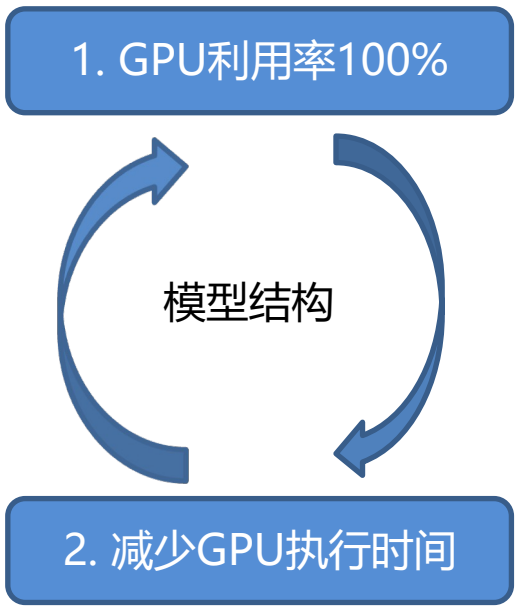
模型性能优化方法

2022年11月10日

单机训练性能优化

深度学习单机框架性能优化的本质：充分压榨AI加速芯片算力（GPU）



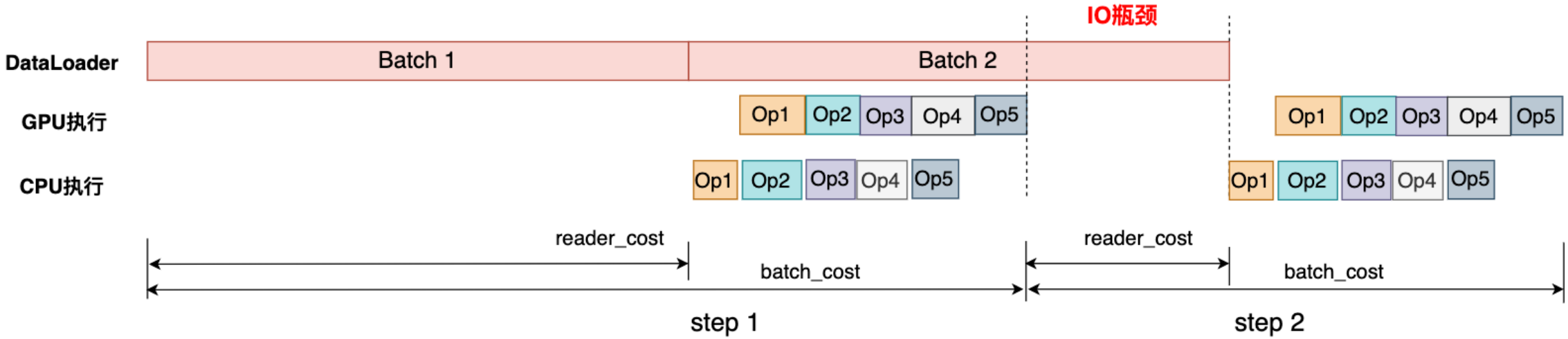
框架关键点	解决方案
数据读取	数据预处理、**异步读取**、**并行读取**、C++读取、GPU读取
数据拷贝	**异步拷贝**、**控制设备切换**
逻辑调度	减少锁操作、细粒度并发、降低线程开销
场景切换	C++接口绑定、减化python接口内逻辑
低精度运算	**混合精度策略**
图优化(编译优化)	**OP Fusion**、自动Fusion、代码生成
OP计算优化	**GPU Kernel优化**、**第三方库优化**

训练性能两大指标

- Time2train，模型训练收敛所需时间
  - MLPerf Training Rules，MLCommons Training v1.0 Results

- IPS (Instances Per Second)，每秒处理的样本数
  - 计算公式：$\text{ips} = \frac{\text{训练N步处理的样本数}}{\text{训练N步的总时间}}$
  - 如images/sec、sequences/s、tokens/s、words/sec、frames/sec等
  - NVIDIA Data Center Deep Learning Product Performance

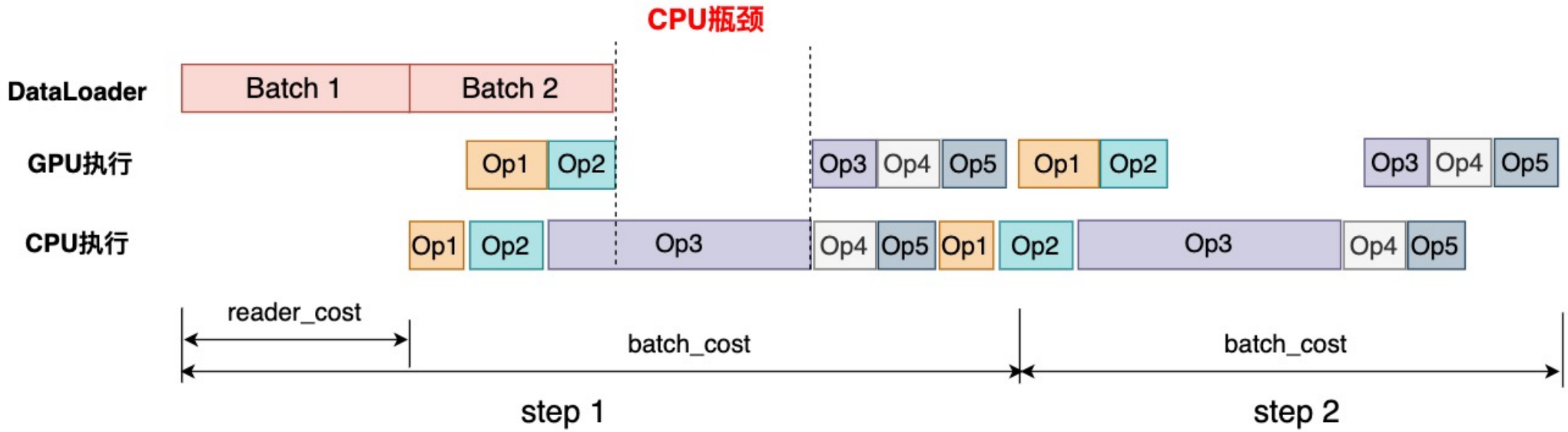
理解训练的3类瓶颈



I/O瓶颈

在模型训练流程中，首先会将磁盘上存储的数据样本读取到CPU内存中，一般还会在CPU上完成数据的预处理，最后将处理好的数据从CPU内存拷贝到GPU内存中，由GPU完成计算。当数据处理过程较慢，则会引起GPU计算等待（图中 reader_cost 部分），造成I/O瓶颈

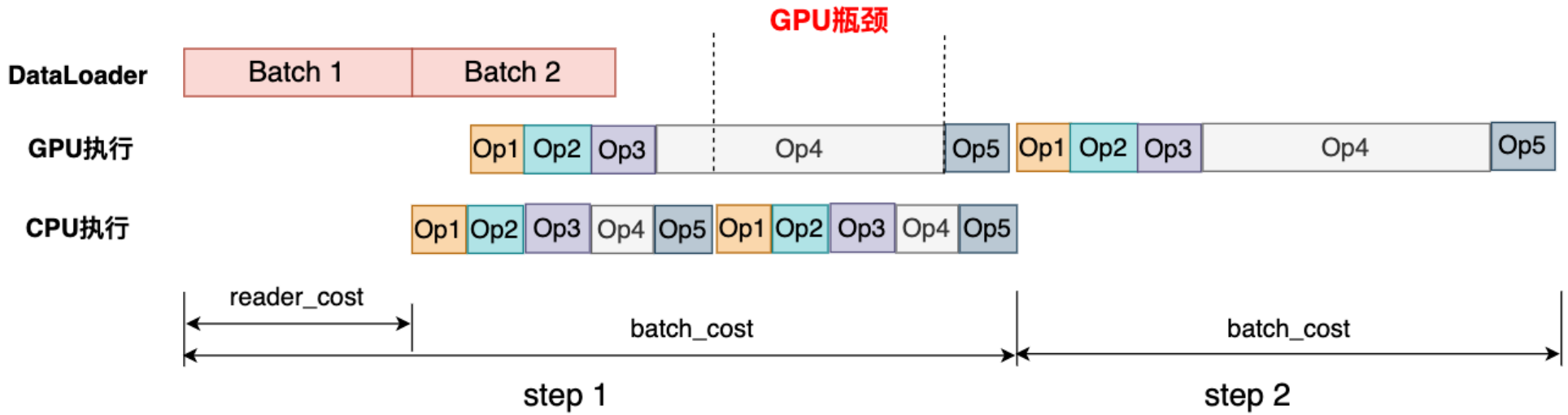
理解训练的3类瓶颈



CPU瓶颈

由于 GPU 运行速度较快，如果CPU端处理任务耗时太久，造成GPU等待，就会拖慢训练速度。Op3的CPU时间过高，导致了这段时间内GPU在等待CPU处理结束，从而影响了性能

