CHAPTER 0
课程介绍

主　讲：韩君

单　位：早稻田大学

公众号：自动驾驶之心

自动驾驶之心
全栈知识矩阵

自动驾驶之心，专注自动驾驶与AI

公众号

干货每日放送

B站

不定期直播+最新视频

知识星球

海量图文教程和Paper分享

视频号

技术视频一站式分享

个人简介

韩君

- 2022年在早稻田大学取得Ph.D.
  - Program logic and mathematical verification
  - Computer architecture and memory optimization
  - Program parallelization, automatic parallelizing compiler
  - High performance computing

- 2022~ now
  - 在日本某自动驾驶头部企业从事深度学习研发
    - Multitask training (2D/3D detection, depth, flow, tracking…)
    - Deep learning model optimization and deployment
    - Active learning, Pseudo labeling
    - Apollo, Autoware
  - 早稻田大学研究员，讲师
    - Transformer accelerator的研发

课程框架

Chapter 1: 并行处理与GPU体系架构

1.1 并行处理简介

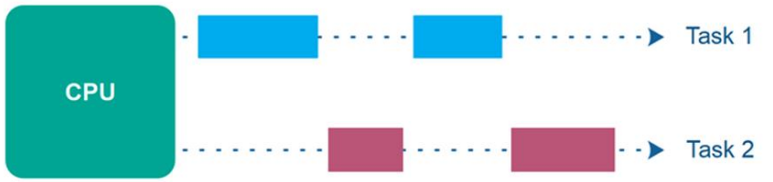
- 串行处理与并行处理的区别
- 并行处理的概念
- 常见的并行处理

1.2 GPU并行处理

- GPU与CPU并行处理的一同
- CPU的优化方式
- GPU的特点

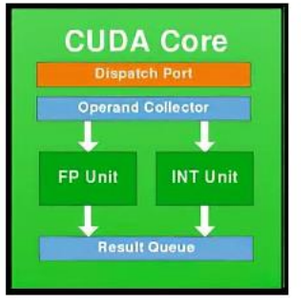
1.3 环境搭建

- CUDA, cuDNN, TensorRT的版本选择
- 通过docker与Dockerfile进行环境搭建
- vscode与neovim编辑器的环境搭建

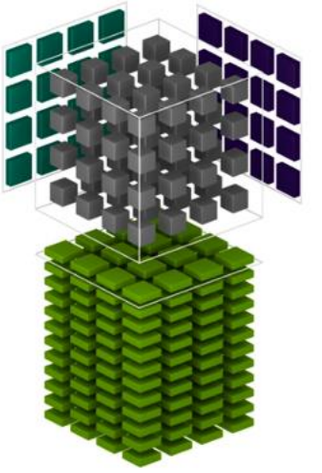


concurrency (并发)

Parallel (并行)



CUDA core[2]



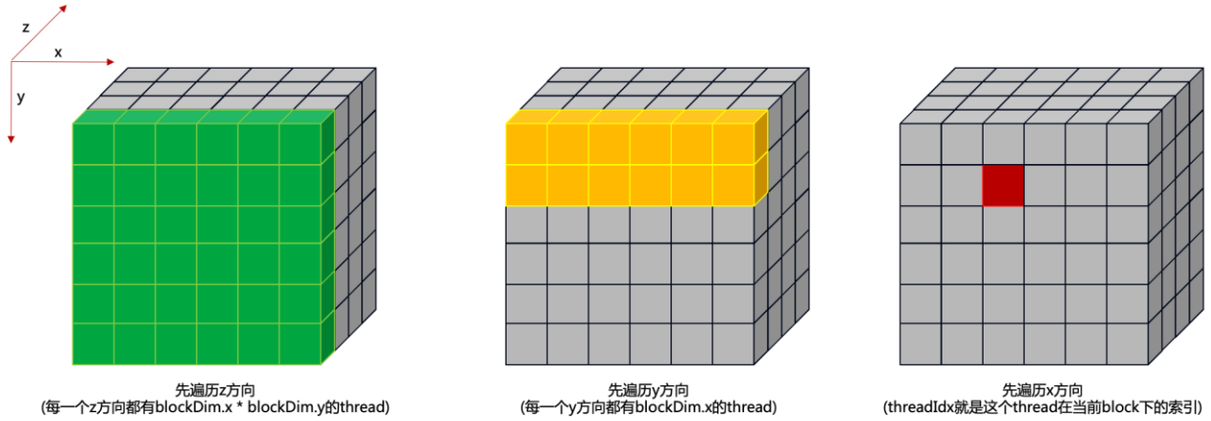
Tensor core[3]

课程框架

Chapter 2: CUDA编程入门

2.1 CUDA中的线程与线程束

- 执行第一个CUDA程序
- 理解CUDA中的Grid和Block的概念
- 理解.cu和.cpp的相互引用以及如何构建Makefile



先遍历z方向
(每一个z方向都有blockDim.x * blockDim.y的thread)

先遍历y方向
(每一个y方向都有blockDim.x的thread)

先遍历x方向
(threadIdx就是这个thread在当前block下的索引)

2.2 使用CUDA进行矩阵乘法的加速

- GPU对矩阵乘法的加速
- 使用Error handler进行快速的错误排查
- 获取GPU硬件信息

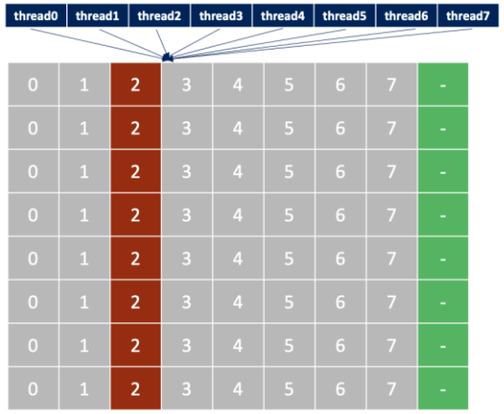
```
int main(){
    int count;
    int index = 0;
    cudaGetDeviceCount(&count);
    while (index < count) {
        cudaSetDevice(index);
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, index);
        LOG("%-40s",            "*********************Architecture related*********************");
        LOG("%-40s%d%s",        "Device id: ",                  index, "");
        LOG("%-40s%s%s",        "Device name: ",                prop.name, "");
        LOG("%-40s%.1f%s",      "Device compute capability: ",  prop.major + (float)prop.minor / 10, "");
        LOG("%-40s%.2f%s",      "GPU global meory size: ",      (float)prop.totalGlobalMem / (1<<30), "GB");
        LOG("%-40s%.2f%s",      "L2 cache size: ",              (float)prop.l2CacheSize / (1<<20), "MB");
        LOG("%-40s%.2f%s",      "Shared memory per block: ",    (float)prop.sharedMemPerBlock / (1<<10), "KB");
        LOG("%-40s%.2f%s",      "Shared memory per SM: ",       (float)prop.sharedMemPerMultiprocessor / (1<<10),
        LOG("%-40s%d%s",        "Device memory bandwidth: ",    prop.memoryBusWidth, "");
        LOG("%-40s%.2f%s",      "Device clock rate: ",          prop.clockRate*1E-6, "GHz");
        LOG("%-40s%.2f%s",      "Device memory clock rate: ",   prop.memoryClockRate*1E-6, "Ghz");
        LOG("%-40s%d%s",        "Number of SM: ",               prop.multiProcessorCount, "");
        LOG("%-40s%d%s",        "Warp size: ",                  prop.warpSize, "");

        LOG("%-40s",            "*********************Parameter related***********************");
        LOG("%-40s%d%s",        "Max block numbers: ",          prop.maxBlocksPerMultiProcessor, "");
        LOG("%-40s%d%s",        "Max threads per block: ",      prop.maxThreadsPerBlock, "");
        LOG("%-40s%d*%d*%d%s",  "Max block dimension: ",        prop.maxThreadsDim[0], prop.maxThreadsDim[1], pro
        LOG("%-40s%d*%d*%d%s",  "Max grid dimension: ",         prop.maxGridSize[0], prop.maxGridSize[1], prop.ma
        index ++;
        printf("\n");
    }
    return 0;
```

课程框架

Chapter 2: CUDA编程入门

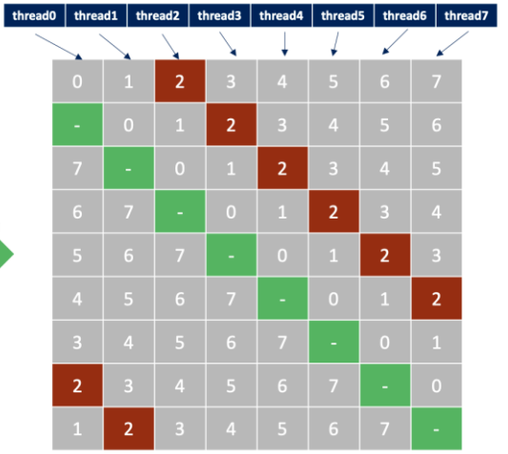
2.3 共享内存以及Bank Conflict

- 学习如何使用共享内存
- 理解何时发生bank conflict以及其缓解策略



发生bank conflict

更改布局



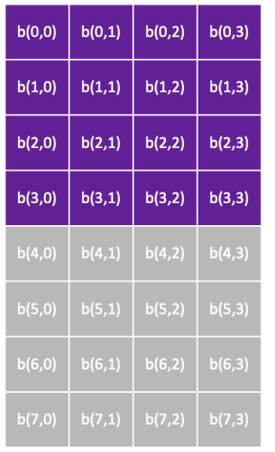
每8个bank进行一次stride，所以最终memory的布局发生改变

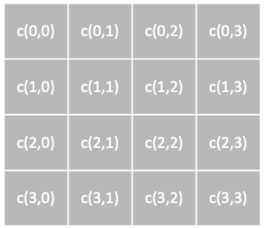
2.4 双线性插值的计算

- 双线性插值的算法介绍
- 如何使用CUDA进行加速以及
- 其他加速技巧

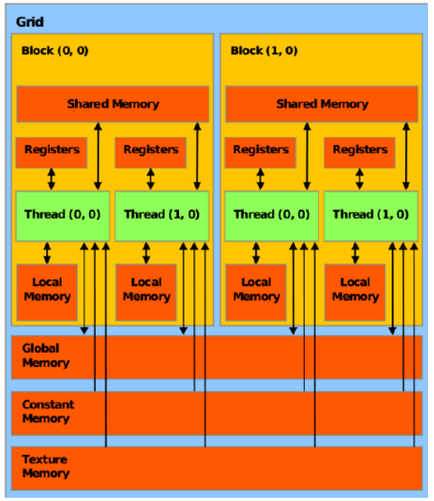
```
__global__ void MatmulSharedStaticKernel(float *M_device, float *N_device, flo
    __shared__ float M_deviceShared[BLOCKSIZE][BLOCKSIZE];
    __shared__ float N_deviceShared[BLOCKSIZE][BLOCKSIZE];
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了，这
    for (int m = 0; m < width / BLOCKSIZE; m ++) {
        M_deviceShared[ty][tx] = M_device[y * width + (m * BLOCKSIZE + tx)];
        N_deviceShared[ty][tx] = N_device[(m * BLOCKSIZE + ty)* width + x];
        __syncthreads();

        for (int k = 0; k < BLOCKSIZE; k ++) {
            P_element += M_deviceShared[ty][k] * N_deviceShared[k][tx];
        }
        __syncthreads();
    }

    P_device[y * width + x] = P_element;
}
```

A(4 x 8)





C(4 x 4)

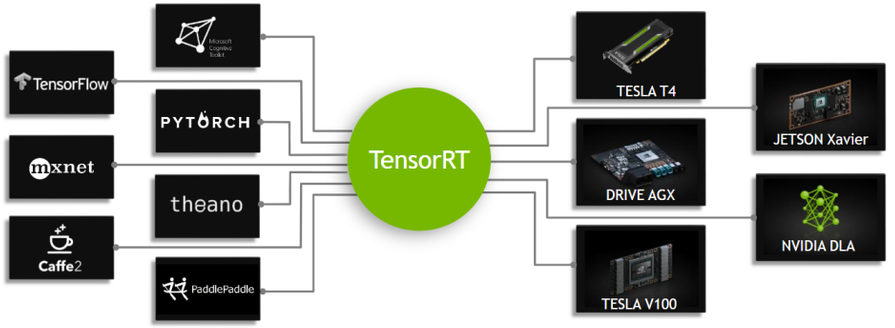


2.5 Stream与Event

- CUDA编程中多流的概念
- 使用Event的一些方式

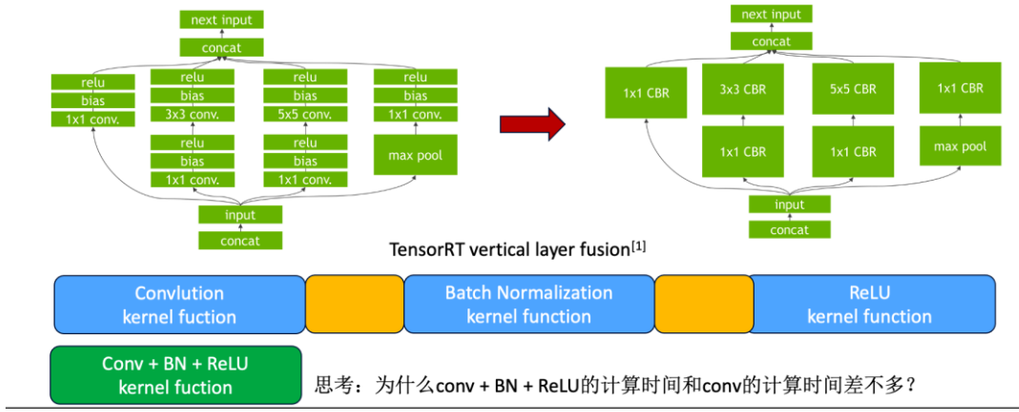
课程框架

Chapter 3: TensorRT基础入门



3.1 TensorRT简介

- TensorRT的工作流、优化极限介绍
- 其他DNN优化编译器介绍

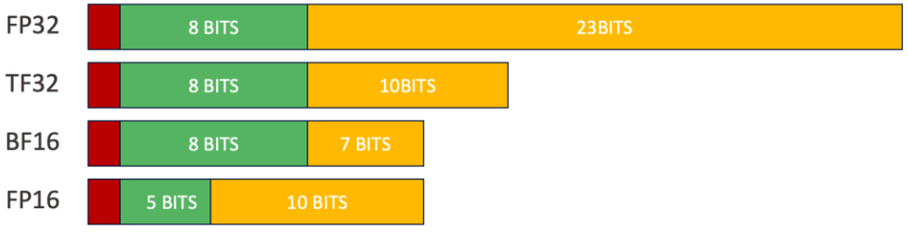


3.2 TensorRT应用场景

- TensorRT优化的意义
- TensorRT的部署常用方法

3.3 TensorRT内部的优化模块

- Layer fusion, Kernel autotuning, quantization介绍



3.4 分析如何生成TensorRT推理引擎

- trtexec介绍
- PyTorch -> ONNX -> TRT的模型转换
- 使用nsys以及trtexec的日志学习TensorRT的优化

课程框架

Chapter 4: TensorRT模型部署优化

4.1 模型部署的基础知识

- FLOPS与TOPS与FLOPs
- Roofline model与计算密度
- FP32/FP16/FP8/INT8/INT4的介绍

4.2 模型部署的几大误区

- FLOPS并不能衡量模型性能
- 不能完全依靠TensorRT
- CUDA Core与Tensor Core的区别
- 1x1 conv和 depthwise conv的部署缺点
- concat和shortcut对memory的不友好

Ampere SM Block diagram[1]

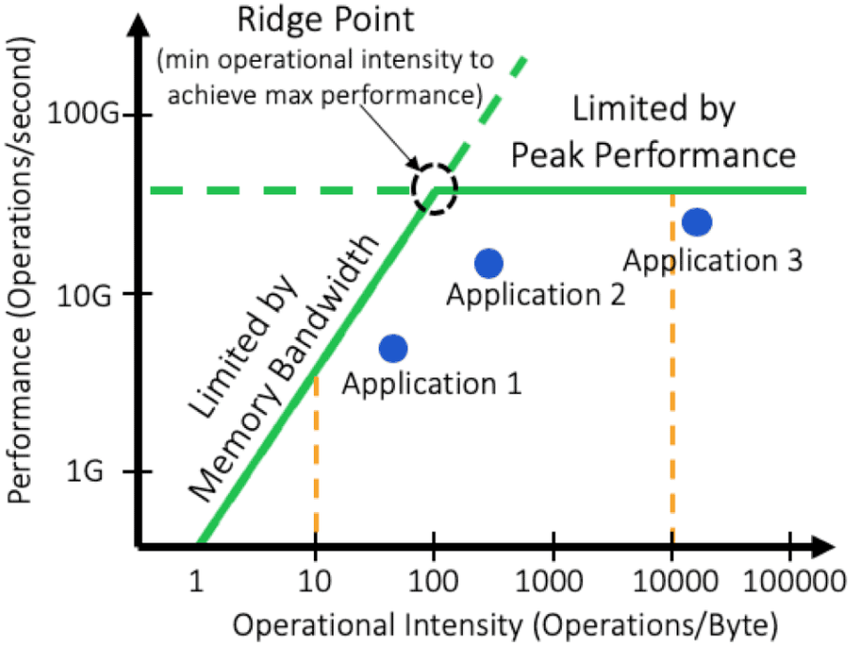
	FP64	FP32	TF32	FP16	INT8	INT4	INT1
CUDA Core	32	64	-	256	256	-	-
Tensor Core	64		512	1024	2048	4096	16484

我们来分析Tensor Core
Ampere架构使用的是第三代Tensor Core，可以一个clk完成一个2048 (= 256 * 2 * 4)个INT8运算。准确来说是4x8的矩阵与8x8的矩阵的FMA

- 频率：1.41 GHz
- SM数量：108
- 一个SM中计算FP16的Tensor core的数量: 4
- 一个Tensor core一个时钟周期可以处理的INT8: 512
- 乘加 : 2

