MegEngine中的动态图Sublinear显存优化

MEGVII 旷视

邓哲也

2021年4月22日

MegEngine 1.4.0rc1 新feature

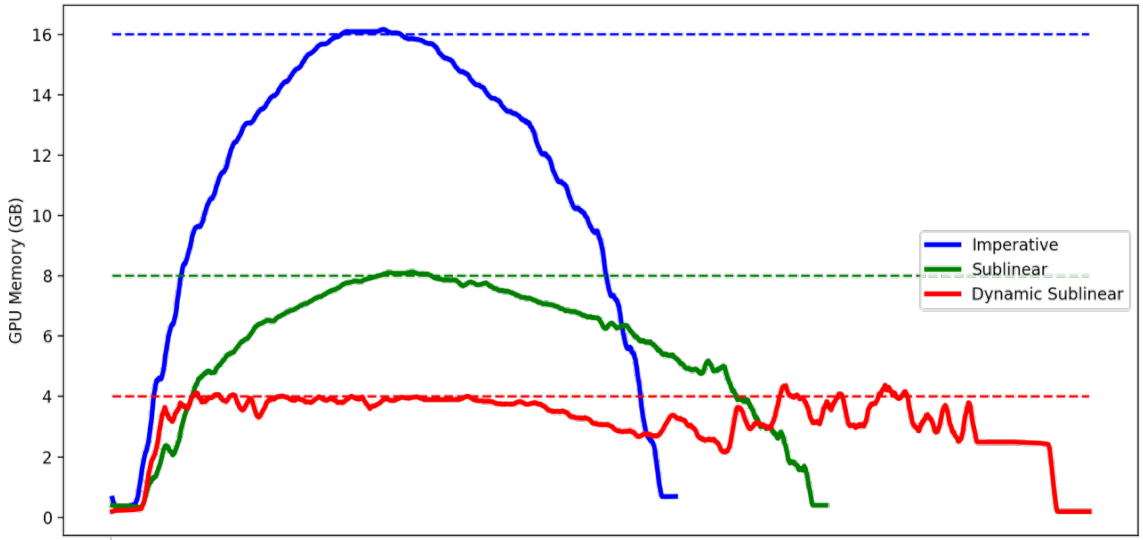
```python
import numpy as np
import resnet
import megengine as mge
import megengine.autodiff as ad
import megengine.functional as F
import megengine.optimizer as optim
```

只需添加两行

```python
batch_size = 400
image = mge.tensor(np.random.random((batch_size, 3, 224, 224)))
label = mge.tensor(np.random.randint(100, size=(batch_size,)))
model = resnet.__dict__["resnet50"]()

gm = ad.GradManager().attach(model.parameters())
opt = optim.SGD(model.parameters(), lr=0.0125, momentum=0.9, weight_decay=1e-4,)

for i in range(n_iter):
    with gm:
        logits = model(image)
        loss = F.nn.cross_entropy(logits, label)
        gm.backward(loss)
        opt.step().clear_grad()
        print("iter = {}, loss = {}".format(i + 1, loss.numpy()))
```

batch_size × 4

动态Sublinear显存优化效果展示




```
from megengine.utils.dtr import DTR

dtr = DTR(memory_budget=4*1024**3)
```

显存占用降至 1/4

训大模型神器

一键开启DTR[1]

[1] Kirisame M, Lyubomirsky S, Haan A, et al. Dynamic tensor rematerialization[J]. arXiv preprint arXiv:2006.09616, 2020.

