AI background

Machine learning内容：data, model, loss function, Optimization Algorithm

Problems: 泛化、过拟合、欠拟合 解决方案: 使用cross validation评价模型

ML算法分类：

- 监督学习：回归，分类，推荐系统
- 无监督学习：聚类，降维
- 强化学习

训练过程：降低loss function

- 线性回归

$$\hat{\boldsymbol{\theta}} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

  线性回归由于计算逆矩阵过于复杂，因此较少使用

- 梯度下降
  - Batch Gradient Descent:一次训练所有样本
  - Stochastic Gradient Descent:一次训练一个样本 (快，不稳定)
  - Gradient Descent with mini-batch:一次训练几个样本 (快，稳定，便于并行化)

选取更新率：Momentum，Adagrad，Adam

Deep learning 从LeNet到AlexNet的挑战

- 数据需求：ImageNet
- 计算需求：GPU
- 避免过拟合：Dropout regularization（随机将神经元输出设为0）
- 梯度消除、爆炸：Residual Net

System for AI

System for AI的目的：

- 训练 Training:
  - 加快训练
  - 训练大模型 scalability
- 推断 Inference:
  - 推断速度
  - 在不同设备上部署（例如边缘设备）

关注点

准确率 & 资源利用率

- 训练 Training: 吞吐量
- 推断 Inference: latency

User API层面

Why not just numpy?

- 只支持部分简单算子
- 需要编程者自己计算梯度
- 需要自己实现更新规则

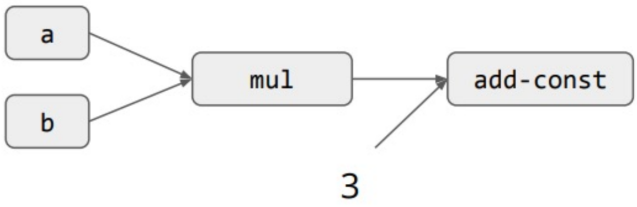
大家更喜欢声明式语言

声明式AI语言实现方法：计算图

Nodes represents the computation (operation)

– E.g., Matrix multiplications, softmax operator, activation functions

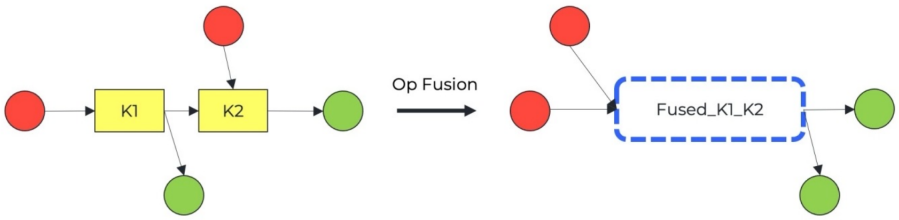
Edge represents the data dependency between operations



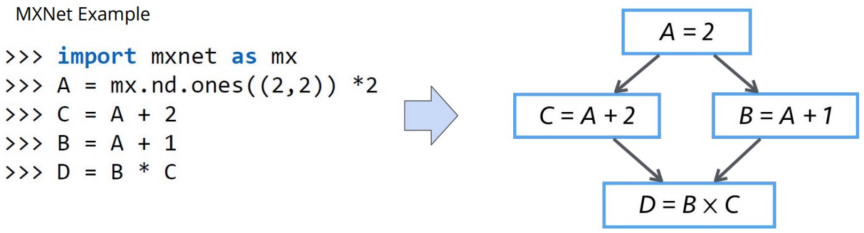
Example: computational Graph for a * b +3

计算图优点：

- 自动求导：利用求导的链式法则
- operation fusion: 避免算子之间的内存拷贝



- parallel scheduling：提高计算效率（例如，将B C过程放到两个计算单元中并行计算）



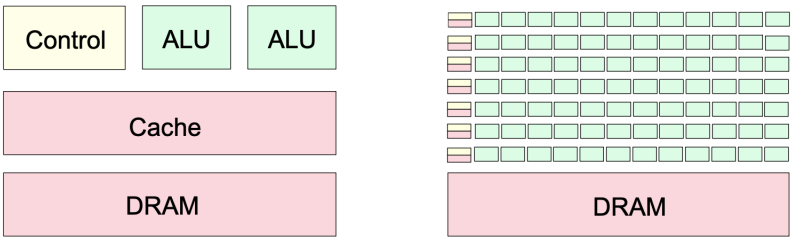
声明式语言更加兼容异构的计算硬件：使用为特定硬件优化的编译器，链接库 etc，无需为特定硬件重新编程

