专题分享：PyTorch加速原理与应用

目 录

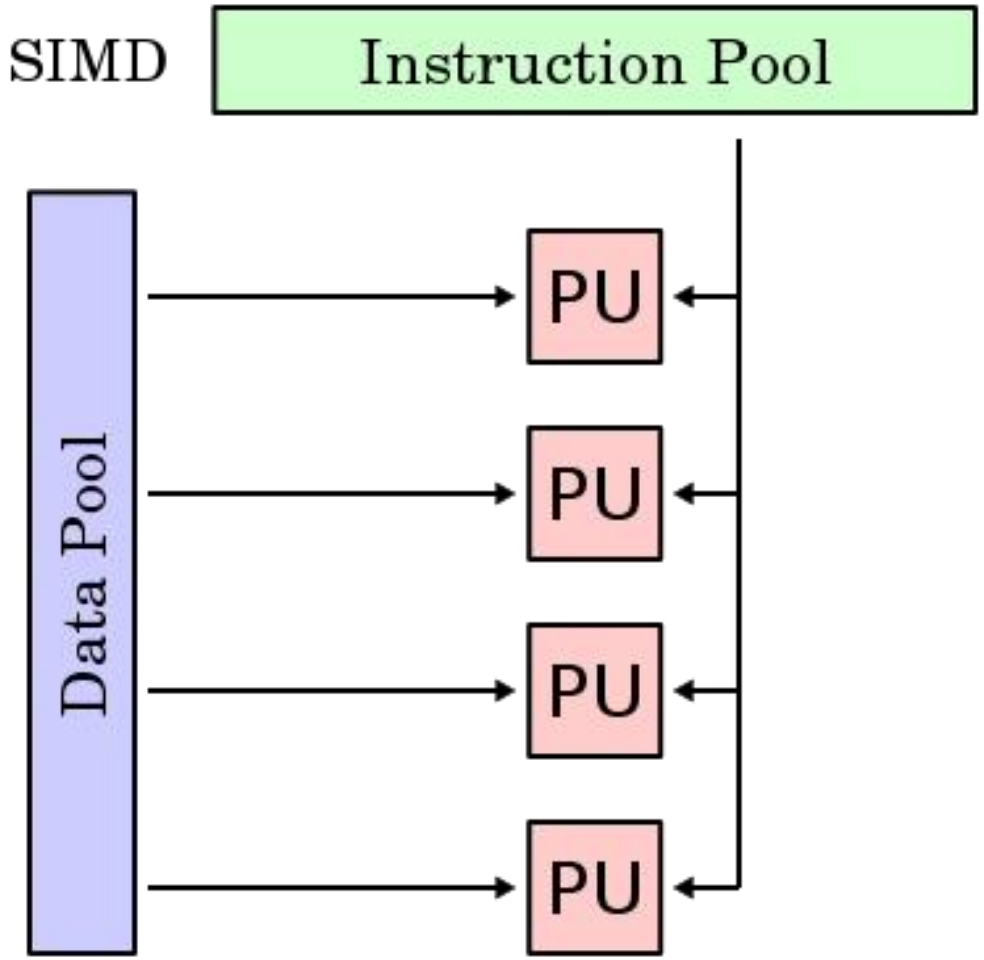
CONTENTS

PyTorch CUDA介绍

Part.01

发展历史

1.随着多媒体技术和电子游戏的发展，为了提高大量单一相同的计算，相近的数据源的计算效率，
上世纪末产生了SIMD(Single Instruction Multiple Data)的技术，单条指令处理多对数据的运算。




```
void simd_add(float* a, float* b, float* result, int n) {
    for (int i = 0; i < n; i += 4) {
        __m128 vec_a = _mm_loadu_ps(&a[i]);
        __m128 vec_b = _mm_loadu_ps(&b[i]);
        __m128 vec_result = _mm_add_ps(vec_a, vec_b);
        _mm_storeu_ps(&result[i], vec_result);
    }
}

void simple_add(float* a, float* b, float* result, int n) {
    for(int i = 0; i < n; i++) {
        result[i] = a[i] + b[i];
    }
}
```

比赛10w~5000w长度向量加法，
SIMD加法比普通加法快18%~24%

发展历史

1.随着多媒体技术和电子游戏的发展，为了提高大量单一相同的计算，相近的数据源的计算效率，通用GPU计算(GPGPU)技术也得到探索，代表有Compute Shader，OpenCL，以及CUDA

```
#version 420 // 使用OpenGL 4.2

in vec2 position; // 输入顶点位置
in vec2 texCoord; // 输入纹理坐标

out vec2 TexCoord; // 输出纹理坐标给Fragment Shader

void main() {
    gl_Position = vec4(position, 0.0, 1.0);
    TexCoord = texCoord;
}
```

```
#version 420 // 使用OpenGL 4.2

in vec2 TexCoord; // 从Vertex Shader接收的纹理坐标

uniform sampler2D texA; // 纹理A
uniform sampler2D texB; // 纹理B

out vec4 color; // 输出颜色

void main() {
    float a = texture(texA, TexCoord).r; // 从纹理A中读取
    float b = texture(texB, TexCoord).r; // 从纹理B中读取
    color = vec4(a + b, 0.0, 0.0, 1.0); // 执行加法，并设置结果为输出颜色的红色通道
}
```

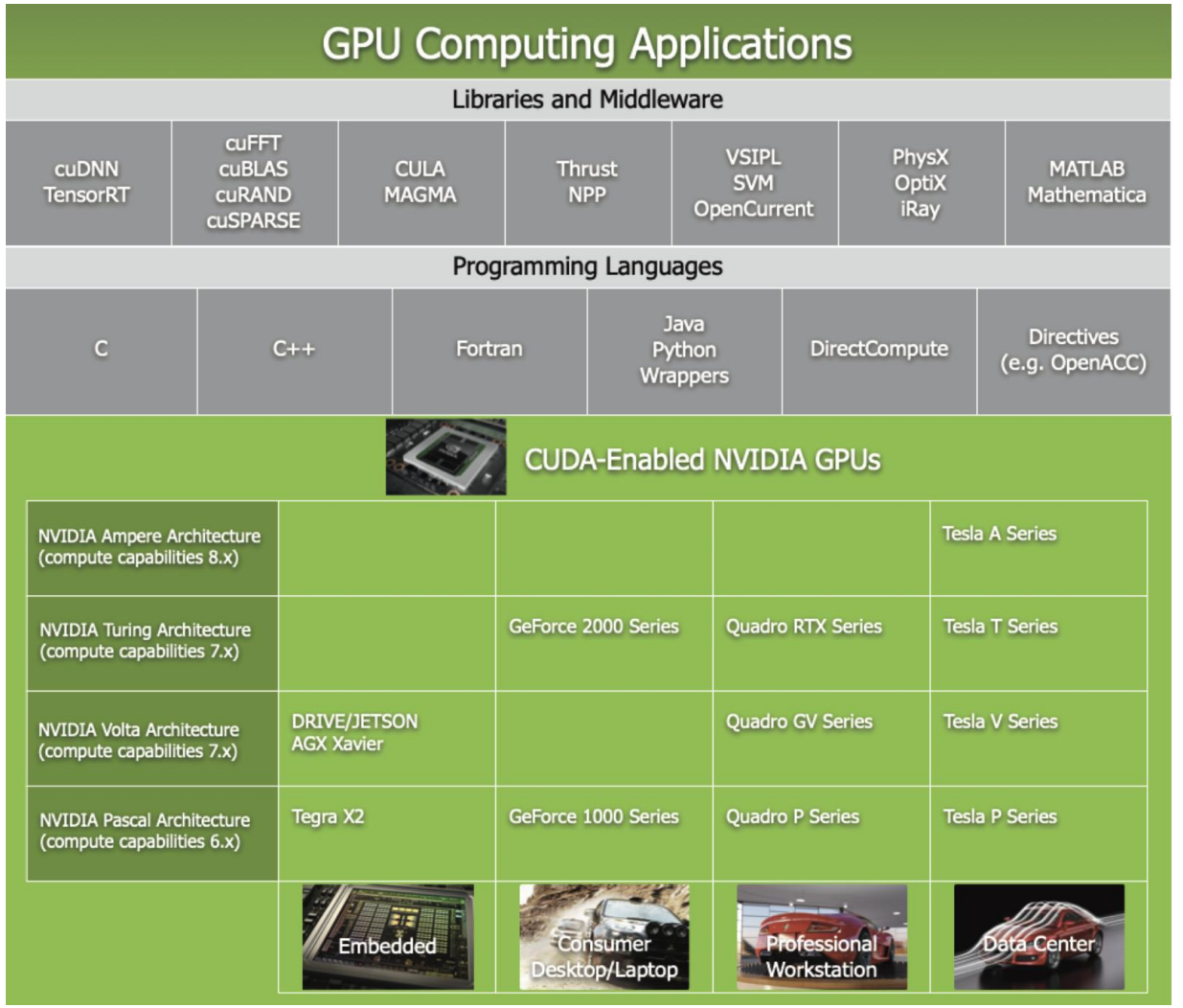
将想要完成的并行计算任务（例如向量加法），转换为一个图形渲染任务（例如图片合成）
数据通过常量缓冲（Constant Buffer或者称Uniform Buffer）或贴图（Texture）传递
走一遍完整的图形渲染管线