Throughput = 1.41 GHz * 108 * 4 * 512 * 2 = 624 TOPS

[1]AI Chips: A100 GPU with Nvidia Ampere architecture

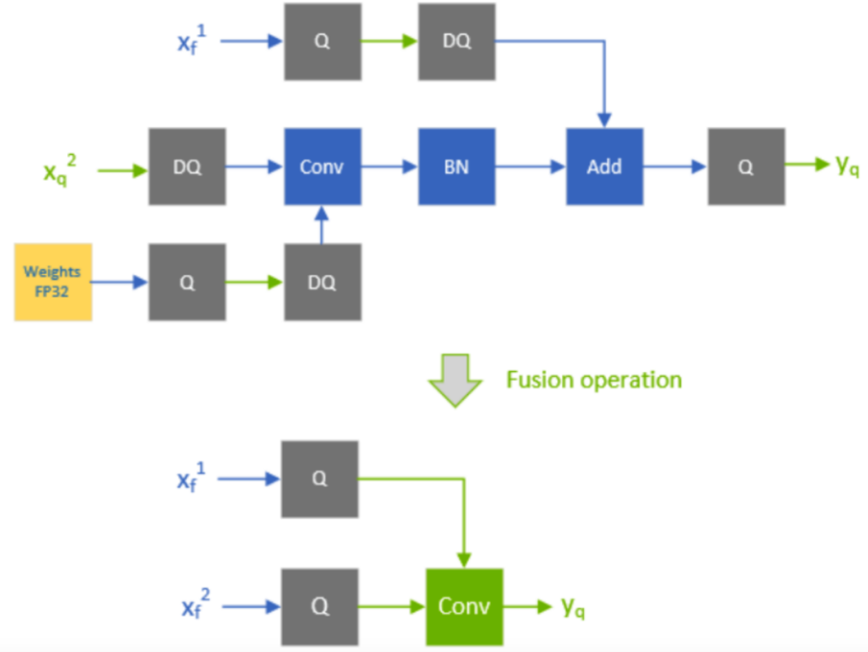


课程框架

Chapter 4: TensorRT模型部署优化

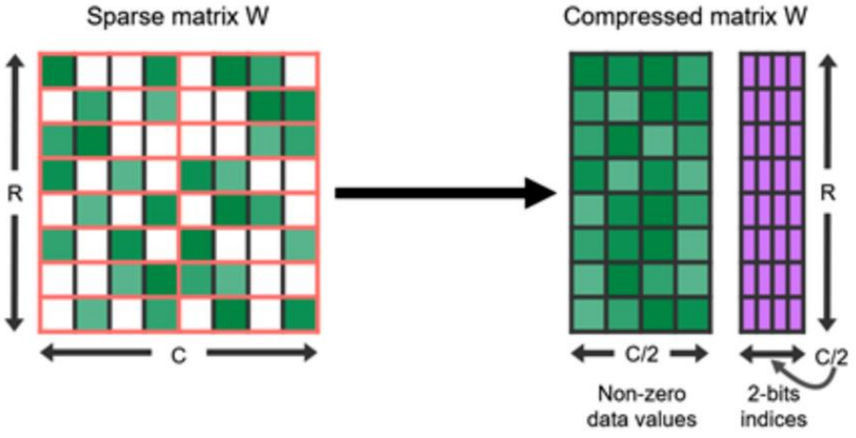
4.3 模型部署优化 – 量化

- 量化的基本概念以及优缺点
- 学习PTQ和QAT的区别
- 分析QDQ节点在TensorRT中的优化
- 学习Per-channel和Per-tensor量化
- 学习常见的量化技巧
- 浅谈TensorRT以外的量化方式



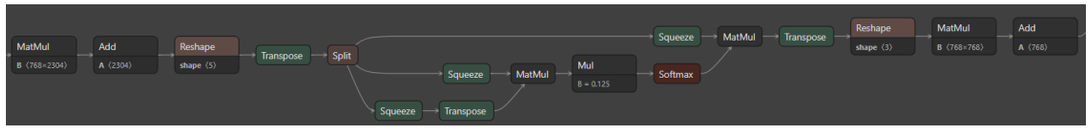
4.4 模型部署优化 – 剪枝

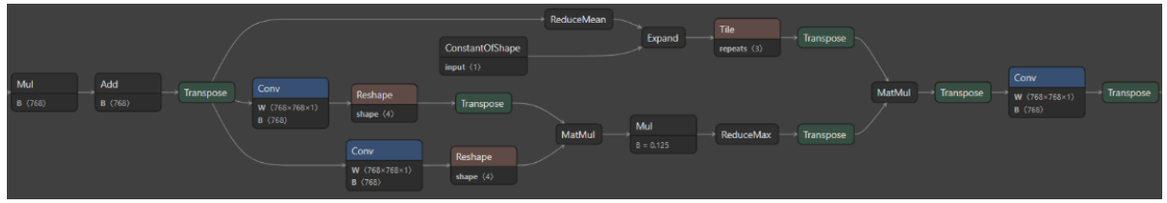
- 学习模型剪枝的基本概念
- 学习Channel pruning与Filter pruning
- 学习常见的剪枝技巧
- 分析剪枝过程所产生的额外的overhead，以及什么情况适用剪枝



课程框架

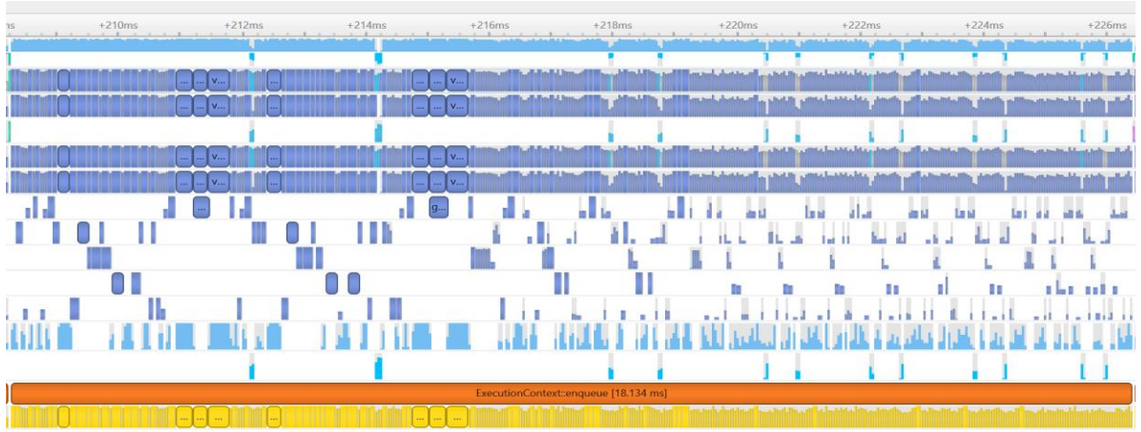
Chapter 5: 实战: TensorRT API的基本使用





5.1 使用TensorRT API生成推理引擎

- 学习TensorRT sample里的MNIST的推理过程
- 加载ONNX模型并比较PyTorch的结果
- 打印TRT优化后的网络结构与精度
- 以resnet/vgg为例，分析QAT/PTQ的TRT底层优化
- 使用nsight进行模型的详细性能分析
- 学习使用polygraphy分析模型



5.2 使用TensorRT对模型进行优化

- 分析不同量化策略的影响
- 查看nsight中kernel函数的技巧
- 构建self-attention模块并使用TRT加速
- 手动实现一个高效的softmax插件的几种方案

课程框架

Chapter 6: 实战: 部署分类器(CNN与Transformer的比较)

6.1 经典分类器的部署以及优化

- CNN (ResNet, VGG, EfficientNet, EfficientNetV2, ConvNeXt)
- Transformer (ViT, DeepViT, Swin Transformer,)
- CNN + Transformer (CvT, PiT, MobileViT, Cross-ViT)

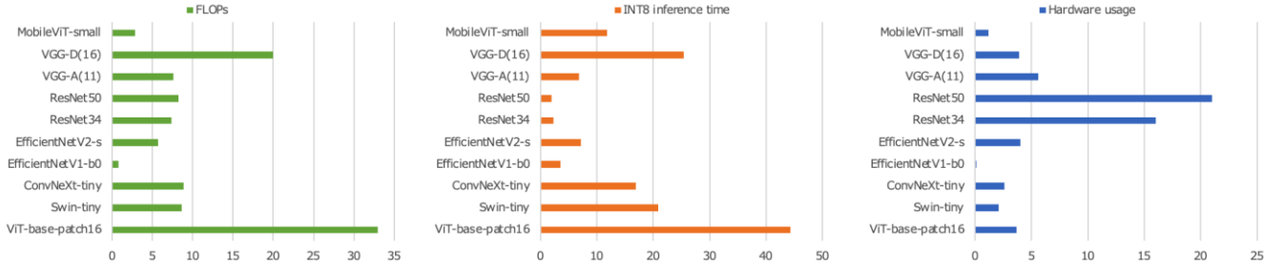
6.2 模型瓶颈分析

- 分析影响计算密度的因素
- 分析感受野和kernel size的关系
- 深入理解attention并优化

6.3 Transformer部署优化

- Overlapping patch embedding
- layerNorm -> BatchNorm
- Faster QKV generation
- Optimzied Softmax

Structure	Model	FLOPs	Params	FP32	FP16	INT8	FLOPS	Hardware usage (20TOPS = 100%)	Activation	Norm	Conv
transformer	ViT-base-patch16	33 GFLOPs	86M	43.45ms	17.275ms	44.257ms	0.76/1.91/0.74	3%/9.5%/3.7%	GELU	LN	-
transformer	Swin-tiny	8.67 GFLOPs	27M	20.497ms	12.837ms	20.832ms	0.42/0.67/0.41	2.1%/3.4%/2.1%	GELU	LN	-
CNN	ConvNeXt-tiny	8.89 GFLOPs	27M	16.425ms	7.50ms	16.963ms	0.54/1.18/0.52	2.7%/5.9%2.6%	GELU	LN	DW_conv
CNN	EfficientNetV1-b0	0.78 GFLOPs	4M	5.702ms	4.764ms	3.589ms	0.14/0.16/0.21	0.7%/0.8%/0.11%	SiLU/Swish	BN	DW_conv
CNN	EfficientNetV2-s	5.72 GFLOPs	20.2M	13.254ms	8.310ms	7.2ms	0.43/0.68/0.79	2.2%/3.4%/4.0%	SiLU/Swish	BN	DW_conv
CNN	ResNet34	7.34 GFLOPs	21M	8.309ms	2.316ms	2.278ms	0.88/3.17/3.21	4.4%/16%/16%	ReLU	BN	3x3
CNN	ResNet50	8.2 GFLOPs	23M	7.511ms	2.123ms	1.953ms	1.10/3.86/4.20	5.5%/19%/21%	ReLU	BN	1x1, 3x3
CNN	VGG-A(11)	7.62 GFLOPs	128M	13.433ms	5.160ms	6.861ms	0.567/1.48/1.11	2.8%/7.4%/5.6%	ReLU	BN	3x3
CNN	VGG-D(16)	20 GFLOPs	134M	52.741ms	22.159ms	25.436ms	0.38/0.92/0.78	1.9%/4.6%/3.9%	ReLU	BN	3x3
Hybrid	MobileViT-small	2.85 GFLOPs	4M	13.769ms	8.375ms	11.77ms	0.21/0.34/0.24	1.1%/1.7%/1.2%	SiLU/Swish	BN, LN	DW_conv



课程框架

Chapter 7: 实战: 部署YOLOv8

7.1 YOLOv8的ONNX模型分析

- YOLOv8的模型结构以及ONNX分析，导出方式

7.2 YOLOv8的部署-检测

- 创建builder并序列化引擎
- 实现目标检测与分割的基本infer

7.3 前处理/后处理优化

- Crop & Resize优化 (CUDA加速)
- Sigmoid优化 (CPU转GPU)
- NMS计算加速 (CUDA加速)
- 使用Halide实现自动并行优化
- 前后处理部分转到DNN优化
- 使用Look up table进行优化

课程框架

Chapter 7: 实战: 部署YOLOv8

7.4 YOLOv8的量化

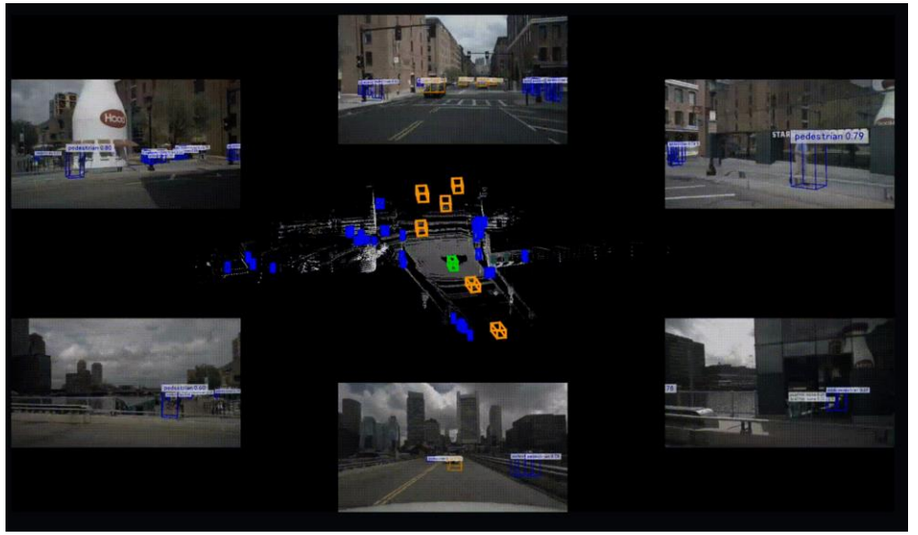
- PTQ量化与calibration
- QAT量化分析

7.5 浅谈multi-task模型的异步执行优化

- 分析Multi-task的部署
- 分析如何高效的实现multi-task模型的多线程部署

课程框架

Chapter 8: 实战: 部署BEVFusion模型



8.4 分析BEVFusion模型架构

- BEVFusion简介
- 分析Camera backbone与融合部分
- 学习针对LiDAR点云的3D Sparse Convolution Network的加速
- 融合部分Camera + LiDAR的DNN
- 分析BEVFusion的head部分

8.4 NVIDIA-AI-IOT的CUDA-BEVFusion的代码分析

- 环境搭建与部署测试
- 代码流程的注释介绍
- 学习RAII接口与模块化的封装技巧
- 学习模型各个模块的拼接方式
- 学习前/后处理的优化方式
- 分析目前在INT8下的bottleneck

```
//将上面所初始化的东西赋值给bevfusion的参数
//注意到目前为止并没有对lidar.backbone.xyz.onnx进行设置model，可能在后续的地方进行设置？
bevfusion::CoreParameter param;
param.camera_model = nv::format("model/%s/build/camera.backbone.plan", model.c_str());
param.normalize = normalization;
param.lidar_scn = scn;
param.geometry = geometry;
param.transfusion = nv::format("model/%s/build/fuser.plan", model.c_str());
param.transbbox = transbbox;
param.camera_vtransform = nv::format("model/%s/build/camera.vtransform.plan", model.c_str());

//根据参数进行初始化。对每一个网络进行创建engine。包括
//  camera_backbone:       命名空间camera
//  camera_bevpool:        命名空间camera
//  camera_vtransform:     命名空间camera
//  camera_depth:          命名空间camera
//  camera_geometry:       命名空间camera
//  transfusion:           命名空间fuser
//  transbbox:             命名空间head
//  lidar_scn:             命名空间lidar
//  normalizer:            命名空间camera
//  在device和host上分配lidar点云的内存空间

//这里体现了RAII的接口封装模式。接口类只提供最小限度的接口。实现类通过接口类调用实现封装上的隐蔽
return bevfusion::create_core(param);
```

课程框架

Chapter 8~: (计划中...)

- ch.9　开源项目学习之 - tensorRT_Pro
- ch.10 开源项目学习之 - tensorrtx
- ch.11 开源项目学习之 – lightNet-TRT
- ch.12 详解TensorRT plugin
- ch.13 针对Point cloud的CUDA加速
- ch.14 TVM介绍
- ch.15 Edge device: Jetson Orin上部署以及高效使用DLA

Thank you !!

CHAPTER1

并行处理与GPU体系架构

主　讲：韩君

单　位：早稻田大学

公众号：自动驾驶之心

主要内容

1. 并行处理简介
2. GPU并行处理
3. CUDA, CUDNN, TENSOR, NEOVIM环境搭建

并行处理简介

Goal: 理解并行处理的基本概念，理解SIMD, 以及编程中常见的并行处理方式

串行处理与并行处理的区别

Sequential processing(串行)
- 指令/代码块依次执行
- 前一条指令执行结束以后才能执行下一条语句
- 一般来说，当程序有数据依赖or分支等这些情况下需要串行

```
1  a = b + c;  //BB1
2  d = b + a;  //BB2
```

data dependency
(BB2中的a依赖BB1中的a)

```
1  if ( a == b) {  // COND3
2    c = d;        // BB4
3  } else {
4    c = e;        //BB5
5  }
```

branch
(COND3执行完了以后才能知道是执行BB4还是BB5)

(扩展)data dependency的种类
- Flow dependency
- Anti dependency
- Output dependency
- Control dependency

串行处理与并行处理的区别

Sequential processing(串行)

- 使用场景：
  - 复杂的逻辑计算(比如：操作系统)

statement 2 依赖 statement 1
statement 5 依赖 statement 4
statement 3 是一个 循环，跟所有的statement没有任何关系
有四个core可以使用

赋值语句 (a = b)
条件语句 (if – then – else)
循环语句 (for, while)
I/O (print)
…

时钟周期

- statement 1, statement 2: 5 cycles
- statement 4, statement 5: 8 cycles
- statement 3: 20 cycles

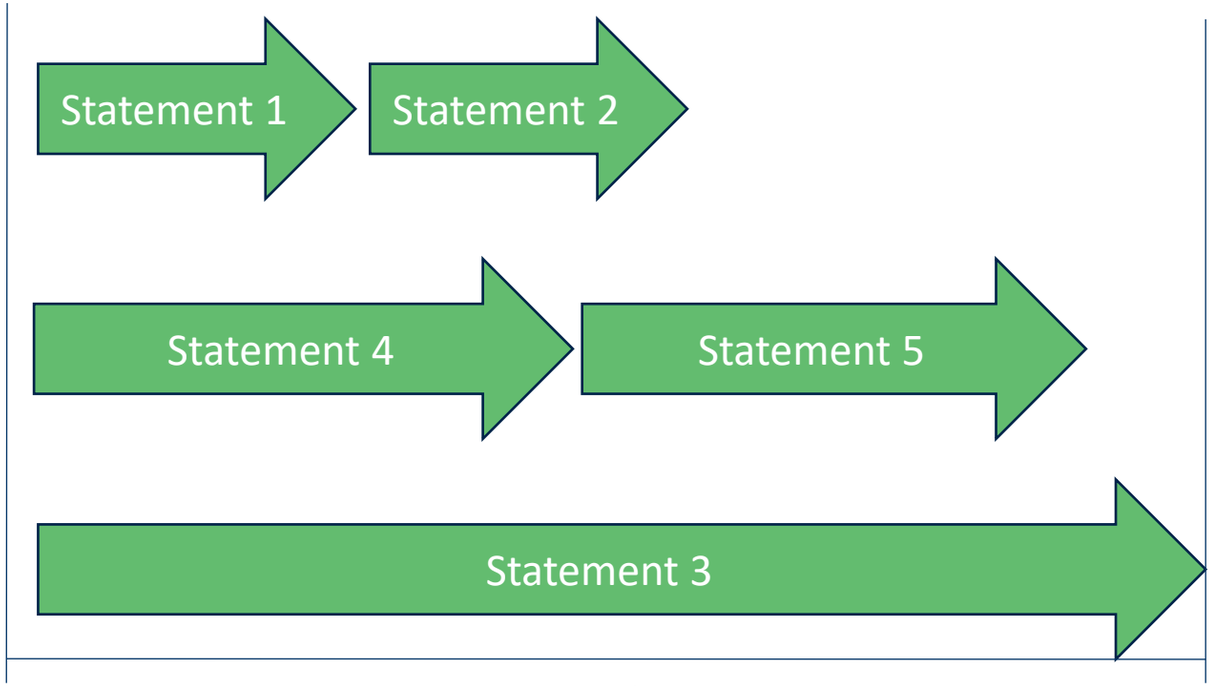
太慢了！

46 cycles

串行处理与并行处理的区别

- statement 2 依赖 statement 1
- statement 5 依赖 statement 4
- statement 3 是一个 循环，跟所有的statement没有任何关系
- 有四个core可以使用