MEGVII 旷视

目录

Contents

1 背景介绍

2 动态图显存优化

3 MegEngine的工程实现

4 未来工作

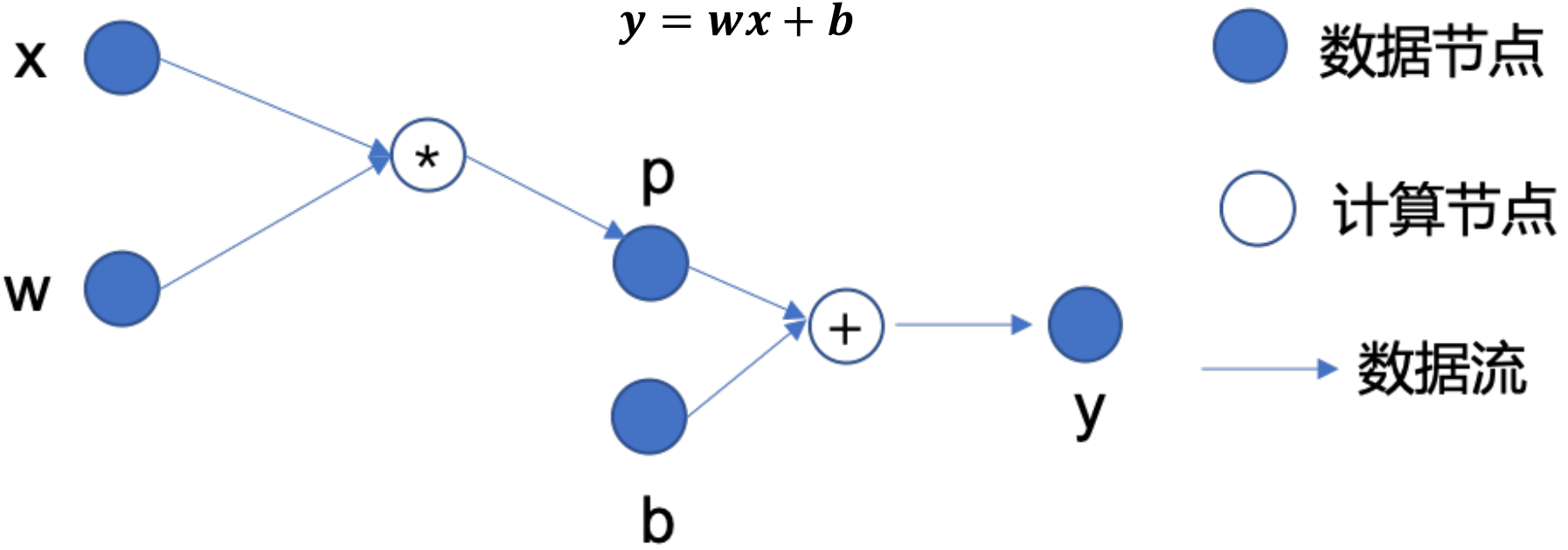
MEGVII 旷视

01

背景介绍

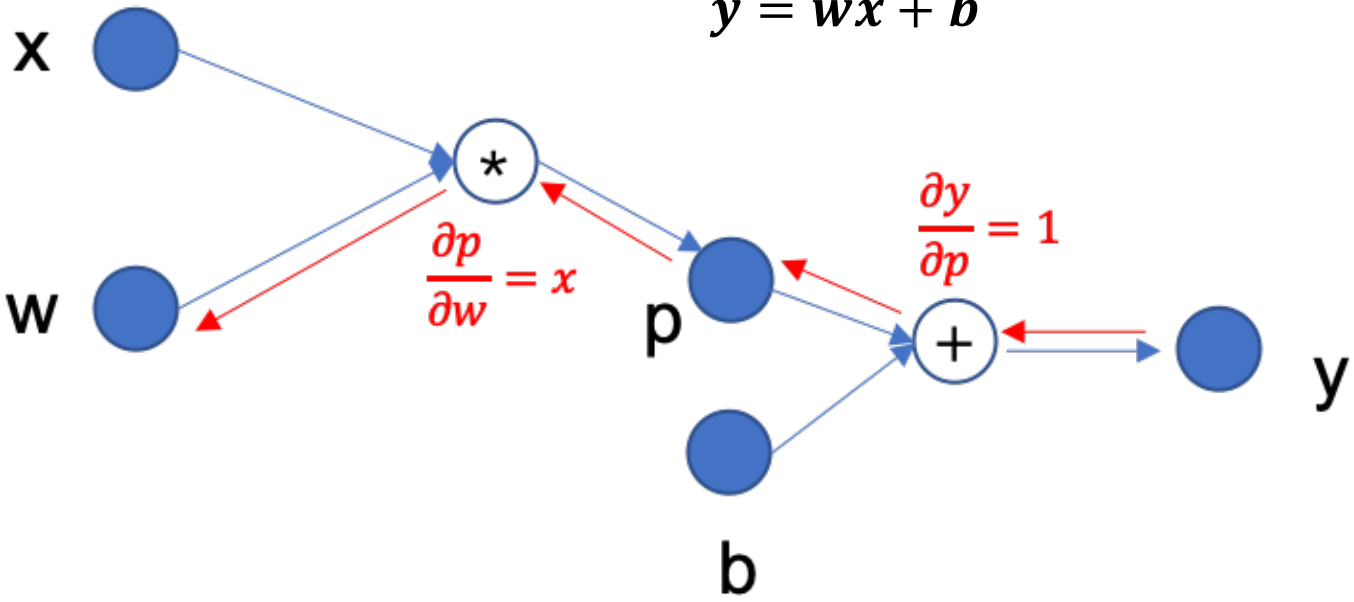
计算图

$$y = wx + b$$

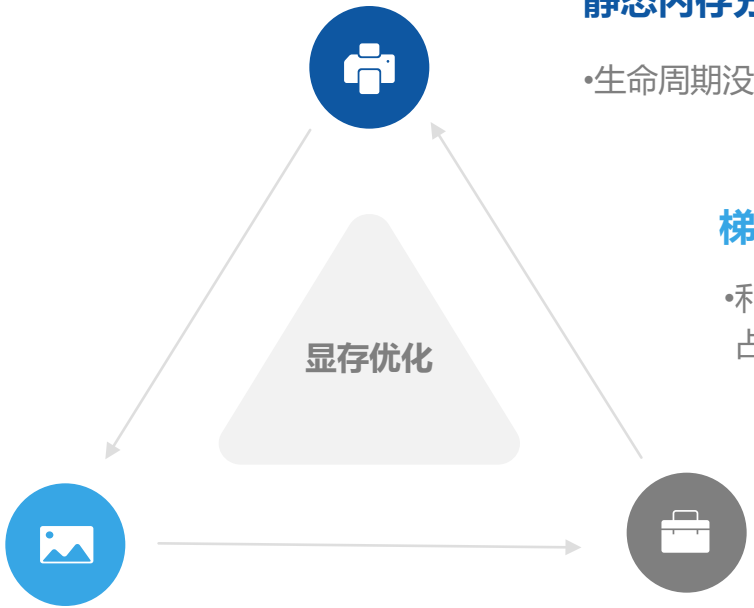


计算图

$$y = wx + b$$



静态图上的显存优化



静态内存分配

•生命周期没有重叠的算子之间可以共享显存

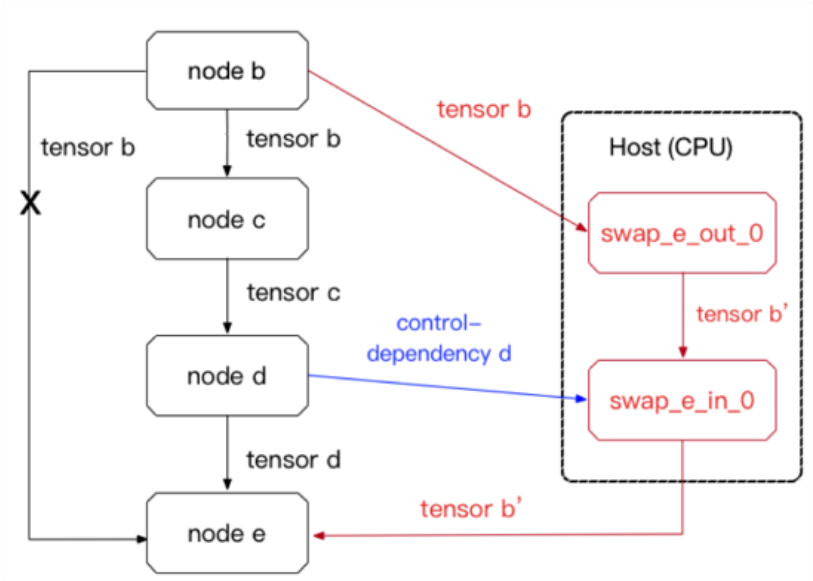
梯度检查点（Gradient Checkpointing）

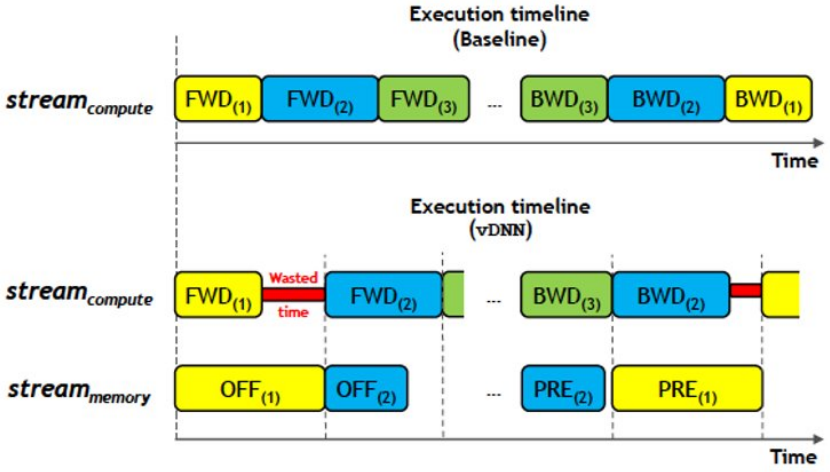
•利用梯度检查点重新计算中间结果减少显存占用

内存交换（Memory Swap）

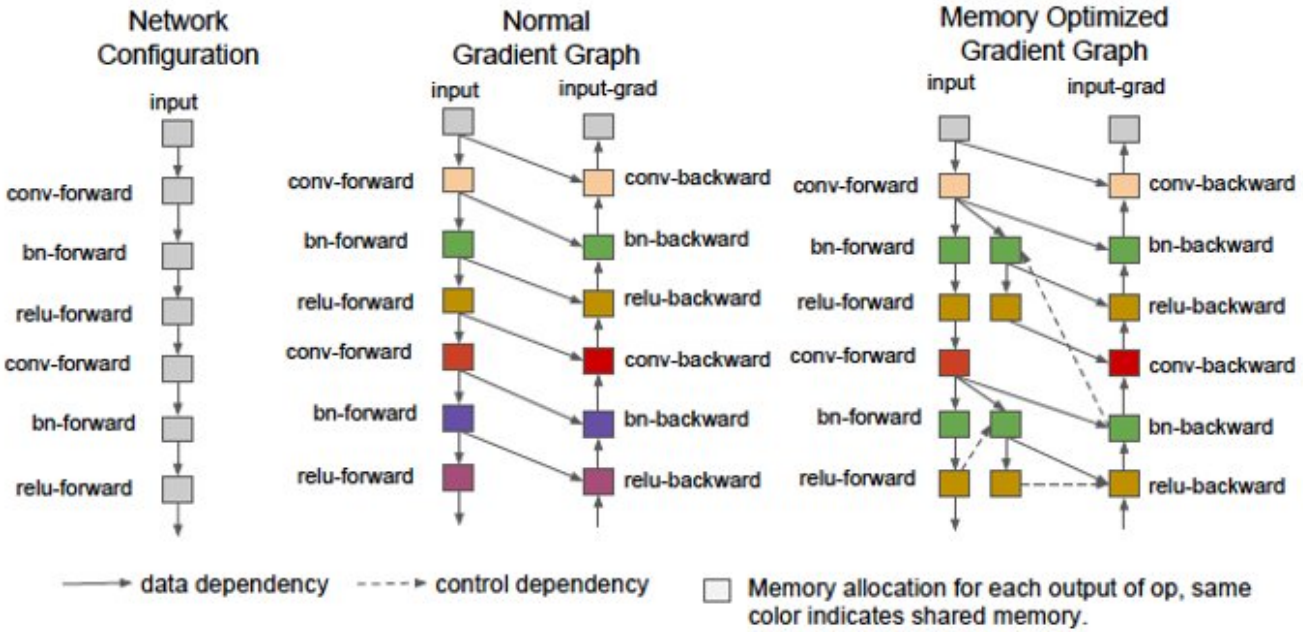
•把暂时不用的数据从 GPU 交换到 CPU，需要时再交换回来

用带宽换显存


node b
tensor b
tensor b
node c
tensor c
node d
tensor d
node e
X
tensor b
Host (CPU)
swap_e_out_0
tensor b'
control–
dependency d
swap_e_in_0
tensor b'


Execution timeline
(Baseline)
stream compute
FWD(1)
FWD(2)
FWD(3)
BWD(3)
BWD(2)
BWD(1)
Time
Execution timeline
(vDNN)
stream compute
Wasted
time
stream memory
OFF(1)
OFF(2)
PRE(2)
PRE(1)
Time

用计算换显存——静态图Sublinear显存优化



- $Memory = \max\{BlockSize\} + \sum Checkpoint = O(n/k) + O(k)$
- 取 $k = \sqrt{n}$，内存减少为原来的 $1/\sqrt{n}$，时间最多慢一倍