数据格式受限

发展历史

2. CUDA（Compute Unified Device Architecture）

- 一种GPU通用计算编程技术
- 原生支持C、C++、Fortran编程语言，截止目前最新版CUDA几乎支持完整的C++20规范，及部分C++标准库。
- Python、Java等编程语言可以通过调用CUDA编译器或使用JIT即时编译技术实现对CUDA的调用。
- PyTorch的算子实际上也是C++写的(LibTorch)，包装了python的接口成为PyTorch



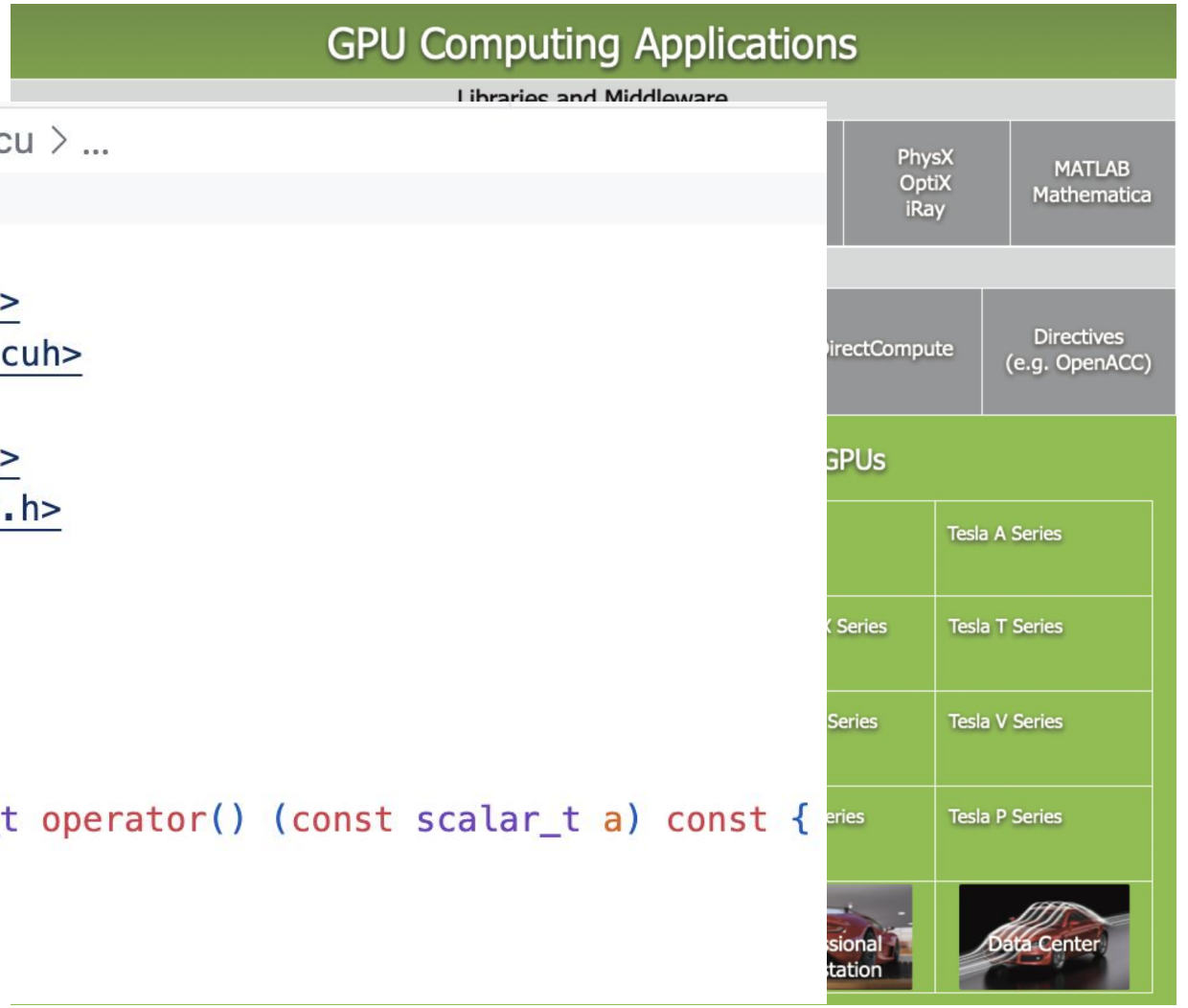
发展历史

2. CUDA（Compute Unified Device Architecture）

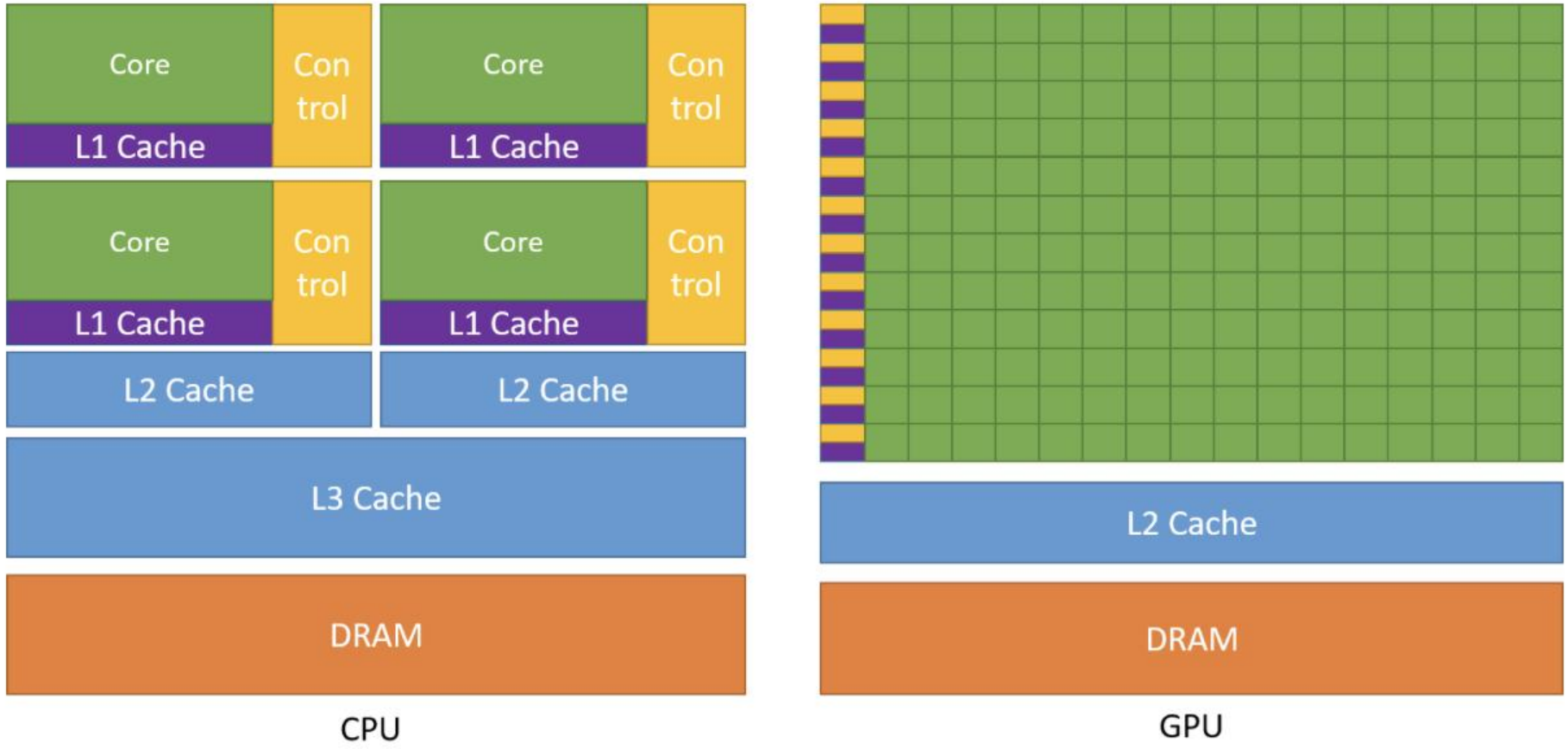
- 一种GPU通用计算编程技术

aten > src > ATen > native > cuda > AbsKernel.cu > ...

```cpp
#define TORCH_ASSERT_NO_OPERATORS
#include <ATen/native/UnaryOps.h>
#include <ATen/native/cuda/Loops.cuh>
#include <ATen/native/cuda/JitLoops.cuh>
#include <ATen/Dispatch.h>
#include <ATen/native/DispatchStub.h>