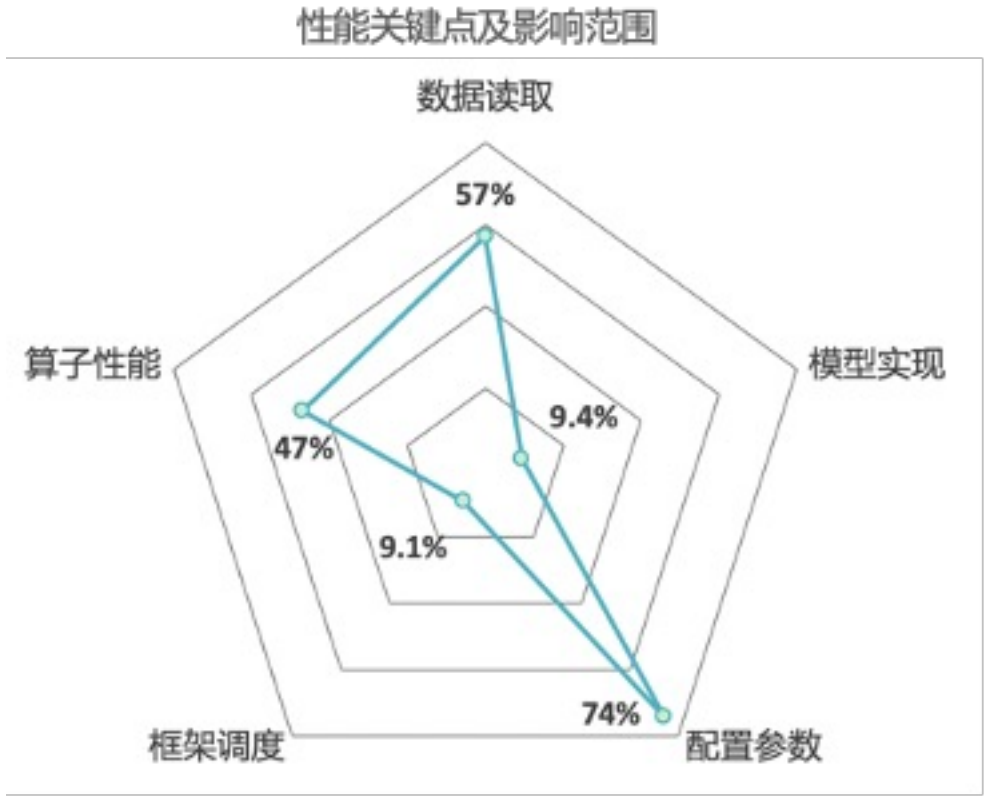
理解训练的3类瓶颈

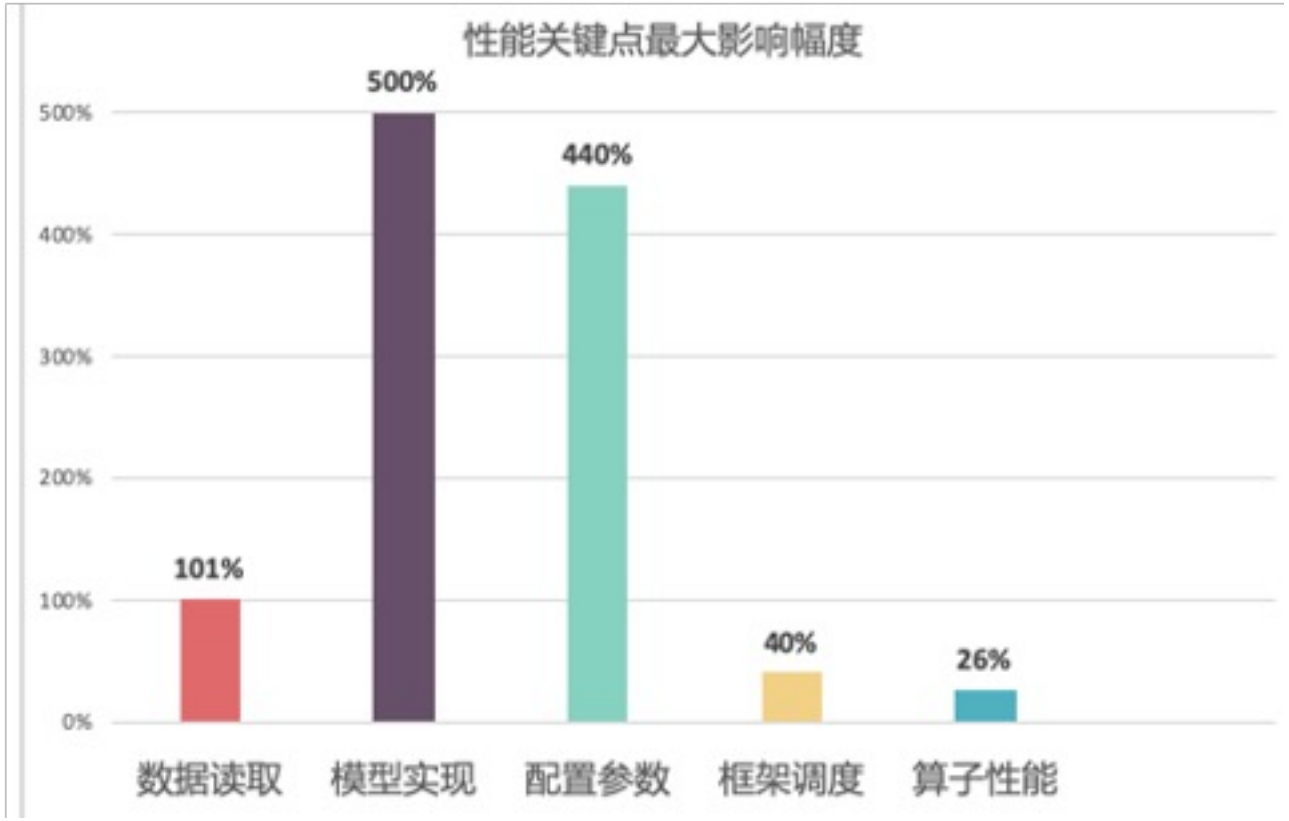


GPU瓶颈

虽然GPU提供了很高的算力，但如果GPU算子实现不够高效，也会导致性能较差。如Op4的GPU时间明显很长，可能存在较多的优化空间

影响性能的5个关键点


性能关键点及影响范围
数据读取
57%
模型实现
9.4%
算子性能
47%
9.1%
框架调度
74%
配置参数


性能关键点最大影响幅度
500%
400%
300%
200%
100%
0%
101%
500%
440%
40%
26%
数据读取
模型实现
配置参数
框架调度
算子性能

影响性能的5个关键点

如何压缩batch_cost?

（1）数据读取：框架提供了多进程数据读取的功能，计算机视觉类的模型在数据的预处理上也会耗费较多时间，可以使用高性能的预处理库DALI

（2）配置参数：提升模型在GPU上的训练性能，其目标是使GPU的利用率达到100%。尽可能地设置较大的Batch Size、使用混合精度训练时使Tensor的尺寸设置满足加速条件限制

（3）算子性能：在各种硬件、各种尺寸和数据类型的计算场景下，算子使用一种算法很难满足最佳性能求

（4）框架调度：一些模型GPU计算开销不大，性能主要受到CPU开销的影响，框架执行调度的开销需要压缩

（5）模型实现：一些模型本身实现不够高效，但这一点很容易被开发者忽略

通用调优方法一：多进程异步数据加载

- 场景：I/O瓶颈，reader_cost较高
- 方法：
  - DataLoader使用多进程异步方式读取数据（设置num_workers > 0，默认值为0）
  - DataLoader使用shared memory（设置use_shared_memory = True，默认值为True）
- 案例：
  - DeepLabv3+模型将num_workers从2增加到8，性能提升1倍

num_workers=2， reader_cost / batch_cost = 0.5

```
epoch: 1, iter: 20/500, loss: 1.7898, lr: 0.009657, batch_cost: 0.4842, reader_cost: 0.23764, ips: 8.2615 samples/sec | ETA 00:03:52
epoch: 1, iter: 30/500, loss: 1.4706, lr: 0.009476, batch_cost: 0.5069, reader_cost: 0.25629, ips: 7.8912 samples/sec | ETA 00:03:58
epoch: 1, iter: 40/500, loss: 1.0206, lr: 0.009295, batch_cost: 0.4977, reader_cost: 0.25142, ips: 8.0372 samples/sec | ETA 00:03:48
```