MEGVII 旷视

02

动态图显存优化

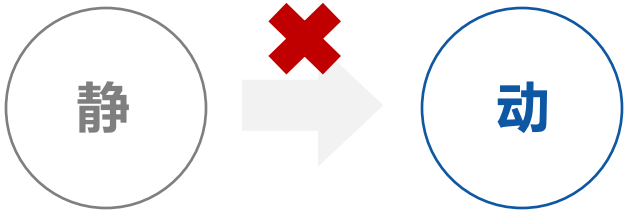
动态显存优化

MEGVII 旷视

需求：动态图里需要有显存优化

需求的背后：
训练超大模型

不能提前获得全局计算图信息



静态内存分配	→	无法提前得到生命周期
梯度检查点	→	可以动态计算
内存交换	→	可以有

方案一：用计算换显存

- 动态图版的 Sublinear，用计算换显存
- 反向求导时需要的中间结果从 checkpoint 重新计算

方案二：用带宽换显存

- GPU 和 CPU host 之间存储交换

用计算换显存 V.S. 用带宽换显存

MEGVII 旷视

算子	显存占用	计算耗时	交换耗时
Convolution	612.5MB	**3.71ms**	199.38ms
BatchNorm	153.1MB	**1.42ms**	49.85ms
ReLU	153.1MB	**0.81ms**	49.85ms

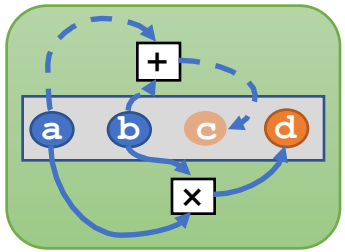
技术实现路径

MEGVII 旷视

需求：动态图里需要有显存优化

用计算换显存

实现基础设施



用户提供策略

```
y = Conv(x)
z = BN(y)
y.drop()
```

框架提供最优策略

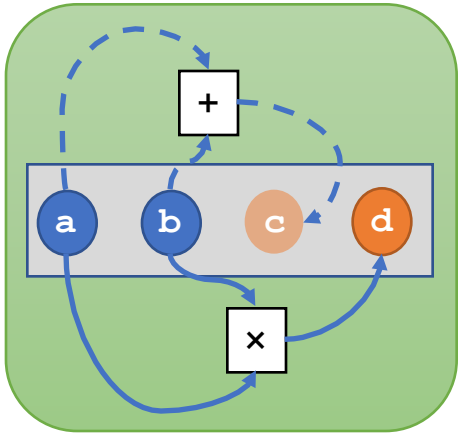
MegEngine

计算历史

MEGVII 旷视

```
struct ComputePath {
      std::shared_ptr<OpDef> op;
      SmallVector<TensorInfo*> inputs;
      SmallVector<TensorInfo*> outputs;
      double compute_time = 0;
} *producer;
SmallVector<ComputePath*> users;
size_t ref_cnt = 0;
```

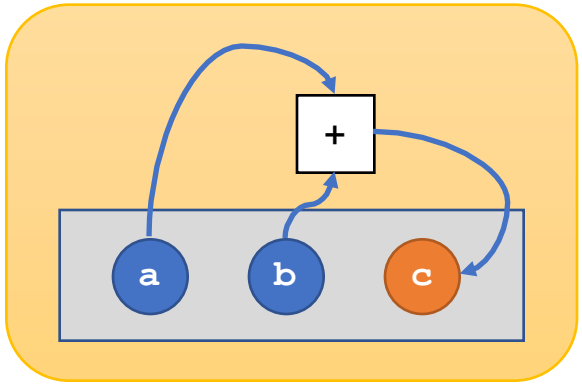
Tensor	op	inputs	outputs	user.outputs	ref_cnt
a	/	/	/	{c,d}	2
b	/	/	/	{c,d}	2
c	Add	{a,b}	{c}	/	0
d	Mul	{a,b}	{d}	/	0

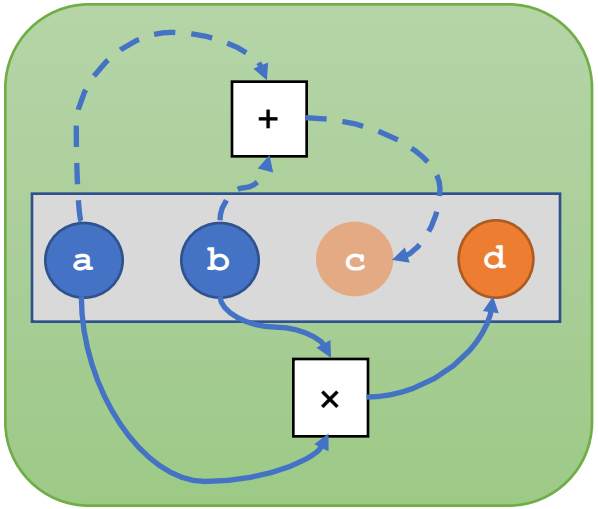


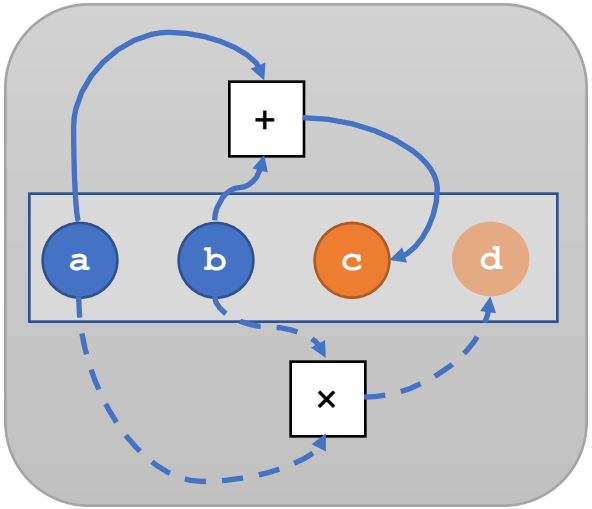
计算历史应用实例

MEGVII 旷视

```
from megengine import tensor

a = tensor([1, 2])

b = tensor([2, 1])

c = a + b

d = a * b

print(c)

print(d)
```



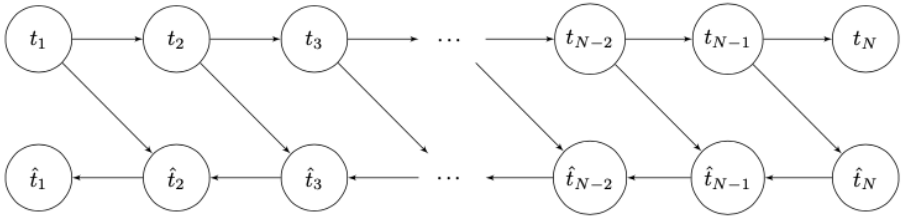




DTR[1] (Dynamic Tensor Rematerialization)

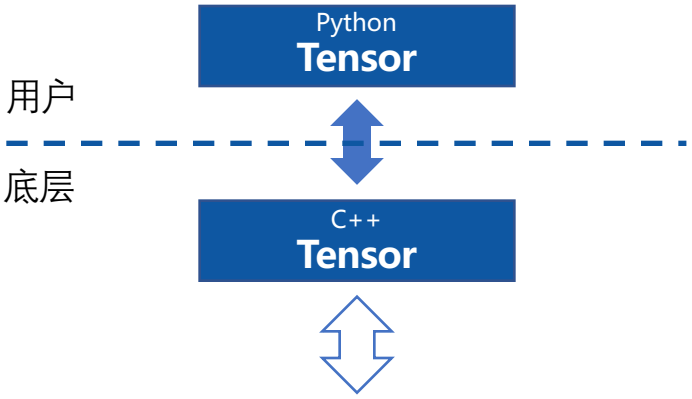
MEGVII 旷视

- 完全动态的启发式策略
- 每当显存超过一个阈值时，动态选择 drop 的 Tensor 释放掉一部分内存
- 对于 N 层的线性前馈网络，可以做到 op 执行次数为 $O(N)$ ，占用内存 $\Omega(\sqrt{N})$



[1] Kirisame M, Lyubomirsky S, Haan A, et al. Dynamic tensor rematerialization[J]. arXiv preprint arXiv:2006.09616, 2020.