#include <ATen/native/TensorIterator.h>

namespace at { namespace native {

template<typename scalar_t>
struct AbsFunctor {
  __device__ __forceinline__ scalar_t operator() (const scalar_t a) const {
    return std::abs(a);
  }
};
```



CUDA编程模型



GPU相比于CPU：拥有更多核心（CUDA Core/Execution Unit），而控制单元相对较少

CUDA编程模型

```
1  __global__ void vector_add_kernel(const float* a, const float* b, float* c, int ndata){
2      int idx = threadIdx.x + blockIdx.x * blockDim.x;
3      if(idx >= ndata) return;
4      c[idx] = a[idx] + b[idx];
5  }
```

C[0]=A[0]+B[0]

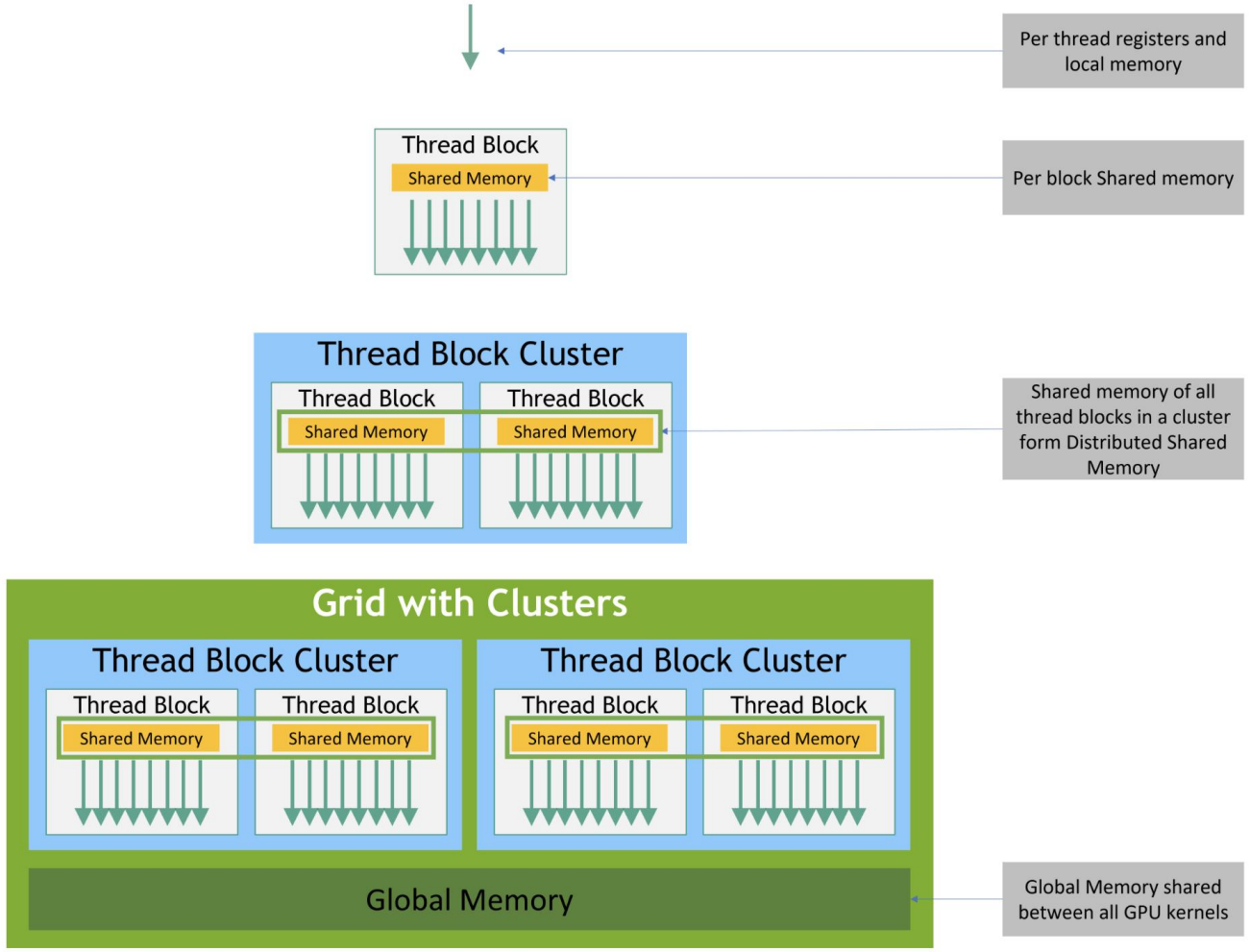
C[1]=A[1]+B[1]

C[2]=A[2]+B[2]

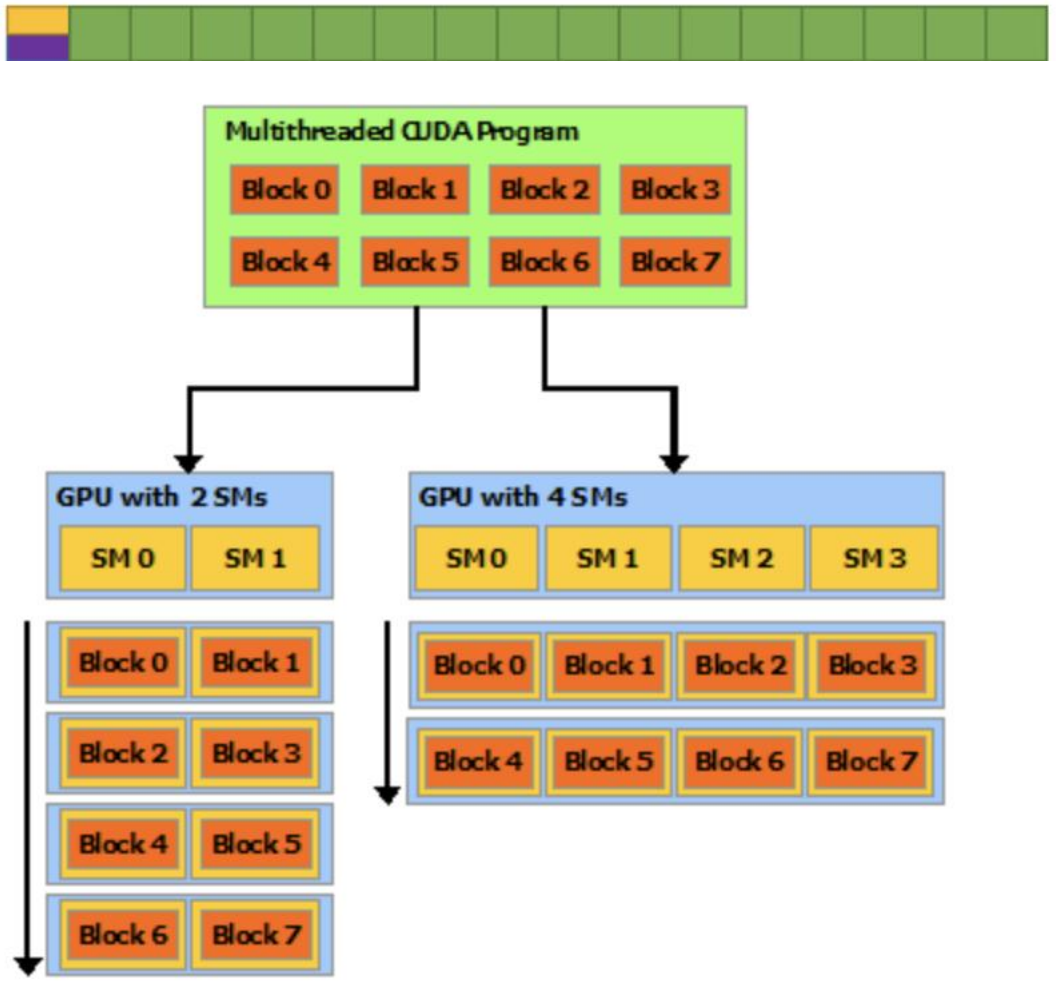
C[15]=A[15]+B[15]