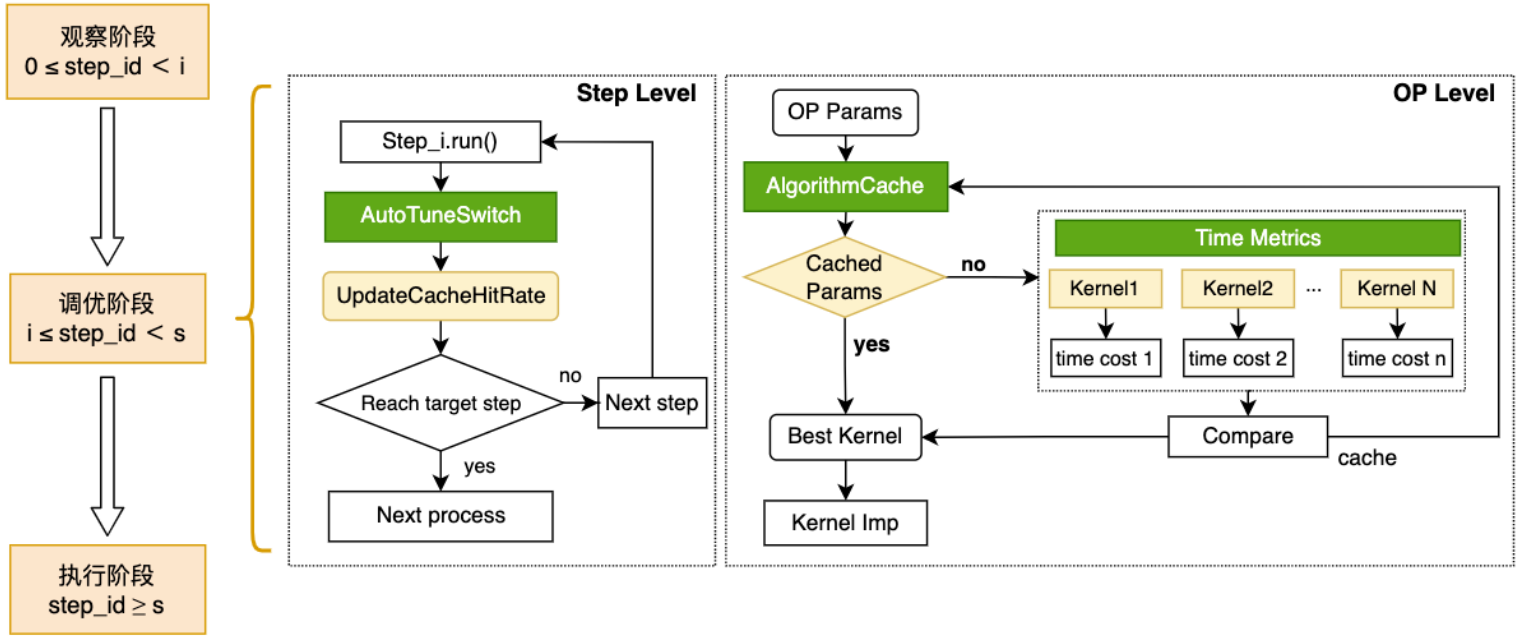
num_workers=8， reader_cost / batch_cost = 0.0006

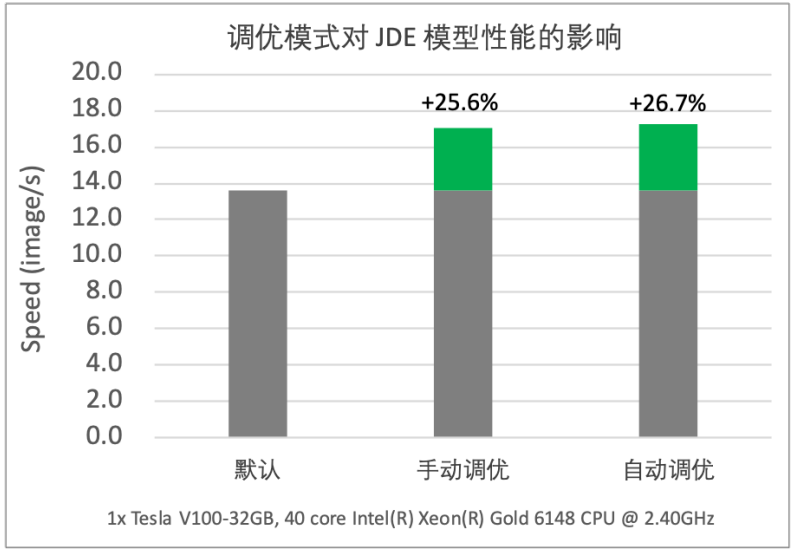
```
epoch: 1, iter: 20/500, loss: 1.6476, lr: 0.009657, batch_cost: 0.2332, reader_cost: 0.00078, ips: 17.1497 samples/sec | ETA 00:01:51
epoch: 1, iter: 30/500, loss: 1.5144, lr: 0.009476, batch_cost: 0.2376, reader_cost: 0.00014, ips: 16.8328 samples/sec | ETA 00:01:51
epoch: 1, iter: 40/500, loss: 1.4258, lr: 0.009295, batch_cost: 0.2341, reader_cost: 0.00047, ips: 17.0885 samples/sec | ETA 00:01:47
```

注意：设置过小会导致速度较慢，设置过多内存开销大，CPU负担较重，最优值需要调节

通用调优方法二：多版本Kernel自动选择

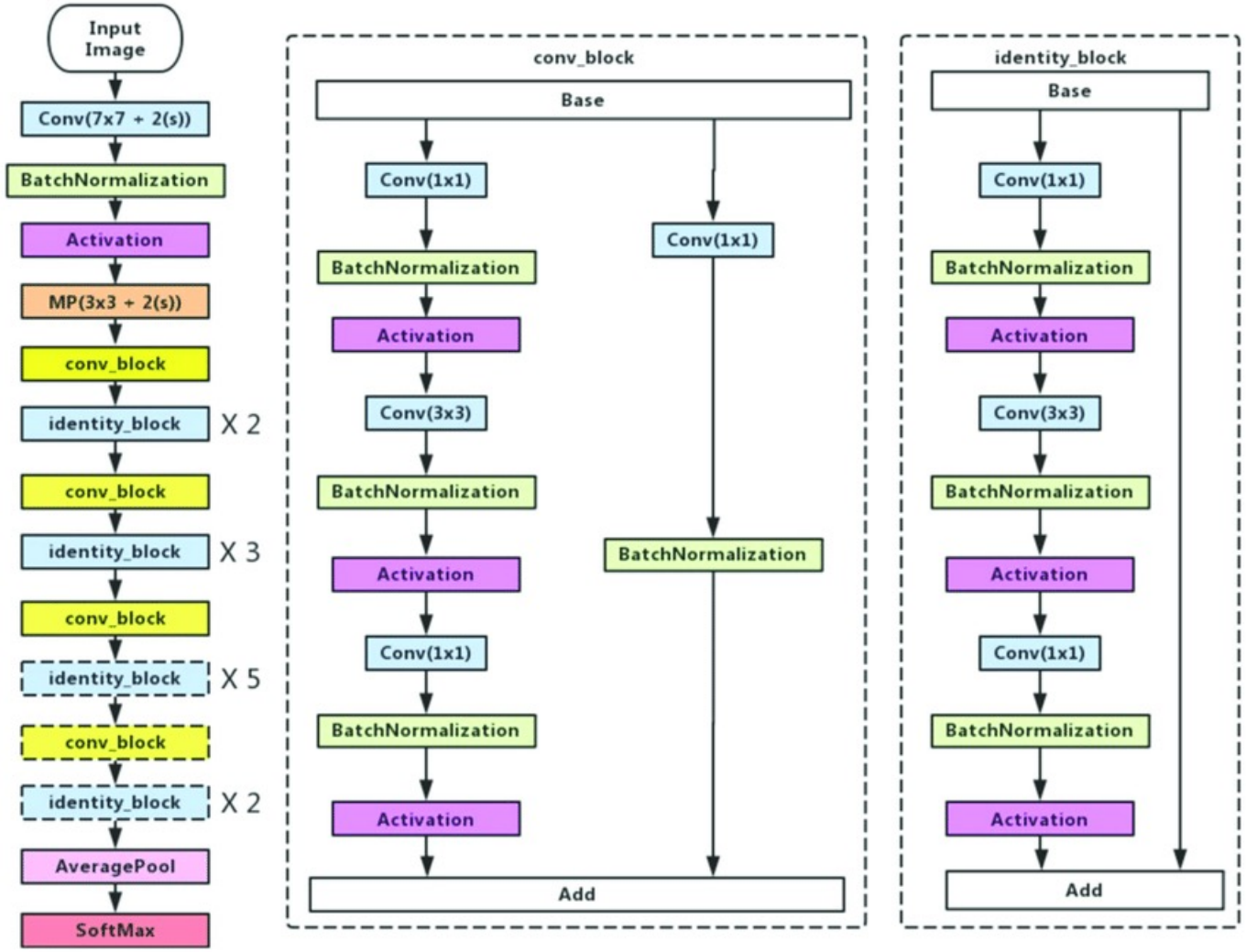
- 场景：同一个算子在不同的配置下最优的算法不同，如卷积
- 方法：
  - 启发式搜索：默认方法，根据经验选择，选择的开销更低，但选到的卷积算法不一定最优
  - 多版本 Kernel自动选择：分为观察阶段、优化阶段、执行阶段组成





