AI训练显存优化
论文综述

汇报人：田子昂

2023.12.25

武漢大學
WUHAN UNIVERSITY

目录 CONTENTS

武漢大學 WUHAN UNIVERSITY

1 背景：DL的发展趋势 — 参数多，层数深

- 768个隐层的BERT，在Batch size设置为64时需要**73GB**的显存空间[2]；
- 使用ImageNet训练Wide ResNet-152，并设置Batch size为64需要显存**180GB**

指标 / GPU	显存容量/GB	显存带宽/Gbps	Tensor Core	FP32 峰值/TFLOPS
V100(SXM2)	32 HBM2	900	640	15.7
TITAN RTX	24 GDDR6	672	576	16.3
P100(SXM2)	16 HBM2	732	NA	10.6
TITAN V	12 HBM2	652.8	640	15
RTX 2080Ti	11 GDDR6	616	544	13.4
RTX 2080	8 GDDR6	448	368	10.1
RTX 2070	8 GDDR6	448	288	7.5
TITAN Xp	12 GDDR5X	547.7	NA	12
RTX 1080Ti	11 GDDR5X	484	NA	11.3
TITAN X	12 GDDR5	336.5	NA	11
GTX 1080	8 GDDR5X	484	NA	8.9
RTX 1070Ti	8 GDDR5	256	NA	8.1
RTX 1070	8 GDDR5	256	NA	6.5
RTX 1060	6 GDDR5	256	NA	4.4

目前(2023年)最高性能的H100刚发布时显存仅80GB，价格昂贵

2 训练过程：显存占用

假设一个batch在训练开始的时候的尺寸是16*3*224*224，使用fp32格式，那么占用的显存也就是16*3*224*224*4B≈9MB.

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTe`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTe`
16-bit floating point	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTens`
8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTens`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTens`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortTe`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTens`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTens`

Pytorch的数据格式

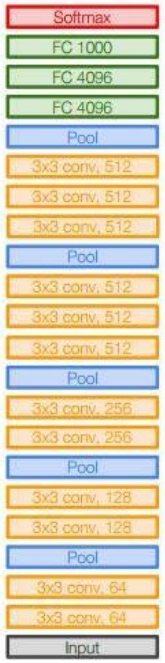
2 训练过程：显存占用

但是还有巨大的内存开销
产生于：

- 各个层的中间结果的存储（feature map）
- 梯度的存储和更新
- 模型的权重矩阵

INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters



VGG16

各个层计算产生的中间变量

2 训练过程：显存占用

设备的显存不足模型的训练所用，即显存不足。

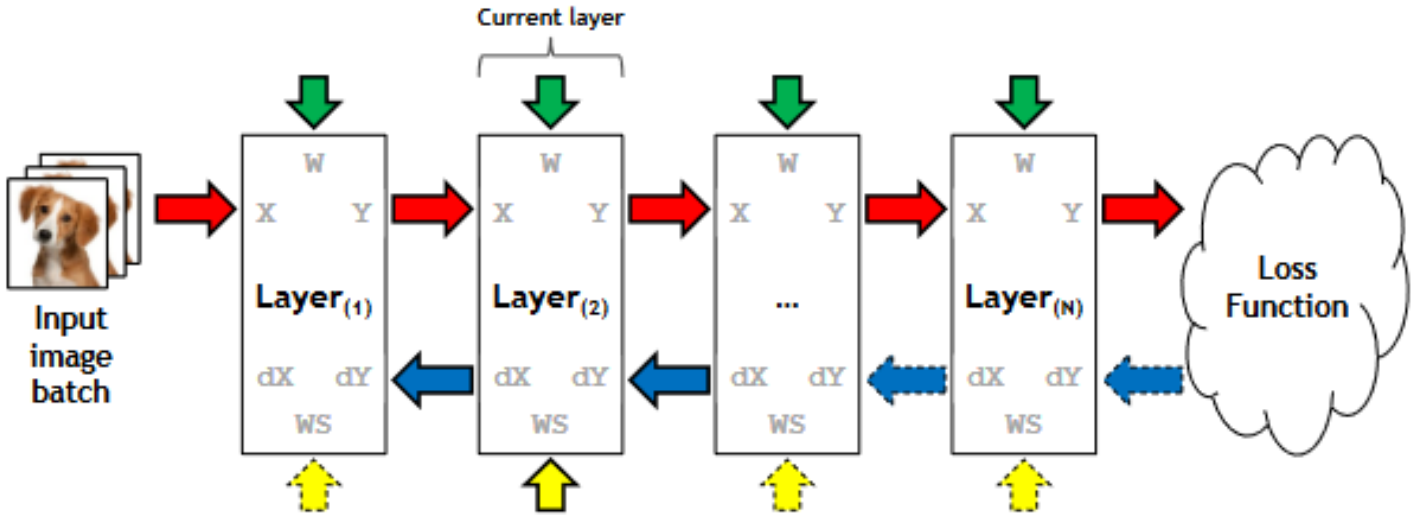
而从2016年的vDNN开始，前沿工作主要包含了以下的解决方案：

- 数据转移
- 重计算
- 压缩
- 利用CPU分流计算

3 GPU-CPU的数据转移

核心思想：

- CPU的内存往往比GPU的大得多。
- 因为DNN是按照层进行排列的，每一个时刻GPU其实只训练某个层，因此有一些层的数据暂时不需要访问，所以这些数据便可以转移出去。
- 前向传播时选择卷积层的前一层的输出数据进行转出，反向传播将数据提前转移回来。

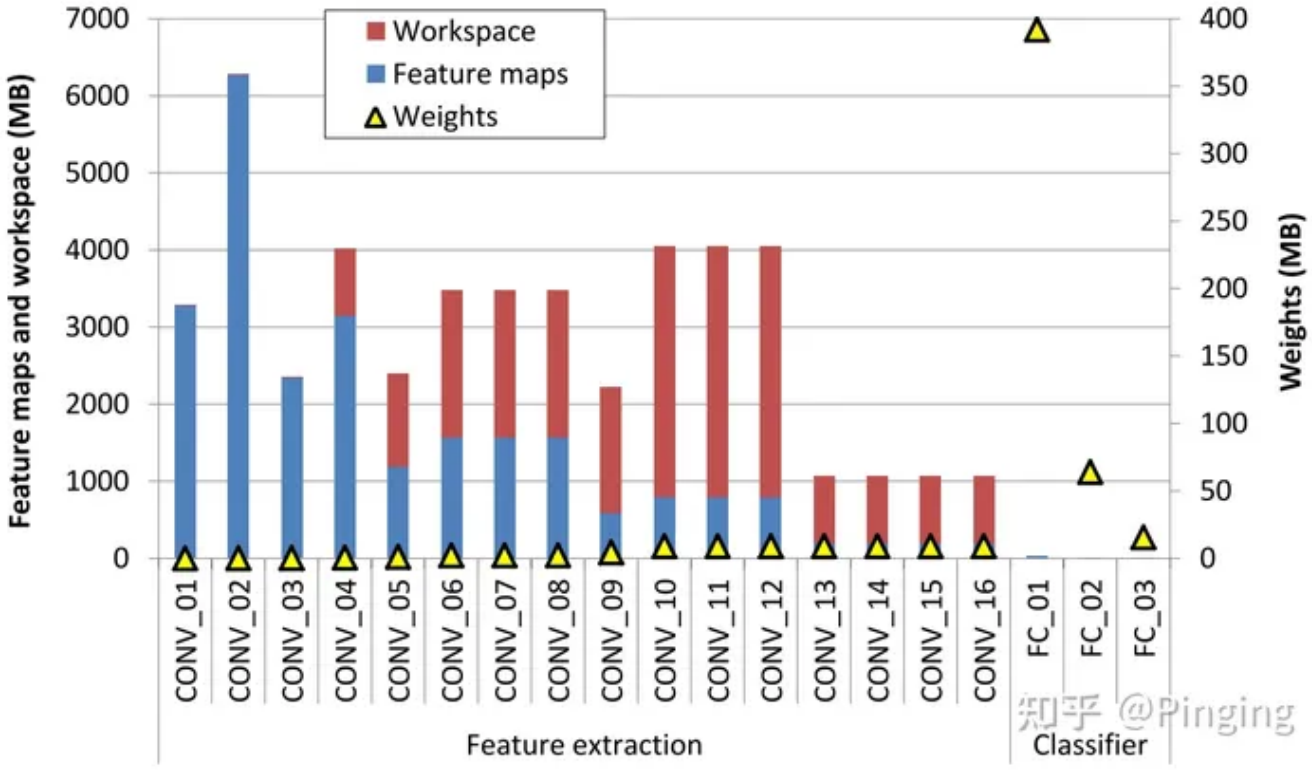


数据转移代表1：vDNN(MICRO 2016)

对当时常用的CV模型进行了测试，发现Feature map类型的数据占空间最大。

而feature map中又属CONV层占用的空间特别多。

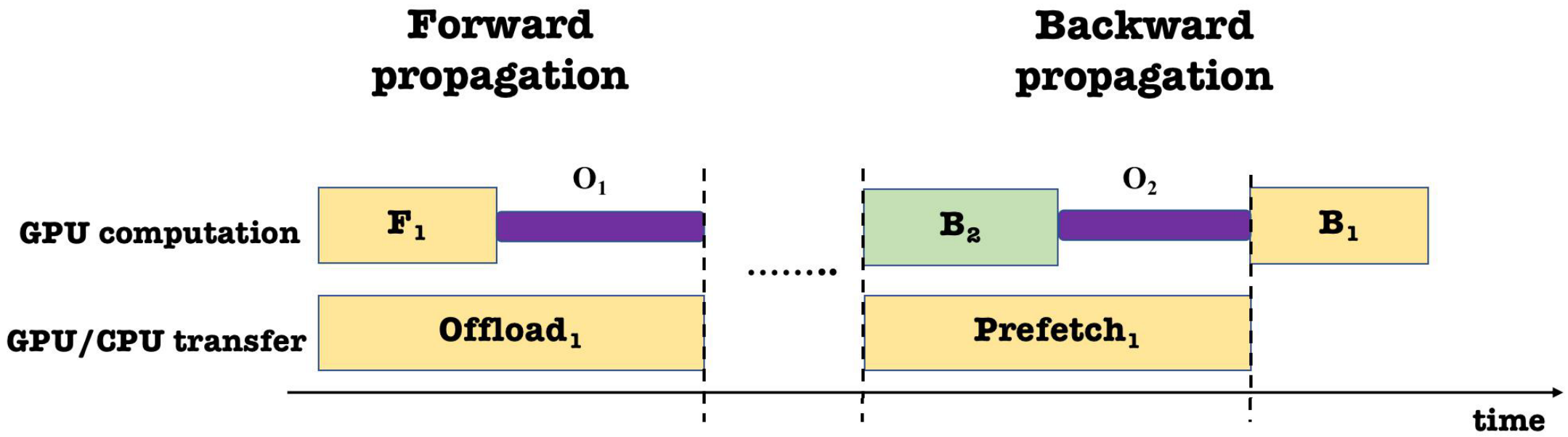
直接把CONV层的Tensor在forward pass的时候转出到CPU里面，等到反向传播的时候的时候再传回来。