CUDA编程模型

编程视角下的CUDA



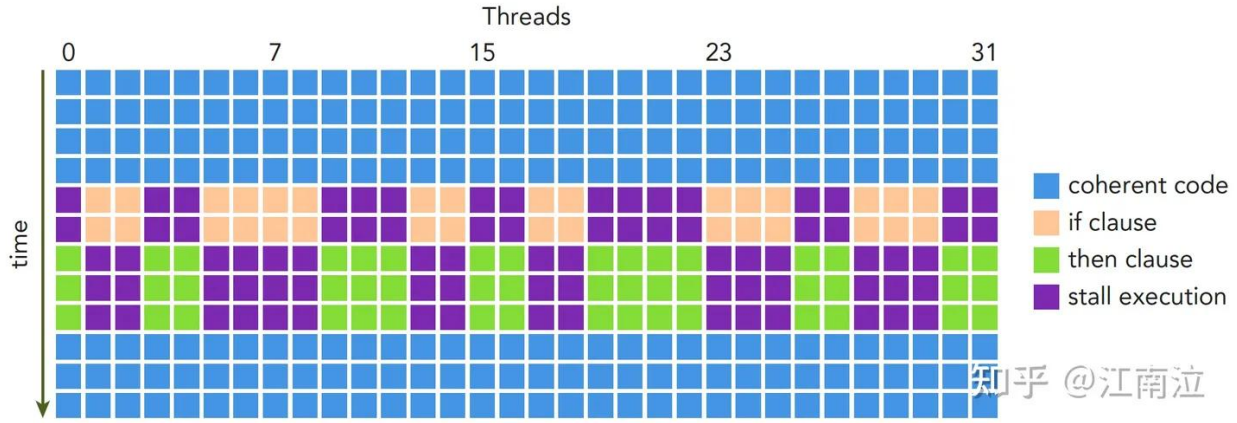
SM：Stream Multiprocessor



CUDA规则

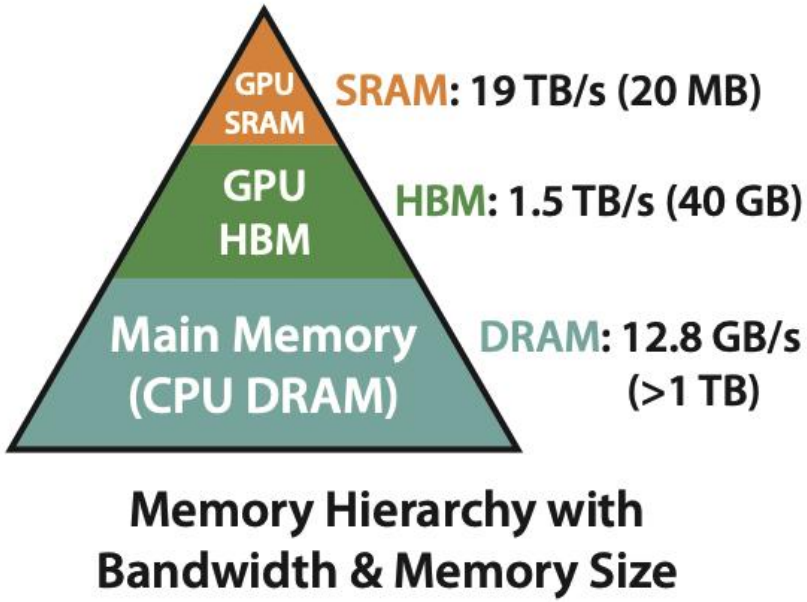
1. 相邻16个线程的访存是可以合并的。
2. 相邻32个线程只能够执行同一条指令。

3. 注意规则2，条件分支可能是非常低效的。



4. GPU全局内存的访存延迟是非常非常高的，通常有几百个指令周期。
5. GPU线程调度是极低开销的，可以通过高并发掩盖访存延迟。
6. 时刻牢记CPU和GPU是两个不同的设备，GPU在计算的时候CPU可以继续工作，CPU将数据或者计算任务提交到GPU上，除了传输和计算，还有额外的调度开销。

CUDA存储系统



Memory Hierarchy with Bandwidth & Memory Size

GPU显存具有很高的延迟，但同时有更大数据带宽（远超内存）
缓存（通常由SRAM构成）兼具低延迟与高带宽，但容量小

在不是很古老的Nvidia GPU上，GPU内部已经实现了和主机上类似的虚拟内存机制（虚拟地址和物理地址机制，并非指将外存当做内存使用）。虚拟内存机制支持了：

1. 设备数据与主机数据的统一编址，主机内存的数据迁移到GPU显存后，可以保持地址不变。在计算能力9.0的设备上甚至支持多台不同设备的统一编址。在统一编制的架构下，Nvidia GPU可以通过缺页中断(Page Fault)来自动在主机内存与显存之间迁移数据。
2. 零拷贝的进程间数据共享。

Dao Tri, et.al. Flash Attention: Fast and Memory-Efficient Exact Attention with IO-Awareness. NIPS 2022
Dao Tri. Flash Attention-2: Faster Attention with Better Parallelism and Work Partitioning. ArXiv 2307. 08691

PyTorch与CUDA

正确测量CUDA运行时间

时刻牢记CPU和GPU是两个不同的设备

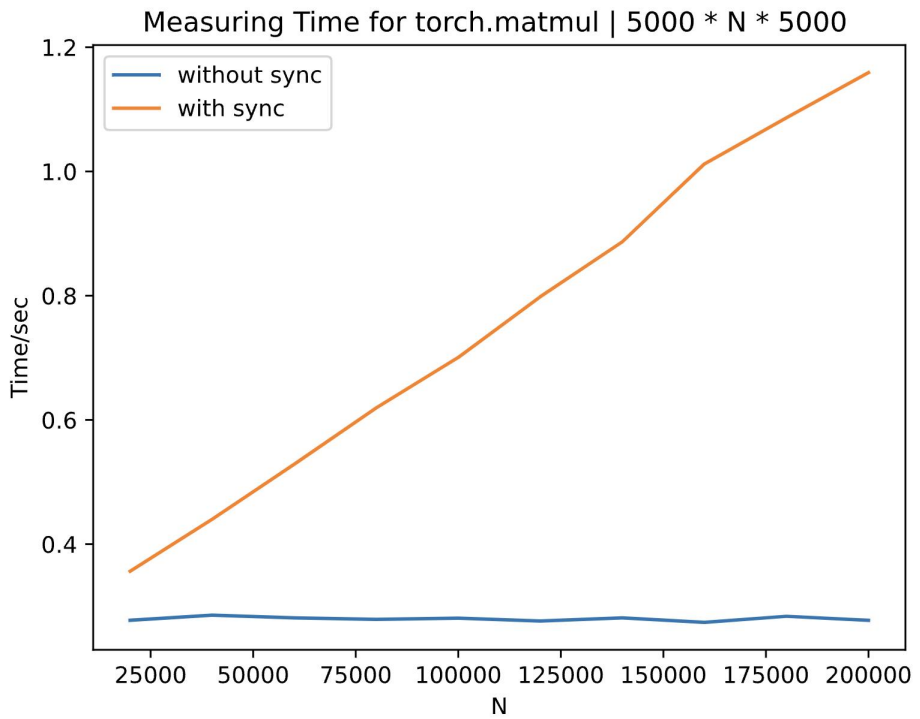
PyTorch CUDA上的运算，由GPU进行，CPU仅负责将计算任务提交给GPU，那么在提交完成后CPU可以选择：