通用调优方法三：自动算子融合

场景：满足固定匹配模式的网络



案例：ResNet50或者以ResNet为backbone的模型

- add + relu
- bn + add + relu
- bn + relu几种模式进行融合

性能都有约6%+的提升。

通用调优方法四：自动混合精度训练（AMP）

场景：大量使用卷积、Matmul的模型； 运行在Volta及Turing架构GPU

示例（动态图）：

```
model = SimpleNet(input_size, output_size)  # 定义模型
optimizer = paddle.optimizer.SGD(learning_rate=0.0001, parameters=model.parameters())  # 定义优化器
# Step1：定义 GradScaler，用于缩放loss比例，避免浮点数溢出
scaler = paddle.amp.GradScaler(init_loss_scaling=1024)
start_timer() # 获取训练开始时间
for epoch in range(epochs):
    datas = zip(train_data, labels)
    for i, (data, label) in enumerate(datas):
        # Step2：创建AMP上下文环境，开启自动混合精度训练
        with paddle.amp.auto_cast():
            output = model(data)
            loss = mse(output, label)
        # Step3：使用 Step1中定义的 GradScaler 完成 loss 的缩放，用缩放后的 loss 进行反向传播
        scaled = scaler.scale(loss)
        scaled.backward()
        # 训练模型
        scaler.minimize(optimizer, scaled)
        optimizer.clear_grad()
```

用法参考：混合精度训练-动态图，混合精度训练-静态图

通用调优方法四：自动混合精度训练（AMP）

- **CV模型加速要点**
  - 卷积的输入/输出通道数为8的倍数
  - FC层的size参数设置为8的倍数
  - 使用NHWC layout

Table 1. Tensor Core requirements by cuBLAS or cuDNN version for some common data precisions. These requirements apply to matrix dimensions M, N, and K.

Tensor Cores can be used for...	cuBLAS version < 11.0 cuDNN version < 7.6.3	cuBLAS version ≥ 11.0 cuDNN version ≥ 7.6.3
INT8	Multiples of 16	Always but most efficient with multiples of 16; on A100, multiples of 128.
FP16	Multiples of 8	Always but most efficient with multiples of 8; on A100, multiples of 64.
TF32	N/A	Always but most efficient with multiples of 4; on A100, multiples of 32.
FP64	N/A	Always but most efficient with multiples of 2; on A100, multiples of 16.

- **NLP模型加速要点**
  - CUDA10需严格遵循维度要求，矩阵乘输入维度[M, K]和[K, N]，应满足8的倍数
  - CUDA11不满足8的倍数也能使用TensorCore加速，但满足8的倍数更高效
  - NVIDIA详细说明：Tensor Core Requirements

V100, CUDA10.1 cuDNN7.6.5, MatMul性能测试结果

x.shape [*, M, K]	y.shape [K, N]	FP32	FP16	加速比 FP32 / FP16	是否使用 TensorCore加速	说明
[16, 36, 1504]	[1504, 10000]	1.340 ms	0.267 ms	5.02x	✅	M,K,N均为8的倍数
[16, **35**, 1504]	[1504, 10000]	1.330 ms	0.264 ms	5.04x	✅	M不是8的倍数
[16, 36, **1500**]	[**1500**, 10000]	1.433 ms	1.367 ms	1.05x	❌	K不是8的倍数
[16, 36, 1504]	[1504, **9999**]	1.406 ms	1.388 ms	1.01x	❌	N不是8的倍数

使用Nsight分析：基本用法

NVIDIA Nsight Systems是系统级别的调优工具，在GPU训练中常用来获取以下信息

Compute：
- CUDA API. Kernel launch and execution correlation
- Libraries: cuBLAS, cuDNN, etc.

OS Thread state and CPU utilization, pthread, etc.

User annotations API (NVTX)
- 在程序中加入NVTX标识，更好地理解模型中时间的分布
- Paddle支持

安装和使用：Nsight Systems User Guide

1）在安装了nsight system的运行示例命令：
nsys profile --stats=true -t cuda,nvtx -c cudaProfilerApi -o model.qdrep -f true python train.py

2）将输出的.qdrep文件导入NSight GUI工具中查看timeline

使用Nsight分析：基本用法

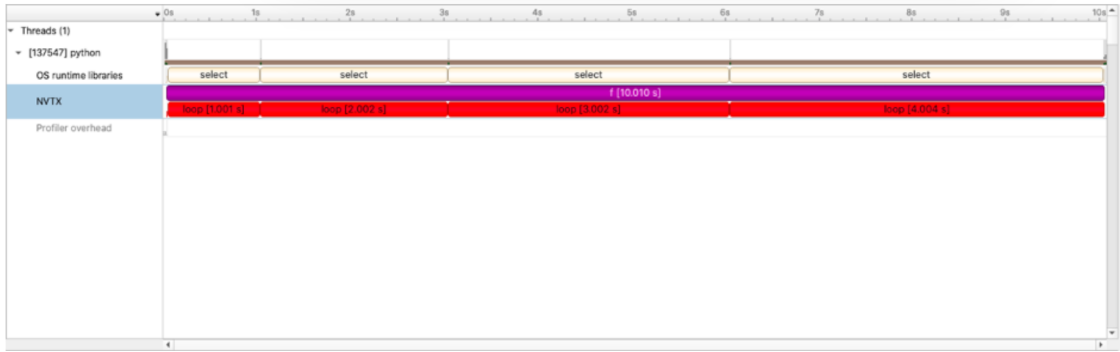
NVTX is a code annotation tool and can be used to mark functions or chunks of code. Python developers can either use decorators `@nvtx.annotate()` or a context manager `with nvtx.annotate(..):` to mark code to be measured. For example:

```
import time
import nvtx


@nvtx.annotate("f()", color="purple")
def f():
    for i in range(5):
        with nvtx.annotate("loop", color="red"):
            time.sleep(i)

f()
```

Python程序使用NVTX工具标记

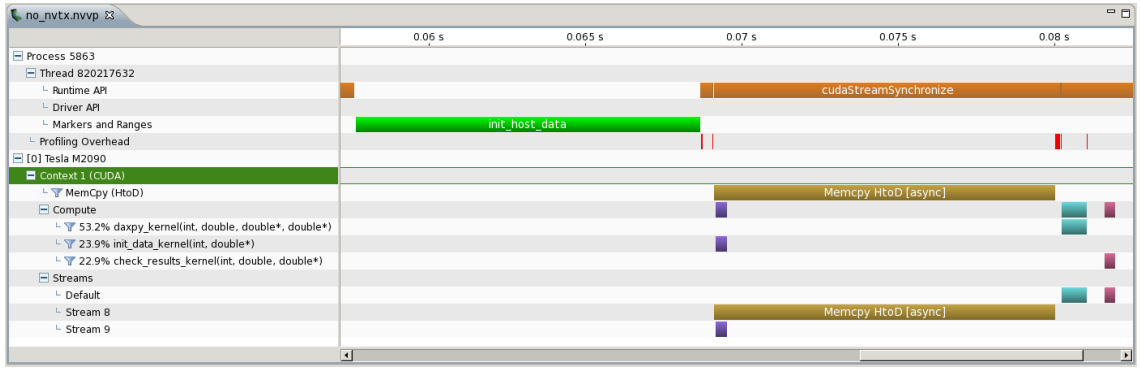


Use NVTX like any other C library: include the header "nvToolsExt.h", call the API functions from your source and link the NVTX library on the compiler command line with -lnvToolsExt.

To see the duration of init_host_data you can use nvtxRangePushA and nvtxRangePop:

```
#include "nvToolsExt.h"
...
void init_host_data( int n, double * x ) {
    nvtxRangePushA("init_host_data");
    //initialize x on host
    ...
    nvtxRangePop();
}
...
```