动态的启发式策略



计算历史
- 算子
- 输入张量

属性值
- $M(t)$ 占用的内存
- $L(t)$ 进入显存的时长
- $C(t)$ 算子的运行时间

- 最小化启发式估价函数

$$f(t) = \frac{C(t)}{L(t) \cdot M(t)}$$

- 选择 Tensor 的标准：
  - 重计算的开销越小越好
  - 占用的显存越大越好
  - 在显存中停留的时长越大越好

DTR 的运行时优化

MEGVII 旷视

- Overhead 主要来自于**寻找应该被释放掉的最优 tensor**

$$f(t) = \frac{C(t)}{L(t) \cdot M(t)}$$

- 每个 tensor 的 $L(t)$ 在不同时刻是不一样的，所以只有在内存超过阈值的时候才会现场计算 $f(t)$
- 两个优化：
  1. 不考虑小 tensor（ < 0.01 * tensor的平均大小 ）
  2. 每次随机采样 $\sqrt{n}$ 个 tensor 进行遍历（ n 为目前可释放的 tensor 候选集的大小 ）

静态图 Sublinear V.S. 动态图 DTR

MEGVII 旷视

	静态图Sublinear	动态图DTR
• 静态分配显存	✓	
• 静态图优化	✓	
• 动态网络结构		✓
• 重计算多次		✓

03

MegEngine中的工程实现

Tensor Interpreter 介绍

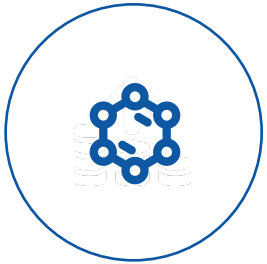
MEGVII 旷视

- Imperative Runtime 的 C++ 实现
- 多生产者单消费者队列，执行 4 种基础操作

Put

(HostTensorND)
→
Handle

把外部数据加载进一个Tensor中



ApplyOp

(OpDef, vector<Handle>)
→
vector<Handle>

执行一个 Op

Del

(Handle)

释放一个 Tensor

GetValue

(Handle)
→
HostTensorND

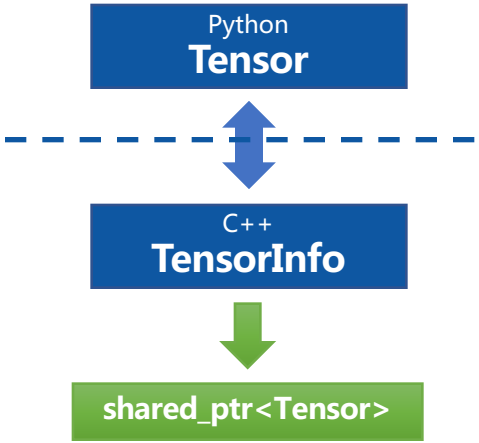
获取 Tensor 的值

Python → Interpreter → C++

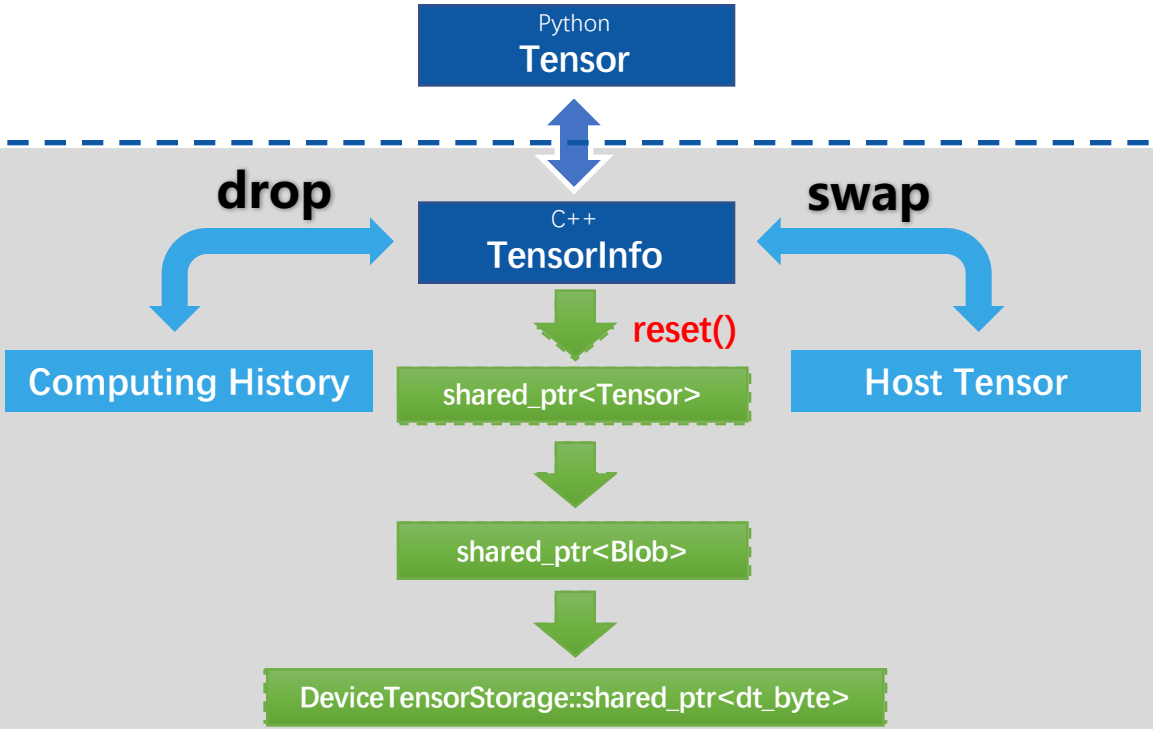
MEGVII 旷视

```
① x = Tensor(0)
② x = x + 1
③ x.numpy()
④ del x
```

```
① %1 = Put()
② %2 = Put()
③ %3 = ApplyOp(add, [%1, %2])
④ Del(%1)
⑤ Del(%2)
⑥ GetValue(%3)
⑦ Del(%3)
```



Tensor 重计算与换入/换出的底层实现



伪代码

MEGVII 旷视

ApplyOp(op, inputs):

对 inputs 的每个元素打上标记，禁止释放

AutoEvict()

检查 inputs 的每个元素 x：

若 x 不在内存中，调用 **Regenerate**(x)

apply_on_physical_tensor(op, inputs)

对 inputs 的每个元素撤消标记

Regenerate(x):

读取 x 的计算历史：op, inputs

x := ApplyOp(op, inputs)

AutoEvict():

while 显存占用 > 阈值:

x := **FindBestTensor**()

if x = NULL:

break

Drop(x)

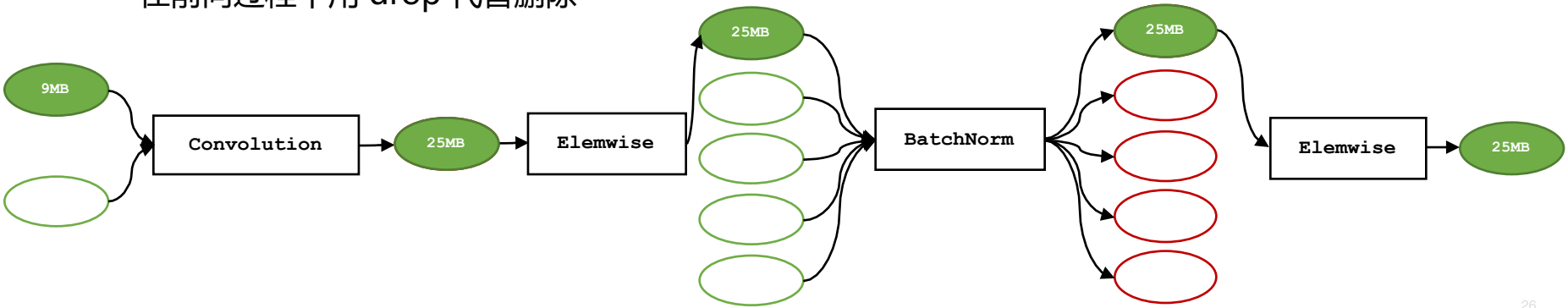
FindBestTensor():

枚举当前在显存中未被打上标记的 Tensor t

返回 f(t) 最小的 Tensor

删除的处理方法

- 删除用户释放的 tensor 可以立即节省内存，但是可能使策略变得局限
- 如下图
  - 绿色的 tensor 前向计算完之后仍被持有
  - 红色的 tensor 前向计算完之后被释放
  - 绿色的 tensor 永远不可能被 drop，因为失去了计算源
- 解决方法：
  - 在前向过程中用 drop 代替删除



删除的实现

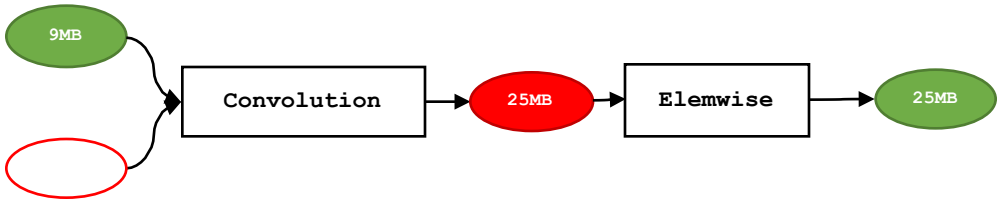
MEGVII 旷视

```
Free(x):
    x.delete = True
    if 当前状态 = 前向计算:
        Drop(x)
    else:
        RealFree(x)

RemoveDep(x):
    枚举依赖 x 的输出(dep_outputs) i:
        Regenerate(i)
```

```
RealFree(x):
    枚举 x 的计算历史输入 i:
        i.ref_cnt := i.ref_cnt - 1
    RemoveDep(x)
    析构 x
    枚举 x 的计算历史输入 i:
        if i.delete and i.ref_cnt == 0:
            RealFree(i)
```

x 的输入的 ref_cnt --

若 x 的输入之前被假删除，并且ref_cnt=0，执行真删除



DTR 使用方法

MEGVII 旷视

```python
import numpy as np
import resnet
import megengine as mge
import megengine.autodiff as ad
import megengine.functional as F
import megengine.optimizer as optim

from megengine.utils.dtr import DTR
dtr = DTR(memory_budget=5 * 1024 ** 3)

batch_size = 64
image = mge.tensor(np.random.random((batch_size, 3, 224, 224)))
label = mge.tensor(np.random.randint(100, size=(batch_size,)))
model = resnet.__dict__["resnet50"]()

gm = ad.GradManager().attach(model.parameters())
opt = optim.SGD(model.parameters(), lr=0.0125, momentum=0.9, weight_decay=1e-4,)

for i in range(n_iter):
    with gm:
        logits = model(image)
        loss = F.nn.cross_entropy(logits, label)
        gm.backward(loss)
        opt.step().clear_grad()
        print("iter = {}, loss = {}".format(i + 1, loss.numpy()))
```