1. 等待GPU完成这个计算任务后再继续
2. 不等GPU完成这个计算任务，直接继续往后执行

做个实验：torch.cuda.synchronize() 可以阻塞CPU等待GPU上的所有任务完成才往下执行。

```
A = torch.randn(size=(args.m, args.n), device='cuda')
B = torch.randn(size=(args.n, args.p), device='cuda')
torch.cuda.synchronize()  # 这里总是同步一下

s = perf_counter()

C = torch.matmul(A, B)

if args.sync:
    torch.cuda.synchronize()    # CUDA设备同步，让CPU等GPU上的运算完成再往下执行
    # torch.cuda.current_stream().synchronize() # 这个也可以，只同步当前的流

t = perf_counter()
print(f"Used time = {t - s} sec.")
```

结论：默认情况下是2，不等待



PyTorch与CUDA

正确测量CUDA运行时间

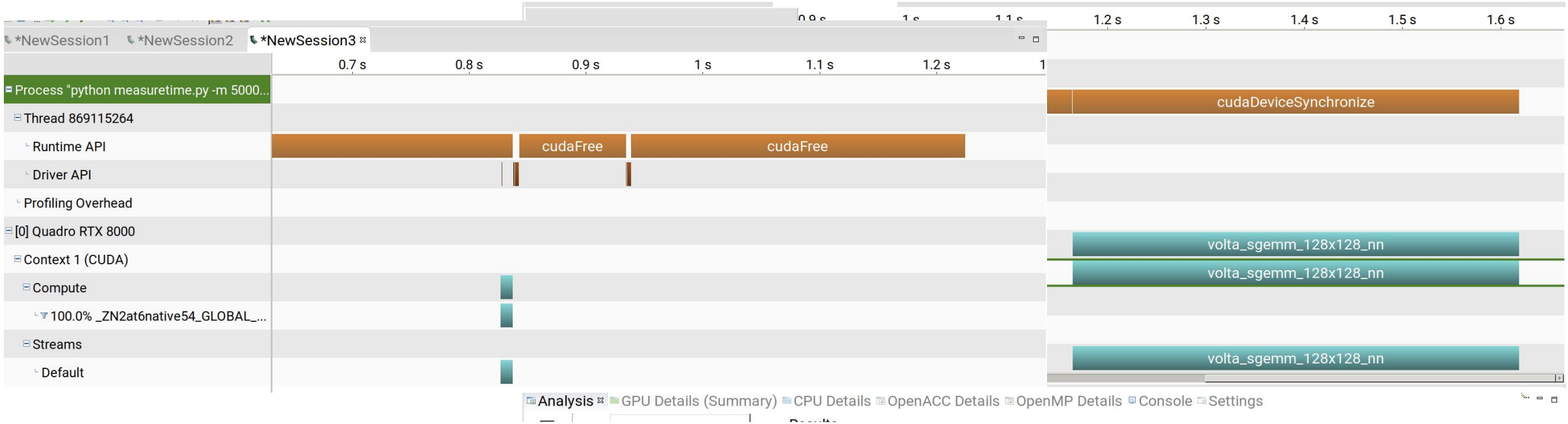
时刻牢记CPU和GPU是两个不同的设备

PyTorch CUDA上的运算，由GPU进行，CPU仅负责将计算任务提交给GPU，那么在提交完成后CPU可以选择：

1. 等待GPU完成这个计算任务后再继续
2. 不等GPU完成这个计算任务，直接继续往后执行

做个实验：torch.cuda.synchronize() 可以阻塞CPU等待GPU上的所有任务完成才往下执行。

结论：默认情况下是2，不等待

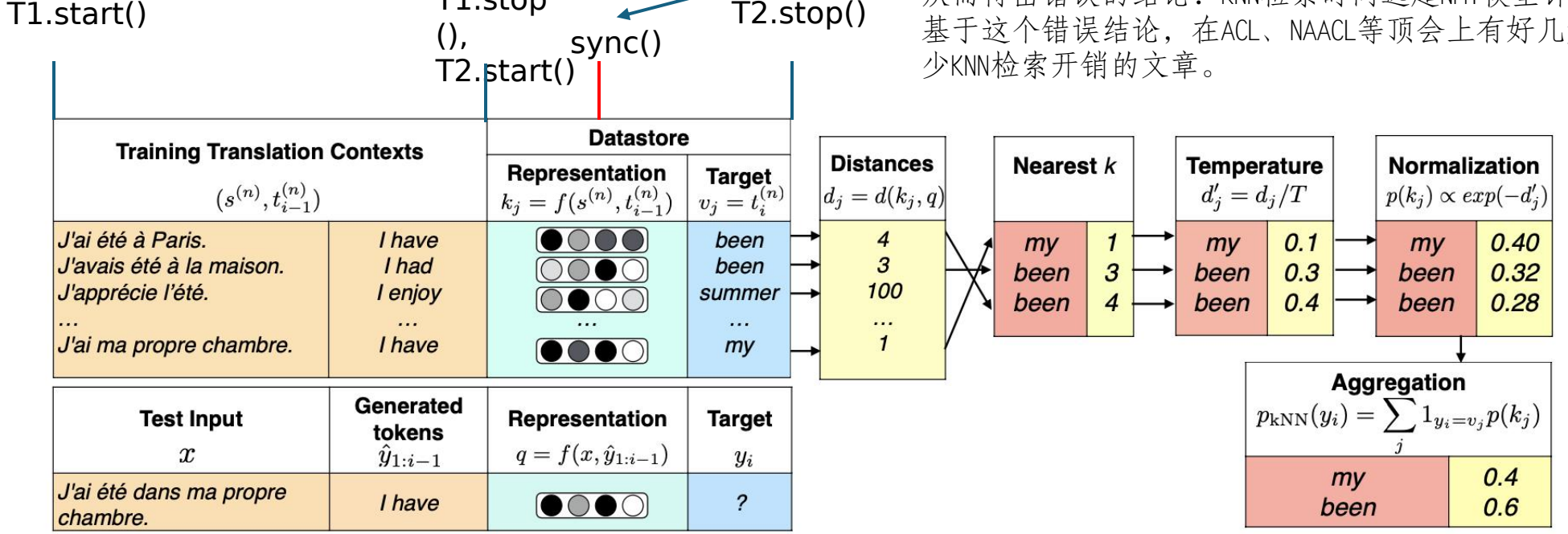


使用错误的测量方法

计时器T1只记录了NMT模型forward过程中CPU向GPU提交计算任务的时间，没有计入实际计算使用时间。

此处的同步操作，导致将NMT的计算时间计入了计时器T2内

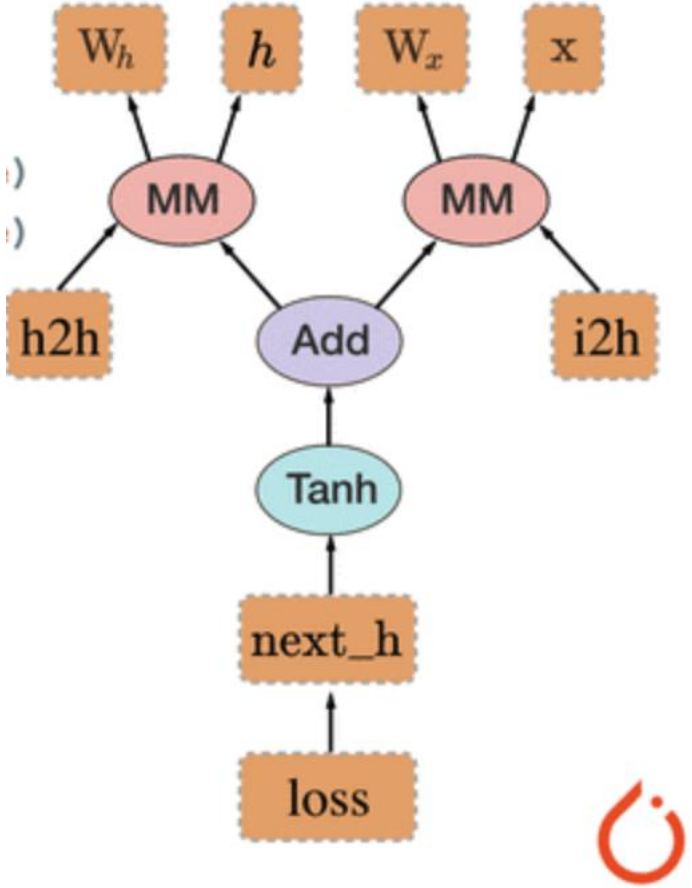
从而得出错误的结论：KNN检索时间远超NMT模型计算时间。基于这个错误结论，在ACL、NAACL等顶会上有好几篇研究减少KNN检索开销的文章。