C++程序使用NVTX工具标记



代码和图片引用自NVIDIA官网

使用Nsight分析：Paddle基本用法

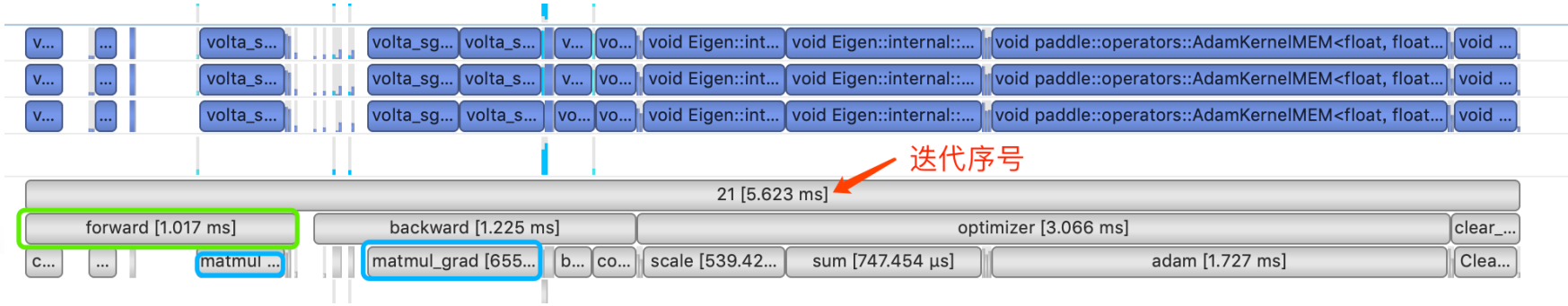
```
for i in range(max_steps):
    if i == 20:
        core.nvprof_start()
        core.nvprof_enable_record_event()
        core.nvprof_nvtx_push(str(i))
    if i == 30:
        core.nvprof_nvtx_pop()
        core.nvprof_stop()
        sys.exit()
    if i > 10 and i < 30:
        core.nvprof_nvtx_pop()
        core.nvprof_nvtx_push(str(i))
    image = paddle.to_tensor(np.random.random((batch_size, 3, 112, 112)), dtype='float32')
    label = paddle.to_tensor(np.random.randint(0, 1000, (batch_size, 1)), dtype='int64')
    core.nvprof_nvtx_push("forward")
    out = model(image)
    loss = nn_loss(out, label)
    core.nvprof_nvtx_pop()
    core.nvprof_nvtx_push("backward")
    loss.backward()
    core.nvprof_nvtx_pop()
    core.nvprof_nvtx_push("optimizer")
    adam.step()
    core.nvprof_nvtx_pop()
    core.nvprof_nvtx_push("clear_grad")
    adam.clear_grad()
    core.nvprof_nvtx_pop()
```