——只需添加两行

ResNet 1202 训练情况对比

MEGVII 旷视

(Titan V 12G)	ResNet-1202 (Batch Size)			
	64	100	120	140
PyTorch-DTR	0.974s	1.18s	1.28s	1.39s
PyTorch	0.712s	X	X	X

表1: DTR 在 PyTorch 上的实验数据[1]

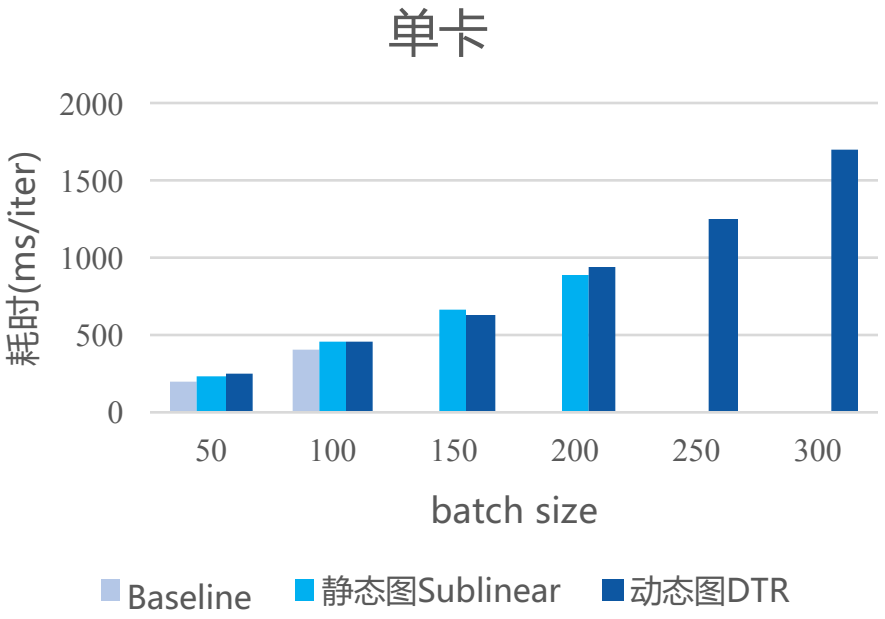
(2080Ti 11G)	ResNet-1202 (Batch Size)						
	64	100	120	140	200	250	300
MegEngine-DTR	0.691s	0.876s	1.11s	1.33s	**2.07s**	**2.42s**	**2.97s**
MegEngine	0.673s	**0.852s**	X	X	X	X	X

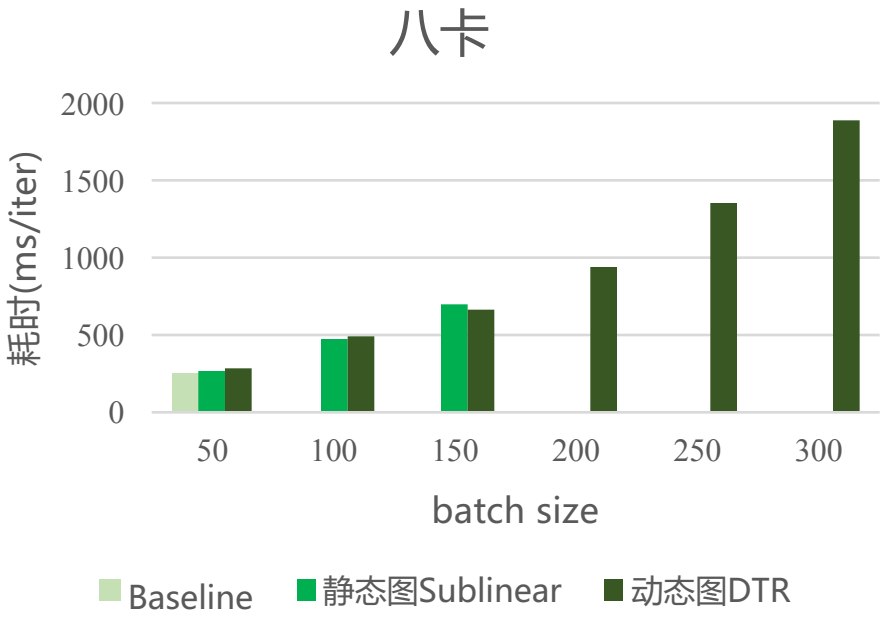
表2: DTR 在 MegEngine 上的实验数据

[1] Kirisame M, Lyubomirsky S, Haan A, et al. Dynamic tensor rematerialization[J]. arXiv preprint arXiv:2006.09616, 2020.