各层数据占用空间情况

数据转移代表1：vDNN(MICRO 2016)

为了保障系统不崩溃正常运行，所以每一次转入转出操作调用的时候，都需要在该层最后加一个同步指令，从而保障这一层已经转出成功了，才能进行下一层的训练。



紫色部分为GPU浪费在同步等待上的时间

数据转移代表2：moDNN(DATE 2018)

vDNN一个明显可以改进的地方：

- 在选择转移的数据的时候，以层为单位，并且一概把CONV层的结果Offload，错失了很多其他优化的机会。

moDNN在此基础上引入的两个新的考虑点是：

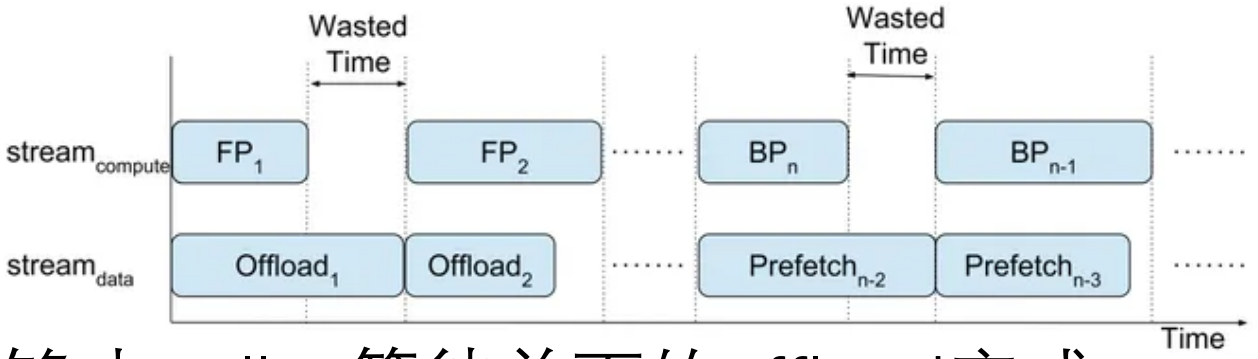
1. 区分不同的卷积层，进行trade-off: 快的卷积函数有可能需要更多的内存释放操作，所以预取数据就会造成delay。所以需要针对“收益”进行最优化的方案选择。

2. 引入sub-batch:大的batch size会提升性能（一次训练的比较多，并行度高），但是反而会使得空间不足，影响性能；小的batch size训练又比较慢。

数据转移代表3：vDNN++(IPDPS 2018)

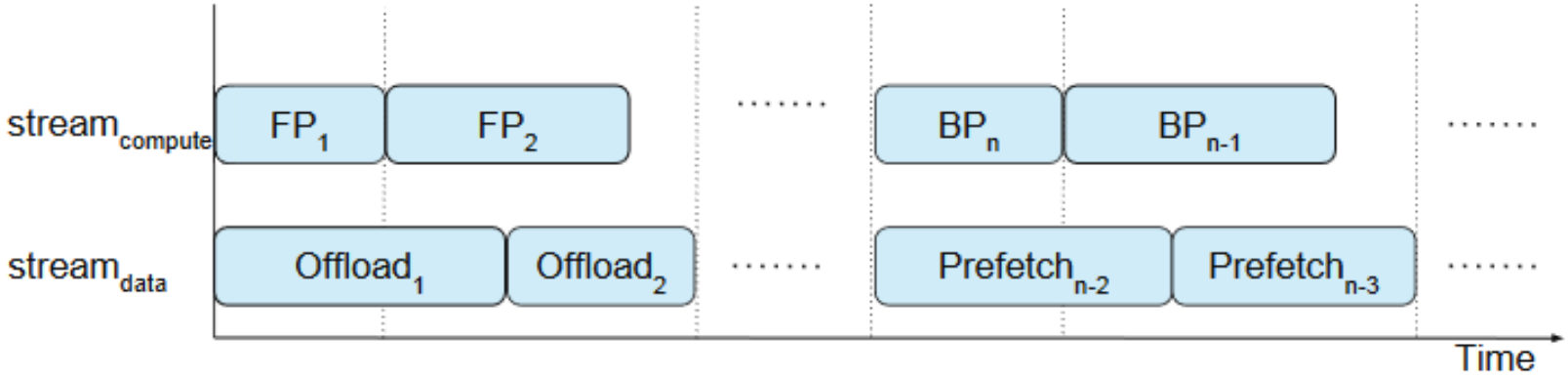
vDNN另一个明显的可以改进的地方：

- 同步过程会浪费大量的时间。



vDNN++提出：

- 在forward的时候，只有当内存不够才stall，等待前面的offload完成；
- 在backward的时候，只有当需要的数据没有，才stall计算来等prefetch。



数据转移代表4：AutoTM(ASPLOS 2020)

背景知识：存储

- Traditional SSD 固态硬盘
- DRAM 用作内存；volatile；是传统的RAM的一种形式。 Byte-addressable
- NVM：non-volatile memory，范围很广
- PM：persistent memory；不只是NVM，而且通常还指代类似于RAM的的访问速度的存储设备。
- NVRAM：和PM是类似的概念，PM更加广泛，包括NVRAM 和3D Xpoint。它们都是介于传统的ROM和RAM之间的技术。 Byte-addressable
- Intel Optane…:Intel推出的存储系列，包括Intel Optane DC Persistent Memory和Intel Optane SSD。都使用了3D Xpoint。 Optane SSD后来演变成了Optane DC PM。Byte-addressable
- Optane PMM有2种模式：2 level mode(PM acts as system memory while DRAM as cache); app direct mode(mount PMM as system file. The total memory is PMM + DRAM).

数据转移代表4：AutoTM(ASPLOS 2020)

1. 使用CPU的DRAM和NVM进行策略搜索。（Optane DC的NVDIMM）
2. 在静态图上优化。规划的结果是：**在何时，将哪个tensor，放在哪个存储区域**

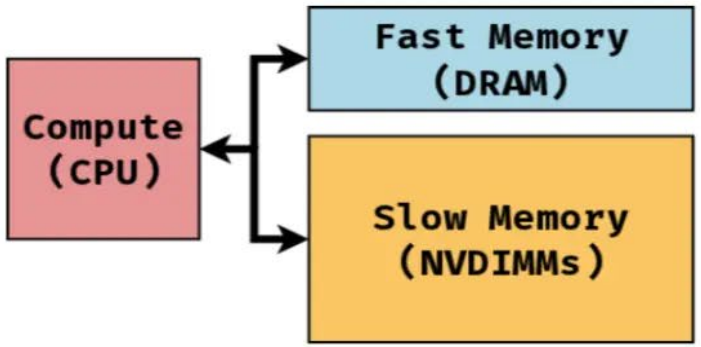
Heterogeneous Memory Systems

- Two types of memory.
- Same memory controller.
- Both are byte addressable.
- NVDIMMs for high **capacity** and low **cost**

Challenges

- All tensors in NVDIMMs memory is **too slow**.
- DRAM as a cache for NVDIMMs also **too slow**.
- Intelligent memory management required.

NVDIMM Style

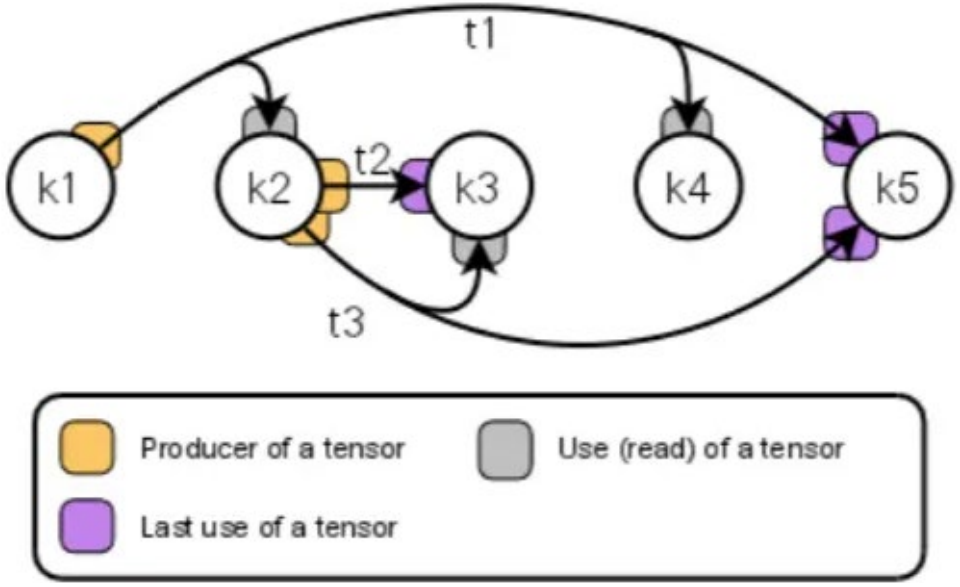


使用NVDIMM进行计算的风格

数据转移代表4：AutoTM(ASPLOS 2020)

编译时形成静态图

关键信息为节点(kernel)和时序关系。



kernel profiling

对给定的kernel，改变输入/输出tensor的位置，测量性能。

Kernel	IO Tensor Locations		
	T1	T2	T3
K2	DRAM	DRAM	DRAM
	DRAM	DRAM	PMM
	DRAM	PMM	DRAM
	DRAM	PMM	PMM
	PMM	DRAM	DRAM
	PMM	DRAM	PMM
	PMM	PMM	DRAM
	PMM	PMM	PMM

以k2为例，改变input/output的位置，测量性能

数据转移代表4：AutoTM(ASPLOS 2020)