架构层面（硬件）

GPU

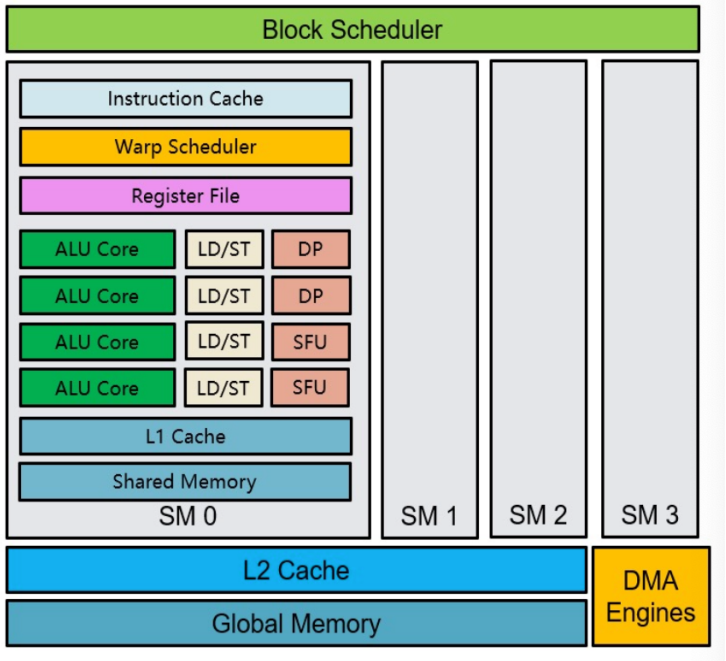
GPU硬件架构简介：

GPU较CPU有更多ALU单元



Simplified architecture of CPU　　Simplified architecture of GPU

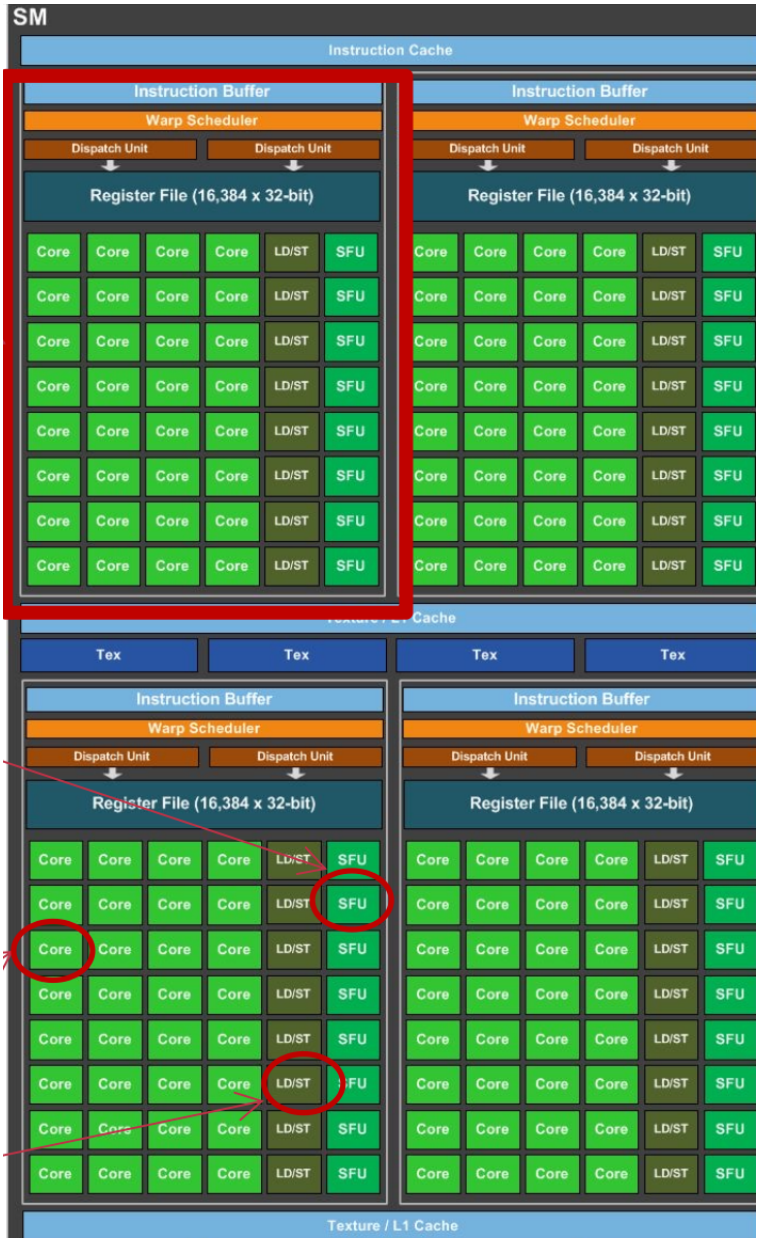
GPU支持SIMD（单指令多数据）并行执行能力更强 （CPU也有支持，但是支持并发程度仍低）