ResNet 50 训练耗时对比

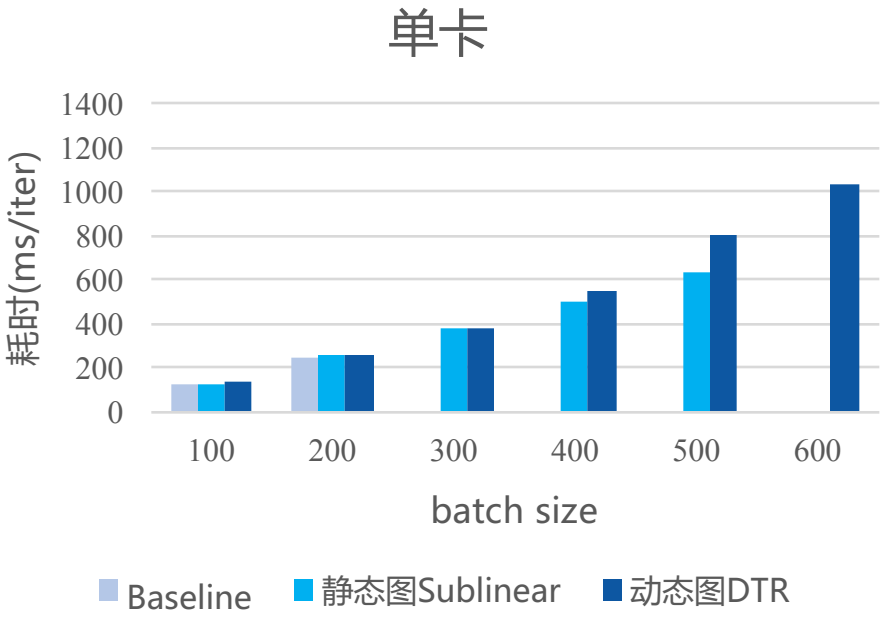
MEGVII 旷视

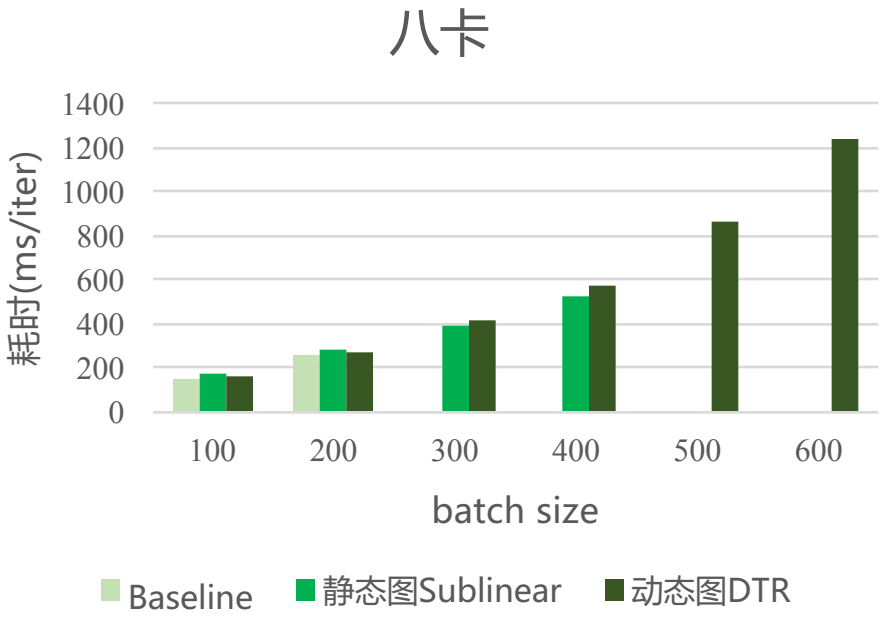




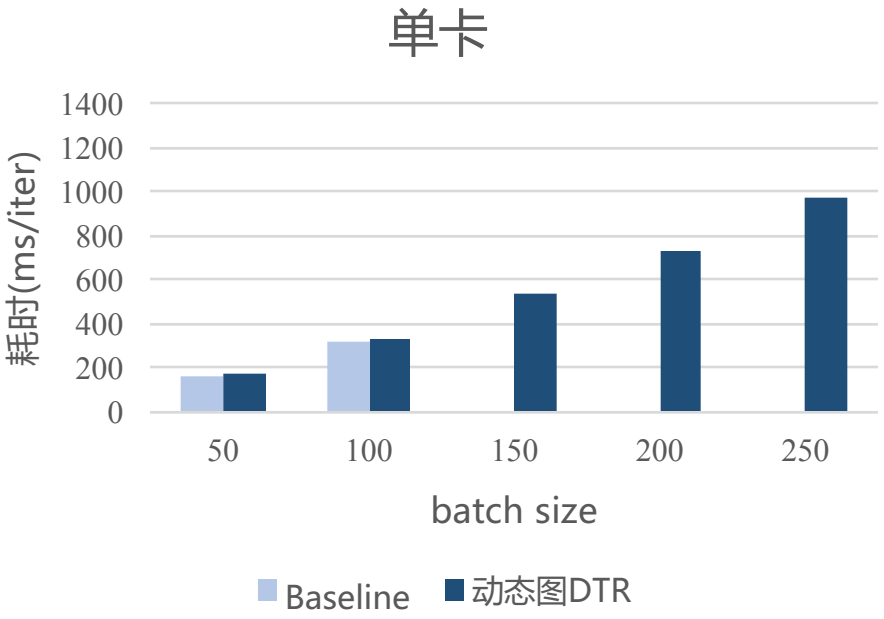
ShuffleNet 训练耗时对比

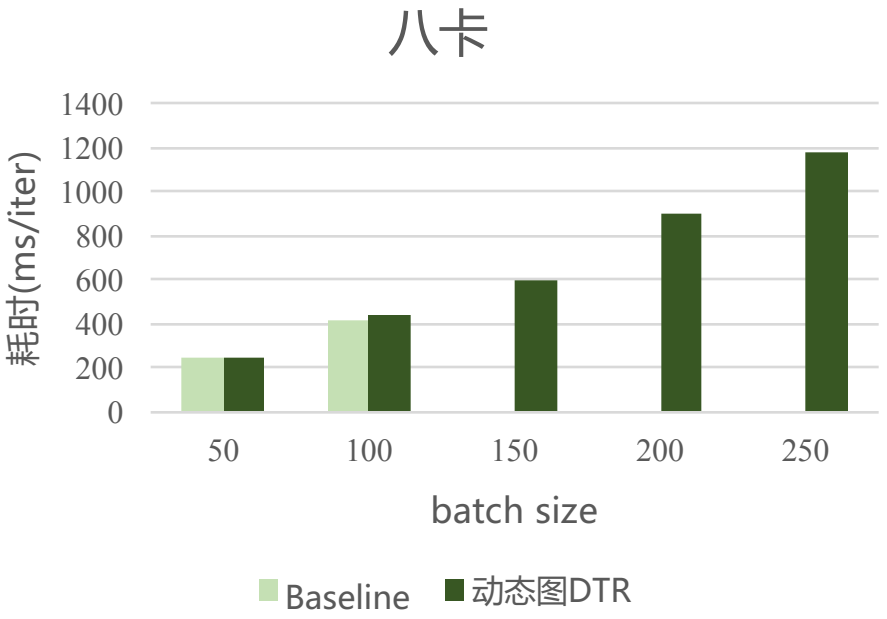
MEGVII 旷视





SPOS[1] 训练耗时对比



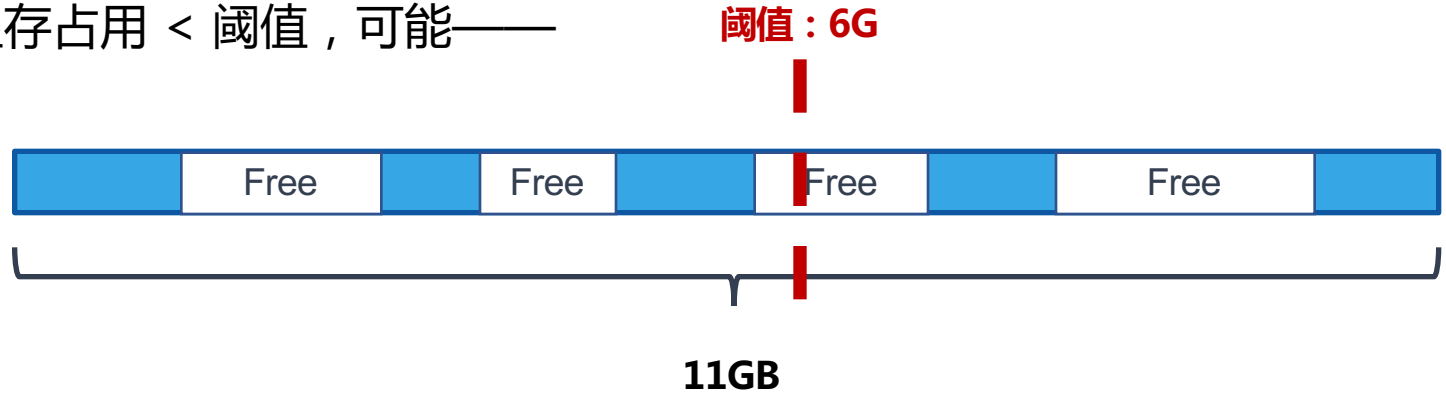


[1] Guo Z, Zhang X, Mu H, Heng W, Liu Z, Wei Y, Sun J. Single path one-shot neural architecture search with uniform sampling. In European Conference on Computer Vision 2020 Aug 23 (pp. 544-560). Springer, Cham.

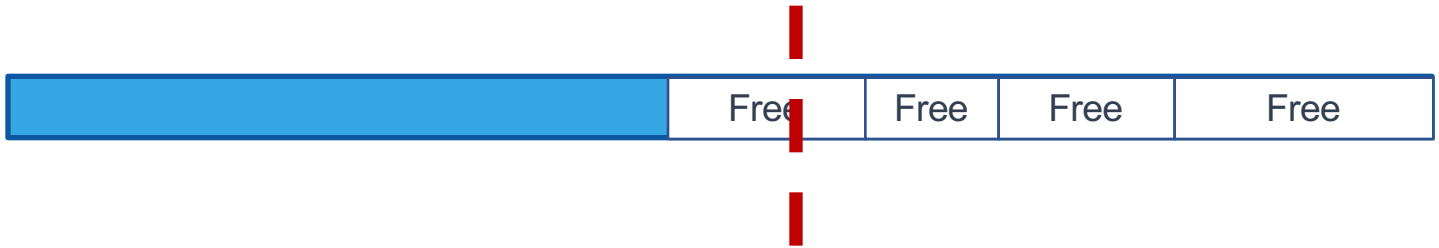
碎片问题

MEGVII 旷视

- 显存占用 < 阈值，可能——



- 申请一块较大的内存时，仍然无法满足请求，需要整理碎片



针对碎片问题的优化

MEGVII 旷视

■ 参数原地更新

- 打开 `MEGENGINE_INPLACE_UPDATE` 环境变量

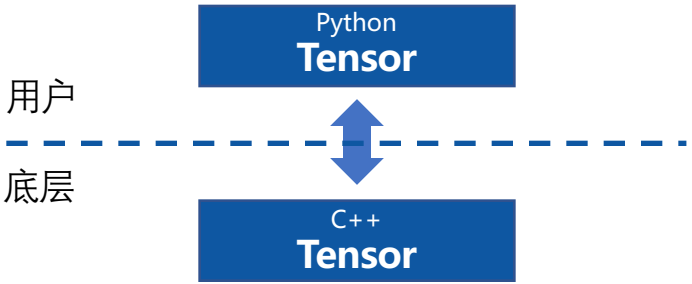
■ 改进估价函数

- 引入碎片相关信息

■ 静态规划策略

- 对显存静态分配

引入碎片相关的信息



计算历史
- 算子
- 输入张量

属性值
- $M(t)$ 占用的内存
- $L(t)$ 进入显存的时长
- $C(t)$ 算子的运行时间
- $R(t)$ 重计算的次数
- $S(t)$ **该 Tensor 内存前后的空闲段大小**