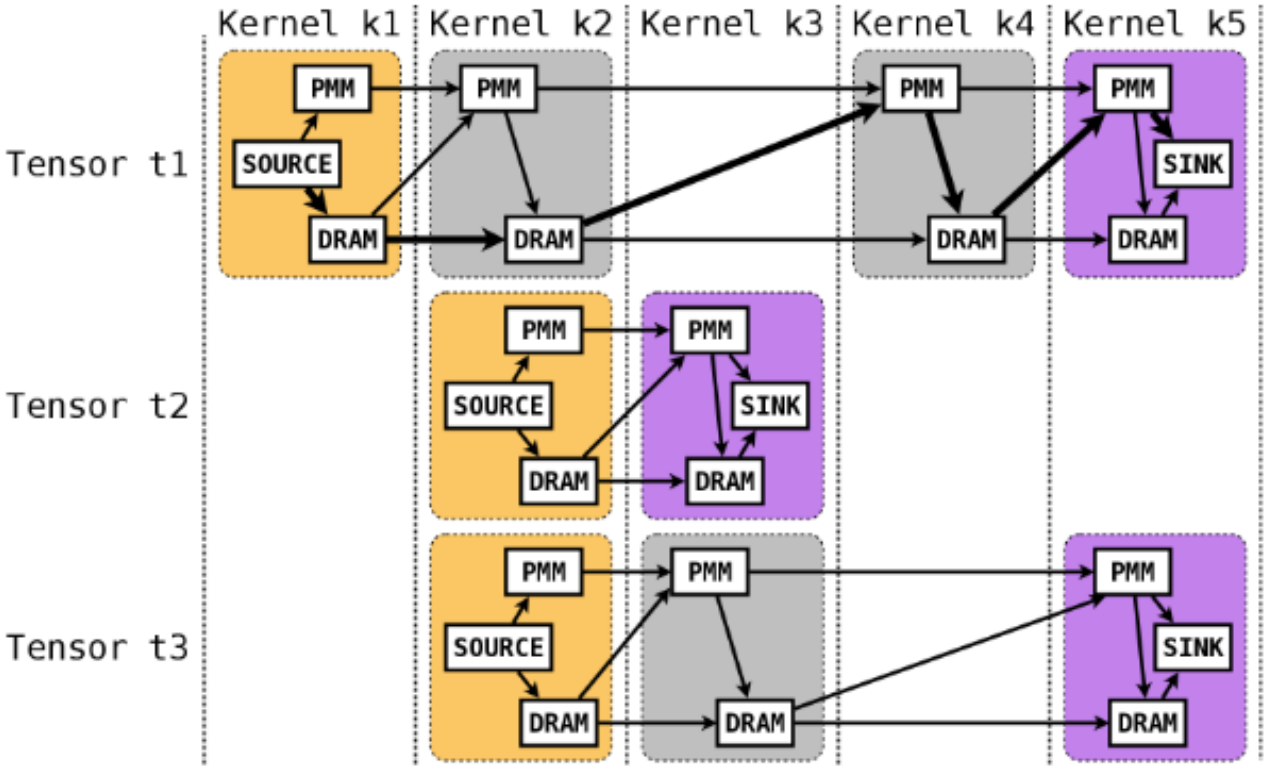
整数规划问题

掌握了每个kernel的“习性”之后，相当于一个规划问题。

- 目标函数: min $(T_{总kernel执行时间} + T_{总转移时间})$

- 约束条件：任意时刻出现在DRAM中的tensor不应该超过其容量

选择一个合适的路径，在约束条件下，使目标函数最小



数据转移代表5：SwapAdvisor (ASPLOS 2020)

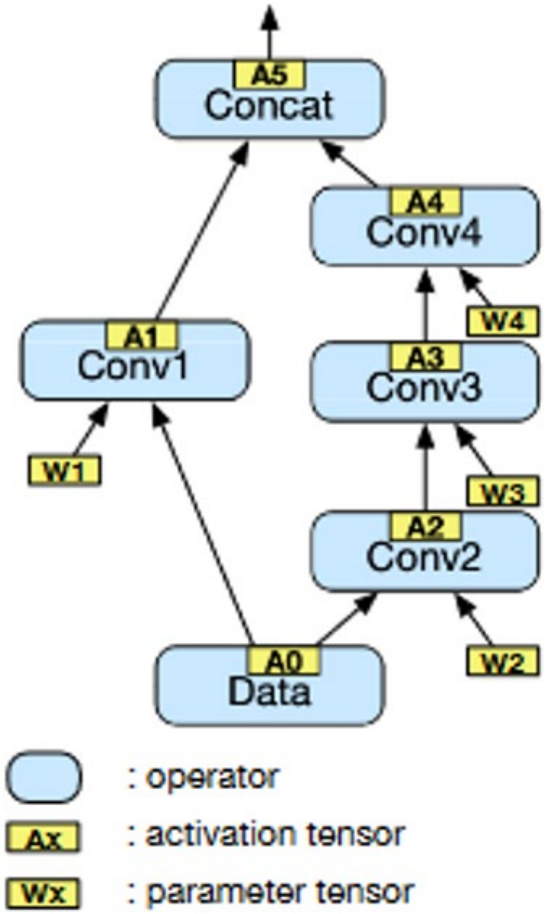
与AutoTM的规划思想一致，只是使用的是遗传算法，
染色体由以下两个调度描述：

Operator Scheduler

对于给定的图G，一个scheduler就是一个拓扑的执行顺序

Memory Allocator

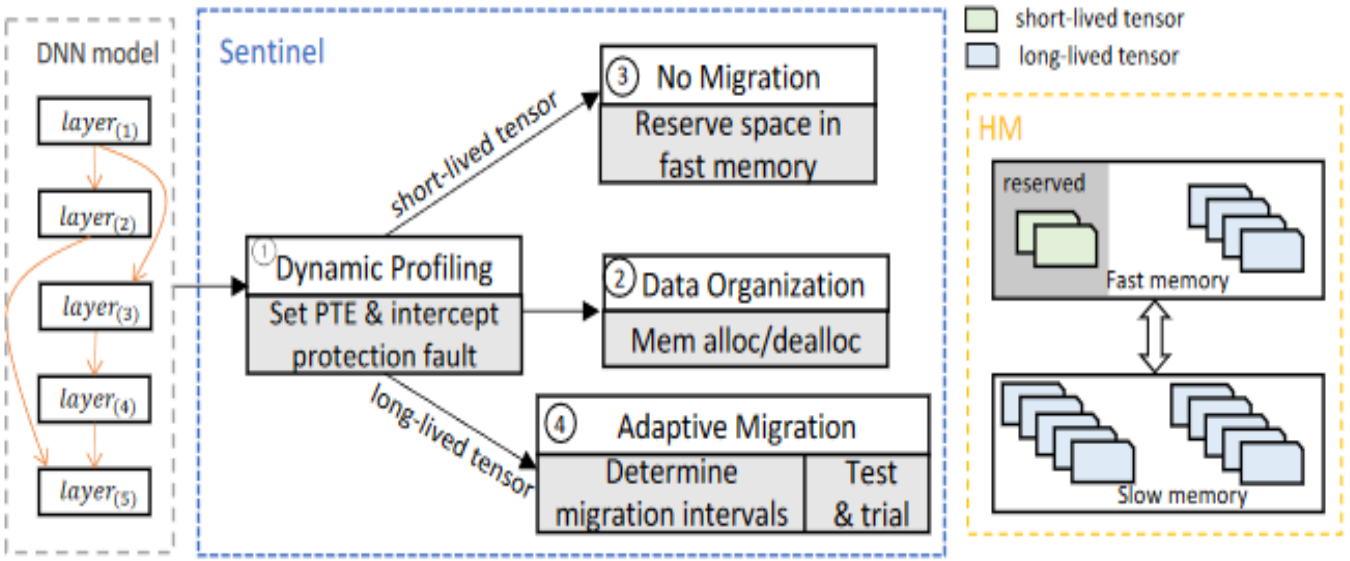
在内存池里，把哪些种类的内存块分给一个tensor



数据转移代表6：Sentinel (HPCA 2021)

与AutoTM一样，Sentinel关注的是NVM与DRAM的结合。

- 更加微观，或许受到了multi-threading里false sharing的启发？
- 抓住的是转移页的时候，生命周期不同的的数据被分配到了同一个页中。这样会像false sharing在同一个cache行的效果一样，造成了大量不必要的转移。



在DRAM中分配连续的内存给短寿命tensor。

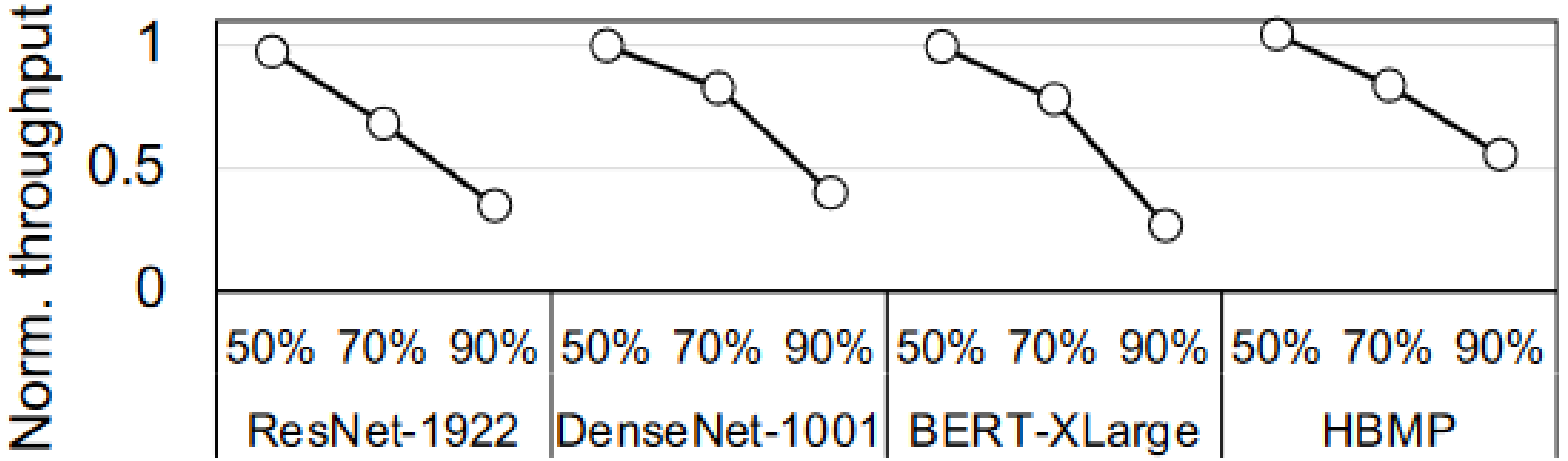
这部分tensor永远不会被转移到NVM上。这个空间在整个训练过程中被短寿命tensor分配和释放重复使用