KNN-MT

流与同步机制

既然CPU提交计算任务后并不会等待GPU计算任务完成，如何确保正确的



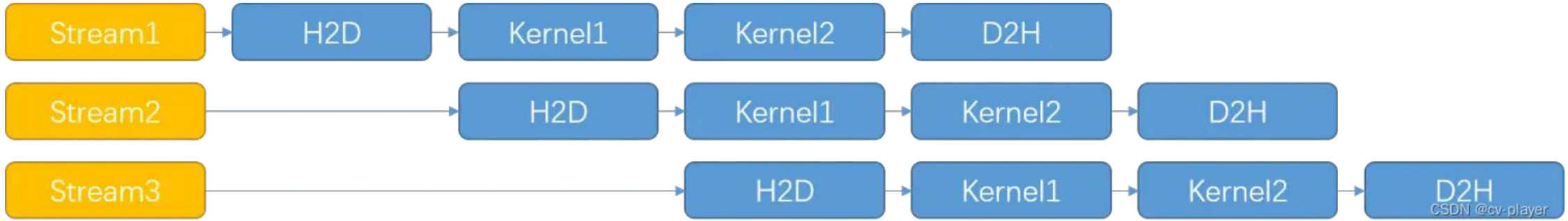
不是因为计算图

PyTorch与CUDA

流与同步机制

既然CPU提交计算任务后并不会等待GPU计算任务
完成，如何确保正确的

流（Stream）：流类似于一个队列，提交到流上工作，工作之间保证是顺序执行的。不同流之间可以并行。



PyTorch的cuda默认会使用一个默认流，
可以使用torch.cuda.Stream和torch.cuda.stream创建新的流以及使用新的流。
CPU可以只对某一个流同步，而不一定需要对整个GPU同步。(Stream.synchronize方法)

PyTorch与CUDA

流加速神经网络推理的例子

```
11      # One stream
12      s = perf_counter()
13      E = torch.matmul(A, B)
14      F = torch.matmul(C, D)
15      torch.cuda.synchronize()
16      t = perf_counter()
17      print(f"Time one stream = {t-s}")
18
19      # Multiple streams
20      st = torch.cuda.Stream()
21      s = perf_counter()
22      E = torch.matmul(A, B)
23      with torch.cuda.stream(st):
24          F = torch.matmul(C, D)
25      torch.cuda.synchronize()
26      t = perf_counter()
27      print(f"Time multiple streams = {t-s}")
```

```
(ML) (base) ➜  20240328code python torch_multistream.py
Time one stream = 0.595285496674478
Time multiple streams = 0.2989145922474563
(ML) (base) ➜  20240328code python torch_multistream.py
Time one stream = 0.5964349377900362
Time multiple streams = 0.2984891929663718
(ML) (base) ➜  20240328code python torch_multistream.py
Time one stream = 0.5920228101313114
Time multiple streams = 0.298232588917017
```

无数据依赖时，可以使用流并行化数据的运算，尤其可以提高推理效率。

例如CLIP模型，文本编码器和图像编码器可以在两个不同的流上运行，实现文本和图像的并行编码，提高GPU利用率。

PyTorch与CUDA

其它实用小trick

```
X = torch.rand(50000, 50000, device='cuda')

torch.cuda.synchronize()
s = perf_counter()
torch.nn.functional.softmax(X, dim=0)
torch.cuda.synchronize()
t = perf_counter()
print(f"Softmax on dim 0, time = {t-s}")

torch.cuda.synchronize()  # 这行其实不必要
s = perf_counter()
torch.nn.functional.softmax(X, dim=-1)
torch.cuda.synchronize()
t = perf_counter()
print(f"Softmax on dim -1, time = {t-s}")
```

最后一个维度上的归约操作更快

```
(ML) (base) ➜  20240328code python softmax_speed.py
Softmax on dim 0, time = 0.15947460662573576
Softmax on dim -1, time = 0.08734449977055192
(ML) (base) ➜  20240328code python softmax_speed.py
Softmax on dim 0, time = 0.151559847872704 27
Softmax on dim -1, time = 0.08632054412737489
(ML) (base) ➜  20240328code python softmax_speed.py
Softmax on dim 0, time = 0.14664118504151702
Softmax on dim -1, time = 0.08748611900955439
(ML) (base) ➜  20240328code python softmax_speed.py
Softmax on dim 0, time = 0.15154830878600478
Softmax on dim -1, time = 0.08650919888168573
(ML) (base) ➜  20240328code
```

PyTorch与CUDA

避免每batch都使用loss.item()

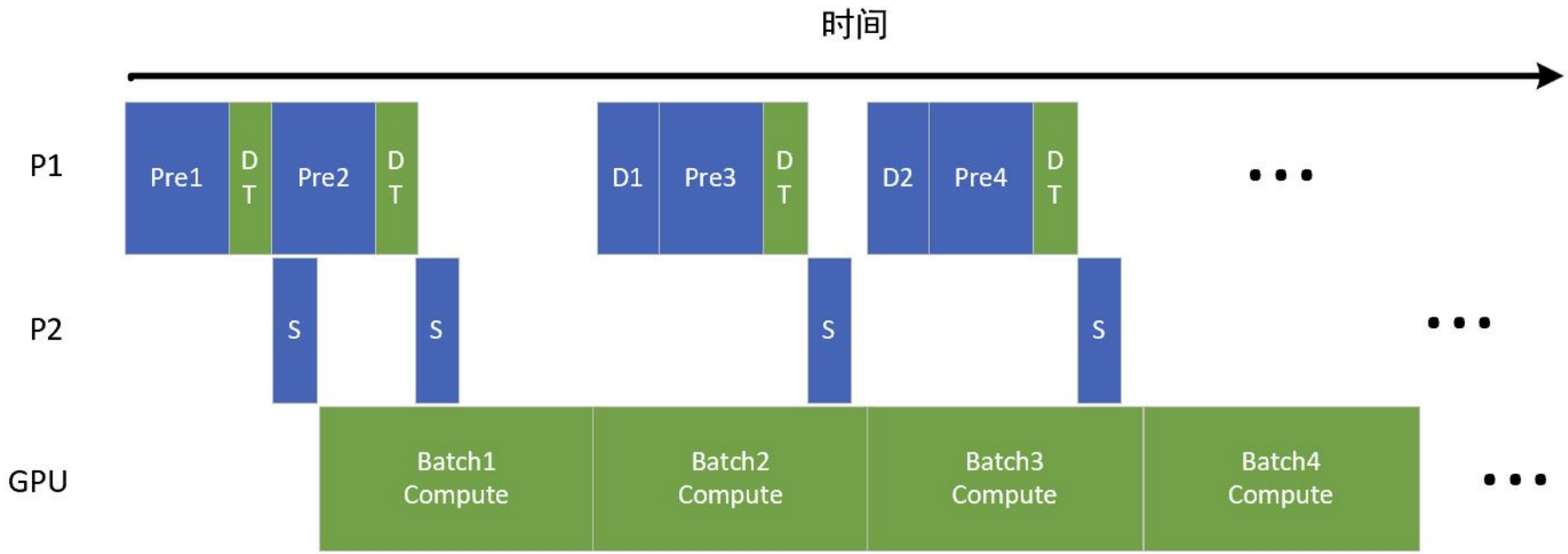
除了使用synchronize，其他操作也可能导致CPU对GPU的同步，一个非常场景的情况就是Tensor.item方法

```
for batch in data_loader:
    loss = model(**batch)
    loss.backward()

    print(loss.item())
```

```
for batch in data_loader:
    loss = model(**batch)
    loss.backward()

    batch_loss += loss.detach()

    #....

    # Accumulate some batches
    print(batch_loss.mean().item())
```