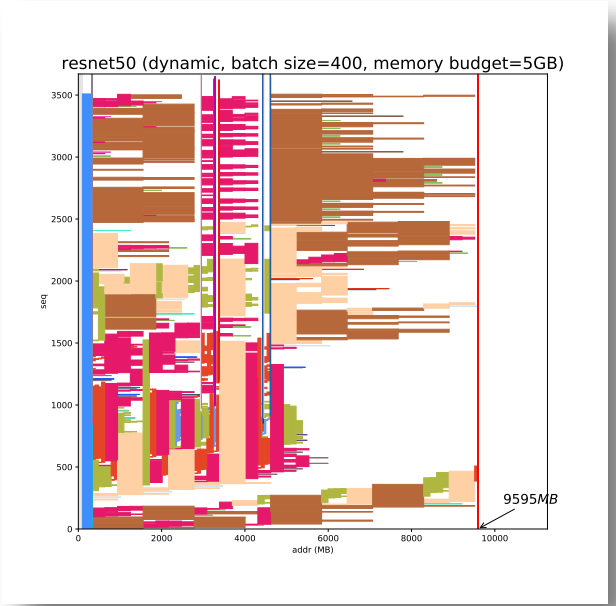
- 最小化启发式估价函数

$$f(t) = \frac{\text{计算耗时}^{\alpha}}{\text{停留时长}^{\beta}\,(\text{显存} + \text{空闲段大小})^{\gamma}}\,\delta^{\text{重计算次数}}$$