数据转移代表7：FlashNeuron (USENIX 2021)

至此为止，转移都是限于CPU的DRAM(或者NVM)

但是没有考虑过内存、CPU正在执行数据预处理操作，从而使得内存总线始终在忙碌，从而使得转移性能极差。

FlashNeuron 采用GPU-direct技术，直接把数据转移到SSD上。



随着CPU运行的任务与GPU-CPU数据转移过程争夺带宽变激烈，转移性能下降。

数据转移代表6：FlashNeuron (USENIX 2021)

主要技术

- Memory Manager: 规划哪些张量应该从GPU转移到SSD上（综合了压缩；同时解决了内存碎片化问题）

- P2P-DSA: 一个用于实现GPU到SSD通信的层，调用GPUDirect实现。具体过程类似DMA，需要提前填写目的地址和各种命令，再进行数据传输。

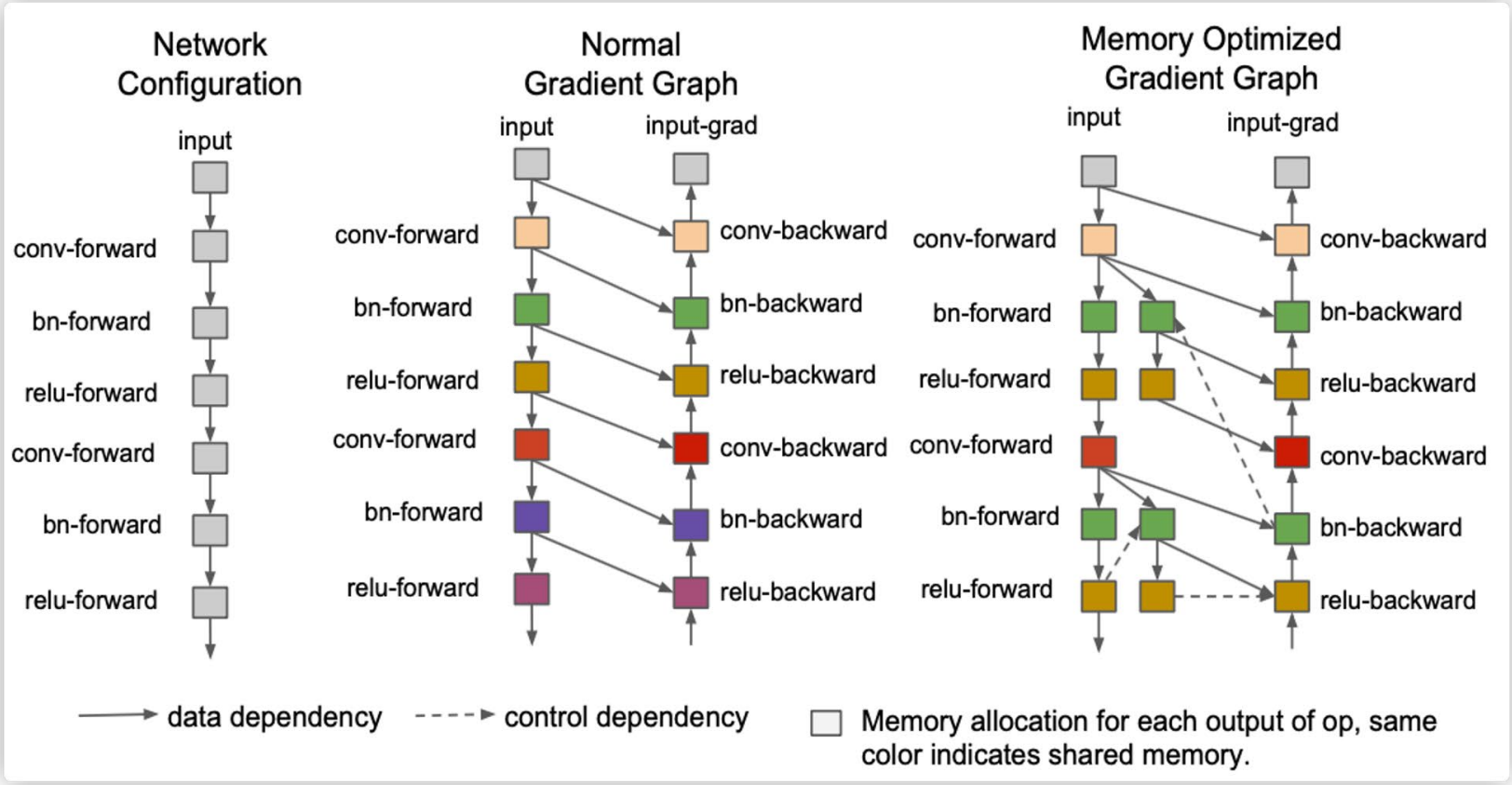
4 重计算

核心思想：

- 把一些方便计算的结果从内存丢弃，之后再计算

Checkpointing

- 设置一些梯度检查点，检查点之外的中间结果先释放掉
- 将来在反向传播的过程中，若发现forward pass的结果不在显存中，就找到最近的梯度检查点再进行前向计算，恢复出被释放的张量



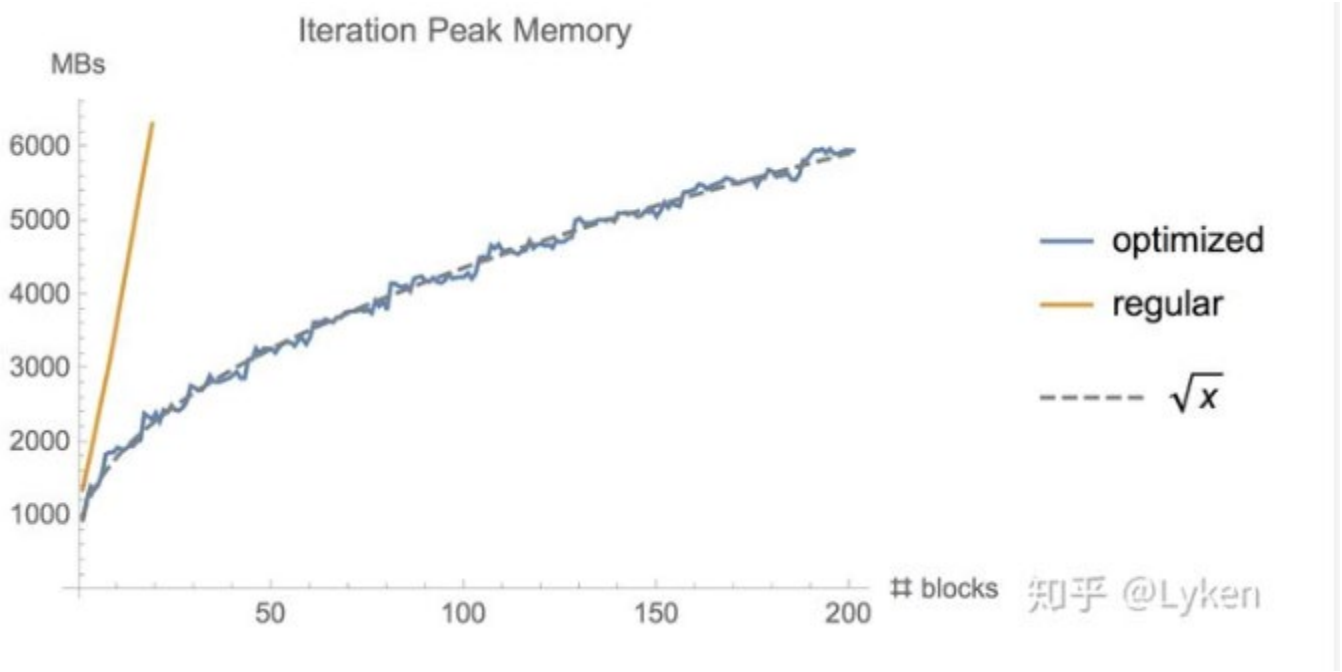
梯度检查点，可见是高度串行化的

重计算代表1：Sublinear Cost (2016)

第一篇提出DNN模型层重计算的文章。

用时间换空间：

对于一个n层的网络，每隔$\sqrt{n}$ 的层数保留一个feature map，其他的反向传播的时候再计算出来。

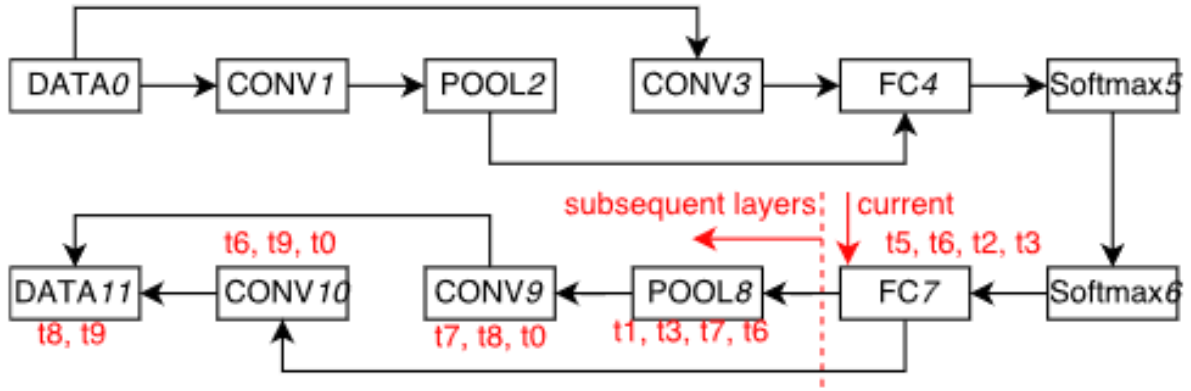


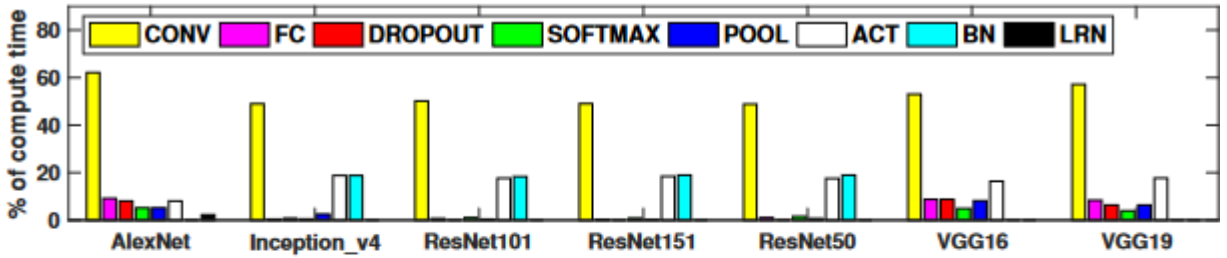
重计算代表2：SuperNeurons (PPoPP 2018)