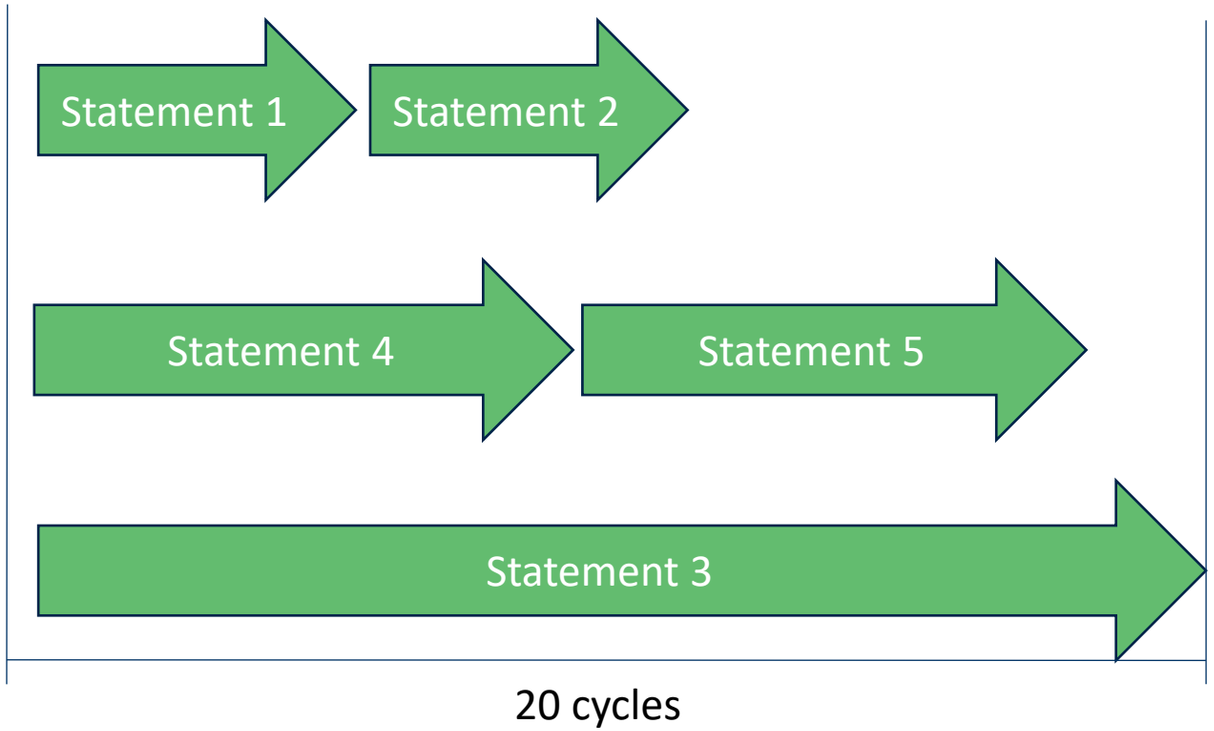
46 cycles

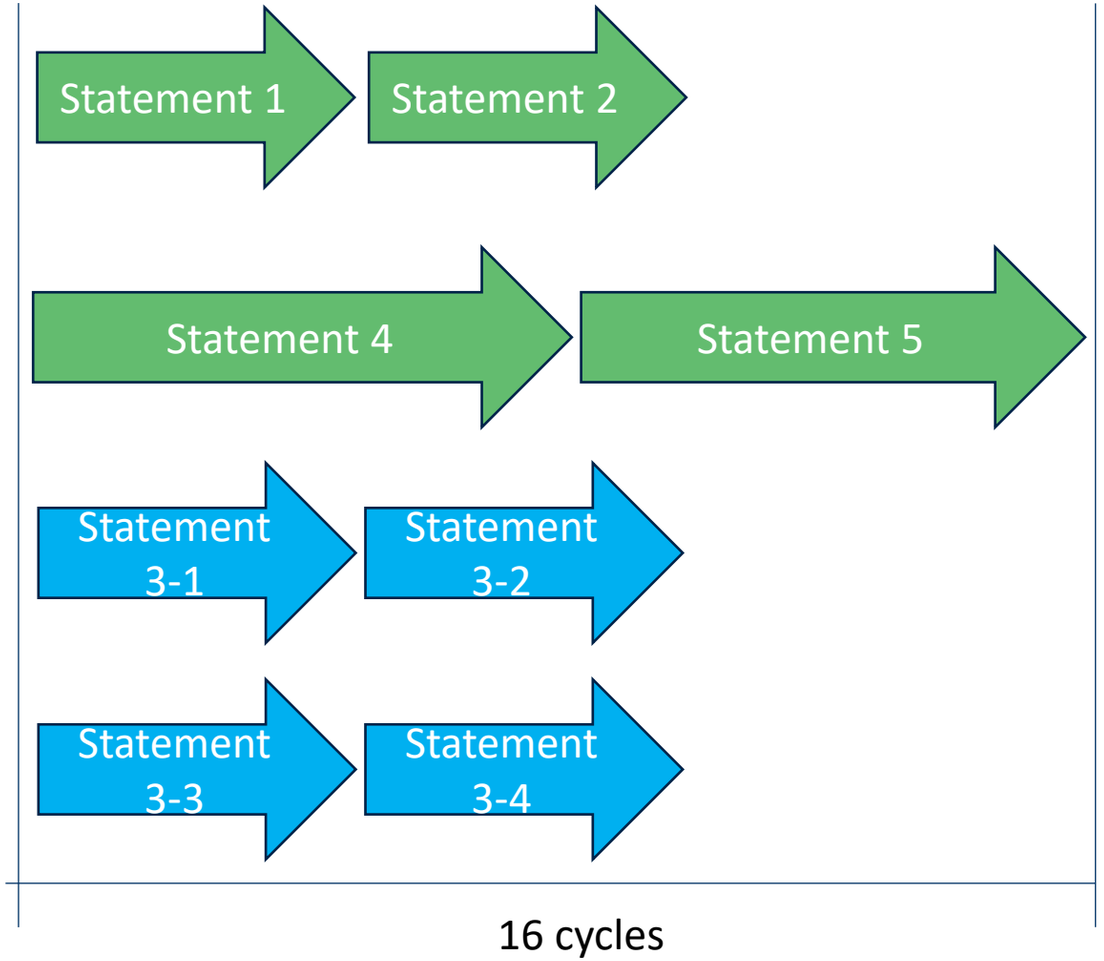


20 cycles

串行处理与并行处理的区别

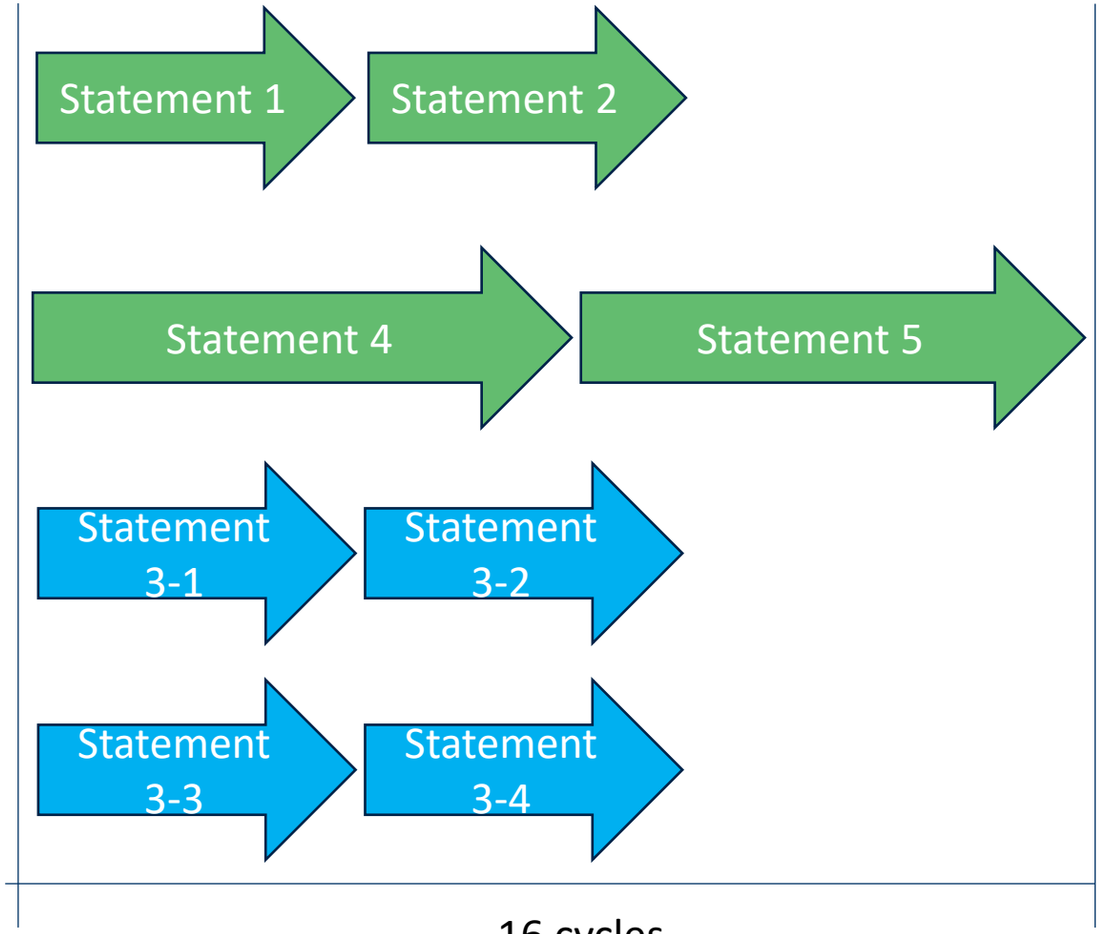
- statement 2 依赖 statement 1
- statement 5 依赖 statement 4
- statement 3 是一个 循环，跟所有的statement没有任何关系
- 有四个core可以使用
- statement3的循环可以分割成多个子代码执行(每个子代码快 5cycles)

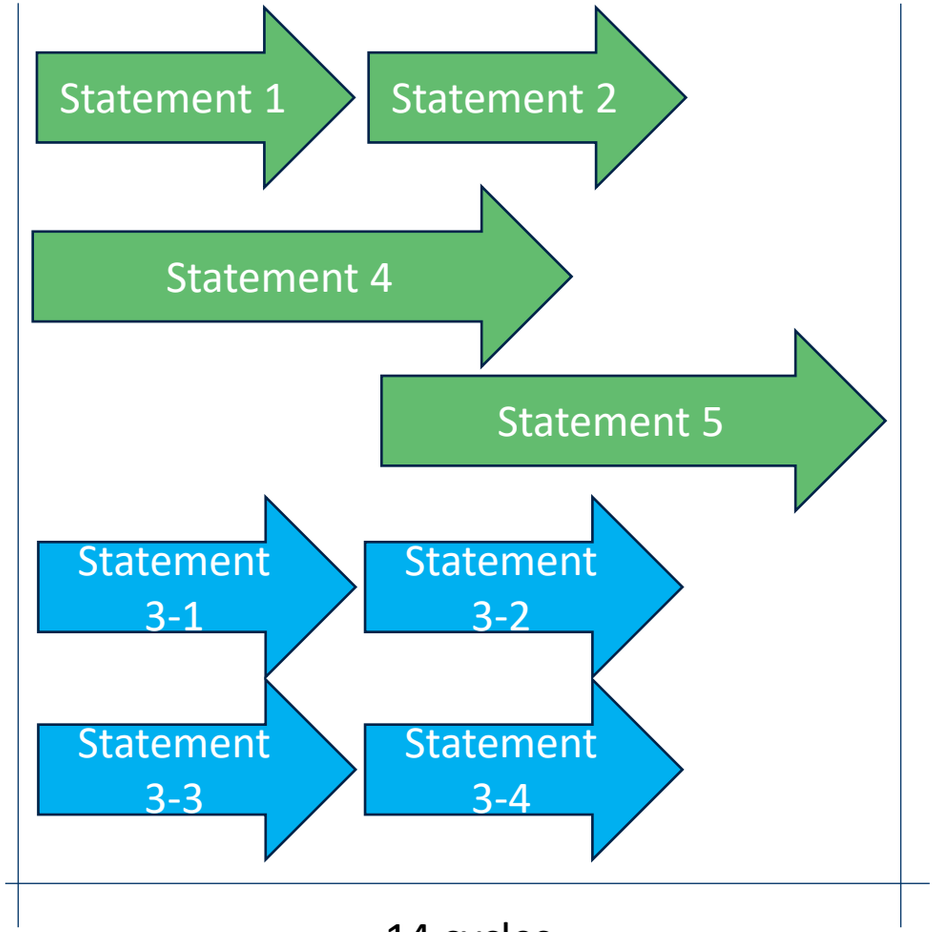




串行处理与并行处理的区别

- statement 2 依赖 statement 1
- statement 5 依赖 statement 4
- statement 3 是一个 循环，跟所有的statement没有任何关系
- 有四个core可以使用
- statement 3 的循环可以分割成多个子代码执行(每个子代码快 5cycles)
- statement 5 可以在 statement 4 彻底执行结束前就知道所依赖的结果了





串行处理与并行处理的区别

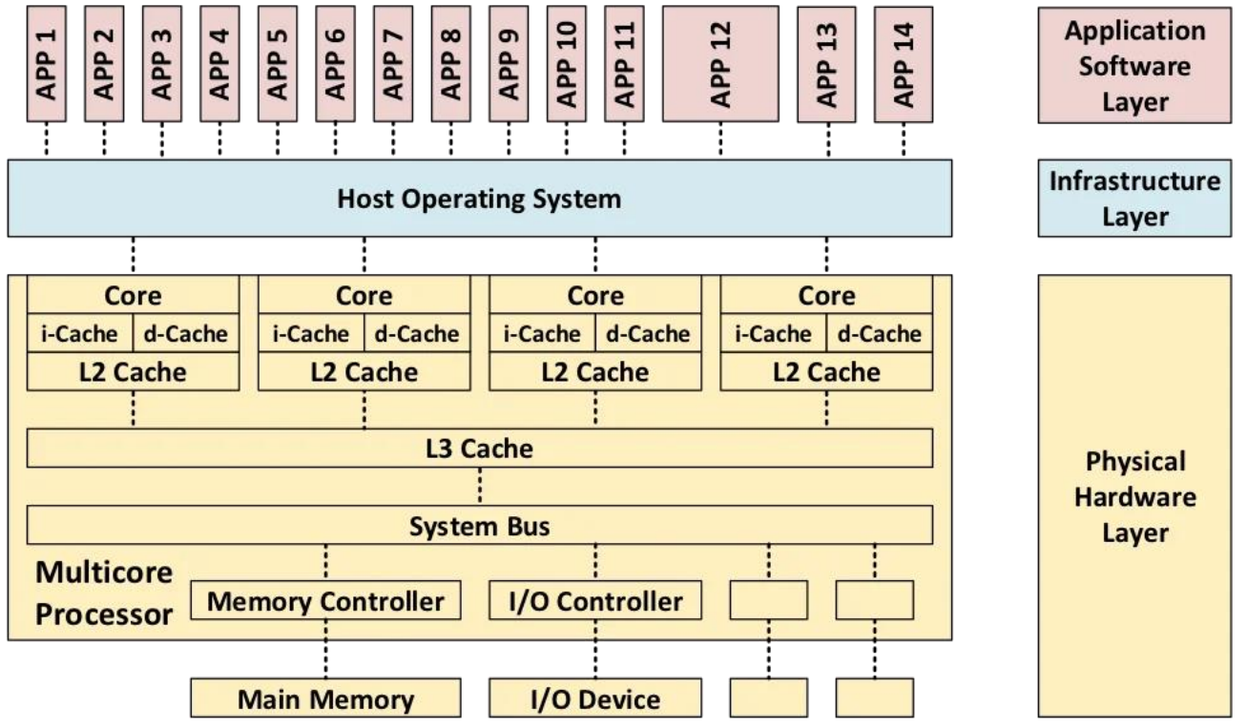
- 回顾一下，我们在这里都做了哪些事情
  - 把没有数据依赖的代码分配到各个core各自执行 (schedule, 调度)
  - 把一个大的loop循环给分割成多个小代码，分配到各个core执行(loop optimization)
  - 在一个指令彻底执行完以前，如果已经得到了想要得到的数据，可以提前执行下一个指令(pipeling, 流水线)

- 我们管这一系列的行为，称作parallelization (并行化)。我们得到的可以充分利用多核多线程的程序叫做parallelized program(并行程序)

并行执行

Parallel processing(并行)

- 指令/代码块同时执行
- 充分利用multi-core(多核)的特性，多个core一起去完成一个或多个任务
- 使用场景：科学计算，图像处理，深度学习等等



Multi-core processor[1]

[1]Multicore Processing

并行执行

Parallel processing(并行)

- Loop parallelization
  - 大部分消耗时间长的程序中，要不然就是在I/O上的内存读写消耗时间上长，要不然就是在loop上。针对loop的并行优化是很重要的一个优化策略
  - 在图像处理/深度学习中很多地方都是用到了循环
  - 比如说：pre/post process (前处理后处理)
    - resize, crop, blur, bgr2rgb, rgb2gray, dbscan, findCounters
  - 在比如说：DNN中的卷积(convolution layer)以及全连接层(Fully connected layer)

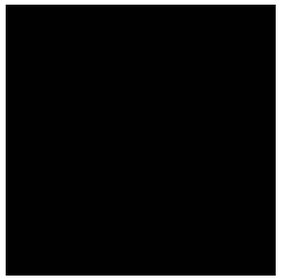
并行执行

(*)请注意，不同编程语言的默认的ordering是不同的。这里举几个例子：

- row major: C/C++/Objective-C, Pascal, C#
- column major: Fortran, OpenGL, MATLAB, R, Julia