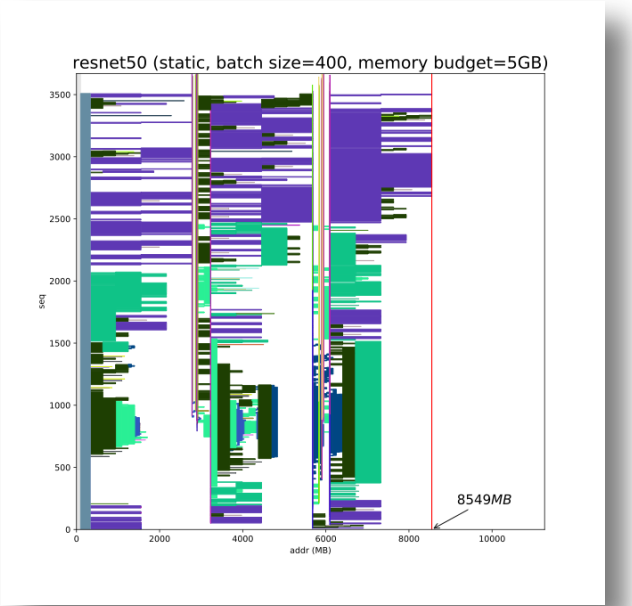
静态分配显存

MEGVII 旷视

9595MB → 8549MB



Best-fit算法
动态分配



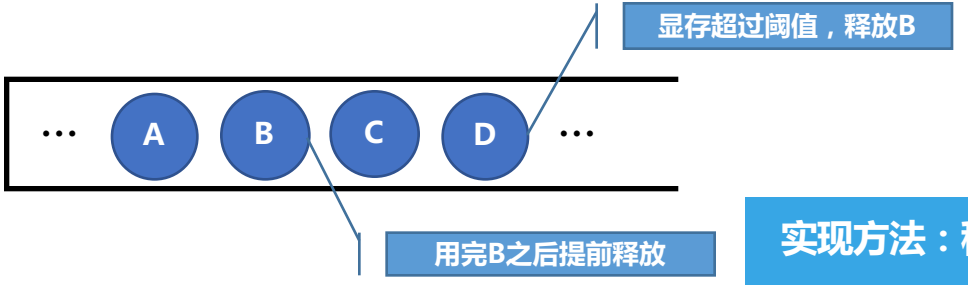
Pushdown算法
静态分配

MEGVII 旷视

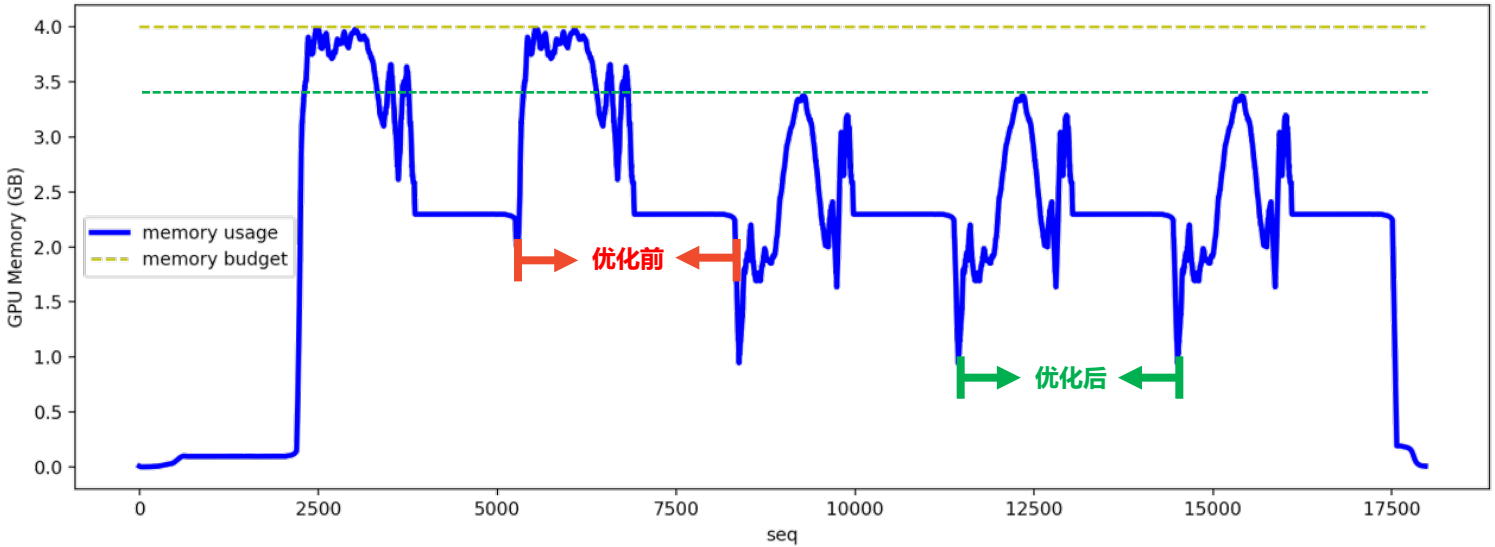
04

未来工作

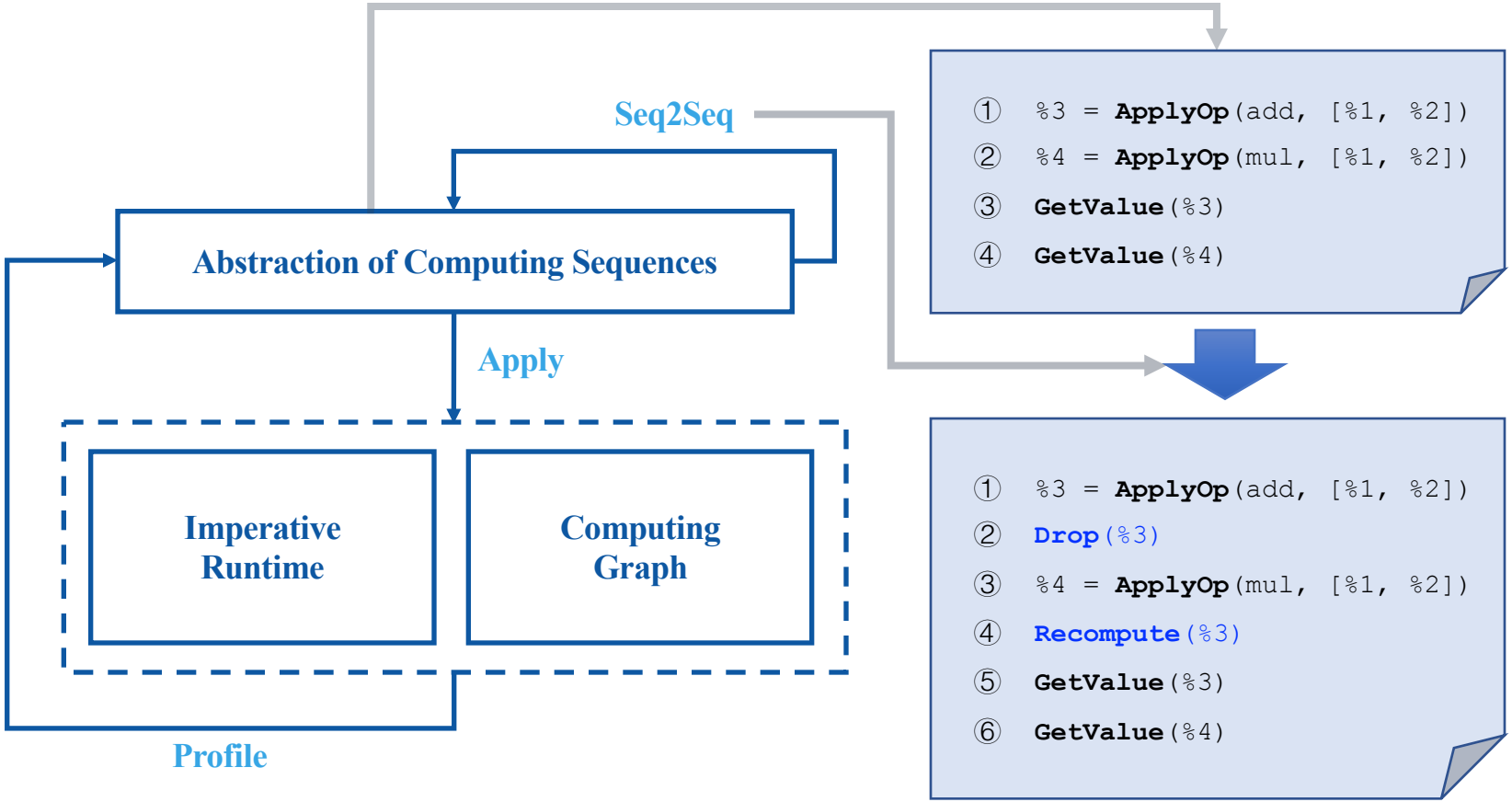
提前释放被选中的 tensor



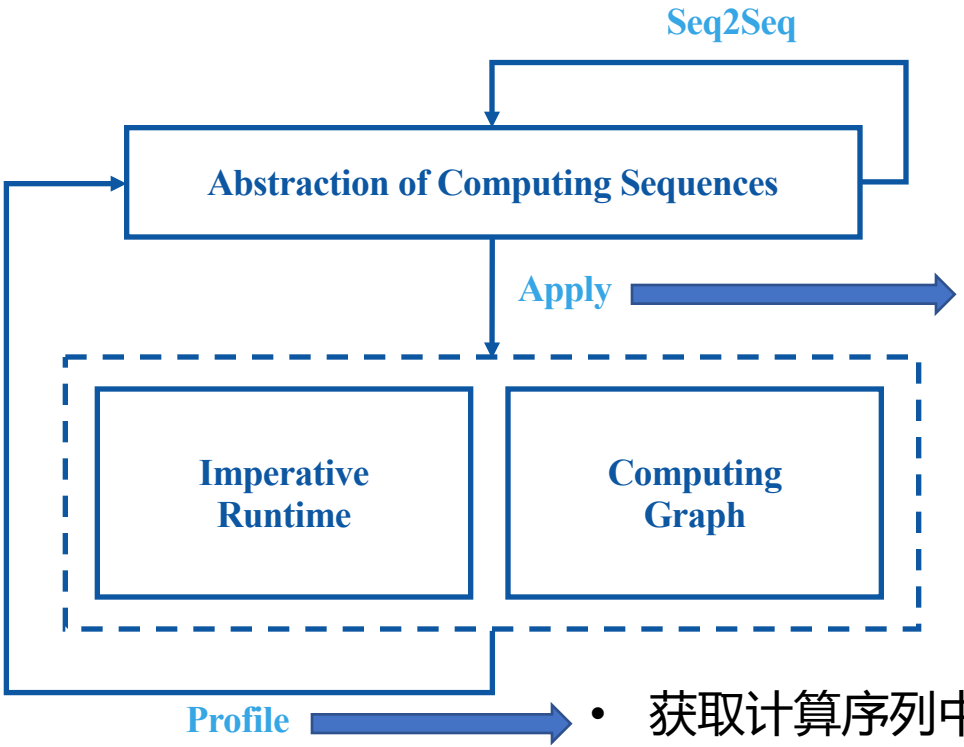
实现方法：移动 drop(B) 的位置



显存优化策略的抽象


Seq2Seq
Abstraction of Computing Sequences
Apply
Imperative Runtime
Computing Graph
Profile
① %3 = ApplyOp(add, [%1, %2])
② %4 = ApplyOp(mul, [%1, %2])
③ GetValue(%3)
④ GetValue(%4)
① %3 = ApplyOp(add, [%1, %2])
② Drop(%3)
③ %4 = ApplyOp(mul, [%1, %2])
④ Recompute(%3)
⑤ GetValue(%3)
⑥ GetValue(%4)

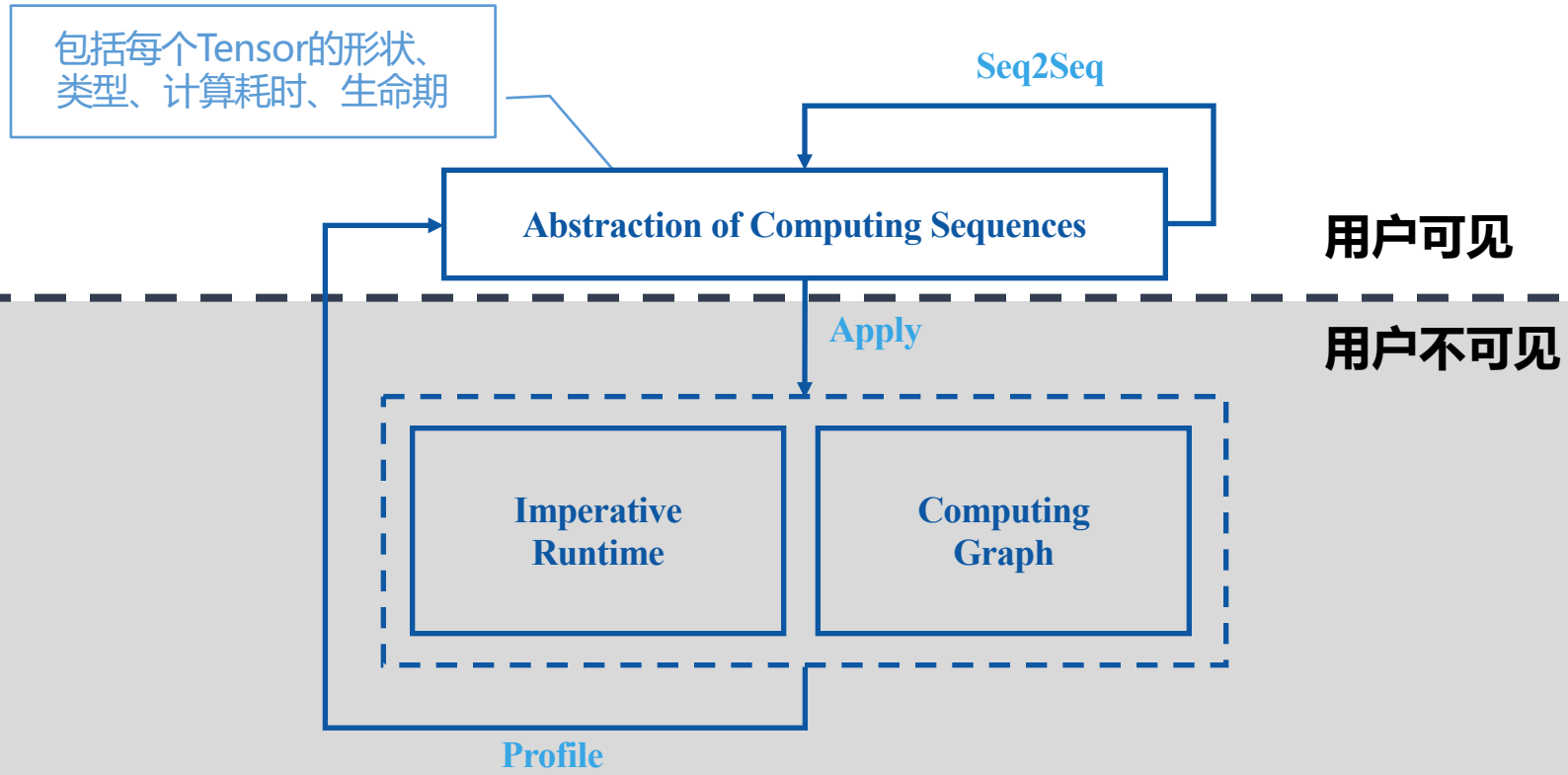
显存优化策略的抽象



- Imperative Runtime
  - 解释执行
- Computing Graph
  - 生成计算图后执行

- 获取计算序列中每个 op 的运行时间，每个 tensor 的生命期等信息
- 根据 Profile 的结果来调整计算序列

用户定制策略


包括每个Tensor的形状、
类型、计算耗时、生命期
Seq2Seq
Abstraction of Computing Sequences
用户可见
用户不可见
Apply
Imperative
Runtime
Computing
Graph
Profile

用人工智能造福大众

MEGVII 旷视