首次将转移、重计算结合到一起的文章。

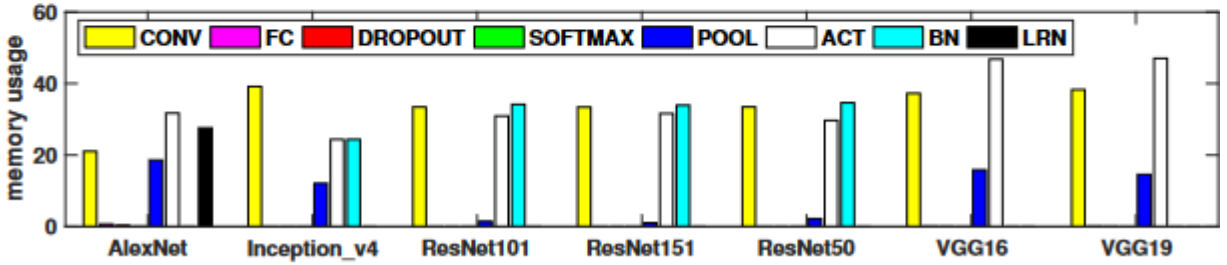
Heuristics：

1. 在反向传播中逐渐释放不再需要的Tensor
2. CONV的计算时间长，不适合重计算，因此仅转移CONV的输出
3. POOL, ACTIV层计算时间短，占用空间多，对这些层重计算





(a) breakdown of execution time by layer types

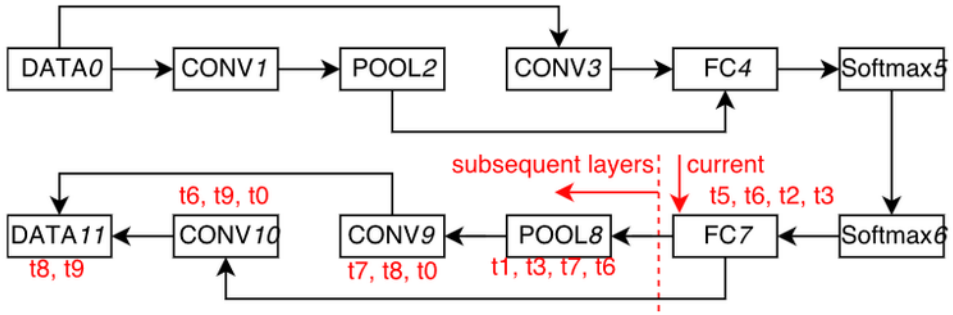


(b) breakdown of memory usages by layer types

重计算代表2：SuperNeurons (PPoPP 2018)

在反向传播中逐渐释放不再需要的Tensor

1. 为每一层构造一个输入集合（in set）和输出集合（out set），用来跟踪在该层之前和之后活跃的张量。
2. 运行时通过检查后续层的依赖关系来填充每一层的输入集合和输出集合。如果后续层不需要输入集合中的张量，就会从输出集合中删除这些张量。每次检查的开销分别为N − 1, N − 2, ⋯, 2, 1，因此总的开销为N(N−1)/2，即O(N^2)。



step:0	step:1	step:2	step:3	step:4	step:5
in:{}	in:{t0}	in:{t0, t1}	in:{t0, t1, t3}	in:{t0, t1, t3, t2}	in:{t0, t1, t3, t2, t4}
out:{t0}	out:{t0, t1}	out:{t0, t1, t3}	out:{t0, t1, t3, t2}	out:{t0, t1, t3, t2, t4}	out:{t0, t1, t3, t2, prob}
step:11	**step:10**	**step:9**	**step:8**	**step:7**	**step:6**
in:{t8, t9}	in:{t0, t6, t8}	in:{t0, t7, t6}	in:{t0, t1, t3, t6}	in:{t0, t1, t3, t2, t5}	in:{t0, t1, t3, t2, prob}
out:{}	out:{t8, t9}	out:{t0, t6, t8}	out:{t0, t7, t6}	out:{t0, t1, t3, t6}	out:{t0, t1, t3, t2, t5}

在step7之后，不在出现t2和t5，因此它们直接被从FC7的out中移走

重计算代表2：SuperNeurons (PPoPP 2018)

在转移的基础上，增加Tensor caching

直接offload和prefetch因为CPU到GPU的传输限制，会带来很大的性能损失。因此提出tensor cache：只有当GPU DRAM不足的时候，才会触发offloading和prefetching。

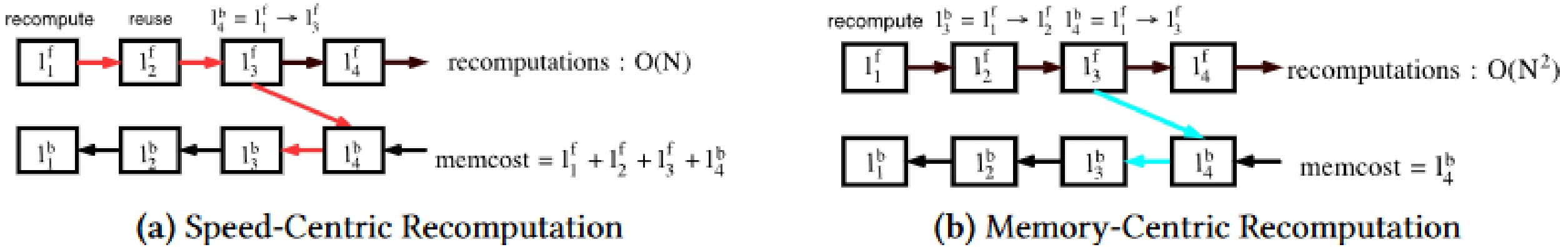
这里的cache使用的是LRU策略，很适合反向传播。

重计算代表2：SuperNeurons (PPoPP 2018)

两种重计算策略

- 对于POOL, ACTIV等层， 占据50% 的内存，计算时间却短。可以直接丢弃，在反向传播的时候重新计算正向依赖。

a. 以速度为核心：保留重新计算的张量，以便其他反向层直接重用。
b. 以内存为核心：总是为每个反向传播层计算依赖



(a) Speed-Centric Recomputation　　(b) Memory-Centric Recomputation

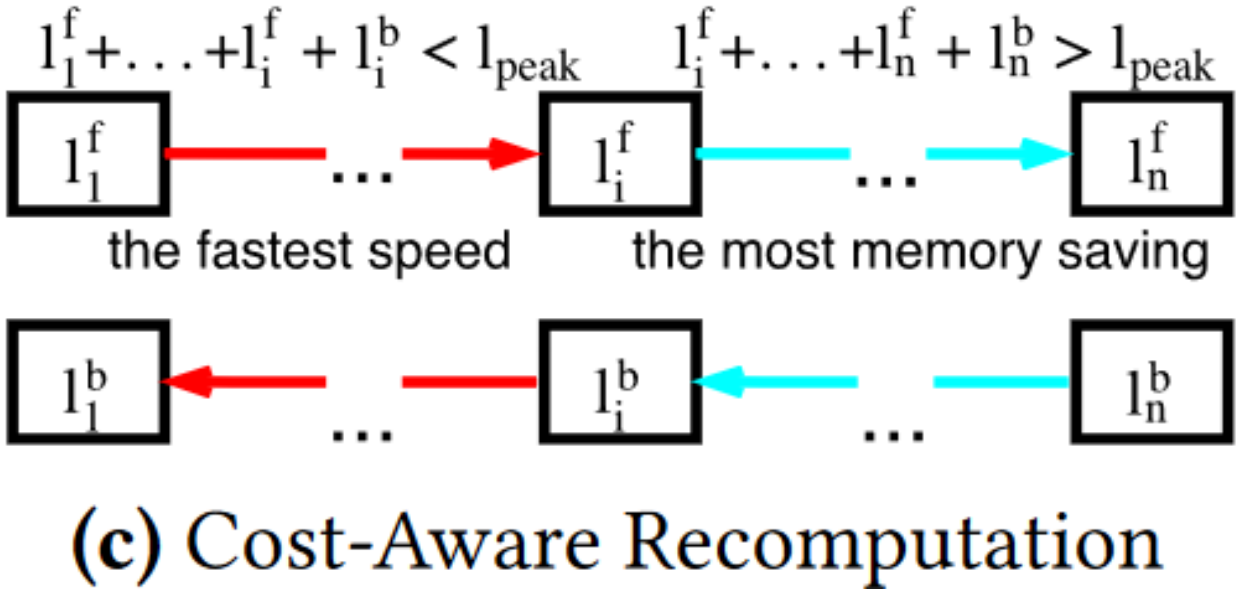
两种计算策略下的时空复杂度

重计算代表2：SuperNeurons (PPoPP 2018)

重计算策略折中

运行时找到最大内存消耗的layer，记这个消耗为L;

在重计算的时候，如果第一层到第i层的内存占用<L，则用以速度为中心的策略；反之，采用以内存为中心的策略。



(c) Cost-Aware Recomputation

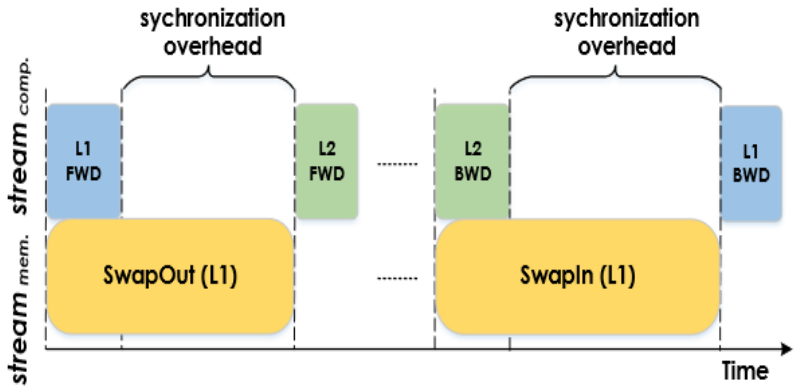
重计算代表3：Capuchin (ASPLOS 2020)