Block：一组GPU线程

Streaming Multiprocessors (SM): 一组ALU核，每个SM有一个L1缓存

SM中分为多个Warp，每个warp中的核执行同一个指令



CPU vs GPU summary：

- CPU
  - 对内存访问优化
  - 对乱序执行控制更优
- GPU
  - 算力高

GPU的内存层级结构仍然是复杂的，因此需要编程时注意，因此有CUDA模型：

- SIMT：单指令，多线程
- 抽象C code：编程者为线程编写C代码，每个线程执行相同代码（支持分支）
- 线程被组合成block
- kernel：一组block组成的grid

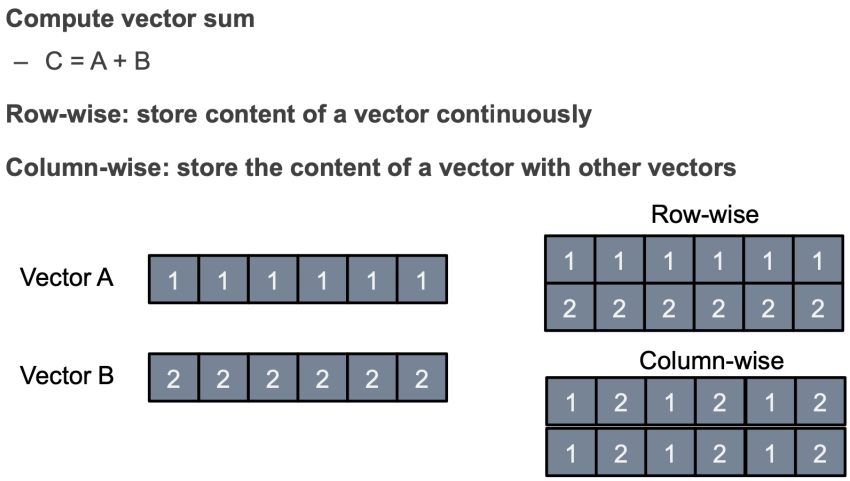
SM调度器会将block分配到SM上，由调度器进行block的换入换出，每个block则被拆分成多个warp，在硬件上执行

每个warp中的线程共享一个program counter，因此在CUDA编程时需要慎重考虑分支

为什么使用block概念(本质就是内存访问优化)：1.block层面共享内存效率高(类似L1 cache) 2.可以有轻量快速的同步barrier

需要考虑的问题：

- 内存分布（需要编程者仔细考虑）



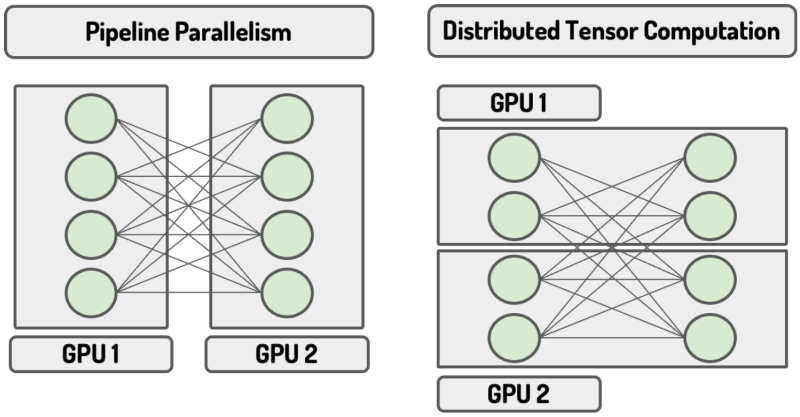
- dead lock（共享PC）

```
if (threadIdx.x == 0) {
    consume();
} else {
    produce();
}
```