所以不同语言的reordering效果会不同。这里建议大家测试一下

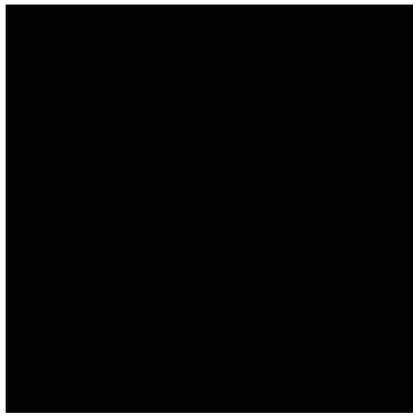
Parallel processing(并行)
- Loop parallelization



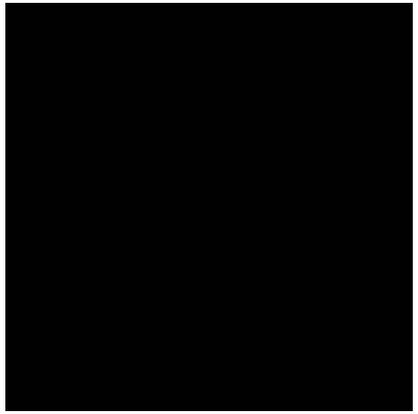
Default ordering(*)

Reordering(*)

Vectorization



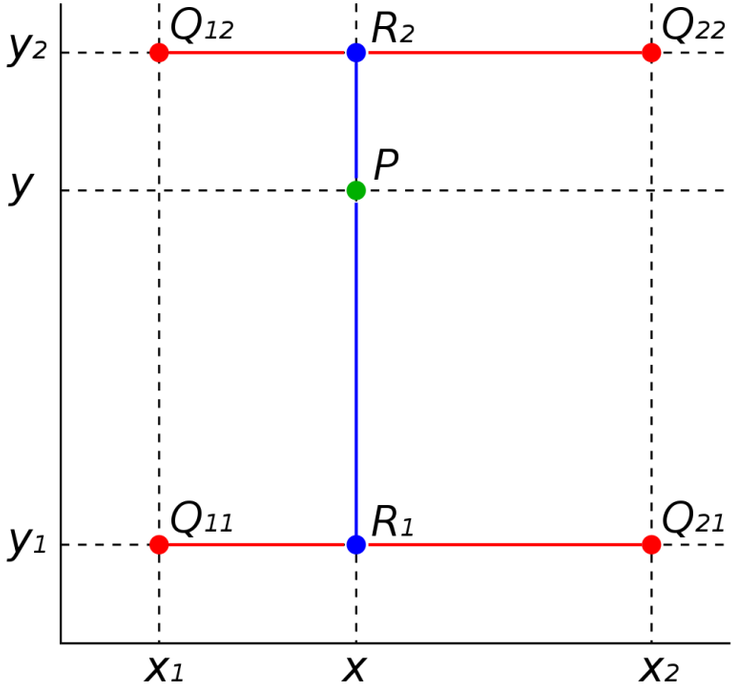
Tiling



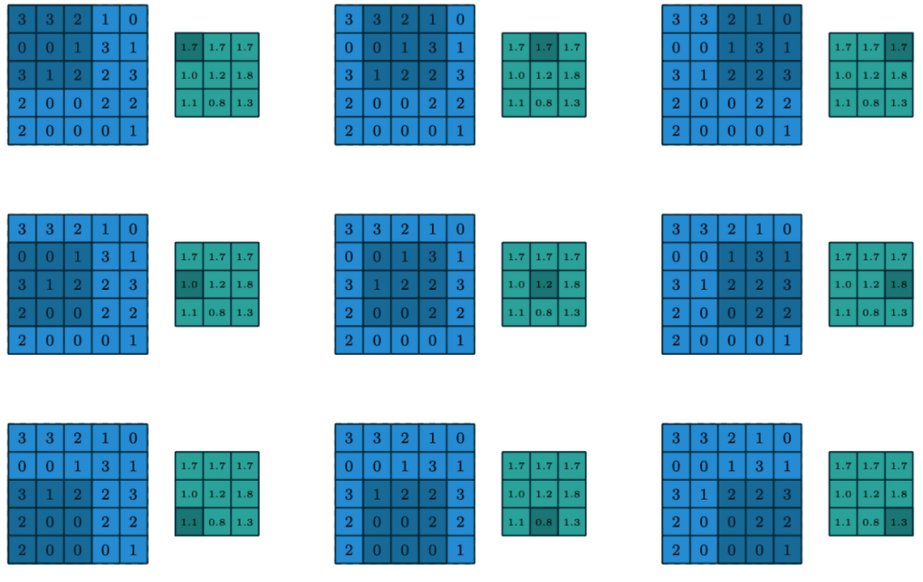
Tiling in parallel

并行执行

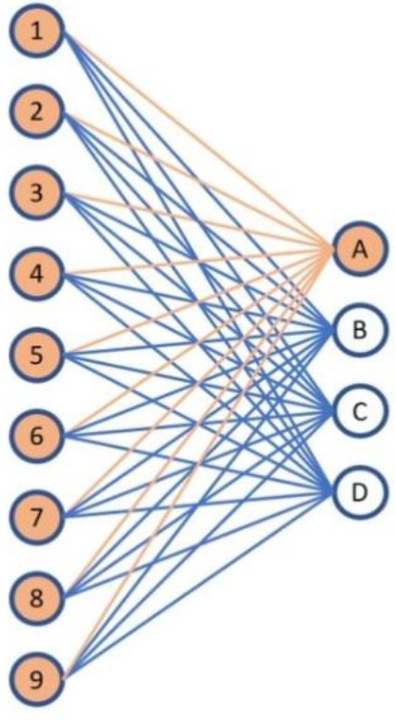
Parallel processing(并行)
- Image processing and DNN



Bilinear interpolation[1]
(双线性插值)



Convolution layer[2]
(卷积层)



Fully connected layer[3]
(全连接层)

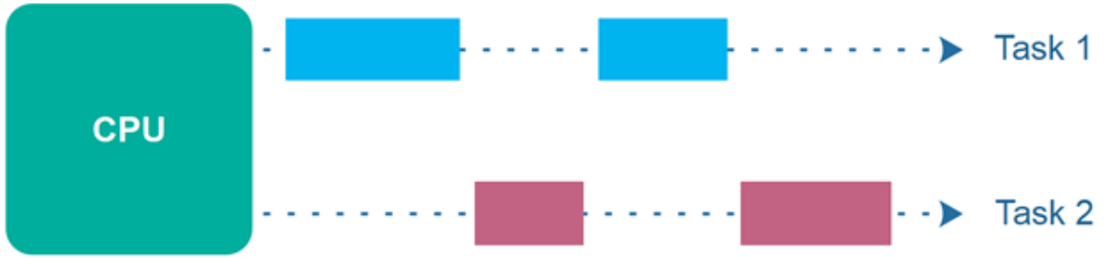
[1]Bilinear interpolation

[2]A guide to convolution arithmetic for deep learning

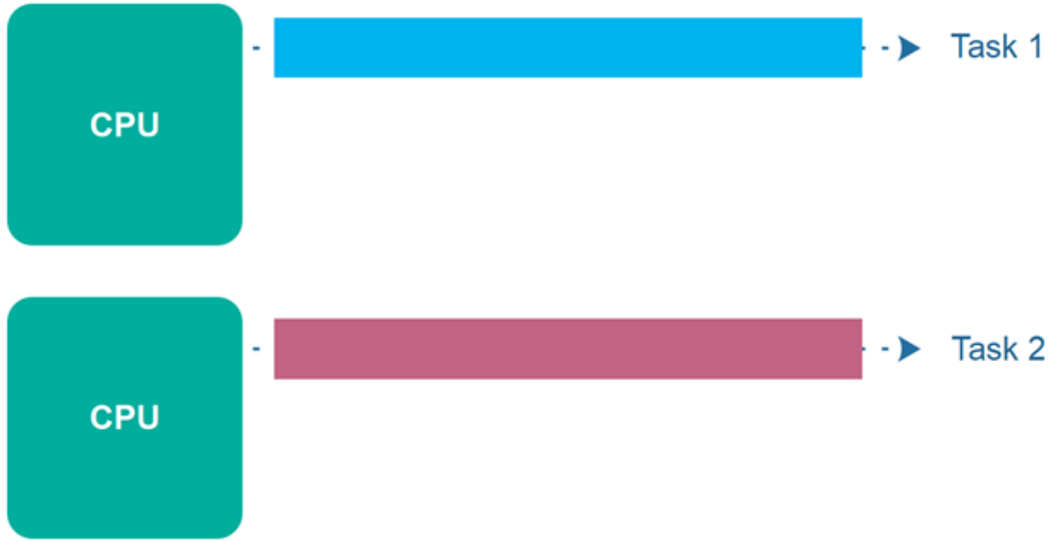
[3]Fully Connected Layer vs. Convolutional Layer: Explained

容易混淆的几个概念

- “并行”与“并发”的区别
  - 并行(parallel)
    - 物理意义上同时执行
  - 并发(concurrent)
    - 逻辑意义上同时执行



concurrency (并发)

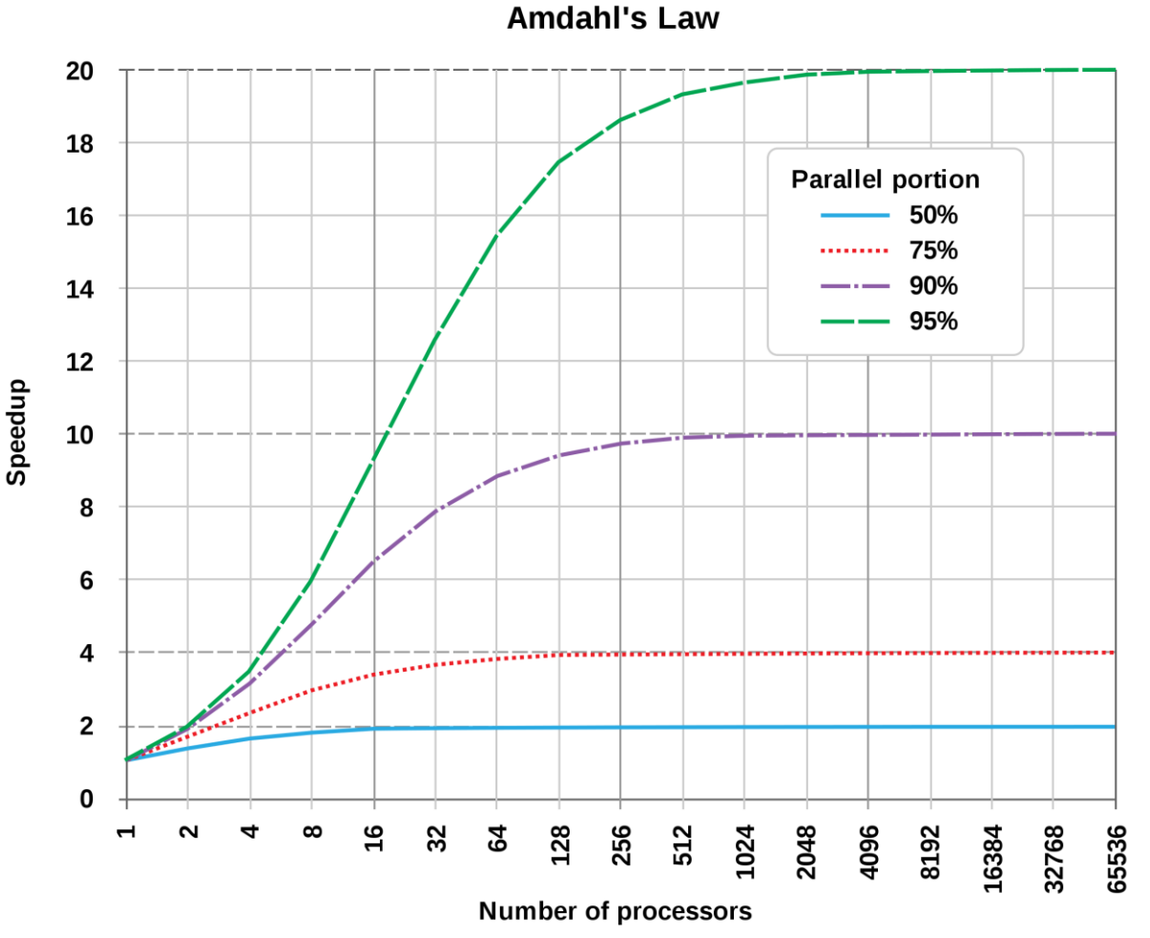


Parallel (并行)

容易混淆的几个概念

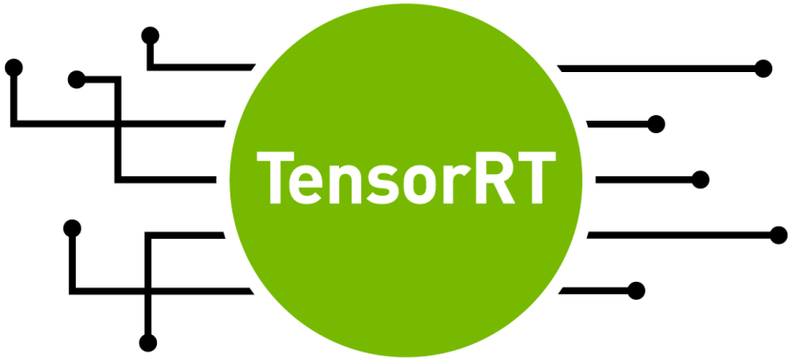
- “进程”与“线程”的关系
  - 线程是进程的子集。一个进程可以有多个线程

- “多核”与“加速比”的关系
  - 双核的加速不一定就是两倍
  - 8核的加速比有时会差于4核



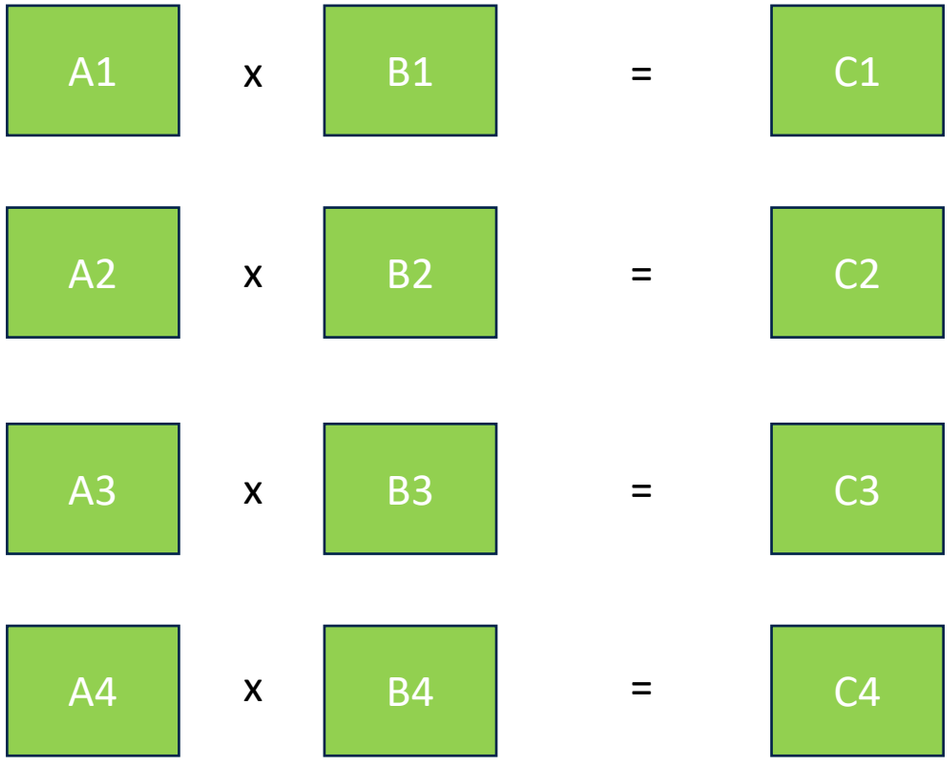
常见的并行处理

- 自古以来的非常非常热门的课题
  - 编译器自动化并行优化
    - GCC, LLVM, TVM, …
  - 针对For循环的并行优化
    - tile, fuse, split, vectorization, …
  - 计算图优化
    - CFG, HTG, MTG, …
  - 数据流
    - Dataflow compiler
    - Dataflow architecture
  - Polyhedral
    - Polyhedral compiler
  - HPC
    - High performance computing
  - …



常见的并行处理

- SIMD
  - Single Instruction Multiple Data

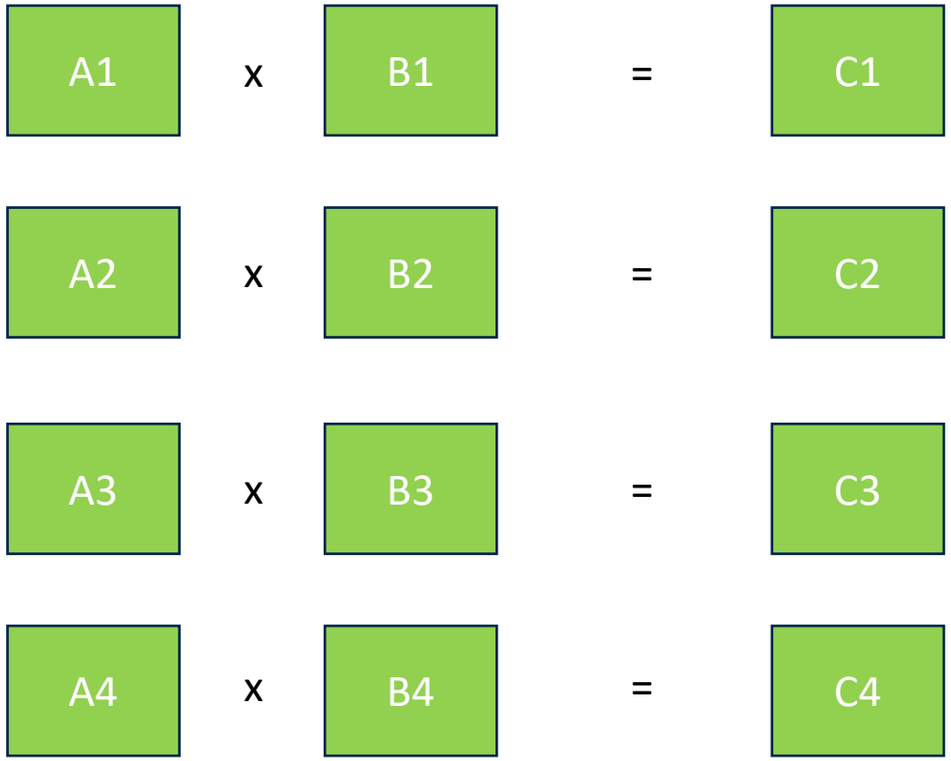


取指，读取数据，计算，写回数据
取指，读取数据，计算，写回数据
取指，读取数据，计算，写回数据
取指，读取数据，计算，写回数据
（同样的操作要做四次，是不是很麻烦？）