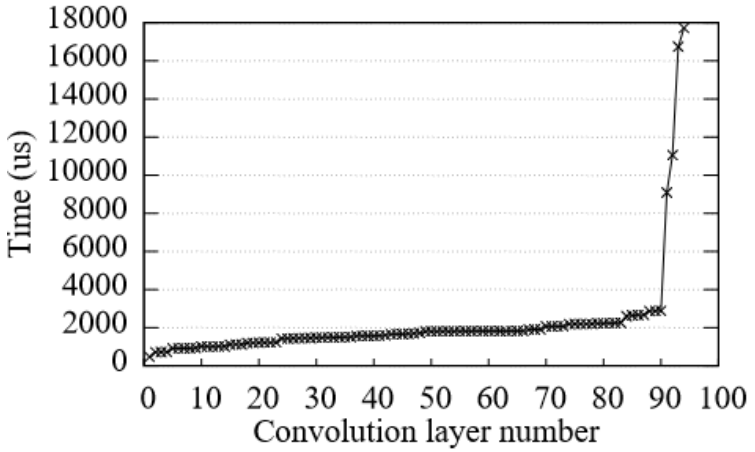
SuperNeurons的升级版，主要解决了以下三个前人遗留的问题：

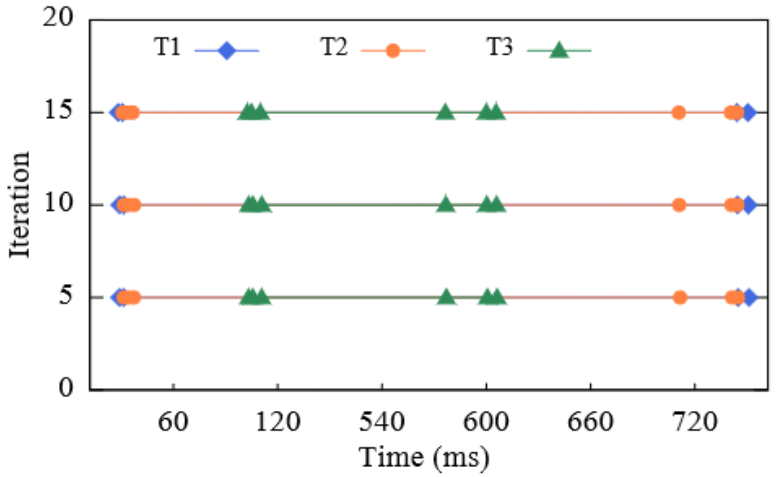
先前研究多遵循heuristics，对某些层的处理先入为主

不能对卷积层一概而论，卷积层执行时间差别大

训练的数据访问pattern一般很规则，但是没有对此的研究和处理







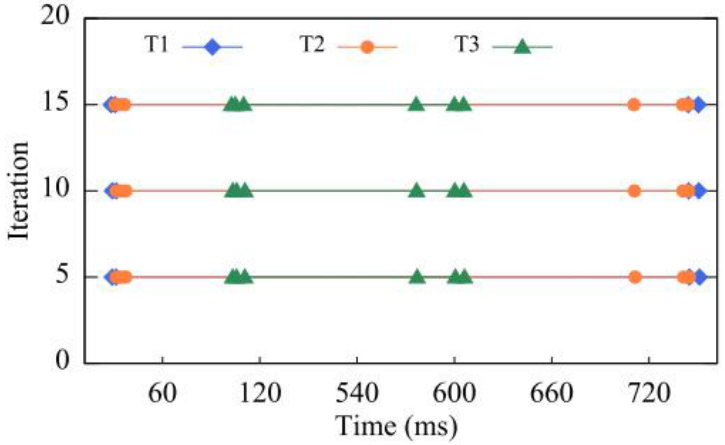
重计算代表3：Capuchin (ASPLOS 2020)

利用规则的访问pattern：每次训练的第一批batch用来获得动态的tensor访问序列，看有哪些优化可以做；后面的batch则使用这些优化。

在设计思路上：不能	不能简单的仅将卷积	前面的数据**生命周期长**，
为等待转移结束	前面的数据进行转移	更值得优先被处理

Capuchin-结合转移+重计算设计新的高效思路[1]

- ①由于转移可以隐藏在计算中，重计算不可避免的会引入额外开销，所以先选择转移的Tensor；即：先根据Tensor的寿命进行排序（可以理解为下图中线长的Tensor优先，左图）。
- ②对排序后的Tensor依次进行转移决策，选择转移开能够完全隐藏的Tensor；
- ③根据MSPS（右图）指标对重计算Tensor进行选择；**-- 保存的空间越大，重计算时间越小的Tensor更值得被重计算；**

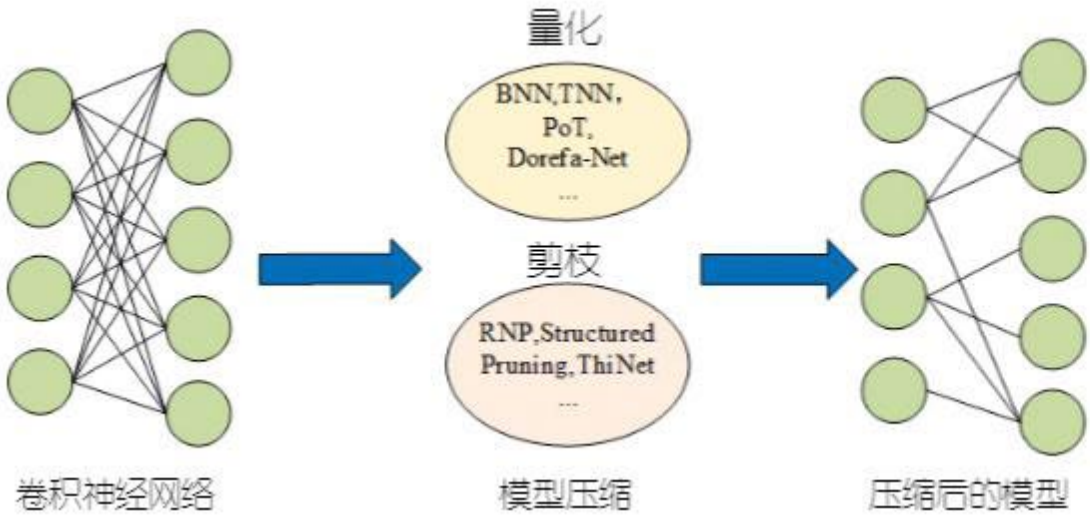


$$MSPS = \frac{Memory\ Saving}{Recomputation\ Time}$$

5 压缩

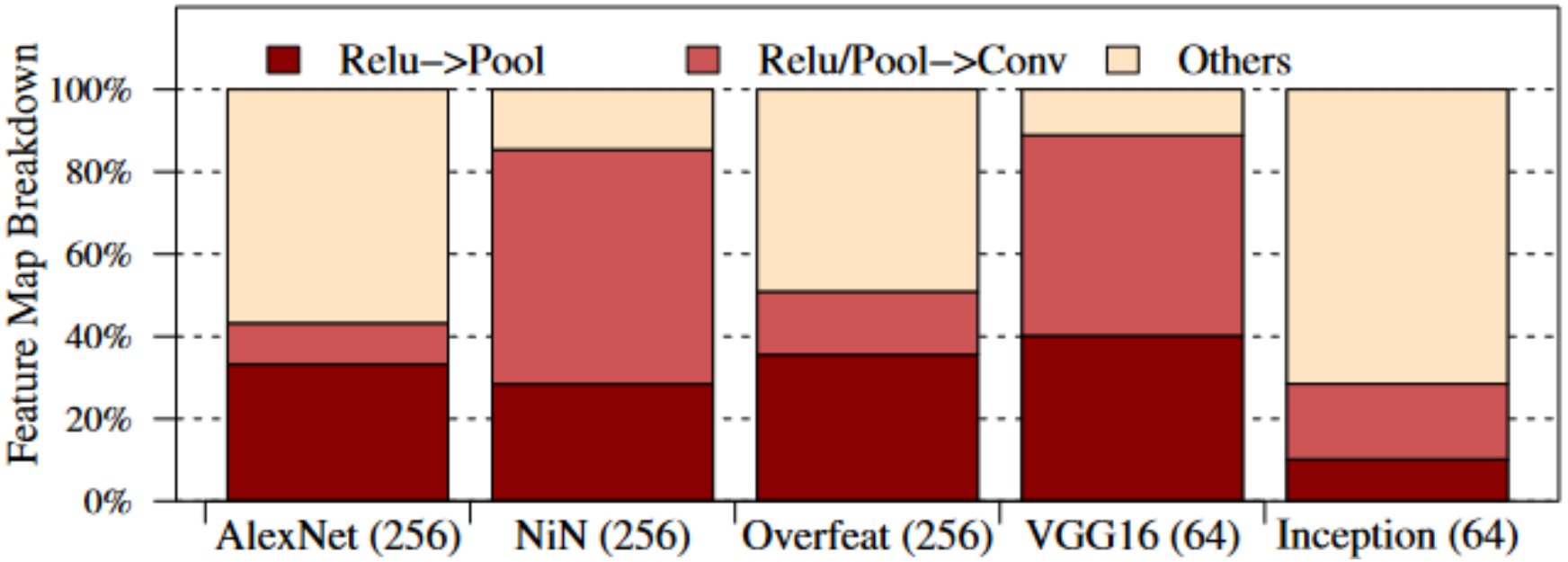
核心思想:

- 通过降低数据精度、或者将数据encode成体积更小的数据，降低内存消耗和交换的带宽占用
- 剪枝：剪去模型的部分参数值（还未阅读相关论文）



压缩代表1：Gist (ISCA 2018)

- 本文针对性较强，针对ReLU的输出进行无损的压缩，对于其他层的输出进行有损的压缩

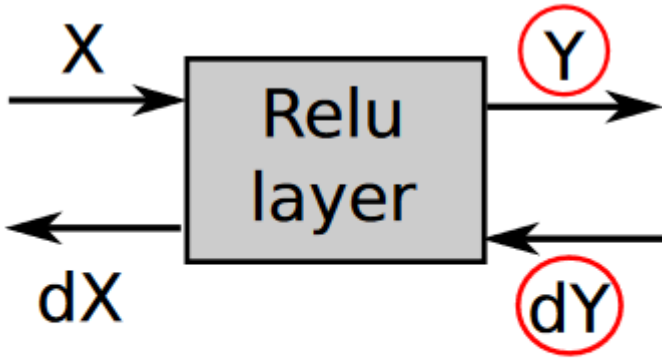


ReLU后接Pool或者ReLU/Pool后接CONV占比很高，值得单独优化

压缩代表1：Gist (ISCA 2018)

