffer_ptr += align_aside_input_bytes_;

    b_i++;
}

```

- 拷贝两组输入，0-3和8-11~

```

vector<int> input_dim = {max_batch_, max_seq_len_, 1};
vector<int> aside_input_dim = {max_batch_, 1, 1};

int binding_idx = start_binding_idx_;
std::vector<std::vector<int>> input_dims = {
    input_dim, input_dim, input_dim, input_dim,
    aside_input_dim, aside_input_dim, aside_input_dim, aside_input_dim,
    aside_input_dim, aside_input_dim, aside_input_dim, aside_input_dim
};

```

- 由于我们profile的维度是固定的了，这样子就是伪动态的感觉，所以就直接可以放到构造函数里面做了set维度

- 插一句，原先是int TrtHepler::Forward函数里面做的：

```

std::vector<std::vector<int>> input_dims = {s.shape_info_0,
s.shape_info_1, s.shape_info_2, s.shape_info_3,
                                         s.shape_info_4,
s.shape_info_5, s.shape_info_6, s.shape_info_7,
                                         s.shape_info_8,
s.shape_info_9, s.shape_info_10, s.shape_info_11};
// set device_bindings_ and setBindingDimensions
for (size_t i = 0; i < input_dims.size(); i++) {
    std::vector<int> dims_vec = input_dims[i];
    nvinfer1::Dims trt_dims;
    trt_dims.nbDims = static_cast<int>(dims_vec.size());
    memcpy(trt_dims.d, dims_vec.data(), sizeof(int) * trt_dims.nbDims);
    context_->setBindingDimensions(binding_idx, trt_dims);
    binding_idx++;
}

if (!context_->allInputDimensionsSpecified()) {
    //gLogFatal << "context_->allInputDimensionsSpecified() error";
    std::cout << ("context_->allInputDimensionsSpecified() error") <<
    std::endl;
    assert(0);
}

// set the input dim

void *device_bindings[13] = {rc_ids_tensor.get(),
sent_ids_tensor.get(), pos_ids_tensor.get(),
                           input_mask_tensor.get(),
                           tmp6_tensor.get(), tmp7_tensor.get(),
                           tmp8_tensor.get(), tmp9_tensor.get(),
tmp10_tensor.get(),
                           tmp11_tensor.get(), tmp12_tensor.get(),
tmp13_tensor.get(),
                           cuda_out_ptr.get()};
//printf("before enqueue\n");
bool ret = context_->enqueueV2(device_bindings, cuda_stream_,
nullptr);

```

```

// set device_bindings_ and setBindingDimensions
for (size_t i = 0; i < input_dims.size(); i++) {
    std::vector<int> dims_vec = input_dims[i];
    nvinfer1::Dims trt_dims;
    trt_dims.nbDims = static_cast<int>(dims_vec.size());
    memcpy(trt_dims.d, dims_vec.data(), sizeof(int) * trt_dims.nbDims);
    context_->setBindingDimensions(binding_idx, trt_dims);
    binding_idx++;
}

```

- 所以就直接在构造函数里面调用setBindingDimensions，而无需在推理里面做了

```
if (!context_->allInputDimensionsSpecified()) {
    std::cout << ("context_->allInputDimensionsSpecified() error") << std::endl;
    assert(0);
}

for (size_t i = 0; i < device_bindings_.size(); i++) {
    s_device_bindings_.push_back(device_bindings_[i]);
}
```

- 这边的`device_bindings_`的大小是650，但是我推测，`s_device_bindings_`的大小是13
 - 感觉不是，得知道这两的区别，对理解代码很重要！
 - 后来实际我发现这个变量没用上啊.....

- 检查函数上网查下就知道干啥的了

```
// warmup
    CUDA_CHECK(cudaMemcpyAsync(d_buffer_, h_buffer_, whole_bytes_ - align_aside_input_bytes_,
                               cudaMemcpyHostToDevice, cuda_stream_));
    cudaStreamSynchronize(cuda_stream_);
}
```

- `warmup`的拷贝

```
template<class T>
void _fill(T* ptr, int size, T v) {
    for (int i = 0; i < size; i++) ptr[i] = v;
}
```

- 小函数