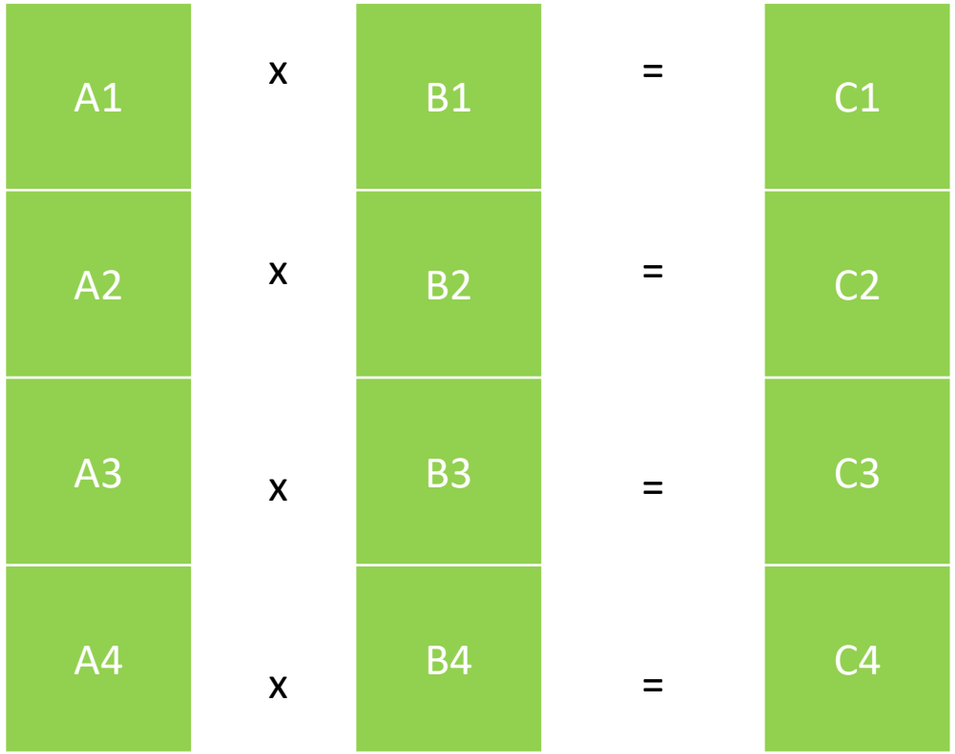
常见的并行处理

- SIMD
  - Single Instruction Multiple Data

取指，读取数据，计算，写回数据
(一次就okay了！)



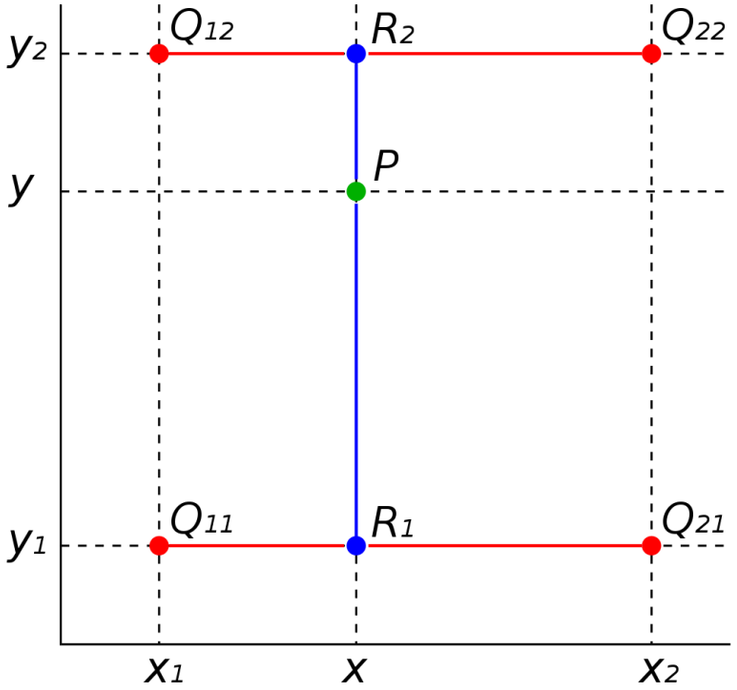
Scalar Operation



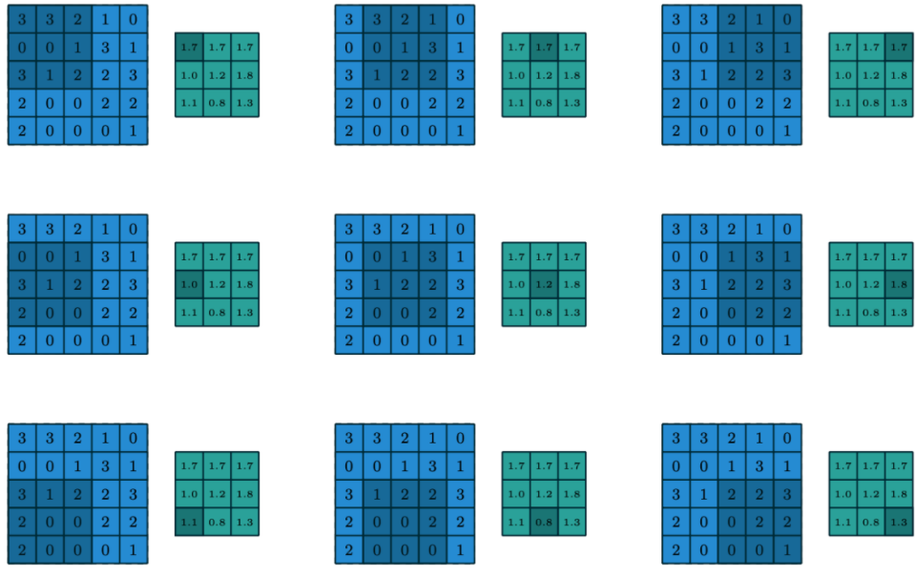
SIMD Operation

常见的并行处理

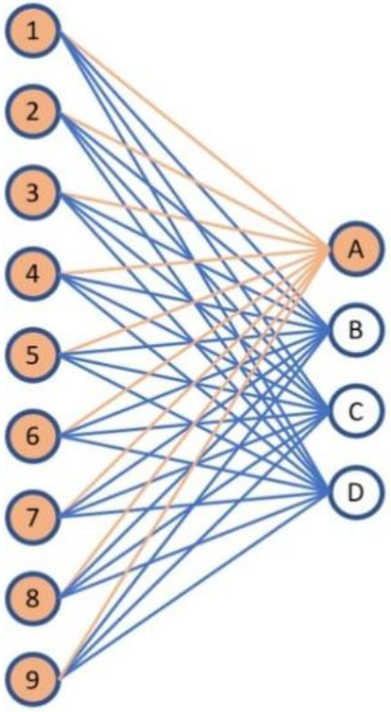
Parallel processing(并行)
- Image processing and DNN



Bilinear interpolation[1]
(双线性插值)



Convolution layer[2]
(卷积层)

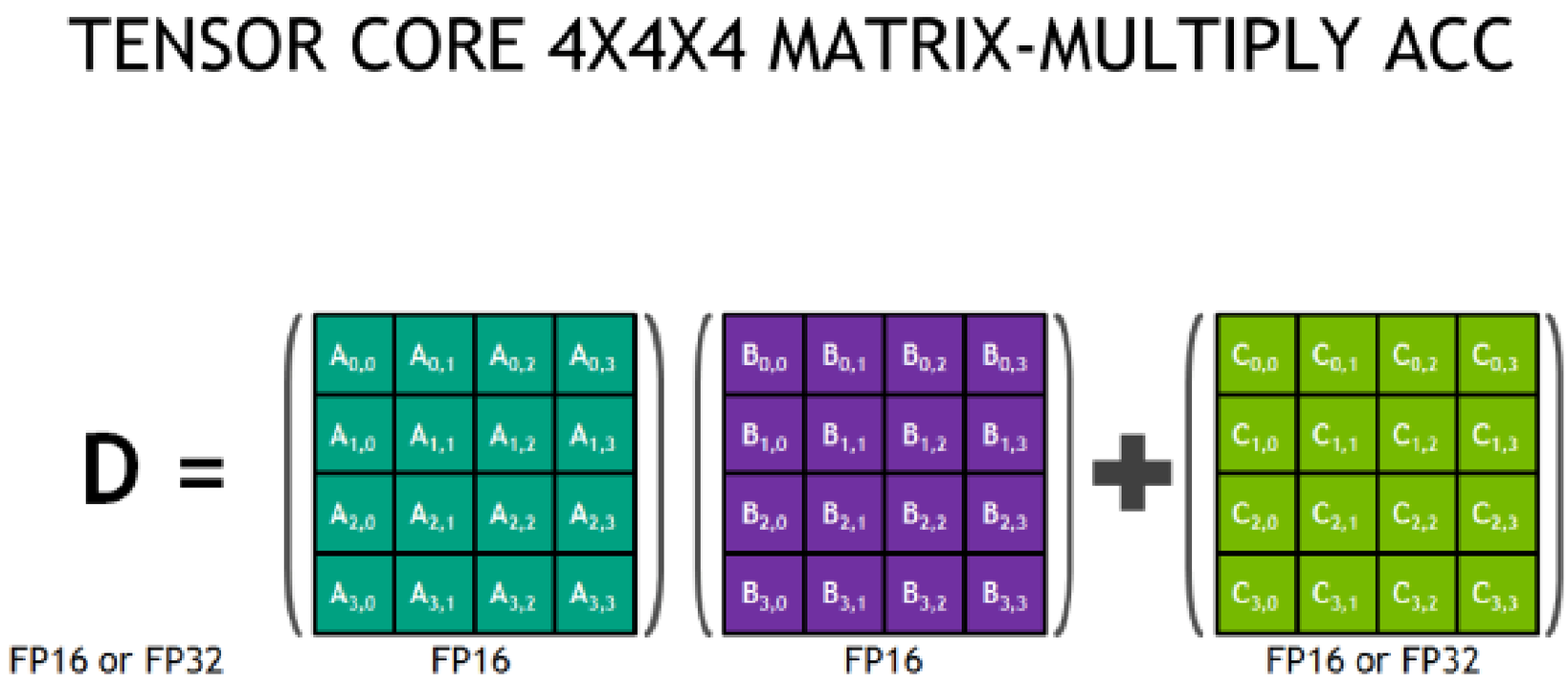


Fully connected layer[3]
(全连接层)

[1]Bilinear interpolation
[2]A guide to convolution arithmetic for deep learning
[3]Fully Connected Layer vs. Convolutional Layer: Explained

常见的并行处理

在cuda编程与TensorRT中，以及NVIDIA中的tensor core的设计理念中都存在着SIMD(*)



(*)准确来说，CUDA编程中是使用的是SIMT(Single Instruction Multiple Thread)，跟SIMD很相像，是SIMD的高级版。这个会在之后详细讲

常见的并行处理

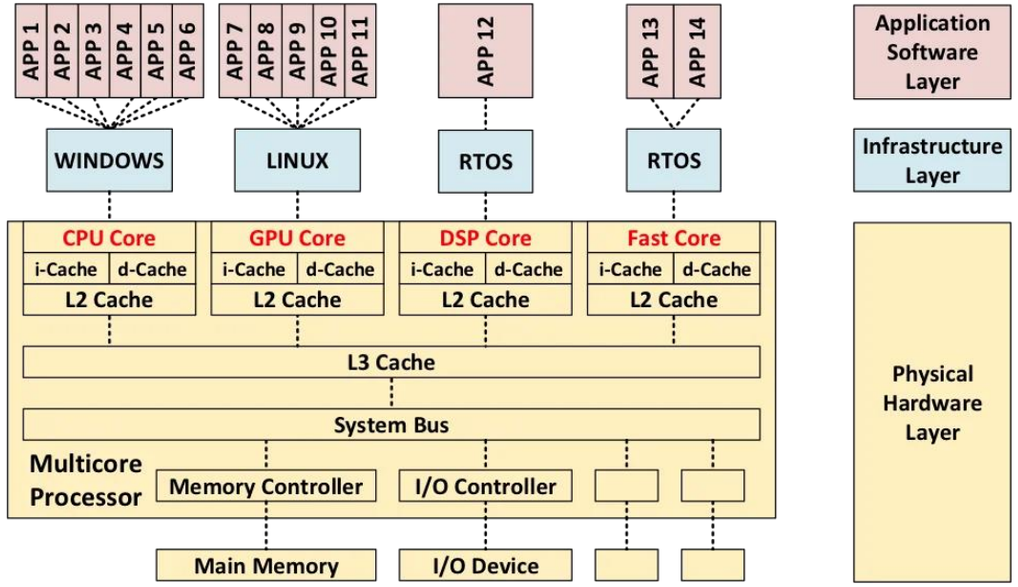
其他比较常见的并行处理方式(这里以在模型部署中比较常见的方法为举例)

CPU上的并行处理
- OpenMP
- pthread
- MPI
- Halide

(应用场景：图像预处理/后处理，Multi-task模型)

GPU上的并行处理
- CUDA programming

(应用场景：DNN优化)



Heterogeneous Multicore Processor (异构架构)[1]

[1]Multicore Processing

GPU并行处理

Goal: 理解CPU和GPU并行处理上的不同，以及影响并行处理的基本因素

这个小节会涉及到的关键字

lantency:

- 完成一个指令所需要的时间

memory latency:

- CPU/GPU从memory获取数据所需要的等待时间
- CPU并行处理的优化的主要方向

throughput(吞吐量):

- 单位时间内可以执行的指令数
- GPU并行处理的优化的主要方向

Multi-threading:

- 多线程处理

CPU与GPU在并行处理的优化方向

CPU:

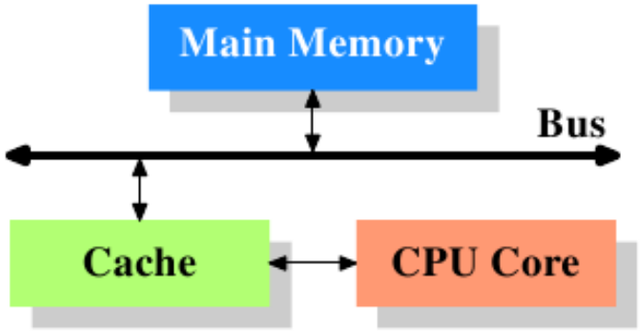
- 目标在于减少memory latency

GPU:

- 目标在于提高throughput

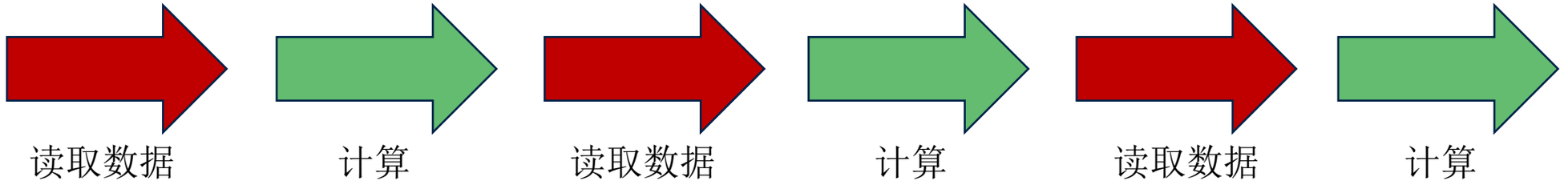
CPU与GPU在并行处理的优化方向

memory latency是什么？

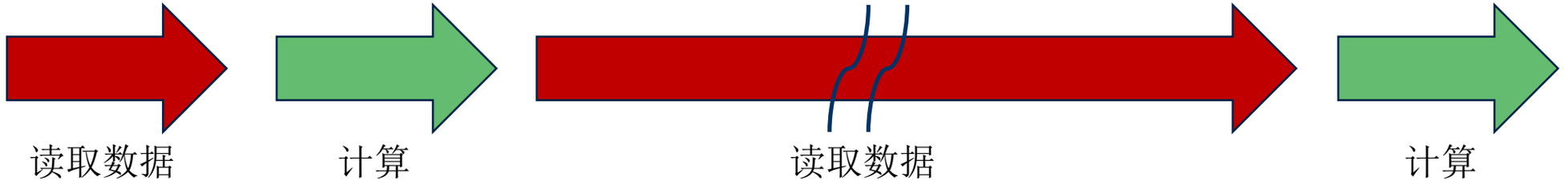


memory access

cache hit

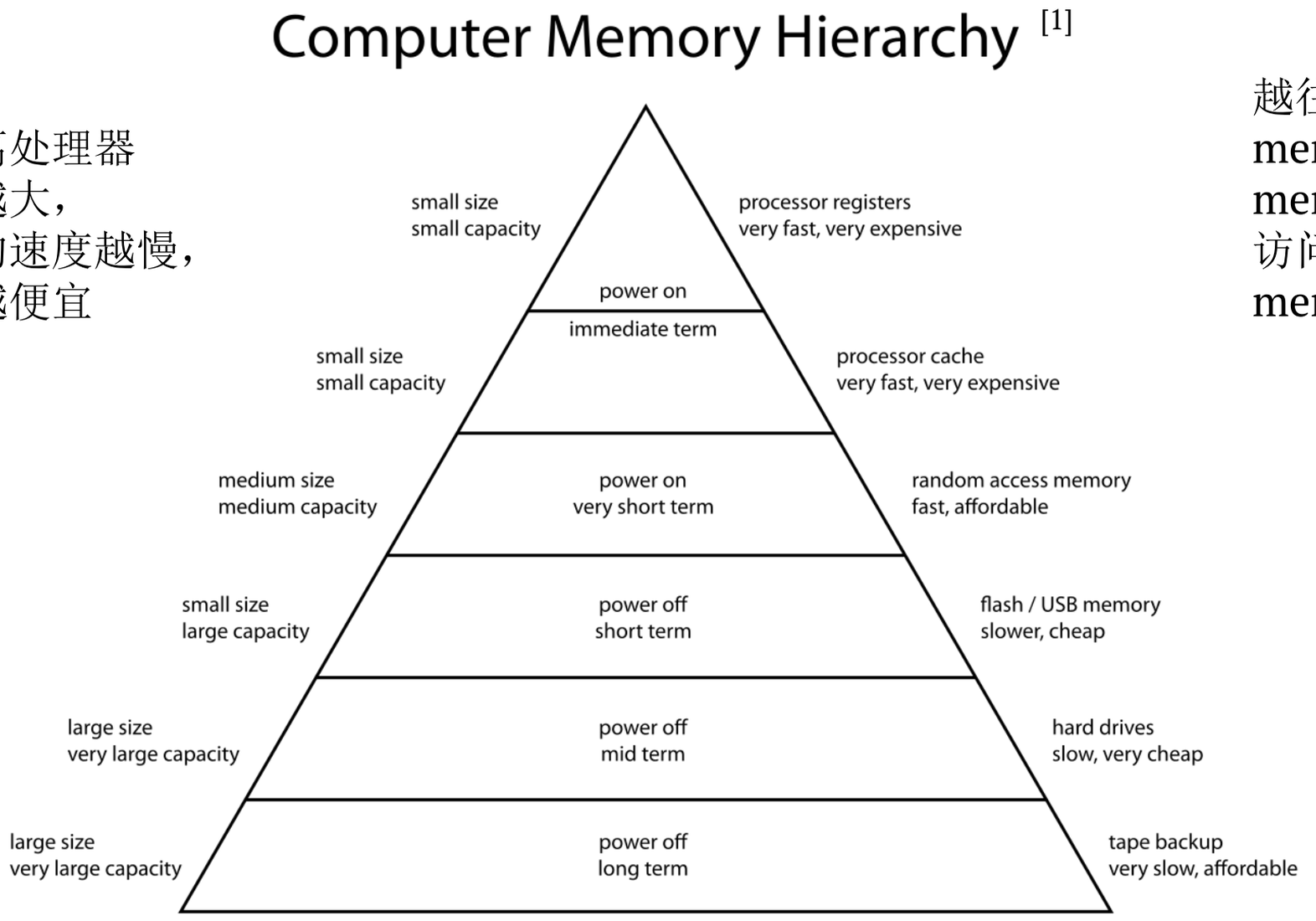


cache miss



CPU与GPU在并行处理的优化方向

越往下走，
memory越远离处理器
memory空间越大，
访问memory的速度越慢，
memory价格越便宜

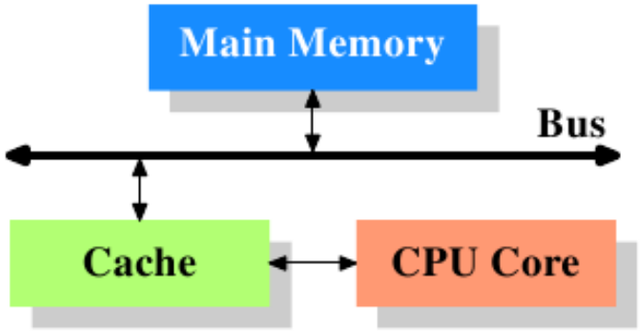


越往上走，
memory越靠近处理器
memory空间越小，
访问memory的速度越快，
memory价格越贵

[1]About memory hierarchy

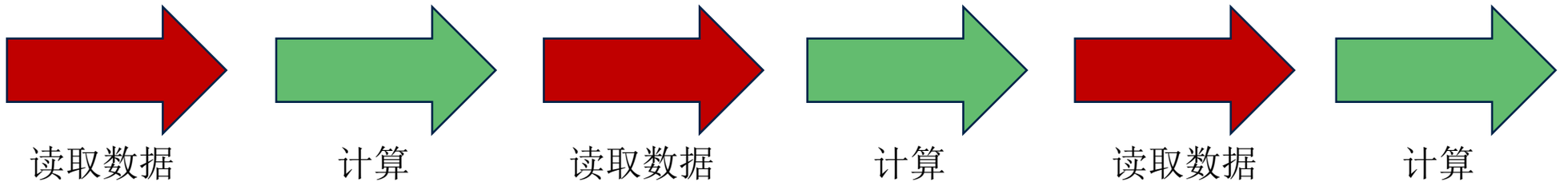
CPU与GPU在并行处理的优化方向

memory latency是什么？

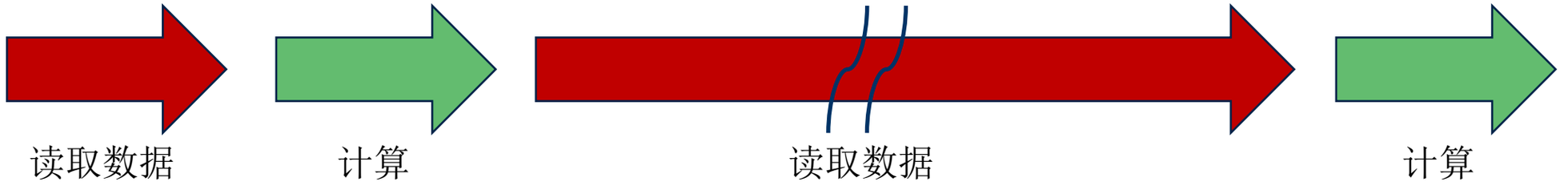


memory access

cache hit



cache miss



这个时候，CPU core由于没有数据，所以在等待数据的到来。这个状态叫做stall

如果数据不在cache里，那么就需要往下级memory中寻找数据，然而访问下级memory是很耗时的

CPU与GPU在并行处理的优化方向

"Locality is efficiency, Efficiency is power, Power is performance, Performance is King", Bill Dally

```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms

Notes
-----
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns
```

L1 cache: 0.5 ns
L2 cache: 7 ns
Main memory: 100 ns

不同memory在latency上的比较[1]

[1]Latency number every programmer should know

CPU与GPU在并行处理的优化方向

CPU是如何进行优化的？

- Pipeline
- cache hierarchy
- Pre-fetch
- Branch-prediction
- Multi-threading

CPU与GPU在并行处理的优化方向

CPU是如何进行优化的？

- Pipeline (流水线执行)
  - 提高throughput的一种优化

CPU instruction pipeline(CPU流水线执行)

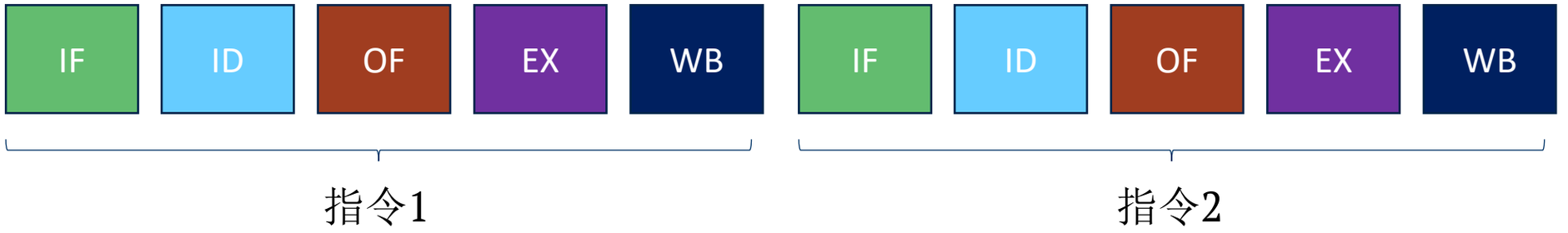
- IF: Instruction fetch
  - 把指令从memory中取出来
- ID: Instruction decode
  - 把取出来的指令给解码成机器可以识别的信号
- OF: Operand fetch
  - 把数据从memory中取出来放到register中
- EX: Execution
  - 使用ALU(负责运算的unit)来进行计算
- WB: write back
  - 把计算完的结果写回register

CPU与GPU在并行处理的优化方向

CPU是如何进行优化的？

- Pipeline (流水线执行)
  - 提高throughput的一种优化

CPU instruction pipeline(CPU流水线执行)

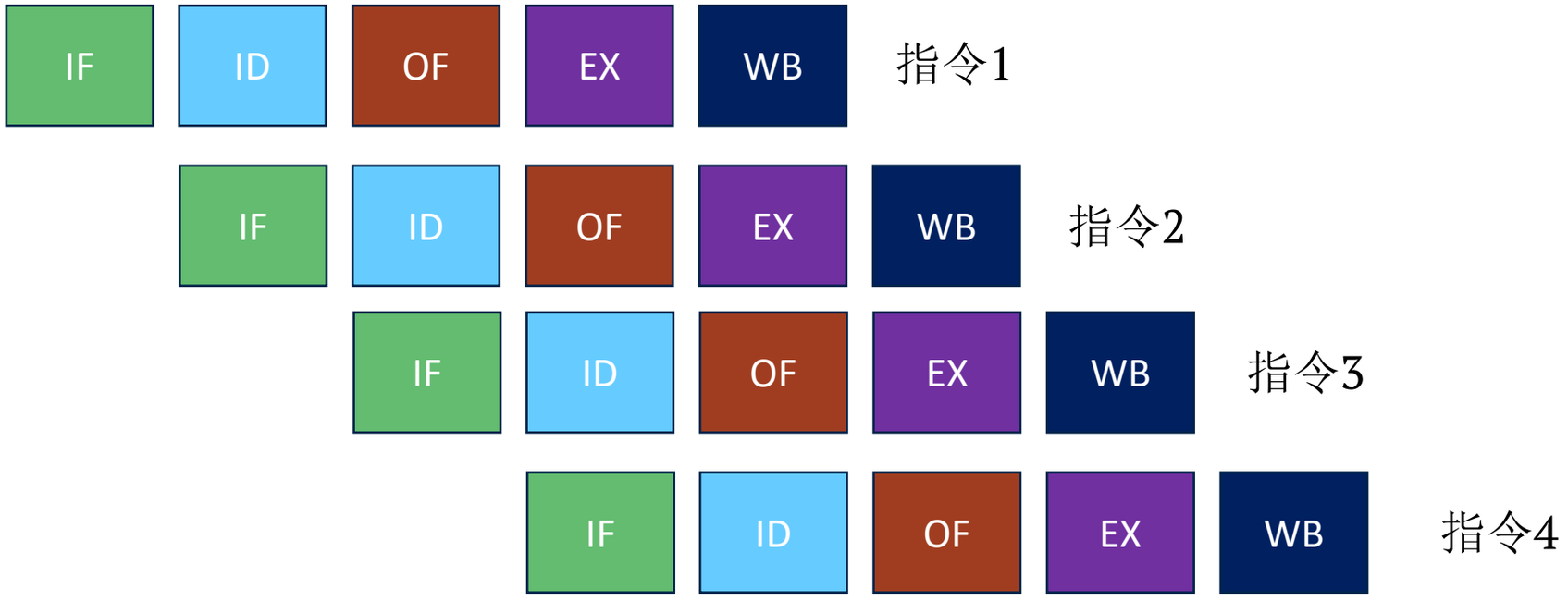


CPU与GPU在并行处理的优化方向

CPU是如何进行优化的？

- Pipeline (流水线执行)
  - 提高throughput的一种优化

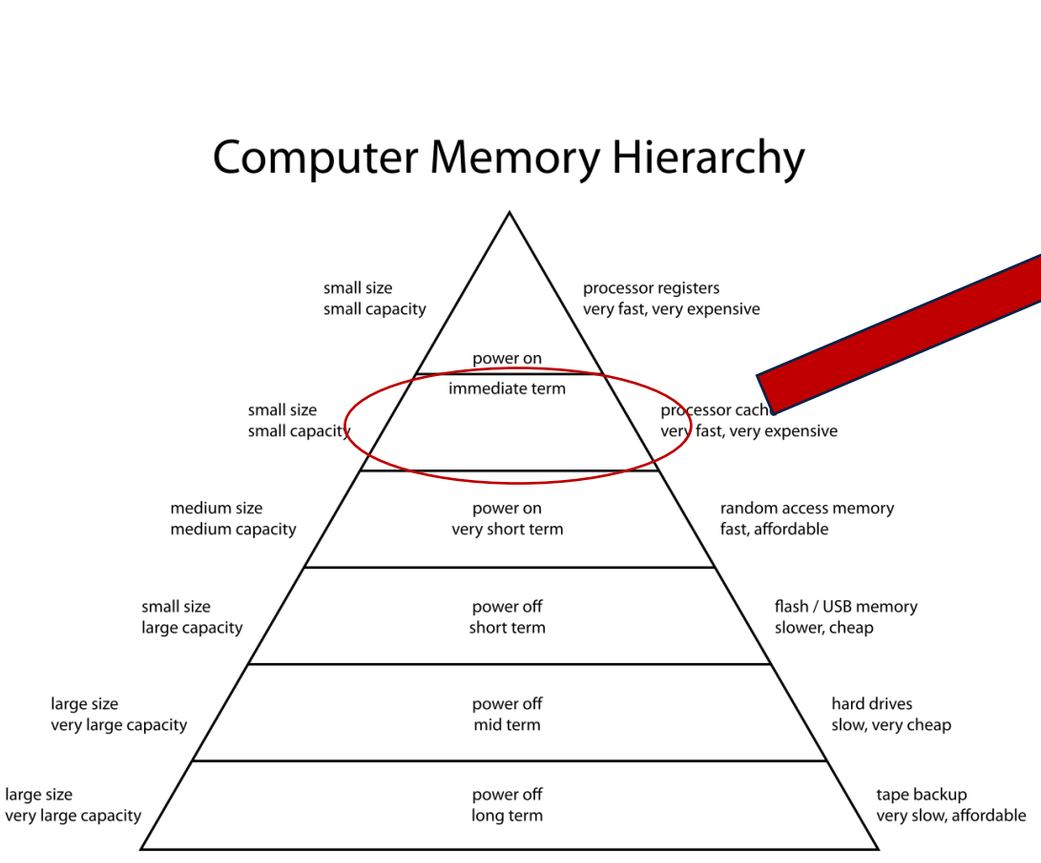
CPU instruction pipeline(CPU流水线执行)

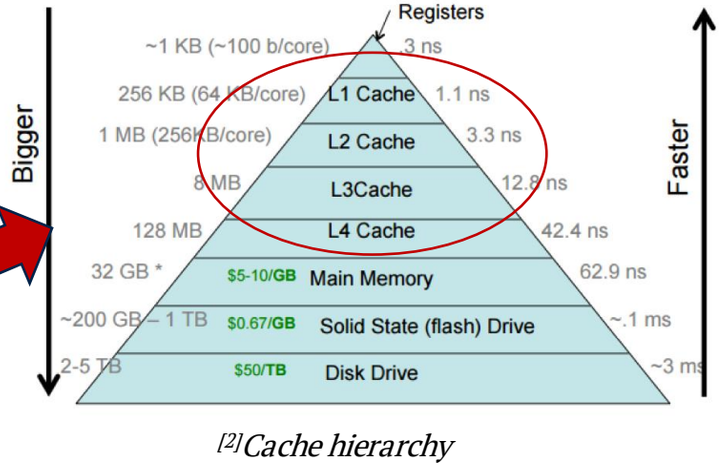


CPU与GPU在并行处理的优化方向

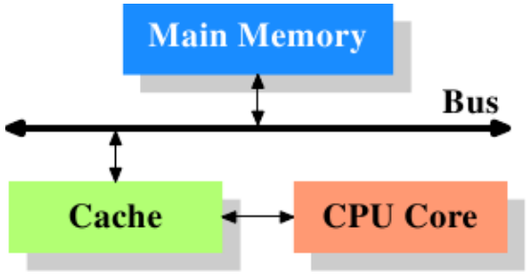
CPU是如何进行优化的？

- cache hierarchy (多级缓存)
  - 减少memory latency的一种优化

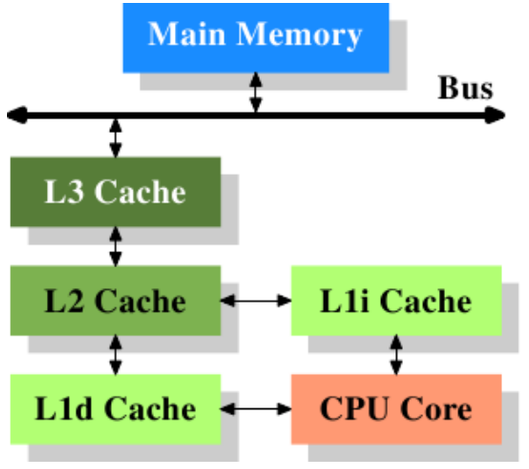




[2]*Cache hierarchy*



[2]*without cache hierarchy*



[2]*with cache hierarchy*

(扩展)

- L1: 逻辑核独占
- L2: 物理核独占
- L3: 所有物理核共享

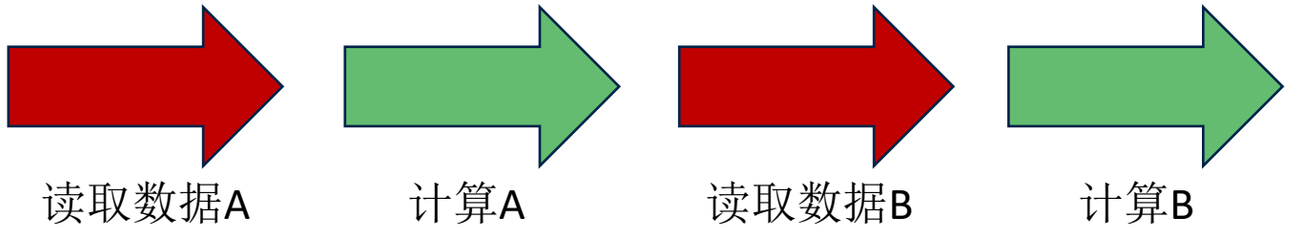
[1]Lab4: Caching
[2]Memory part 2: CPU caches

CPU与GPU在并行处理的优化方向

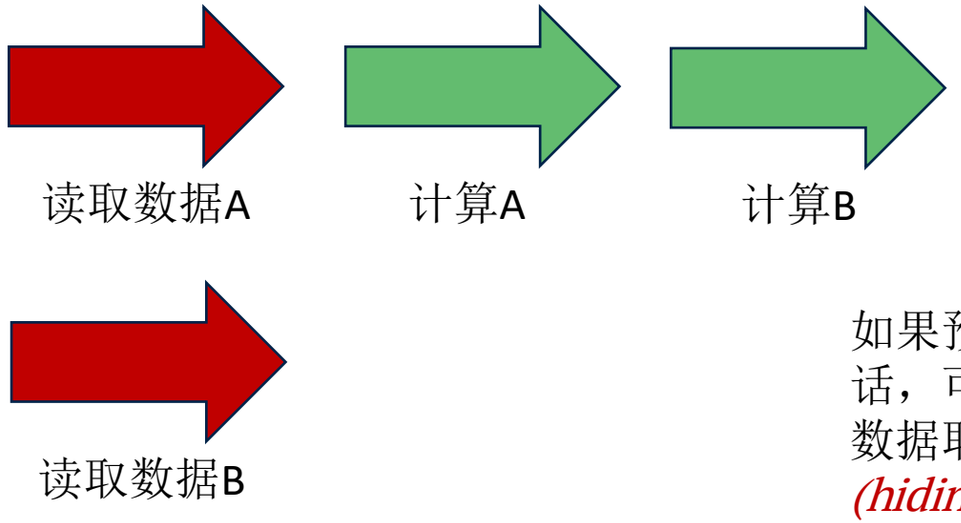
CPU是如何进行优化的？

- Pre-fetch
  - 减少memory latency的一种优化

without pre-fetch



with pre-fetch



Q:当程序中出现branch的话，如何进行pre-fetch?