ReLU层相关的结构，矩阵稀疏，采用无损压缩

- 对于ReLU-Pool层，可以使用binarize的策略，即将输出的feature map的中的每一个数值用一个bit表示是否为0。并且辅助记录Pool层选择的位置，保证反向传播正确进行。这样总体下来，能够减少相当的内存使用。
- 对于ReLU-CONV层，将数据在生命周期的大部分以sparse方式存储，只在被计算前convert back。经过比对实验之后，选择了Compressed Sparse Row格式进行压缩



压缩代表1：Gist (ISCA 2018)

其余层：有损压缩，控制粒度

- 其他层的特征就不明显了，于是作者使用8bit、10bit以及16bit来代替32bit，降低原来FP32格式的数据的大小。
- 若将其余所有数据均压缩，则模型损失严重；但是如果只压缩梯度，那么训练准确度不会被影响。
- 最后，压缩需要滞后到反向传播才真正参与计算；即，在正向传播时，前递的数据保持精度。这样能最小化压缩的误差影响。：

压缩代表2：cDMA (HPCA 2018)

仍然利用ReLU的稀疏特性，直接面向硬件，在GPU中设计了一个部件压缩数据

采用的压缩算法是Zero-value compression。对于32个数字，设置一个32bit的mask，mask的每一个bit就表示那个数字是不是0. 如果最后得到一个全0的Mask，就直接用这个mask来表示；如果不是，就append非零值即可。

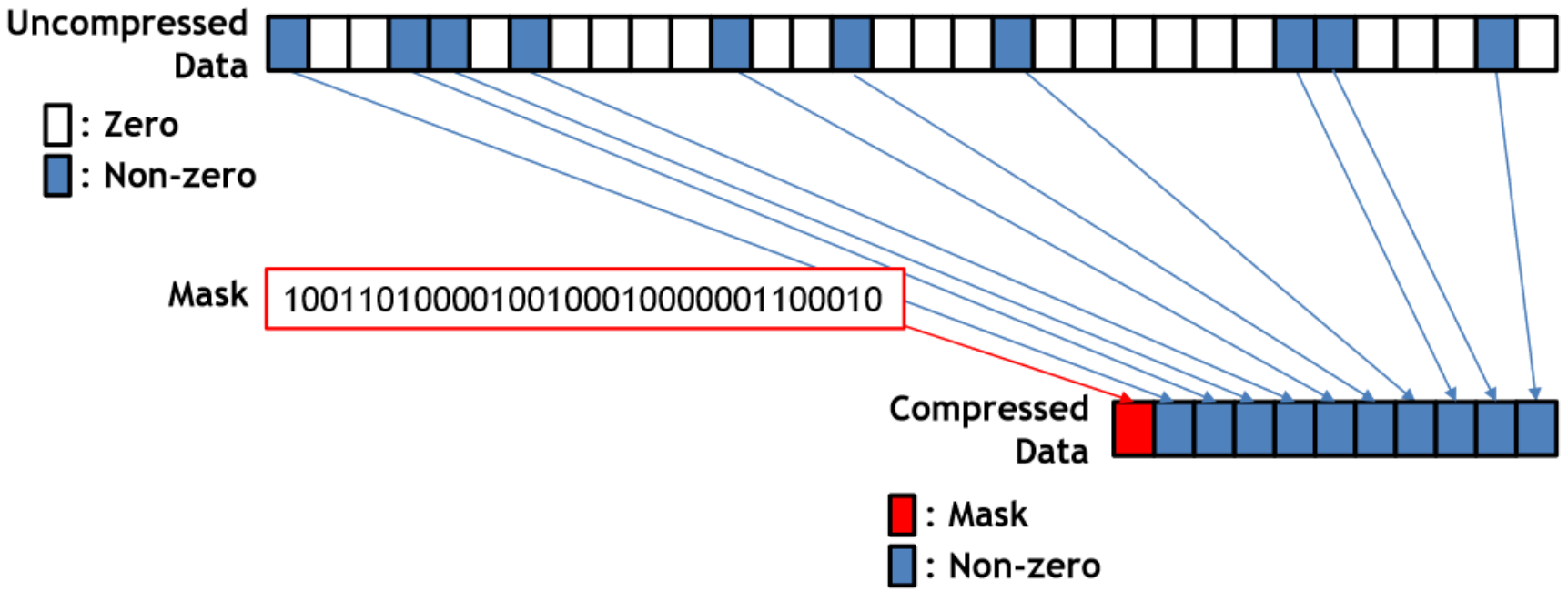
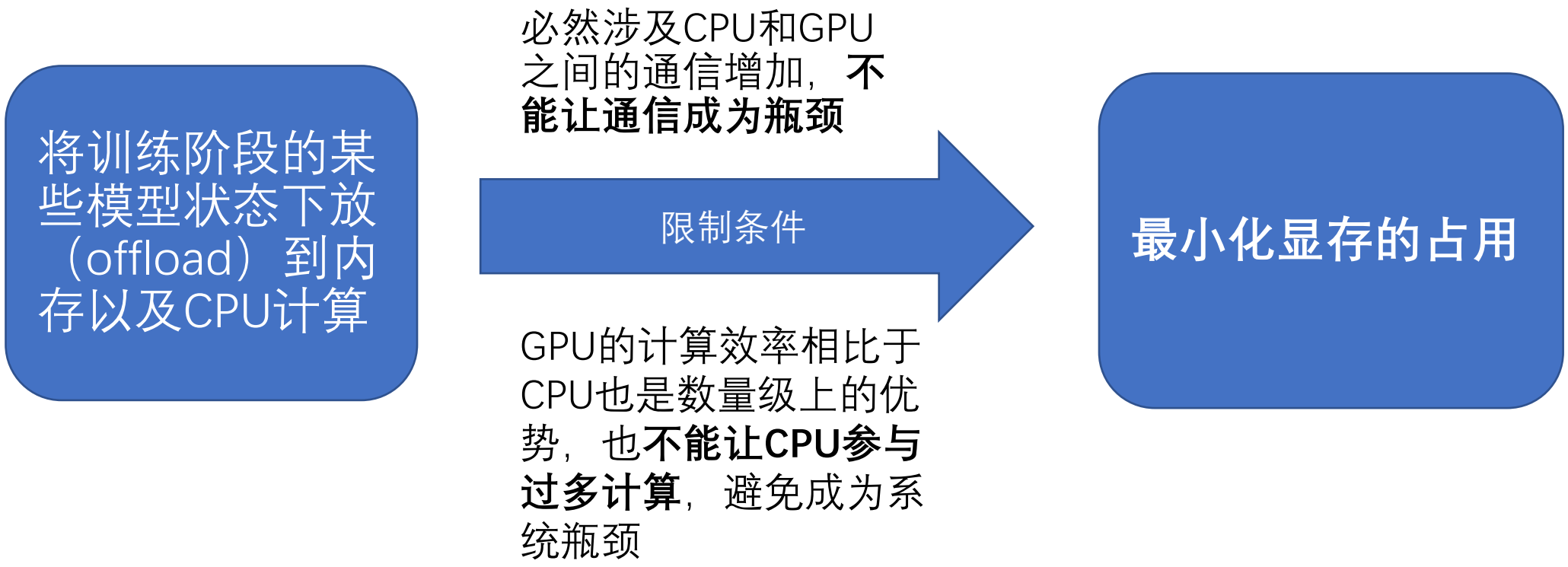


Fig. 8: Zero-value compression.