实际上，在GPU上正在进行运算的Tensor，如果CPU这边需要取值的话（print打印Tensor的值也需要从GPU上取得正确的值），都会触发当前流的同步

一边训练模型一边加载并预处理数据



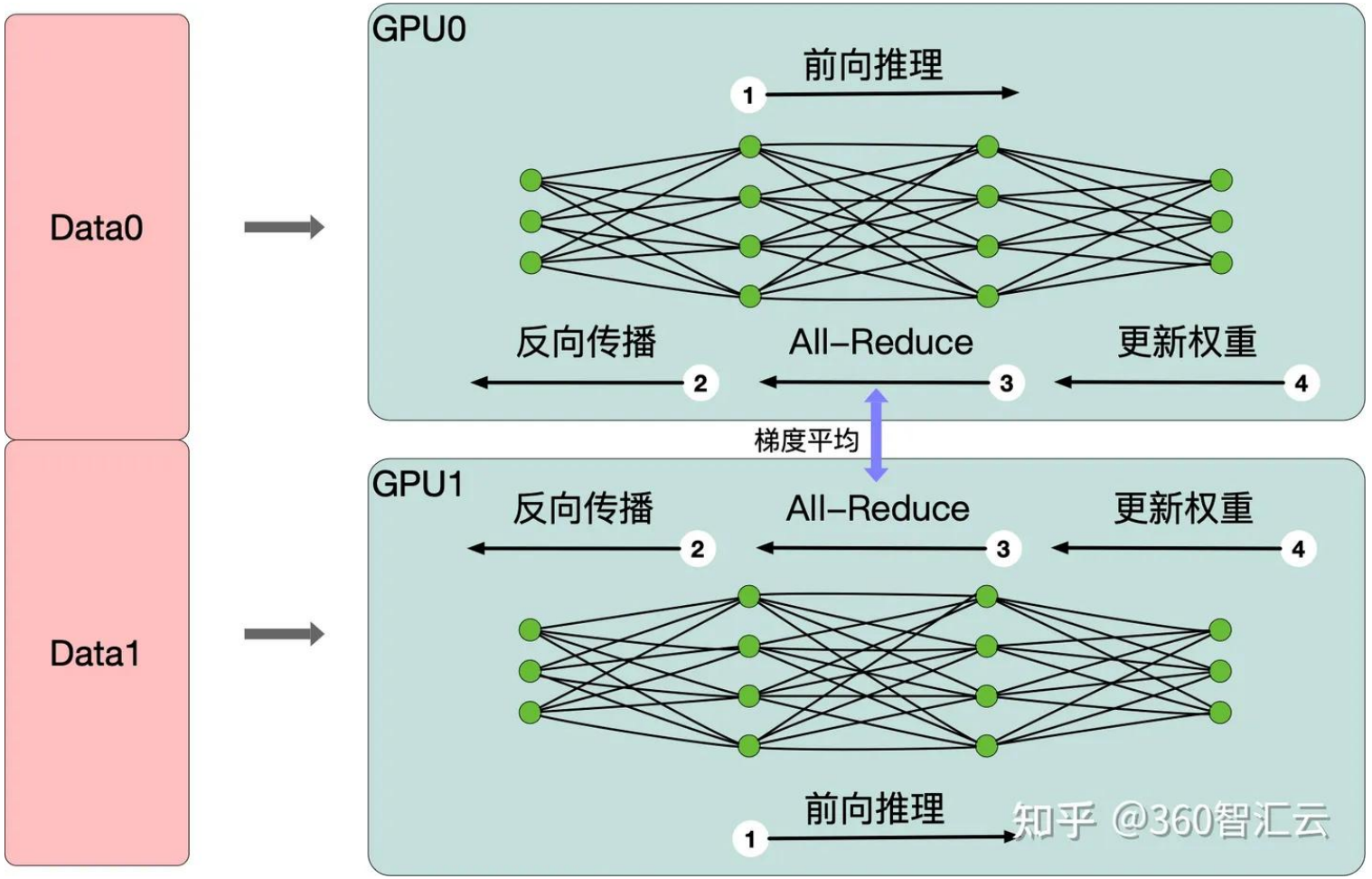
Pre：预处理 DT：数据迁移内存->显存 D：释放数据存储资源 S:提交计算作业　　蓝：CPU 绿：GPU

GPU和CPU之间的内存传输和GPU代码执行是可以并行进行的

利用流水线工作的方式，一边训练一边加载数据并预处理，几乎和提前加载并处理好所有数据再训练的时间相同，因为流水线并行的调度方式掩盖了数据加载和处理的时间

PyTorch 数据级并行

PyTorch数据级并行



每块GPU卡上都存储整个模型
一批数据分组后分发到不同GPU上
汇总所有GPU卡上算出loss取平均，然后多卡并行进行反向传播

PyTorch数据级并行

nn.DataParallel
用法简单，对代码略微修改只需要给模型套一层DataParallel。使用多线程实现，因python GIL的存在，性能损耗较大，有时还会负提升。

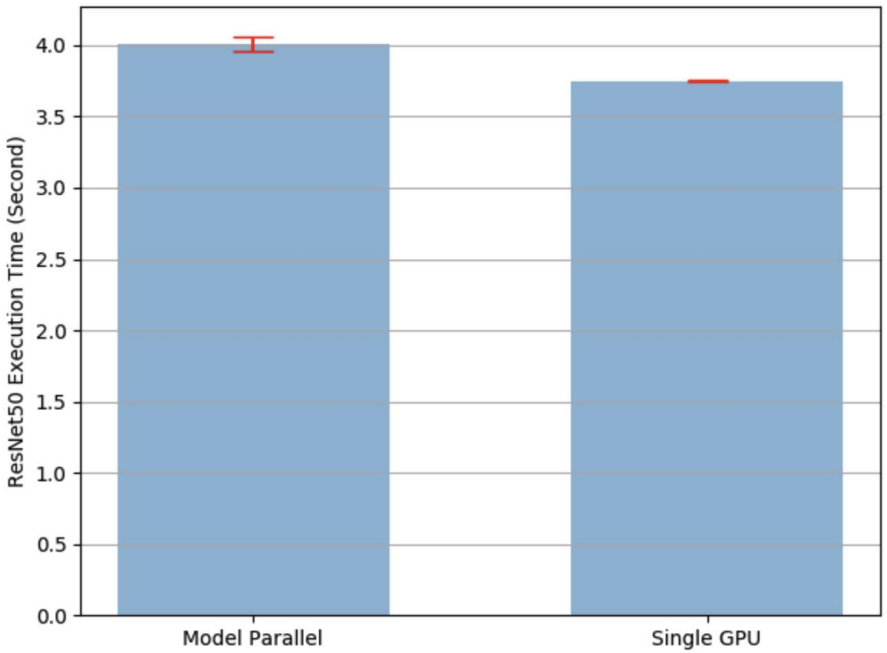
nn.parallel.DistributedDataParallel
用法稍复杂。使用多进程实现，没有GIL的问题，目前pytorch官方推荐使用这种方式实现数据级并行。

缺点：
每一个GPU上都要存储模型的全部参数，严重浪费显存空间
当模型过大，在单块GPU上无法完成至少batch_size=1的计算的时候，就无法使用了。

PyTorch 模型级并行

PyTorch模型级并行

```python
class ModelParallelResNet50(ResNet):
    def __init__(self, *args, **kwargs):
        super(ModelParallelResNet50, self).__init__(
            Bottleneck, [3, 4, 6, 3], num_classes=num_classes, *args, **kwargs)

        self.seq1 = nn.Sequential(
            self.conv1,
            self.bn1,
            self.relu,
            self.maxpool,

            self.layer1,
            self.layer2
        ).to('cuda:0')

        self.seq2 = nn.Sequential(
            self.layer3,
            self.layer4,
            self.avgpool,
        ).to('cuda:1')

        self.fc.to('cuda:1')

    def forward(self, x):
        x = self.seq2(self.seq1(x).to('cuda:1'))
        return self.fc(x.view(x.size(0), -1))
```