如果预先知道计算A结束后就执行计算B的话，可以预先(pre)把计算B所需要的指令和数据取出来(fetch)，放入到cache中 *(hiding memory latency)*

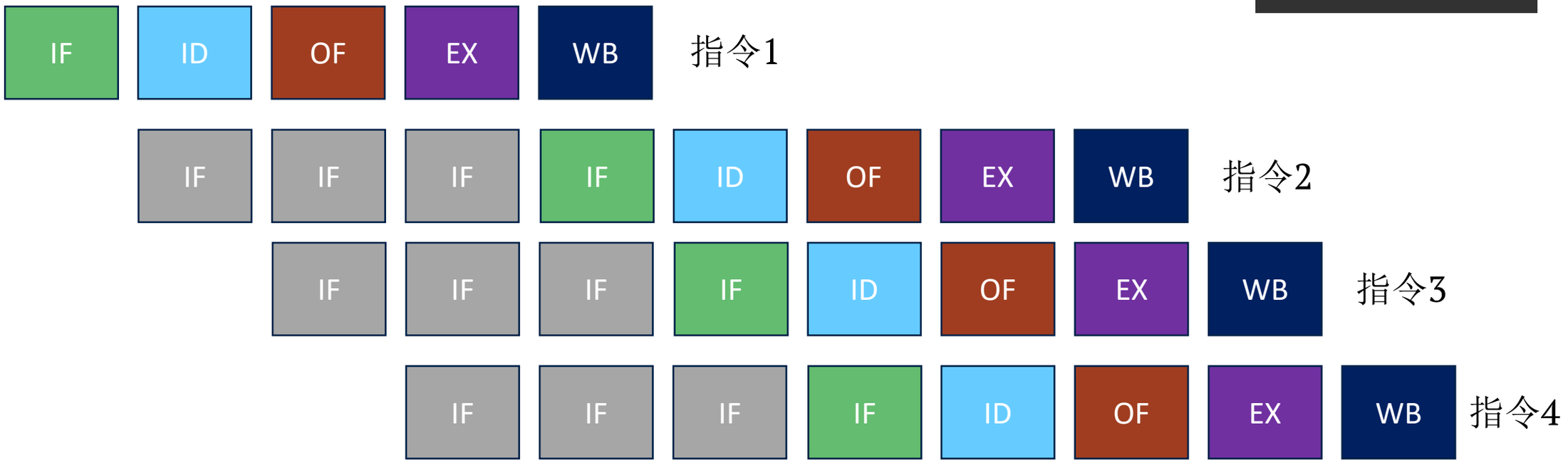
CPU与GPU在并行处理的优化方向

CPU是如何进行优化的？

- Branch-prediction(分支预测)
  - 简单来说，就是根据以往的branch得走向，去预测下一次branch的走向

```
1  while (i < 100){
2    sum += a[i];
3    i ++;
4  }
```

CPU instruction pipeline(CPU流水线执行)



CPU与GPU在并行处理的优化方向

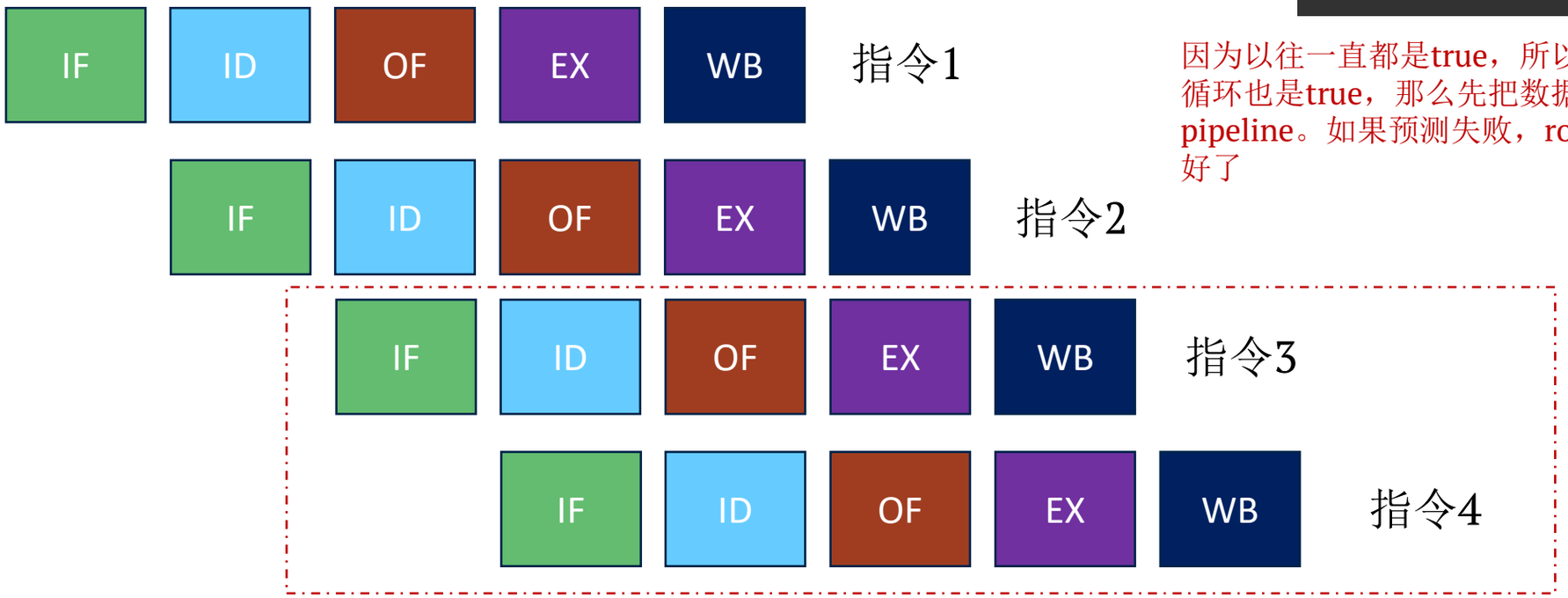
(*)CPU硬件上有专门负责分支预测的Unit

CPU是如何进行优化的？

- Branch-prediction(分支预测)
  - 简单来说，就是根据以往的branch得走向，去预测(*)下一次branch的走向

```
1  while (i < 100){
2    sum += a[i];
3    i ++;
4  }
```

CPU instruction pipeline(CPU流水线执行)

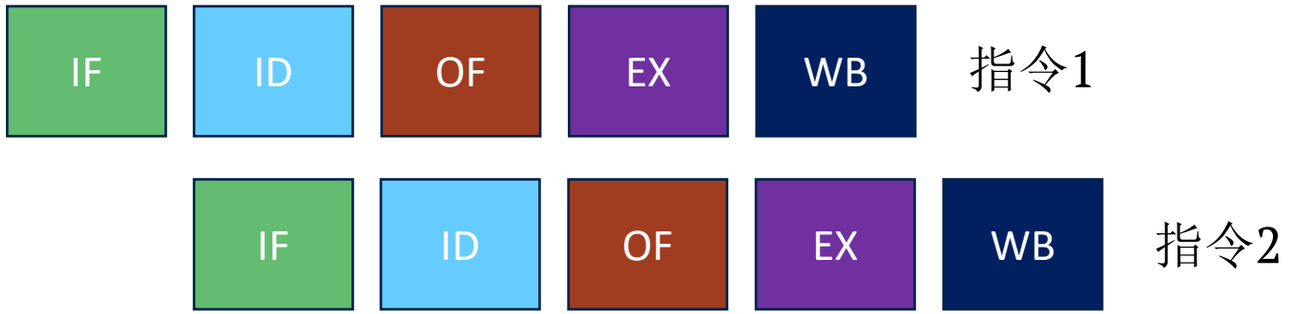


因为以往一直都是true，所以预测下一次循环也是true，那么先把数据取出来进行pipeline。如果预测失败，rollback回去就好了

CPU与GPU在并行处理的优化方向

CPU是如何进行优化的？

- Multi-threading
  - 充分利用计算资源的一种技术
  - 让因为数据依赖或者cache miss而stall的core去做一些其他的事情
  - 提高throughput的一种技术




```
BB1: a = b + 1;
BB2: c = d + 1;
```

指令1与指令2没有data dependency

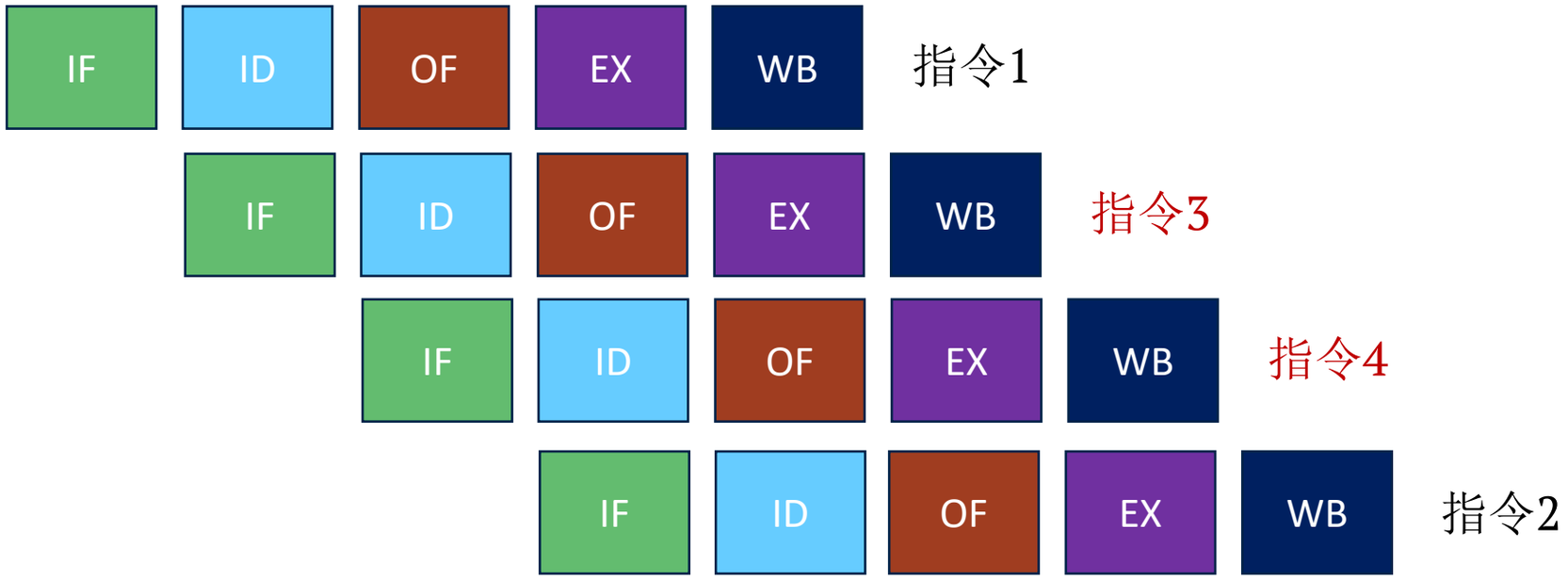

```
BB1: a = b + 1;
BB2: c = a + d;
```

指令1与指令2有data dependency

CPU与GPU在并行处理的优化方向

CPU是如何进行优化的？

- Multi-threading
  - 充分利用计算资源的一种技术
  - 让因为数据依赖或者cache miss而stall的core去做一些其他的事情
  - 提高throughput的一种技术




```
BB1: a = b + 1;
BB2: c = a + d;
```

指令1与指令2有data dependency
(由于存在stall，所以可以让它去做一些其他的指令)

CPU与GPU在并行处理的优化方向

CPU是如何进行优化的？

- 减少memory latency
  - cache hierarchy
  - pre-fetch
  - branch prediction
- 提高throughput
  - pipeline
  - multi-threading

But…

由于CPU处理的大多数都是一些复杂逻辑的计算，有大量的分支以及难以预测的分支方向，所以增加core的数量，增加线程数而带来的throughput的收益往往并不是那么高

So …

去掉复杂的逻辑计算，去掉分支，把大量的简单运算放在一起的话，是不是就可以最大化的提高throughput呢？

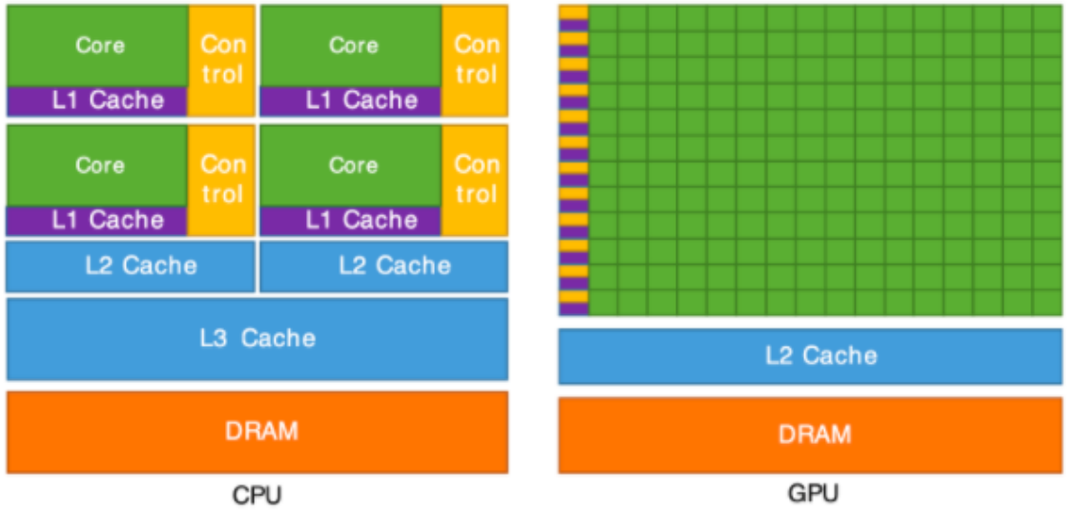
答案是yes，这个就是GPU所做的事情。

CPU与GPU在并行处理的优化方向

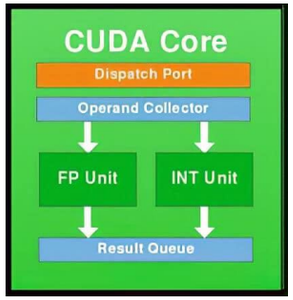
GPU的特点

- multi-threading技术
- 大量的core，可以支持大量的线程
  - CPU并行处理的threads数量规模：数十个
  - GPU并行处理的threads数量规模：数千个

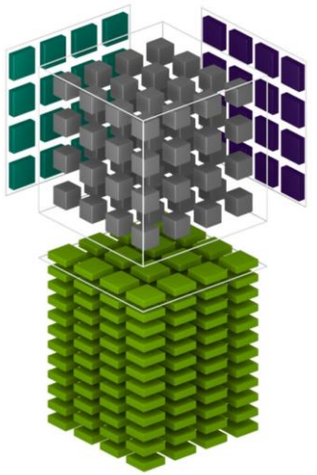
- 每一个core负责的运算逻辑十分simple
  - CUDA core:
    - D = A * B + C
  - Tensor core:
    - 4x4x4的matrix calculation
    - D = A * B + C



CPU体系架构与GPU体系架构的不同[1]



CUDA core[2]



Tensor core[3]

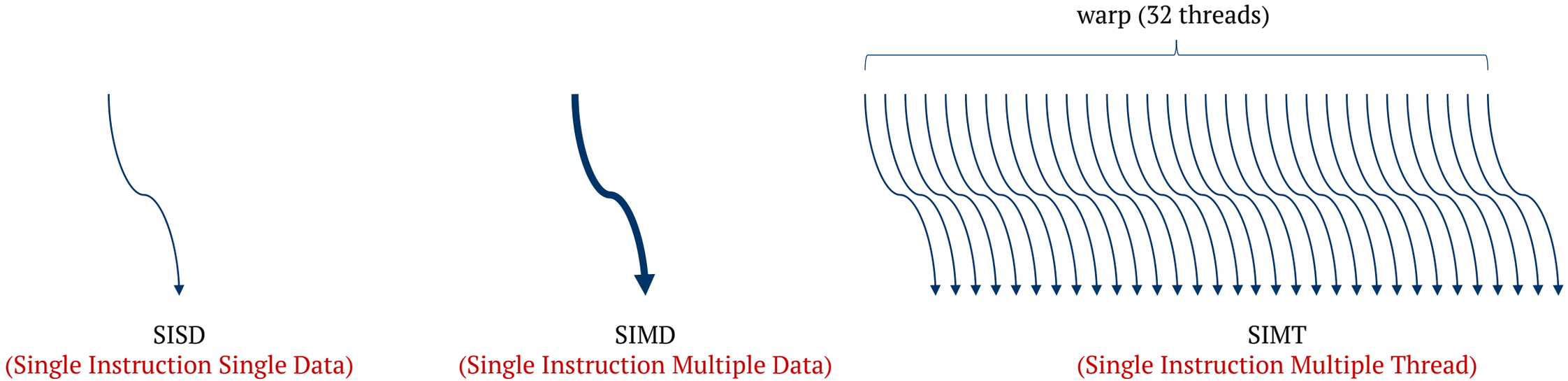
[1]Understanding GPU Architecture: GPU Characteristics
[2]NVIDIA CUDA Cores Explained: How are they different?
[3]Programming Tensor Cores in CUDA 9

CPU与GPU在并行处理的优化方向

(*)Warp schedular是GPU体系架构中特有的概念，CPU中没有这个

GPU的特点

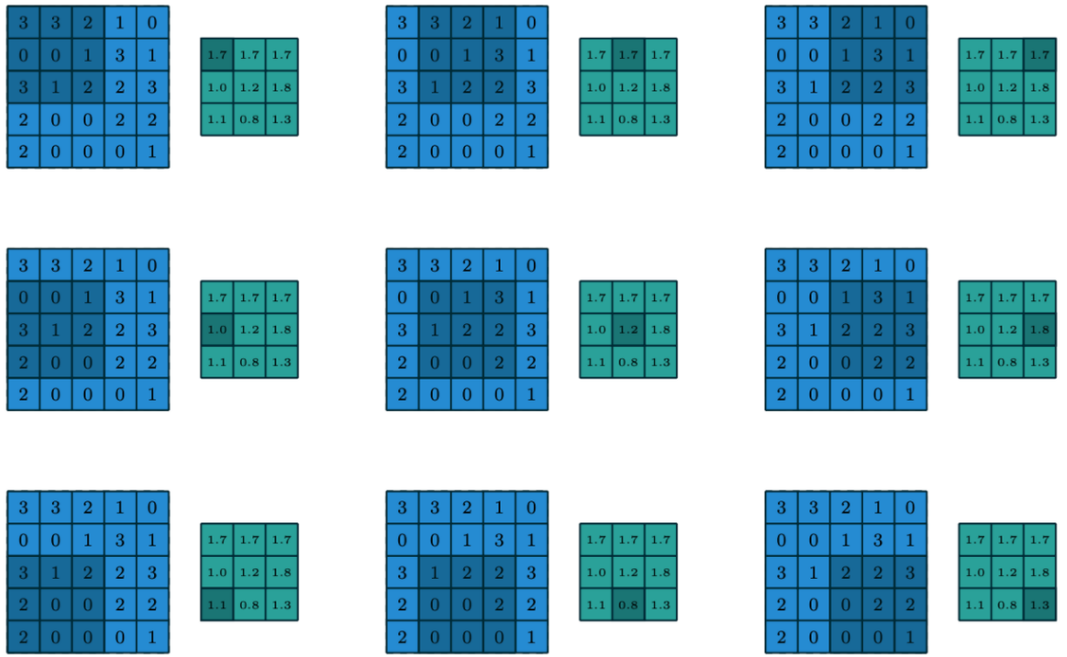
- SIMT
  - 类似于SIMD的一种概念
  - 将一条指令分给大量的thread去执行
  - thread间的调度是由warp来负责管理
    - GPU体系架构中有一个warp schedular(*)，专门负责管理线程调度的