6 CPU辅助计算：ZeRO-Offload （ATC 2021）

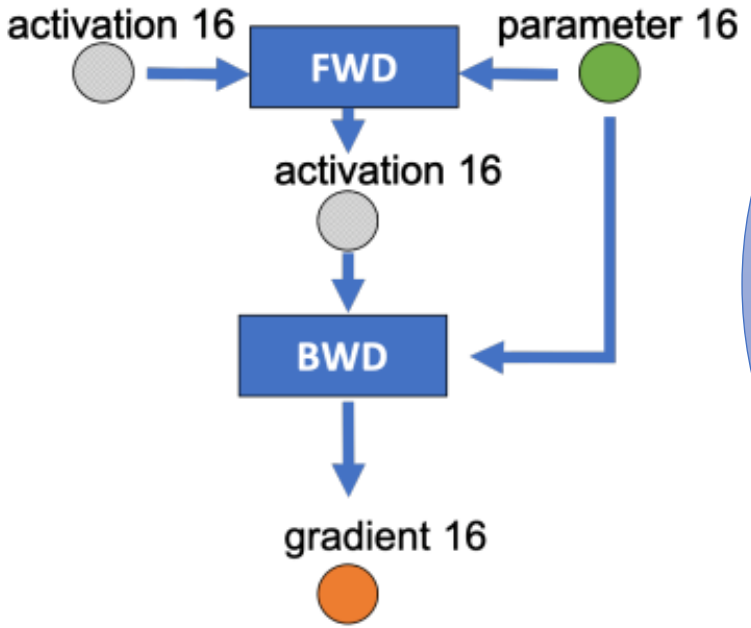
研究思路：前面的研究都只利用了CPU的存储，并没有利用CPU的计算能力。



6 CPU辅助计算：ZeRO-Offload （ATC 2021）

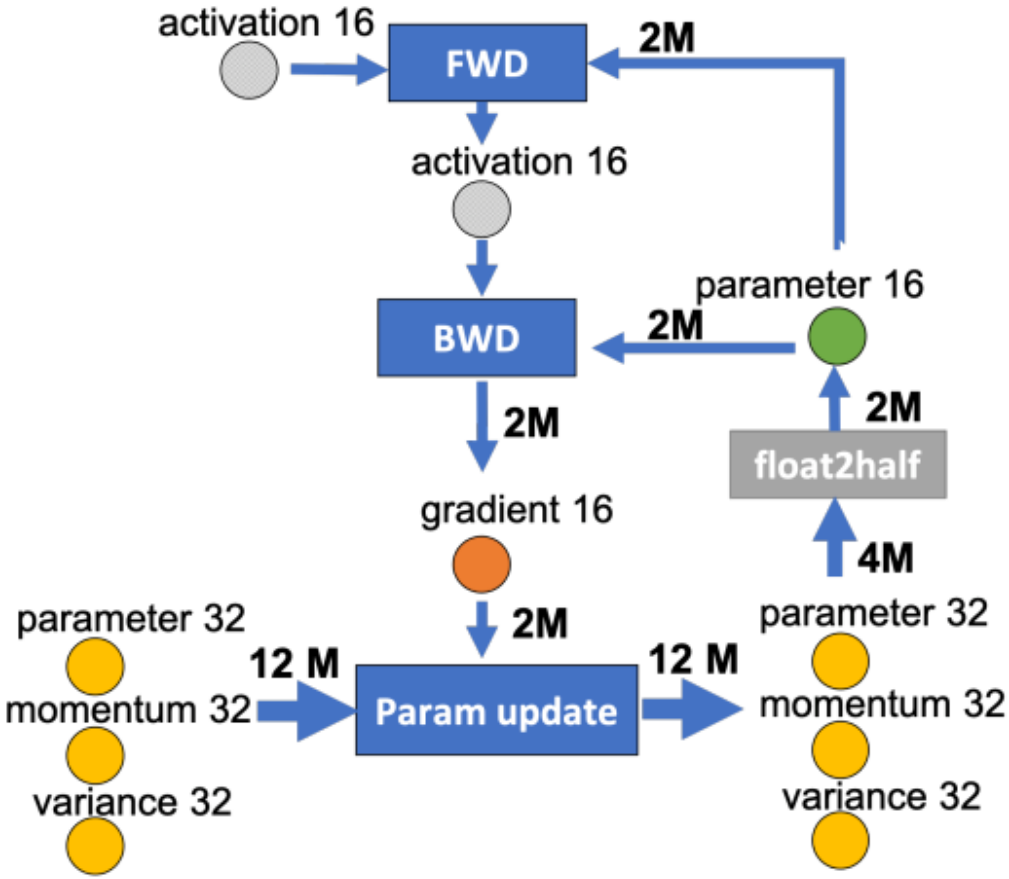
将模型训练过程看作数据流图

- 圆形节点表示模型状态，比如本层的参数、梯度和优化器状态
- 矩形节点表示计算操作，比如前向计算、后向计算和参数更新
- 边表示数据流向



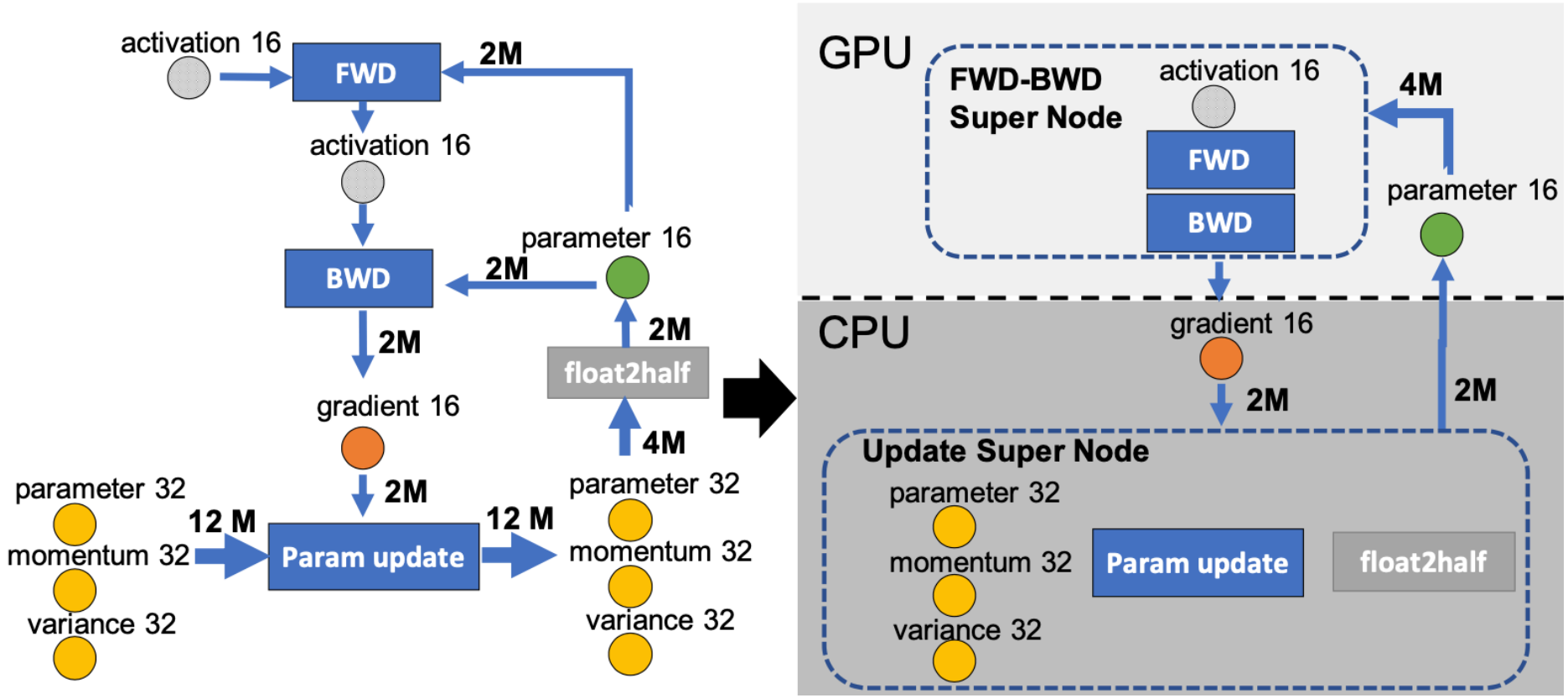
某一层的一次迭代

任务：沿着边把数据流图切分为两部分，分布对应GPU和CPU，计算节点（矩形节点）落在哪个设备，哪个设备就执行计算



用Adam优化器进行参数更新
模型参数量是M
边的数字是传递的权重的字节数

6 CPU辅助计算：ZeRO-Offload （ATC 2021）

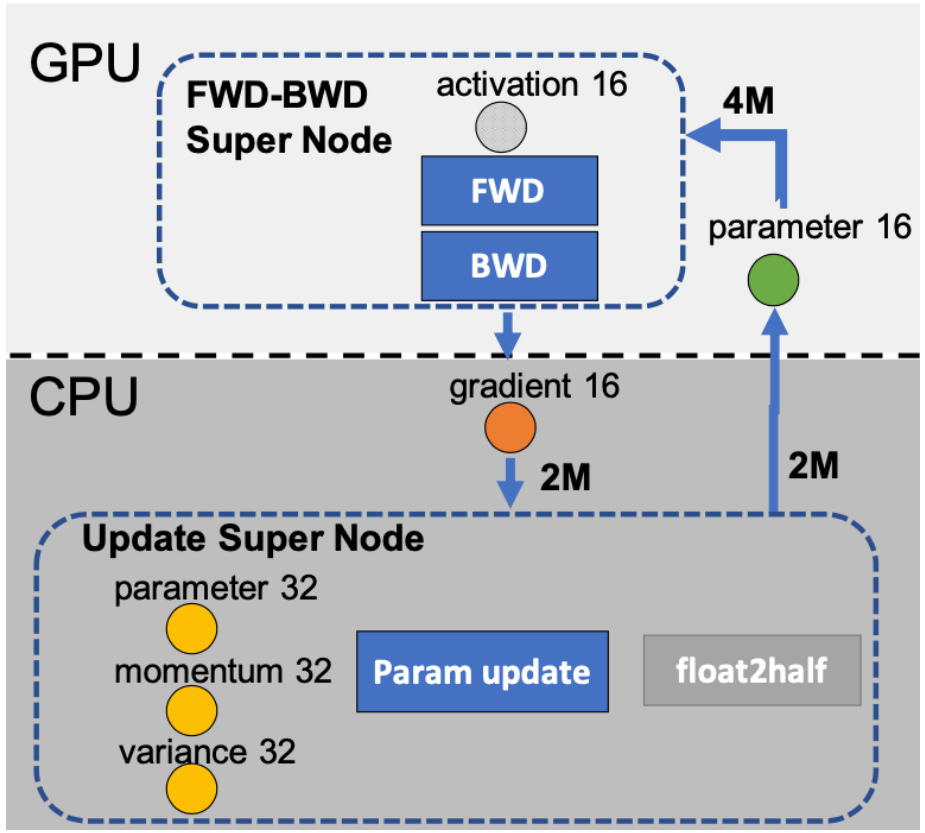


要求1：CPU不能计算大量数据

- 图中有四个计算类节点：FWD、BWD、Param update和float2half，其中FWD与BWD的计算次数随Batch Size增长O(MB)，放在GPU;
- Param update和float2half 的计算复杂度是O(M)，放在CPU上
- Adam状态自然也放在内存中

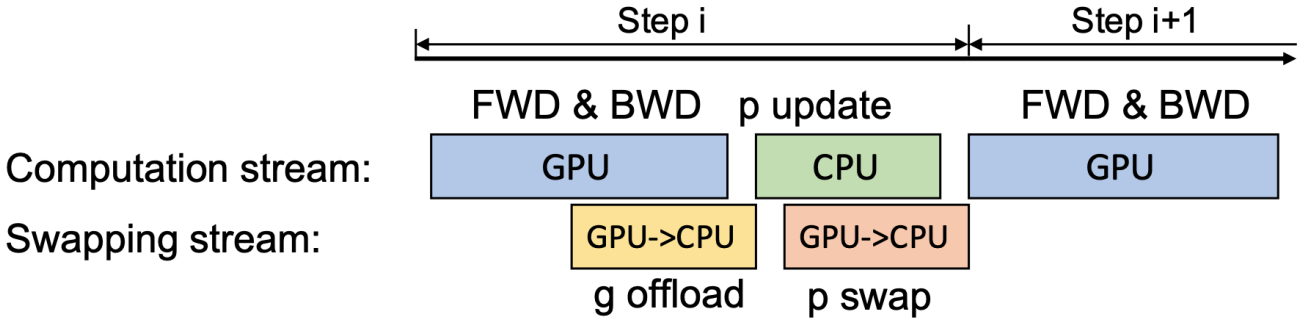
6 CPU辅助计算：ZeRO-Offload （ATC 2021）

要求2： 避免CPU和GPU之间的通信成为瓶颈



将计算和通信并行起来：

- GPU在反向传播阶段，可以一批梯度值计算完后，一边计算新的梯度，一边将计算好的梯度传输给CPU
- 当反向传播结束，CPU基本上已经有最新的梯度值了，同样，CPU在参数更新时也同步将已经计算好的参数传给GPU，如下图所示。



ZeRO-Offload training process on a single GPU.

感谢观看！