在第20个step激活性能分析器

显示event名称，如OP的name

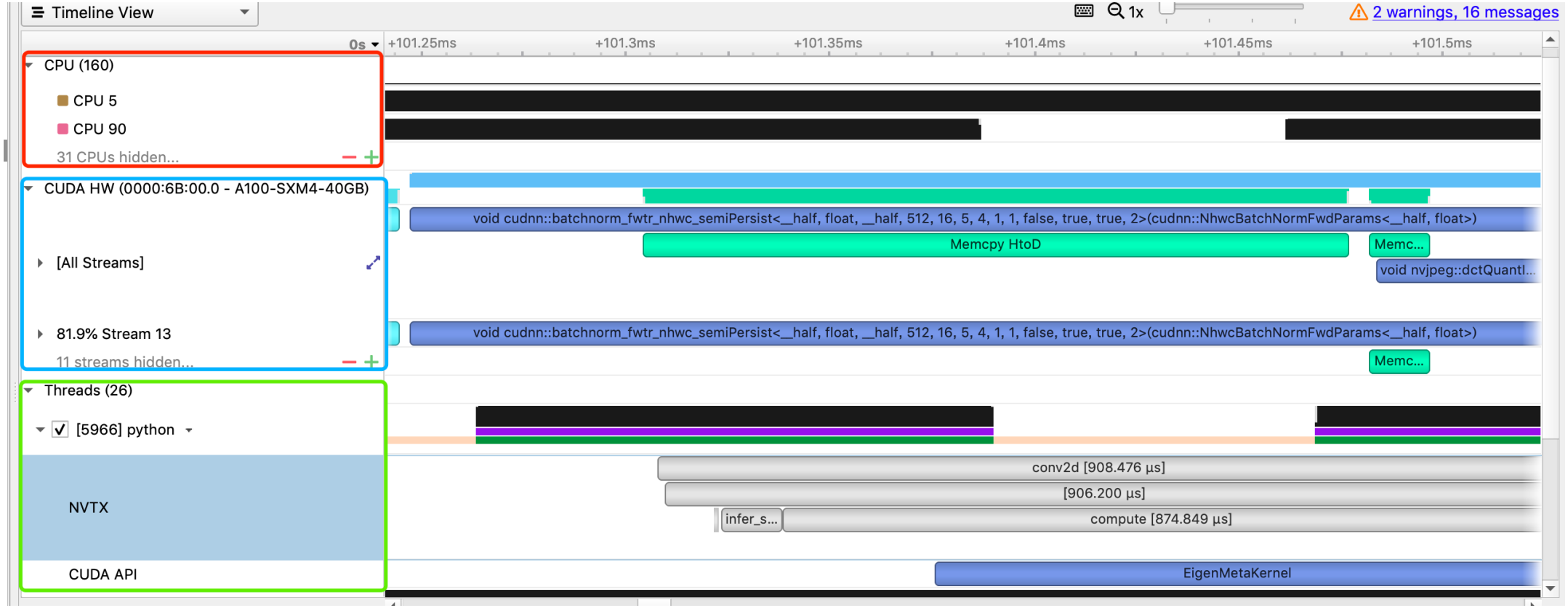
在第30个step停止性能分析

标记特定过程，如forward等



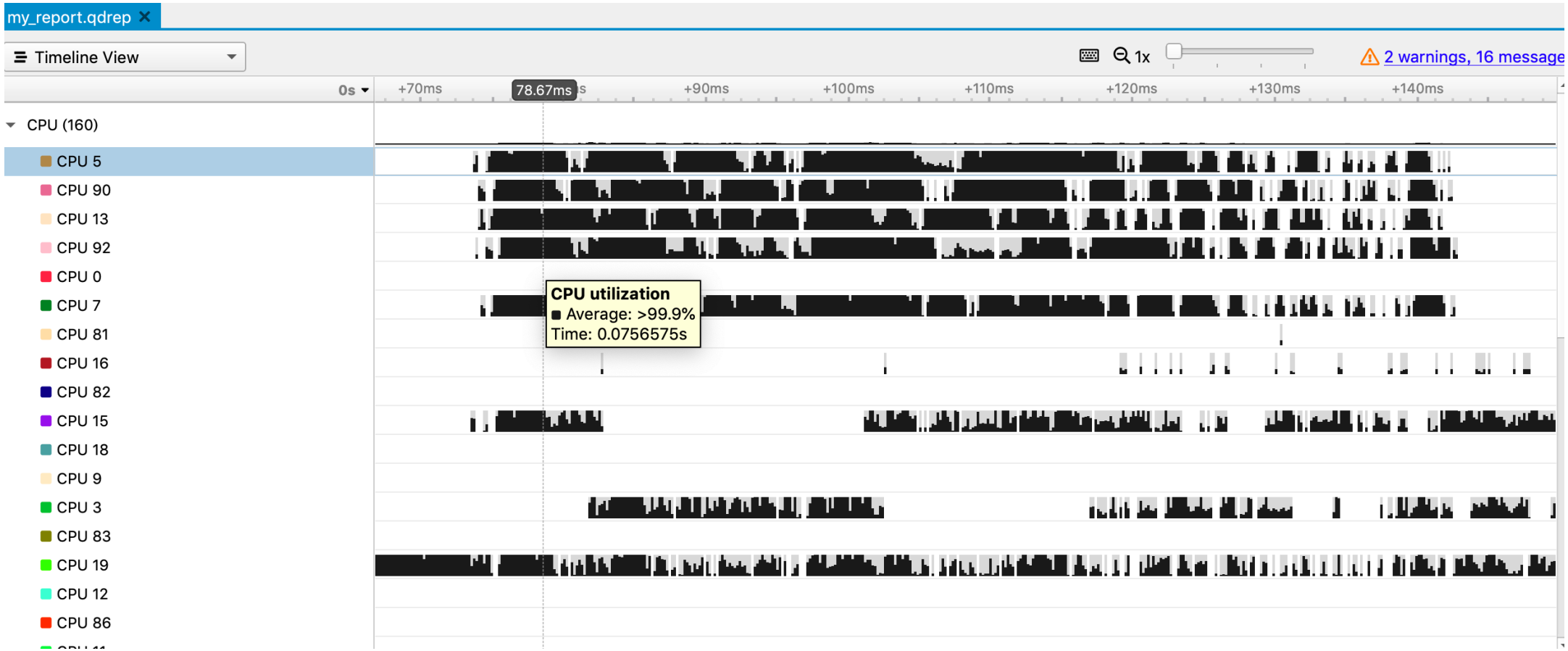
使用Nsight分析：Timeline解读

在Timeline上所有的事件被大体组织为3层内容：**CPU**、**CUDA**、**Threads**



使用Nsight分析：Timeline解读

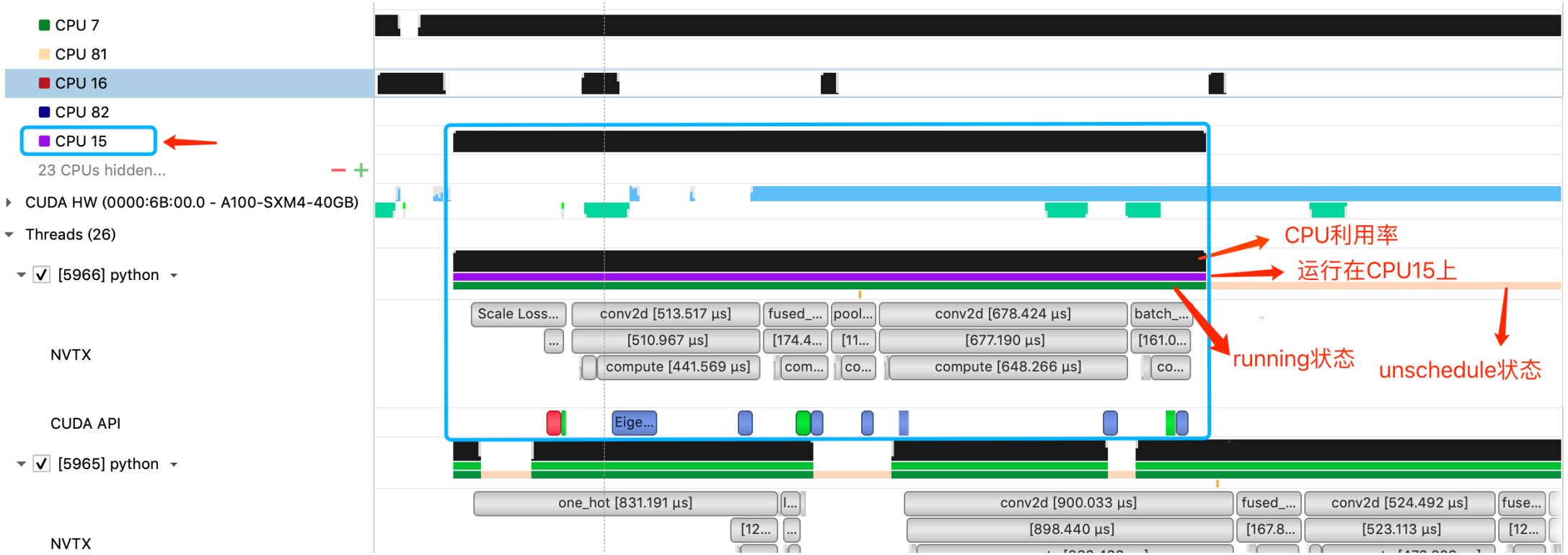
CPU：左侧使用不同颜色标记不同的core，右侧能看到对应的使用情况



使用Nsight分析：Timeline解读

Threads

- 运行在哪个CPU core上，以及对应的利用率
- 线程状态：running、unscheduled、waiting
- OS runtime libraries usage: pthread, etc.
- API usage: CUDA, cuDNN, cuBLAS, etc.

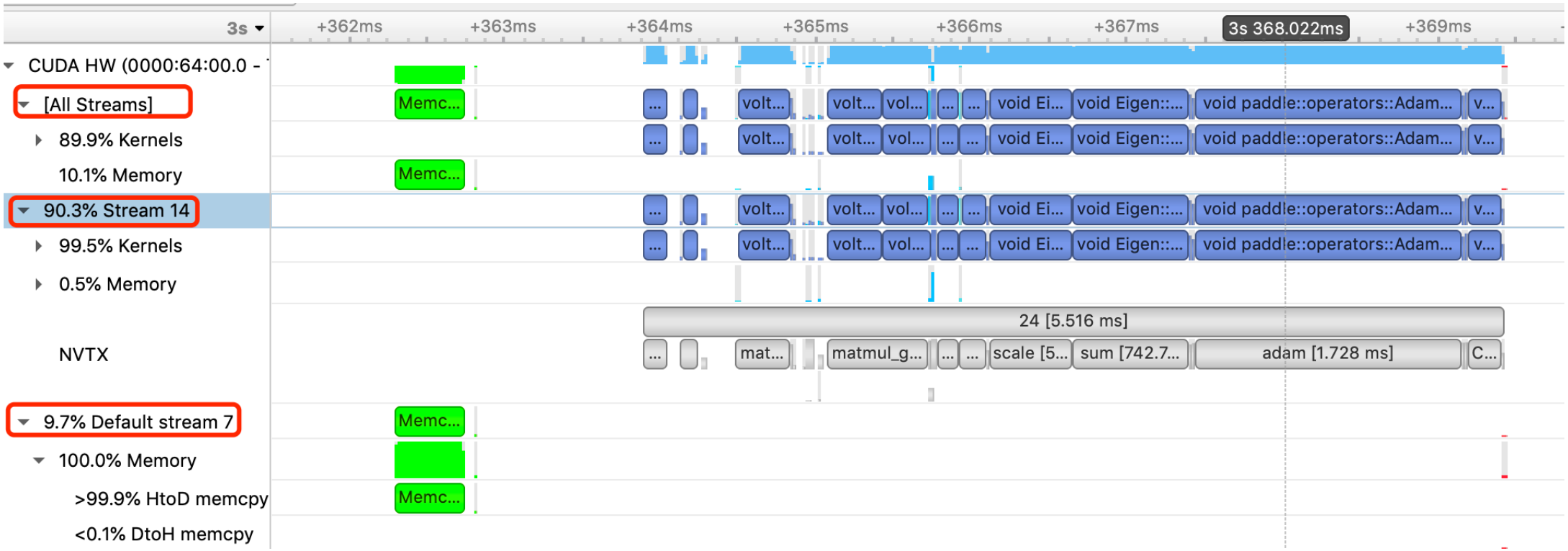


使用Nsight分析：Timeline解读

CUDA：所有事件组织在不同的stream中，可以观察到GPU Kernel执行时间，定位GPU空闲时间

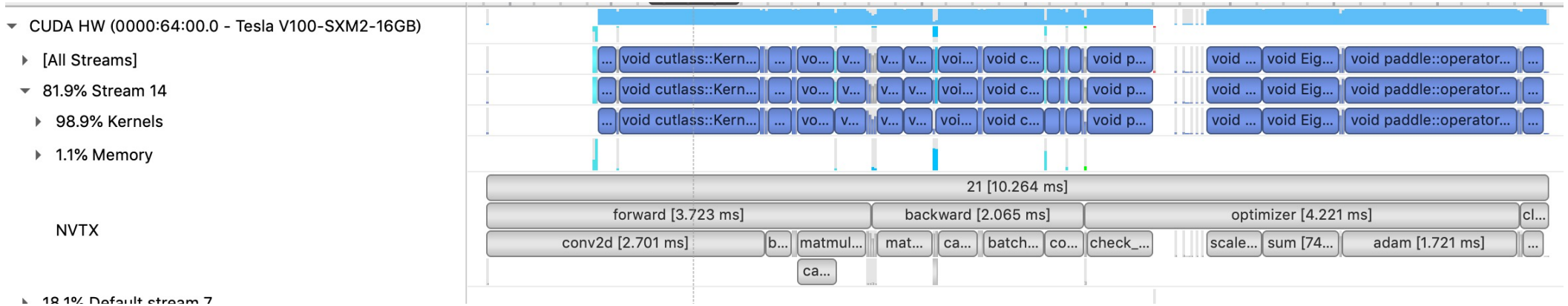
如下案例

- All Stream：所有stream上的事件被合并在一起展示
- Stream 14：计算的stream，即OP Kernel的调用
- Defaul stream 7：数据拷贝stream，如HtoD、DtoH