```

int TrtContext::CaptureCudaGraph() {
    if (graph_created_) return 1;

    // fill test inputs
    auto input_size = max_batch_ * max_seq_len_;
    _fill((int*)host_bindings_[0], input_size, 1);
    _fill((int*)host_bindings_[1], input_size, 1);
    _fill((int*)host_bindings_[2], input_size, 1);
    _fill((float*)host_bindings_[3], input_size, 1.0f);

    _fill((int*)host_bindings_[4], max_batch_, 1);
    _fill((int*)host_bindings_[5], max_batch_, 1);
    _fill((int*)host_bindings_[6], max_batch_, 1);
    _fill((int*)host_bindings_[7], max_batch_, 1);
    _fill((int*)host_bindings_[8], max_batch_, 1);
    _fill((int*)host_bindings_[9], max_batch_, 1);
    _fill((int*)host_bindings_[10], max_batch_, 1);
    _fill((int*)host_bindings_[11], max_batch_, 1);

    CUDA_CHECK(cudaMemcpyAsync(d_buffer_, h_buffer_, whole_bytes_ - align_aside_input_bytes_,
                               cudaMemcpyHostToDevice, cuda_stream_));
}

```

// warm up and let mContext do cublas initialization

```

auto status = context_->enqueueV2((void**)device_bindings_.data(), cuda_stream_, nullptr);
if (!status) {
    cerr << "Enqueue failed\n";
    exit(-1);
}

```

- 和上面类似的，貌似这边初始化bindings的时候，也是要设置随机值的，不能是空

```
CUDA_CHECK(cudaStreamBeginCapture(cuda_stream_, cudaStreamCaptureModeRelaxed));
```

```

status = context_->enqueueV2((void**)device_bindings_.data(), cuda_stream_, nullptr);
if (!status) {
    cerr << "Enqueue failed\n";
    exit(-1);
}

```

```
CUDA_CHECK(cudaStreamEndCapture(cuda_stream_, &graph_));
```

```
CUDA_CHECK(cudaStreamSynchronize(cuda_stream_));
```

```
CUDA_CHECK(cudaGraphInstantiate(&instance_, graph_, NULL, NULL, 0));
```

```
CUDA_CHECK(cudaMemcpyAsync(host_bindings_[12], device_bindings_[12], align_aside_input_bytes_,
                           cudaMemcpyDeviceToHost, cuda_stream_));
```

```
graph_created_ = true;
```

```
cout << "profile_idx=" << profile_idx_ << ", CaptureCudaGraph Done!" << endl;
```

```
return 0;
}
```

- 调用beginCapture
- 捕捉enqueuev2

- `cudaMemcpyAsync`应该也是能捕捉进来的，拷贝回去的应该也可以的
- **forward的代码**

```
struct sample{
    std::string qid;
    std::string label;
    std::vector<int> shape_info_0;
    std::vector<int> i0;
    std::vector<int> shape_info_1;
    std::vector<int> i1;
    std::vector<int> shape_info_2;
    std::vector<int> i2;
    std::vector<int> shape_info_3;
    std::vector<float> i3;
    std::vector<int> shape_info_4;
    std::vector<int> i4;
    std::vector<int> shape_info_5;
    std::vector<int> i5;
    std::vector<int> shape_info_6;
    std::vector<int> i6;
    std::vector<int> shape_info_7;
    std::vector<int> i7;
    std::vector<int> shape_info_8;
    std::vector<int> i8;
    std::vector<int> shape_info_9;
    std::vector<int> i9;
    std::vector<int> shape_info_10;
    std::vector<int> i10;
    std::vector<int> shape_info_11;
    std::vector<int> i11;
    std::vector<float> out_data;
    uint64_t timestamp;
};
```