GPU使用out-memory design，通过PCIe访问，在GPU上执行单次计算效率低

Scale

模型并行



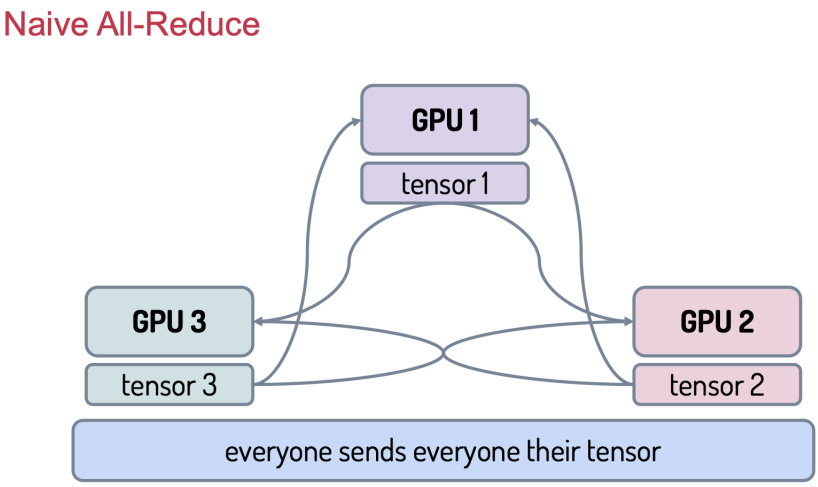
主要讲了pipeline式的模型并行

往往一个GPU中会有多层

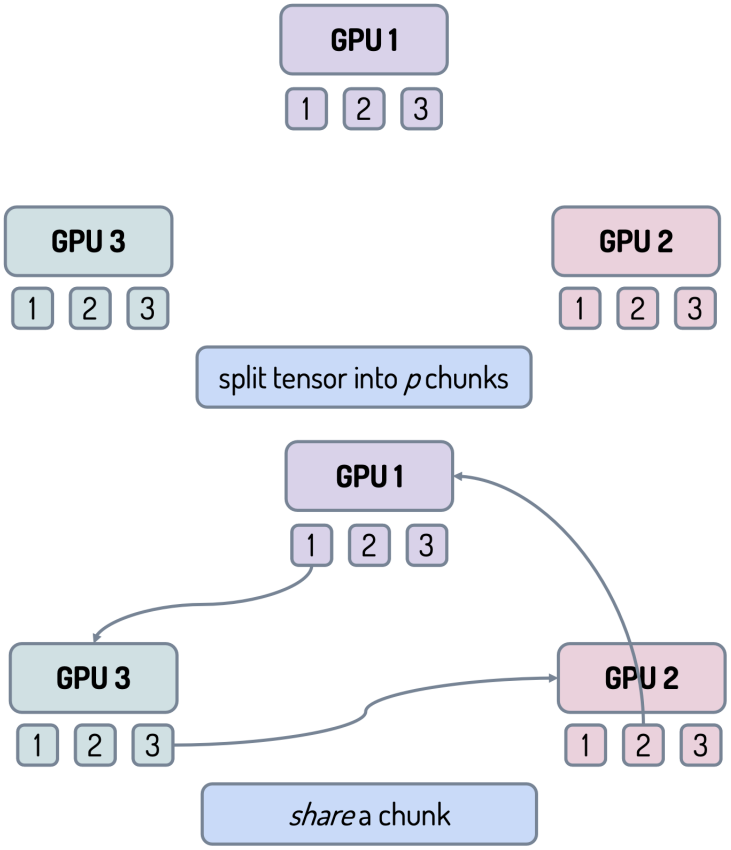
但是pipeline仍然无法解决需要存储activation的问题，activation memory usage睡着minibatch size 和 参数的数量增加。通信开销过大，无法换入换出。解决方案，扔掉GPU中间的层的activation，在方向传播时再重新传播出中间扔掉的activation

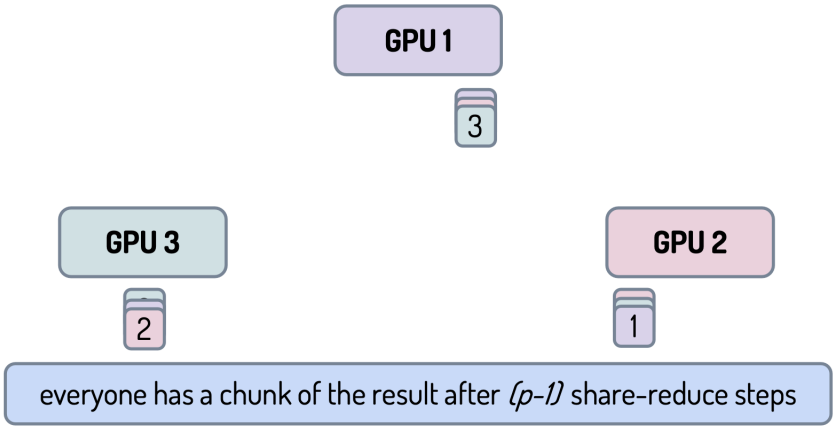
数据并行

需要进行一个all reduce过程来保证模型收敛到同一状态

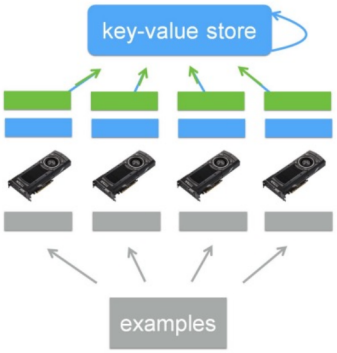


使用Ring ALL reduce方法，降低通信成本O(np^2) -> O(np) (p个GPU, tensor size是n)





或者使用Parameter server来做key-value store，在使用参数前pull，通过barrier保证同步



但是这样性能受最慢节点影响，因此尝试异步方法Async Training

异步会使精度降低，收敛变慢

此外Parameter server带宽可能不足