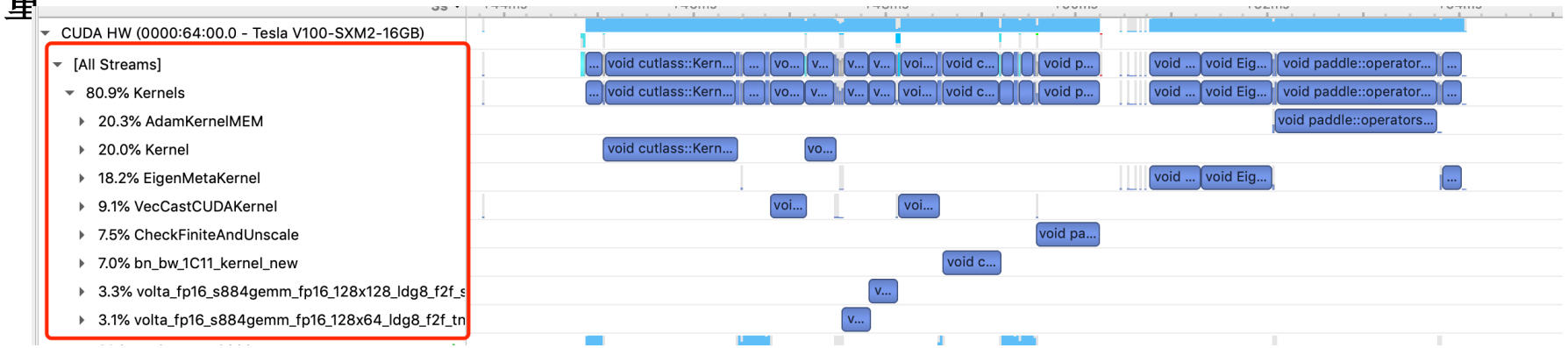


使用Nsight分析：模型使用技巧

（1）通过NVTX标记可以大致确定训练中哪些阶段耗时较高

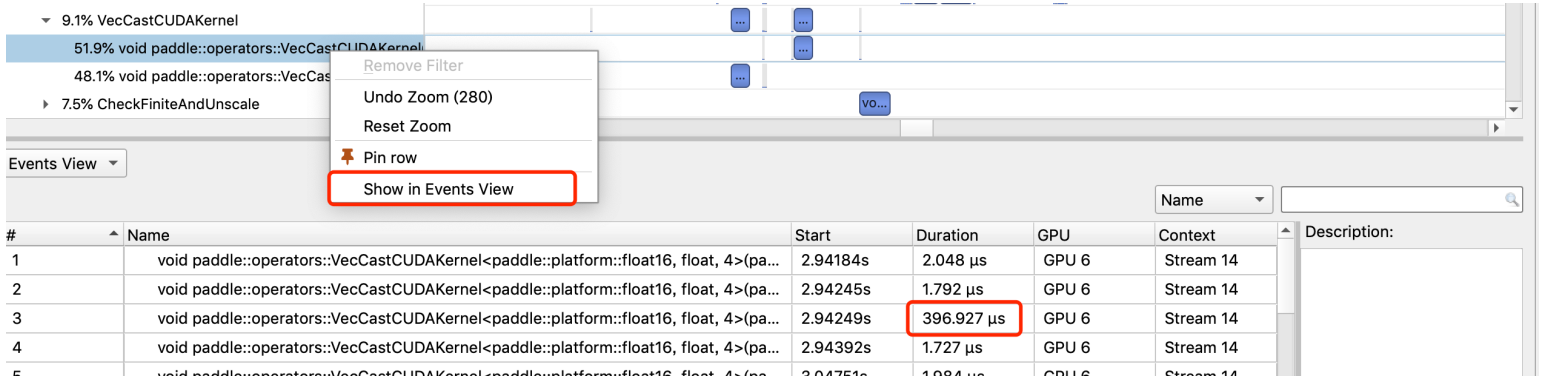


（2）通过Kernel占比，锁定占比较高的Kernel，但无法确定到底是哪种配置（输入、参数）下耗时严重

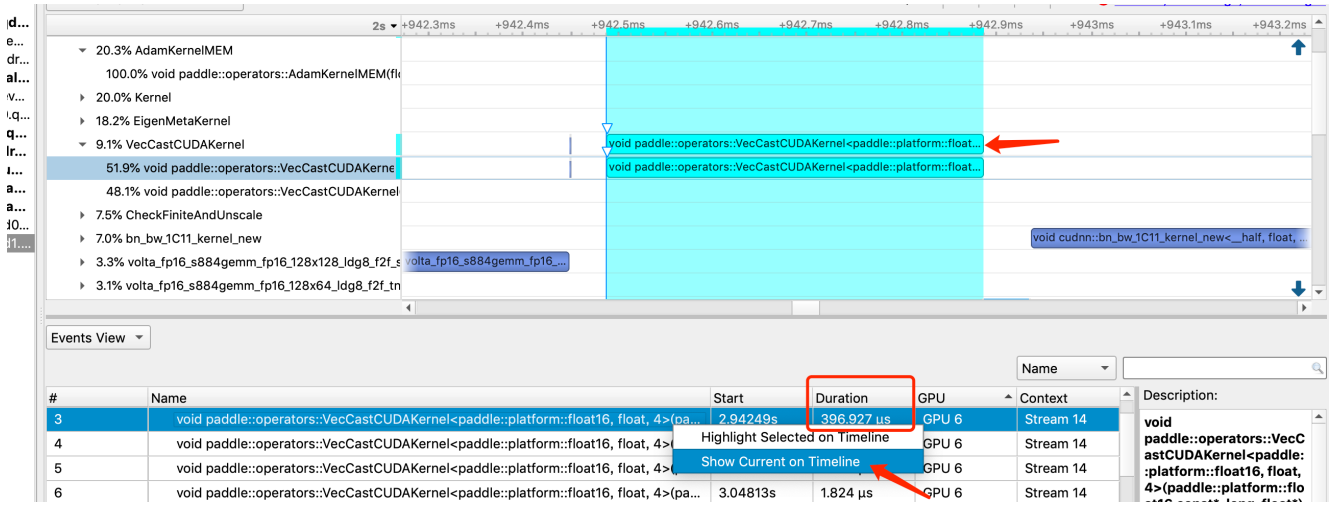


使用Nsight分析：模型使用技巧

（3）通过同一种Kernel的耗时，确定有没有异常的情况

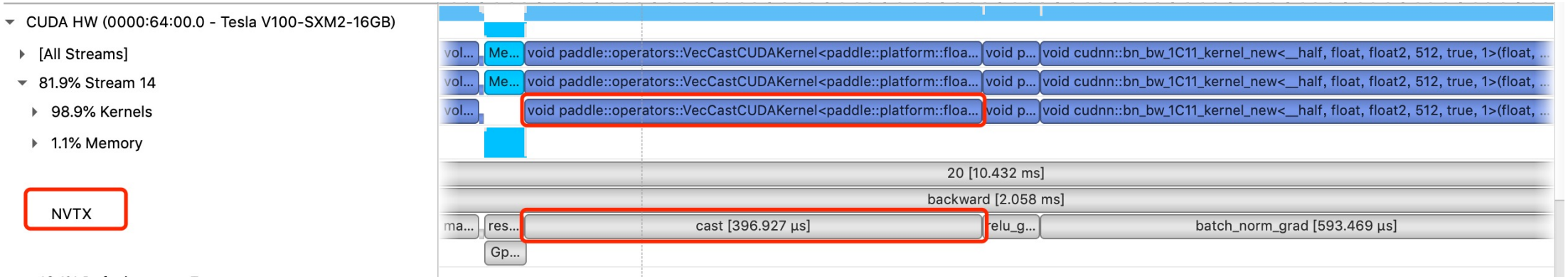


让某一次的调用在timeline中标记出来



使用Nsight分析：模型使用技巧

（4）通过NVTX标记找到Kernel对应的OP名字



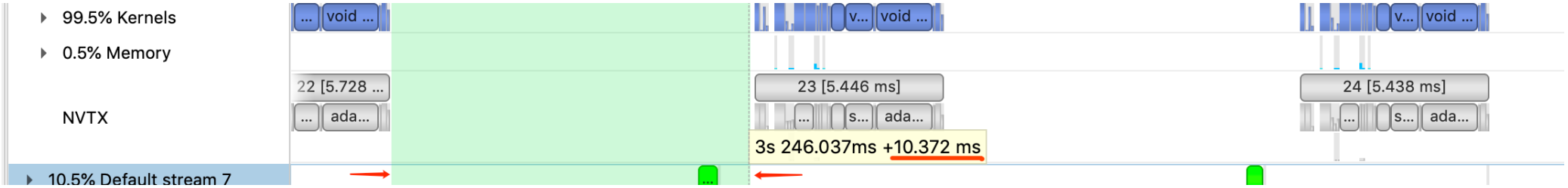
大体定位这次调用发生在训练的哪个阶段（示例为backward）
目前只能结合前后的算子，判断发生在哪处网络结构中，进一步确认OP配置