```

int TrtContext::Forward(sample &s) {
    cudaSetDevice(dev_id_);
    int idx = 0;

    auto batch = s.shape_info_0[0];
    auto seq_len = s.shape_info_0[1];
    auto input_bytes = batch * seq_len * sizeof(int);
    auto aside_input_bytes = batch * sizeof(int);
    memcpy(host_bindings_[0], s.i0.data(), input_bytes);
    memcpy(host_bindings_[1], s.i1.data(), input_bytes);
    memcpy(host_bindings_[2], s.i2.data(), input_bytes);
    memcpy(host_bindings_[3], s.i3.data(), input_bytes);

    memcpy(host_bindings_[4], s.i4.data(), aside_input_bytes);
    memcpy(host_bindings_[5], s.i5.data(), aside_input_bytes);
    memcpy(host_bindings_[6], s.i6.data(), aside_input_bytes);
    memcpy(host_bindings_[7], s.i7.data(), aside_input_bytes);
    memcpy(host_bindings_[8], s.i8.data(), aside_input_bytes);
    memcpy(host_bindings_[9], s.i9.data(), aside_input_bytes);
    memcpy(host_bindings_[10], s.i10.data(), aside_input_bytes);
    memcpy(host_bindings_[11], s.i11.data(), aside_input_bytes);
}

```

cudaEvent_t start, stop;
float elapsed_time = 0.0;

//cudaStreamSynchronize(cuda_stream_);
//cudaEventCreate(&start);
//cudaEventCreate(&stop);
//cudaEventRecord(start, 0);

- 先把12个输入拷贝到host的bindings里面
- 我们刚才的host_bindings_ 和device_bindings_都是根据偏移量算好的，二者的偏移量是相同的

```
CUDA_CHECK(cudaMemcpyAsync(d_buffer_, h_buffer_, whole_bytes_ - align_aside_input_bytes_,
                           cudaMemcpyHostToDevice, cuda_stream_));
```

- 调用一行拷贝把数据拷贝到device buffer，无需for循环12次拷贝了
- 这边有个问题，h_buffer_哪里赋值了？**
- 貌似和host_bindings_ 绑定起来了，但我没找到代码
- 好像就在上面循环13次那~

```
vector<int> v_data(128);
cudaMemcpyAsync(v_data.data(), device_bindings_[13], 128 * sizeof(int),
               cudaMemcpyDeviceToHost, cuda_stream_);
cudaStreamSynchronize(cuda_stream_);
```

- 冗余的测试代码，可能~

```
if (graph_created_) {
    CUDA_CHECK(cudaGraphLaunch(instance_, cuda_stream_));
} else {

    // printf("before enqueue\n");
    auto status = context_->enqueueV2((void**)device_bindings_.data(), cuda_stream_, nullptr);
    if (!status) {
        cerr << "Enqueue failed\n";
        exit(-1);
    }

}
```

- 如果graph已经被捕捉了，那么我们直接launch即可，否则就是传统的enqueuev2

```
s.out_data.resize(batch);
//memcpy(s.out_data.data(), host_bindings_[12], batch * sizeof(float));
CUDA_CHECK(cudaMemcpyAsync(s.out_data.data(), device_bindings_[start_binding_idx_ + 12], batch * sizeof(float),
                           cudaMemcpyDeviceToHost, cuda_stream_));
cudaStreamSynchronize(cuda_stream_);

//cudaEventRecord(stop, 0);
//cudaEventSynchronize(stop);
//cudaEventElapsedTime(&elapsed_time, start, stop);

//cout << "batch=" << max_batch_ << ", seq_len=" << max_seq_len_
//<< ", enqueue time=" << elapsed_time << "ms" << endl;
//cudaEventDestroy(start);           I
//cudaEventDestroy(stop);

// 计算当次推理结束的时间戳
struct timeval tv;
gettimeofday(&tv, NULL);
s.timestamp = tv.tv_sec * 1000000 + tv.tv_usec;
}
```

- 在把结果拷贝回来，还有计时的代码~

调用端调用——ernie_infer_demo.cpp

```

    int main(int argc, char *argv[]) {
        if (argc != 4) {
            cout << "ERROR: argc != 4 && argc != 11" << endl;
            return -1;
        }
        // nvinfer1::plugin::hello();
        // init
        int argc_idx = 1;
        string model_file = argv[argc_idx++];
        std::string test_file = argv[argc_idx++];
        std::string out_file = argv[argc_idx++];

        std::shared_ptr<TrtEngine> trt_engine(new TrtEngine(model_file, 0));
        //auto trt_engine = new TrtEngine(model_file, 0);
        // auto trt_context = new TrtContext(trt_engine, 0);

        vector<int> batchs{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        vector<int> seq_lens{1, 32, 64, 96, 128};
        //vector<int> batchs{1};
        //vector<int> seq_lens{64, 128};
        auto context_num = batchs.size() * seq_lens.size();

        assert(trt_engine->engine_->getNbOptimizationProfiles() == context_num);
    }

```

- 生成一个engine