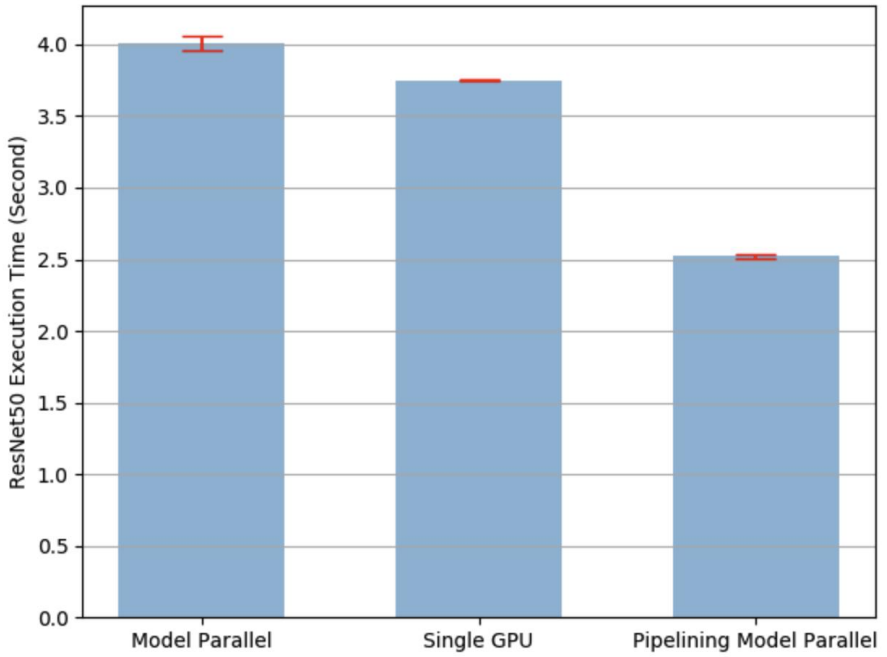


将模型的参数分布到不同GPU卡上

PyTorch模型级并行

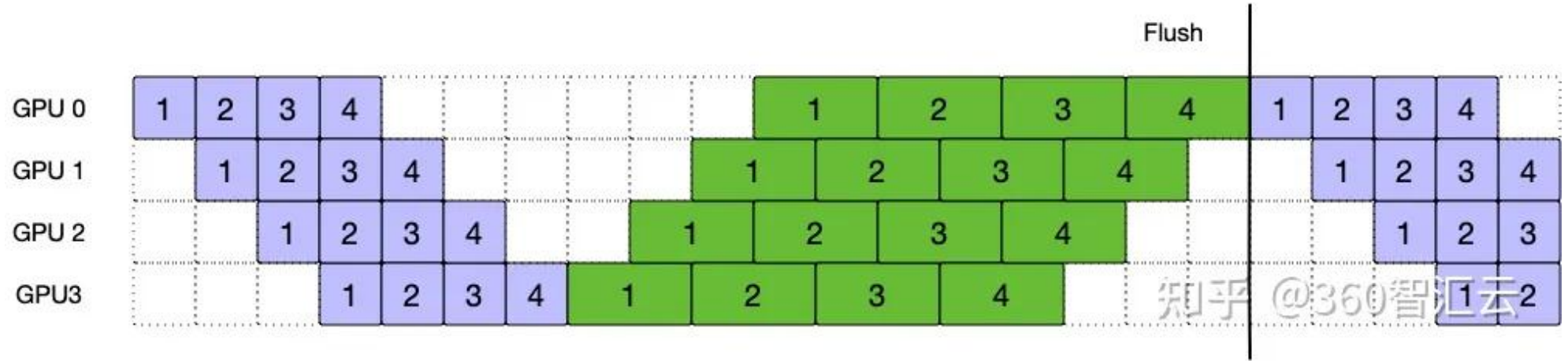
流水线方式

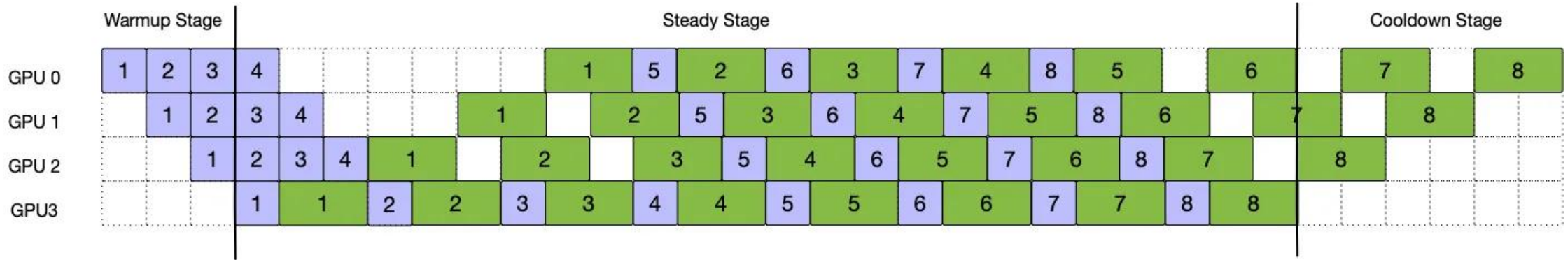
```
def forward(self, x):
    splits = iter(x.split(self.split_size, dim=0))
    s_next = next(splits)
    s_prev = self.seq1(s_next).to('cuda:1')
    ret = []

    for s_next in splits:
        # A. ``s_prev`` runs on ``cuda:1``
        s_prev = self.seq2(s_prev)
        ret.append(self.fc(s_prev.view(s_prev.size(0), -1)))

        # B. ``s_next`` runs on ``cuda:0``, which can run concurrently with A
        s_prev = self.seq1(s_next).to('cuda:1')

    s_prev = self.seq2(s_prev)
    ret.append(self.fc(s_prev.view(s_prev.size(0), -1)))

    return torch.cat(ret)
```



PyTorch模型级并行

流水线方式





PyTorch模型级并行

模型级并行小结

优点

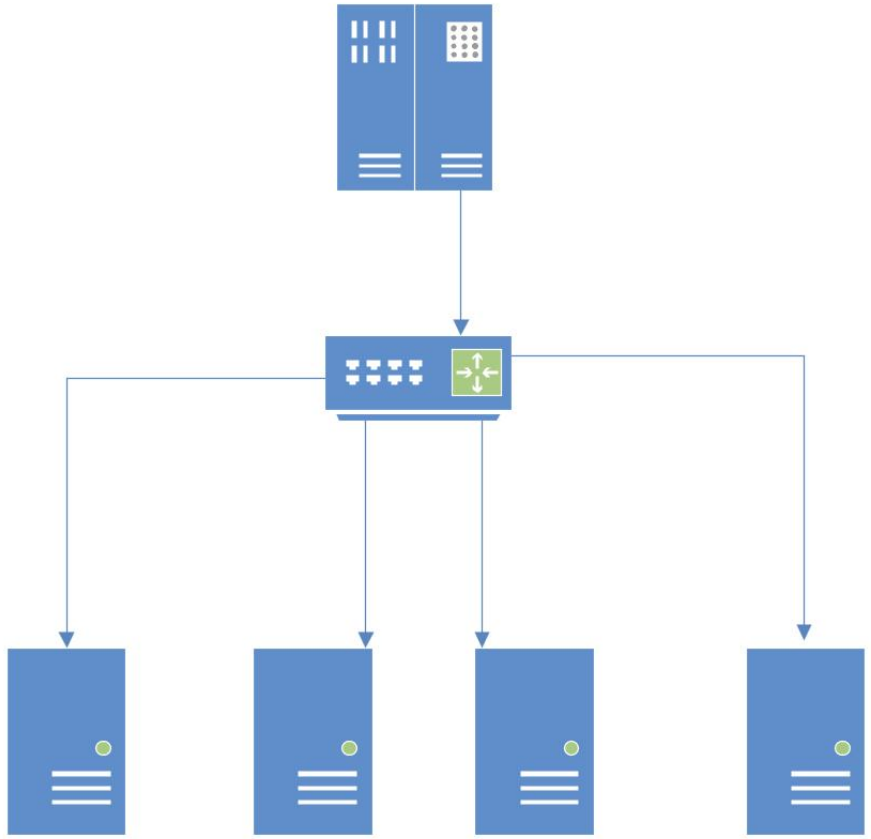
1. 模型级并行可以解决单块GPU显存太小，存放不下模型的问题。
2. 模型级并行通过流水线的方式进行调度，可以实现对所有GPU卡的高效利用。
3. 相比于数据级并行，没有浪费多倍的空间用以存储模型参数

缺点

1. 编程实现比较复杂，尤其是流水线调度的方式
2. 需要了解模型的内部结构，可能需要重新编写forward函数和训练的代码

PyTorch模型级并行

数据级并行与模型级并行结合



网传OpenAI GPT-3的训练方法：

在单台服务器内多块显卡使用模型级并行

在不同服务器之间使用数据级并行

总结

Part.01

总结

随着大模型的研究和落地应用，如何充分利用GPU的计算能力，对大模型进行微调、终端部署等也成为了热门研究问题，在近两年的深度学习顶会上也有越来越多的论文探讨这样的方法。或可成为大模型时代的没有大量GPU的情况下可以选择的研究方向。

Thank you for listening!