（5）对Timeline展开，发现更多模型瓶颈

使用Nsight分析：模型瓶颈分析

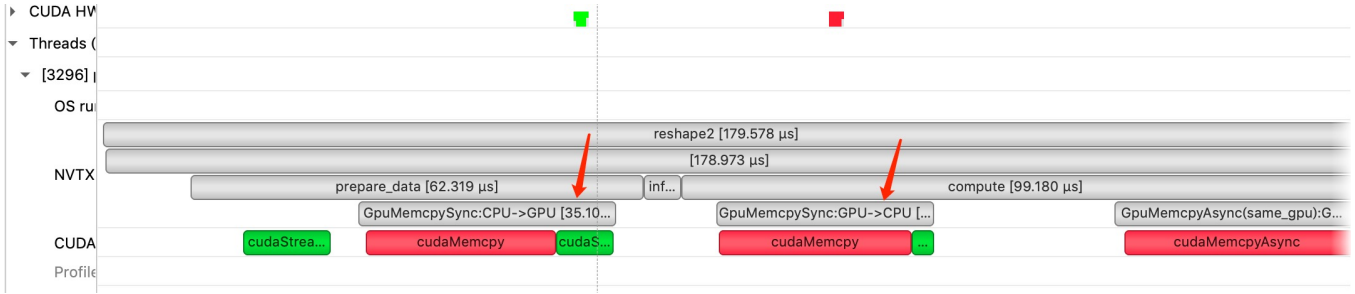
问题1： I/O瓶颈

1）表现：性能监控指标中reader cost耗时较多，timeline中表现为2个iter间空闲较多



2）表现：timeline上出现H2D、D2H

优化方法：尽量避免CPU → GPU、GPU → CPU的同步拷贝

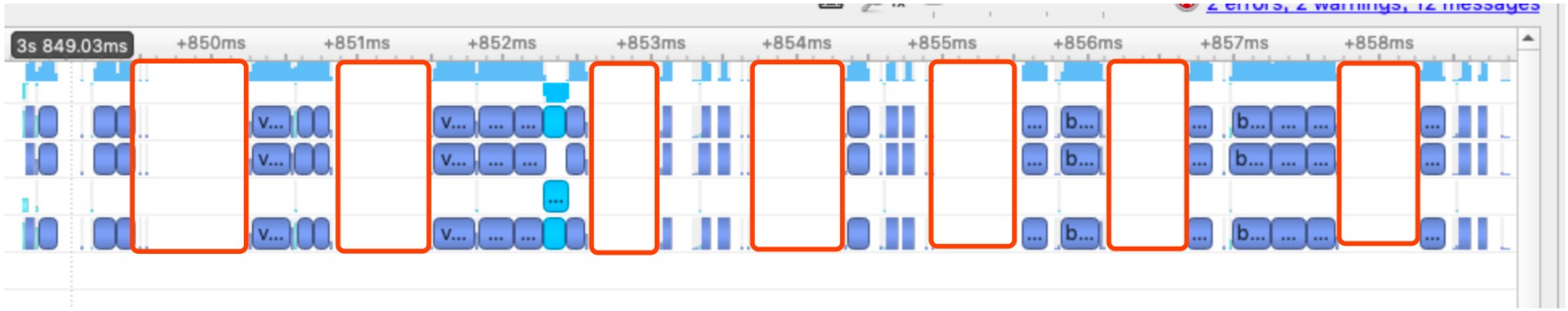


使用Nsight分析：模型瓶颈分析

问题2： CPU瓶颈

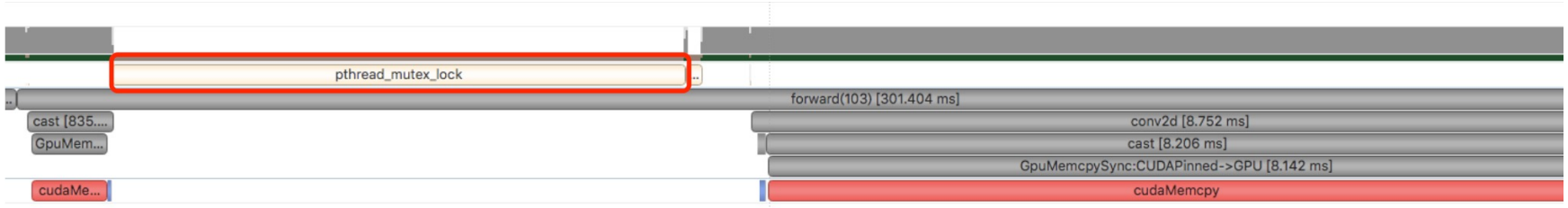
表现1：timeline上CUDA部分出现大段“空白”且占比较高，GPU在等待CPU处理完成

案例： TSM模型conv2d计算input_grad前需要选择算法，CPU端耗时较高



表现2：timeline上Threads部分显示存在一些导致wait的事件

案例： 在HRNet模型中，timeline上pthread_mutex_lock，显示有锁

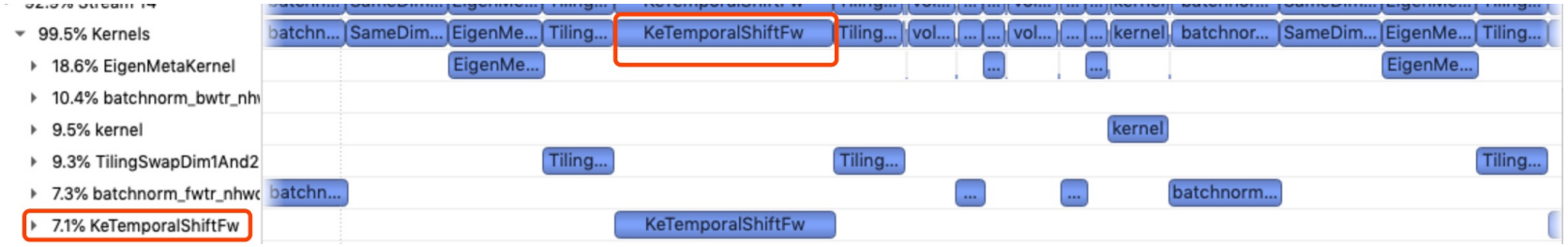


使用Nsight分析：模型瓶颈分析

问题3： GPU瓶颈

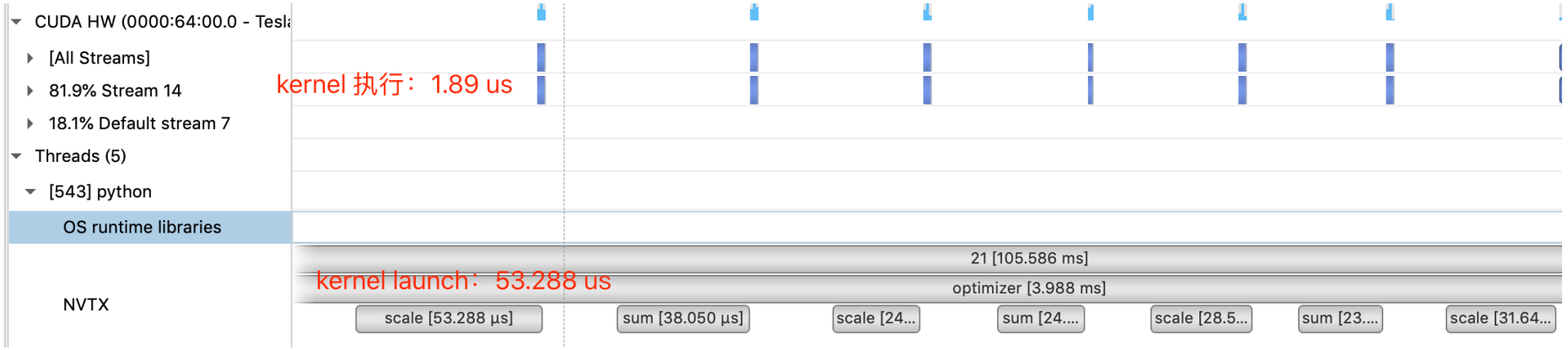
1）表现：观察CUDA部分对应的timeline，显示Kernel执行耗时较高 → Kernel优化

案例：TSM模型temporal_shit算子GPU时间较高



2）表现： Kernel Launch耗时较高 -> Kernel Fusion

案例：SimpleNet中weight_decay引入较多scale、sum算子，将weight_decay和optimizer融合则能提升性能



使用Nsight分析：模型瓶颈分析

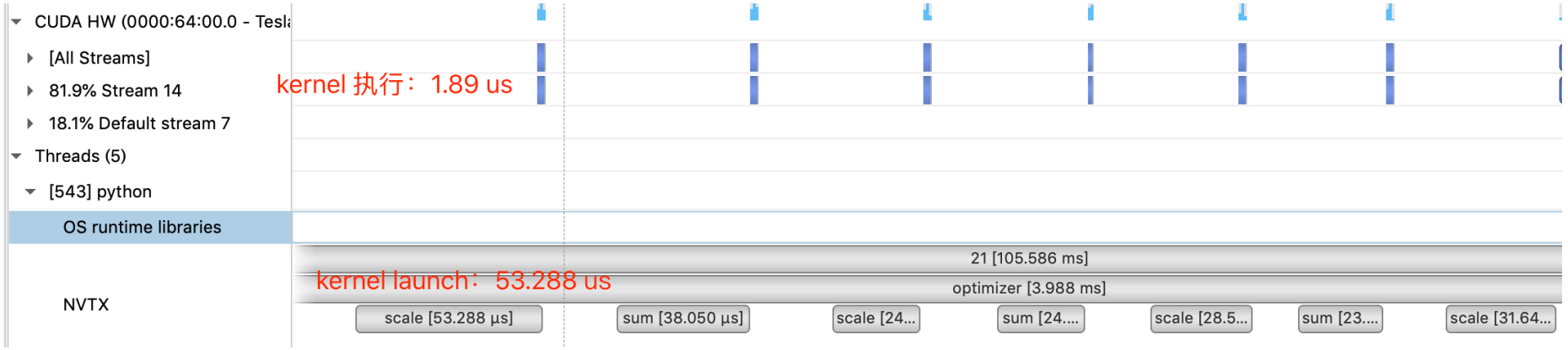
问题3： GPU瓶颈

1）表现：观察CUDA部分对应的timeline，显示Kernel执行耗时较高 → Kernel优化

案例：TSM模型temporal_shit算子GPU时间较高

2）表现： Kernel Launch耗时较高 -> Kernel Fusion

案例：SimpleNet中weight_decay引入较多scale、sum算子，将weight_decay和optimizer融合则能提升性能



谢谢！