```

vector<std::shared_ptr<TrtContext>> trt_contexts(context_num);
for (size_t i = 0; i < context_num; i++) {
    auto context = new TrtContext(trt_engine.get(), i);
    trt_contexts[i].reset(context);
}

```

- 根据profile number生成多个context放到数组里面

```

for (size_t i = 0; i < context_num; i++) {
    trt_contexts[i]->CaptureCudaGraph();
}

```

- 给每一个context捕捉cuda graph

- 所以这边就是感觉是随机数据即可，你只要保证地址是一样的（指针），数据可以每次不一样，数据的shape一样，地址一样应该就行

- 但是需要padding吧~这个应该也是预处理，后面会有吗？

```
// preprocess
std::string aline;
std::ifstream ifs;
ifs.open(test_file, std::ios::in);
std::ofstream ofs;
ofs.open(out_file, std::ios::out);
std::vector<sample> sample_vec;
while (std::getline(ifs, aline)) {
    sample s;
    line2sample(aline, &s);
    sample_vec.push_back(s);
}
```

- 读取一行输入

```
// inference
int idx = 0;
for (auto &s : sample_vec) {
    int batch = s.shape_info_0[0];
    int seq_len = s.shape_info_0[1];

    int s_idx = 0;
    for (int i = 0; i < seq_lens.size(); i++) {
        if (seq_len <= seq_lens[i]) {
            s_idx = i;
            break;
        }
    }

    auto batch_idx = batchs[batch - 1] - 1;
    int context_idx = batch_idx * seq_lens.size() + s_idx;

    trt_contexts[context_idx]->Forward(s);

    idx++;
    if (idx % 100 == 0) cout << "Forward " << idx << endl;
    //break;
}
```

- 根据seq-len判断是第几个profile，还得根据batch判断吧
- padding？看来之前的理解错了啊？
- 也没错，s_idx是padding的
- seq_len的长度是5，分别是1, 32, 64, 96, 128
- 如果当前的seq-len为28，batch为2
 - 那么会跳过[1,32],[1,64]...[1,128]这五个profile，每一个batch都是我五个
 - 然后batch=2的时候，seq-len<32，则i=1
 - 那么batch_idx * 5 + 1就是当前的profile id，也就是context id~

- postprocess

```
// postprocess
for (auto& s : sample_vec) {
```

```
    std::ostringstream oss;
    oss << s.qid << "\t";
    oss << s.label << "\t";
    for (int i = 0; i < s.out_data.size(); ++i) {
        oss << s.out_data[i];
        if (i == s.out_data.size() - 1) {
            oss << "\t";
        } else {
            oss << ",";
        }
    }
    oss << s.timestamp << "\n";
    ofs.write(oss.str().c_str(), oss.str().length());
}
ofs.close();
ifs.close();
return 0;
}
```

- 重点就是device_binding, 以及偏移量

tips

- 在trtexec里面，可以直接加上--useCudaGraph， 默认是不打开的
- 使用多个profile的话，trt的构建过程将会变得十分的漫长，耗时越大，同时对内存的需求也越高，正常我的电脑，profile这个模型大概30个profile，用将近20G的内存~
- 生成engine之后，模型的大小也翻倍了，都是FP16的情况下，从251.2MB翻倍到612.5MB
- 此时，你直接拿原来的代码的话，会发现core_dump，就是在context_->setBindingDimensions(binding_idx, trt_dims);
 - 之前上课的时候说过的，这个接口确实用不起来了
 - 目前代码实践完成了，精度也许是因为128的seq-len和我的profile为64导致的问题，过多个profile会导致内存不够，我只有32G内存
- 这个版本的代码还剩一个节省显存的操作后续考虑尝试