CPU与GPU在并行处理的优化方向

GPU的特点

- 由于throughput非常的高，所以相比与CPU，cache miss所产生的latency对性能的影响比较小
- GPU主要负责的任务是大规模计算(图像处理、深度学习等等)，所以一旦fetch好了数据以后，就会一直连续处理，并且很少cache miss



Convolution layer[1]

[1]A guide to convolution arithmetic for deep learning

Summary

CPU与GPU的分工不同

- CPU:
  - 适合复杂逻辑的运算
  - 优化方向在于减少memory latency
  - 相关的技术有，cache hierarchy, pre-fetch, branch-prediction, multi-threading
  - 不同于GPU，CPU硬件上有复杂的分支预测器去实现branch-prediction
  - 由于CPU经常处理复杂的逻辑，过大的增大core的数量并不能很好的提高throughput

- GPU:
  - 适合简单单一的大规模运算。比如说科学计算，图像处理，深度学习等等
  - 优化方向在于提高throughput
  - 相关的技术有，multi-threading，warp schedular
  - 不同于CPU，GPU硬件上有复杂的warp schedular去实现多线程的multi-threading
  - 由于GPU经常处理大规模运算，所以在throughput很高的情况下，GPU内部的memory latency上带来的性能损失不是那么明显
    - 然而CPU和GPU间通信时所产生的memory latency需要重视

环境搭建

Goal: 理解CUDA, cuDNN, TensorRT的版本选择, 以及如何使用docker

本课程的环境配置


windows用户
(推荐使用windows terminal + wsl2)

编辑器推荐vscode或者neovim


mac用户
(推荐使用iterm2)

server推荐使用
ubuntu22.04 + docker

host端环境配置

安装iTerm2
安装Nerd Font

mac用户
(推荐使用iterm2)

```
[station]~/C/d/i/tensorrt-starter (main|✔) $ ll                                    (base)
drwxrwxr-x   - users 12  6月 13:14 1.0-build-environment
drwxrwxr-x   - users 11  6月 12:41 2.1-dim_and_index
drwxrwxr-x   - users 11  6月 13:42 2.2-cpp_cuda_interactive
drwxrwxr-x   - users 11  6月 14:29 2.3-matmul-basic
drwxrwxr-x   - users 11  6月 14:46 2.4-error-handler
drwxrwxr-x   - users 11  6月 15:00 2.5-device-info
drwxrwxr-x   - users 11  6月 15:31 2.6-matmul-shared-memory
drwxrwxr-x   - users 11  6月 17:10 2.7-bank-conflict
drwxrwxr-x   - users 11  6月 17:29 2.8-bilinear-interpolation
drwxrwxr-x   - users 12  6月 19:51 2.9-affine-transformation
drwxrwxr-x   - users 12  6月 19:51 2.10-stream-and-event
drwxrwxr-x   - users 10  5月 10:54 3.1-generate-onnx
drwxrwxr-x   - users 11  6月 18:40 3.2-export-onnx
drwxrwxr-x   - users 11  6月 11:33 3.3-trtexec
drwxrwxr-x   - users 11  6月 11:33 5.2-load-model
drwxrwxr-x   - users 11  6月 11:33 5.3-infer-model
drwxrwxr-x   - users 11  6月 18:39 5.3-multiple-outputs
drwxrwxr-x   - users 11  6月 23:29 5.4-print-structure
drwxrwxr-x   - users 11  6月 23:49 5.5-deploy-classification
drwxrwxr-x   - users 11  6月 11:33 5.6-deploy-classification-advanced
drwxrwxr-x   - users 12  6月 19:44 6.1-vision-transformer
drwxrwxr-x   - users 12  6月 19:44 6.2-cnn-vs-transformer
drwxrwxr-x   - users 12  6月 19:44 6.3-transformer-deploy-analysis
drwxrwxr-x   - users 12  6月 19:45 6.4-vision-transformer-optimization
drwxrwxr-x   - users 12  6月 19:36 7.1-deploy-yolo
drwxrwxr-x   - users 12  6月 19:37 7.2-deploy-yolo-advanced
drwxrwxr-x   - users 12  6月 19:39 7.3-yolo-model-analysis
drwxrwxr-x   - users 12  6月 19:41 7.4-halide-acceleration
drwxrwxr-x   - users 12  6月 19:42 7.5-multi-task-and-multi-thread
drwxrwxr-x   - users 12  6月 19:46 8.1-OSS_BEVFusion
drwxrwxr-x   - users 12  6月 19:47 8.2-OSS_tensorrtx
drwxrwxr-x   - users 12  6月 19:47 8.3-OSS_tensorRT_Pro
drwxrwxr-x   - users 12  6月 19:50 8.4-OSS_lightNet-TRT
drwxrwxr-x   - users  5  6月 13:15 archive
drwxrwxr-x   - users  3  6月 07:50 config
.rw-rw-r-- 62 users  3  6月 04:11 README.md
[station]~/C/d/i/tensorrt-starter (main|✔) $                                       (base)
```

server端环境配置(without docker)

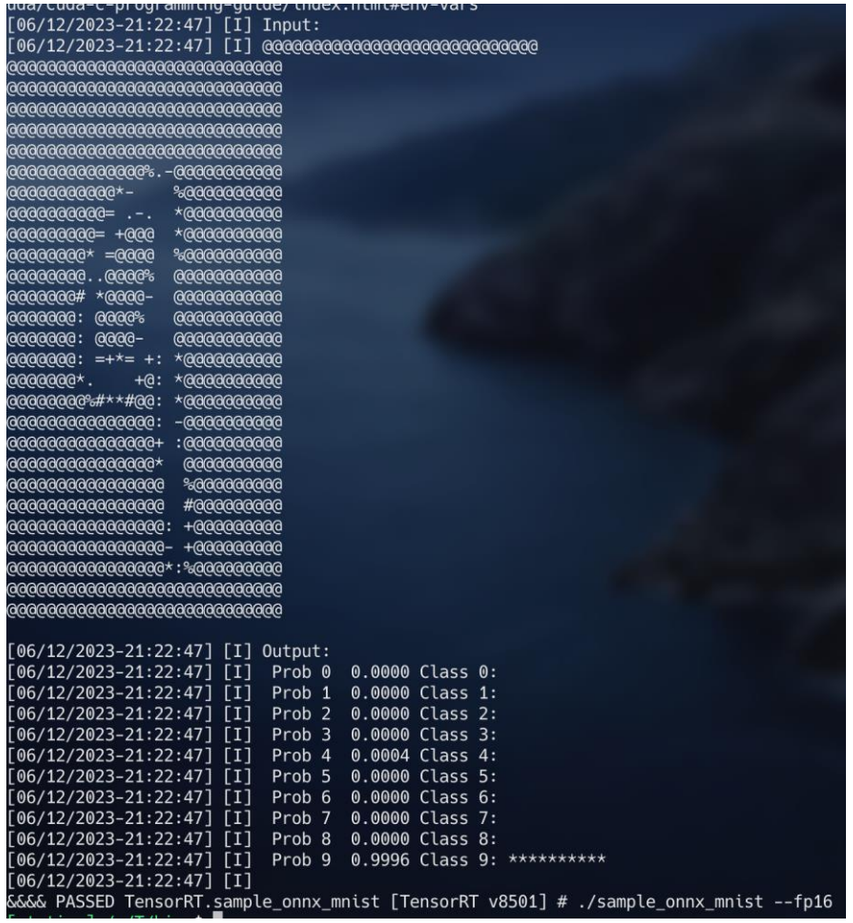
server推荐使用ubuntu22.04

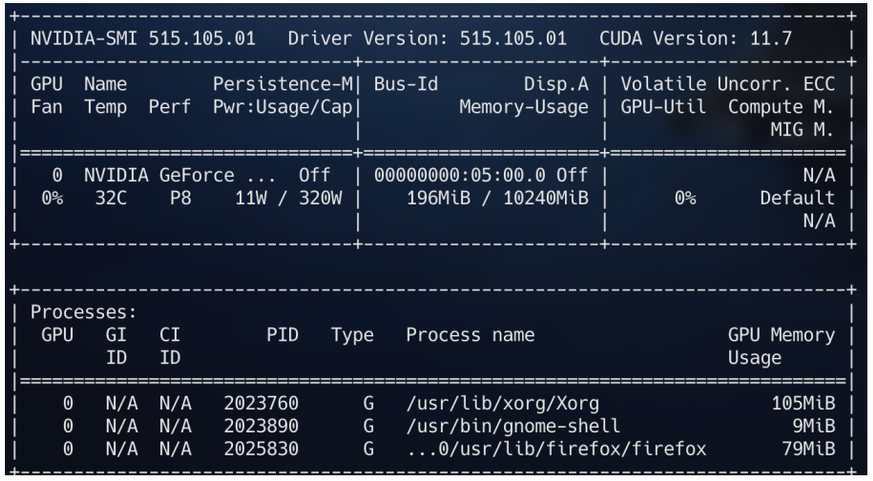
整体的流程(TensorRT, CUDA, cuDNN篇)

1. 配置openssh-server

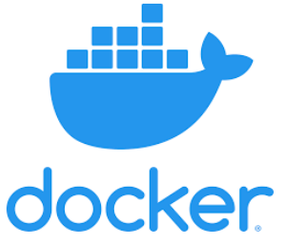
2. 检查是否有GPU
3. 查看linux系统的版本
4. 根据需求寻找TensorRT版本
5. 从TensorRT官方的release note寻找对应的CUDA和cuDNN版本
6. 安装CUDA (注意，这里建议从官方提供的脚本安装，不要apt-get)
7. 安装cuDNN (看官方的文档)
8. 配置TensorRT的路径

9. 参考opencv官方文档安装opencv4.5.2
10. 安装其他便于开发的软件包 (P.8)





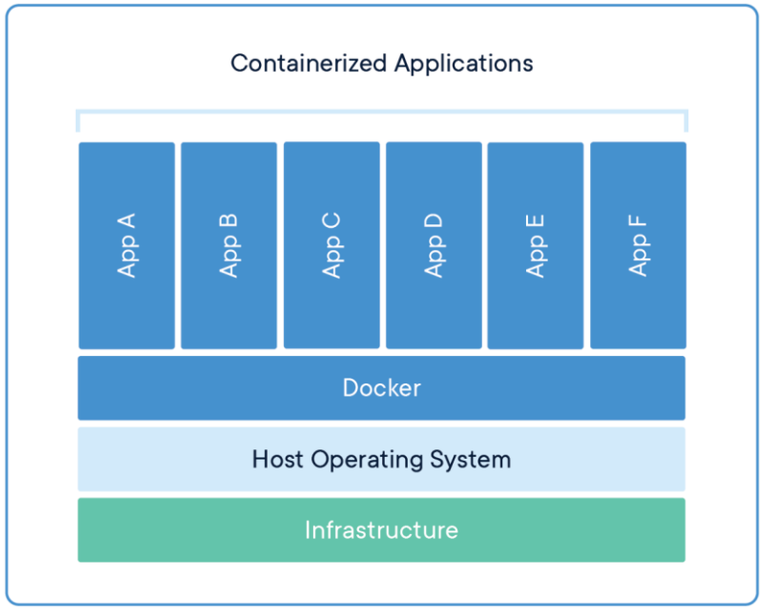
server端环境配置(with docker)



整体的流程(dockerfile篇)

1. 根据官方文档安装NVIDIA Container toolkit
2. 启动一个container查看nvidia-smi的运行状况
3. 从NVIDIA NGC中的官方release note寻找对应版本
4. 根据需求自行创建dockerfile
    1. 配置opencv (P.6)
    2. 配置openssh-server (P.6)
    3. 创建用户组
    4. 配置工作目录
    5. 安装其他便于开发的软件包 (P.8)

```
=> [ 2/21] RUN ln -snf /usr/share/zoneinfo/Asia/Tokyo /etc/localtime && ec
=> [ 3/21] RUN apt-get update
=> [ 4/21] RUN apt install -y --no-install-recommends   build-essential cm
=> [ 5/21] RUN apt-get -y clean && rm -rf /var/lib/apt/lists/*
=> [ 6/21] RUN mkdir /root/opencv_build
=> [ 7/21] WORKDIR /root/opencv_build/
=> [ 8/21] RUN git clone https://github.com/opencv/opencv.git && git clone
=> [ 9/21] WORKDIR /root/opencv_build/opencv/build
=> [10/21] RUN cmake -D OPENCV_GENERATE_PKGCONFIG=ON   -D OPENCV_EXTRA_MOD
=> [11/21] RUN make -j16 && make install
=> [12/21] WORKDIR /root
=> [13/21] RUN rm -rf opencv_build
=> [14/21] RUN apt-get update && apt install -y --no-install-recommends
=> [15/21] RUN apt-get -y clean && rm -rf /var/lib/apt/lists/*
=> [16/21] RUN useradd -rm -c trt -d /home/trt -s /bin/bash -g 100 -G sudo
=> [17/21] RUN echo "trt   ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers
=> [18/21] RUN apt-get update
=> [19/21] RUN mkdir -p /home/trt/workspace/
=> [20/21] RUN chown -R trt:users /home/trt
=> [21/21] WORKDIR /home/trt
=> exporting to image
=> => exporting layers
=> => writing image sha256:2fa25e743e1c67b689a472c6d5af9cb272b3d2380963fd4
=> => naming to docker.io/library/trt_webinar:cuda11.4-cudnn8-tensorrt8.2-
```



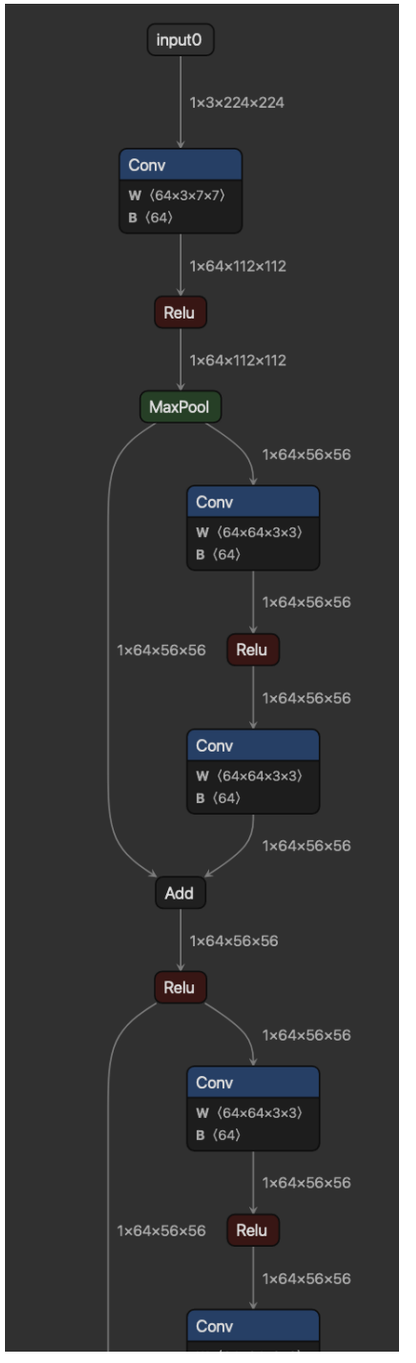
server端环境配置(with/without docker, optional)

NETRON

整体的流程(提高开发效率篇)

1. 安装fish shell
2. 安装fisher进行fish包的管理
3. 使用z进行目录的高速跳转
4. 安装exa
5. 使用peco快速寻找history以及git版本
6. ssh远程linux是转发X11使用GUI
7. 安装图片浏览器从终端打开图片
8. 安装tmux进行多window多session的管理
9. miniconda的安装进行python环境的隔离
10. 安装netron进行DNN网络架构图的分析

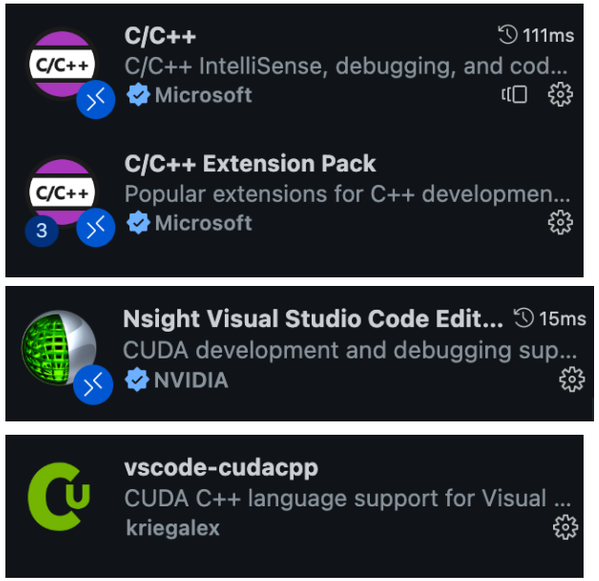
```
netron my.onnx --port 8888
netron demo.change.onnx --port 8888
netron demo.onnx --port 8888
netron yolov5s.pt
netron workspace/yolov5s-raw.onnx
netron workspace/yolov5s.onnx --port 8081
netron demo
netron workspace/demo.onnx
netron mnist.onnx
netron example.
netron example_two_head.onnx
netron
netron --port 8080 --host 192.168.11.11 centerpoint.scn.onnx
netron --port 8081 --host 192.168.11.11 centerpoint.scn.onnx
netron --port 8081 lidar.backbone.xyz.onnx
netron --port 8081 --host 192.168.11.11 lidar.backbone.xyz.onnx
netron resnet18_qat.onnx
netron resnet50.onnx
netron example.onnx
netron sample_qat.onnx
netron ../models/vgg16.onnx
netron ../models/sample2.onnx
netron swinb.onnx
netron ../models/sample.onnx
netron models/resnet50.onnx
netron ../models/resnet50.onnx
netron ../models/resnet18.onnx
netron vgg16.onnx
netron mobilenetV2.onnx
netron squeezenet1_0.onnx
netron densenet161.onnx
netron shufflenetV2
netron shufflenetV2.onnx
netron --port 8081 --host 192.168.11.2 fuser.onnx
netron --port 8082 --host 192.168.11.2 fuser.onnx
netron --port 8082 --host 192.168.11.2 camera.backbone.onnx
netron --port 8081 --host 192.168.11.2 camera.backbone.onnx
netron --port 8082 --host 192.168.11.2 lidar.backbone.xyz.onnx
netron --port 8082 --host 192.168.11.2 example_two_head.onnx
netron --port 8082 --host 192.168.11.2 resnet18.onnx
pip install netron
netron sample.onnx
```

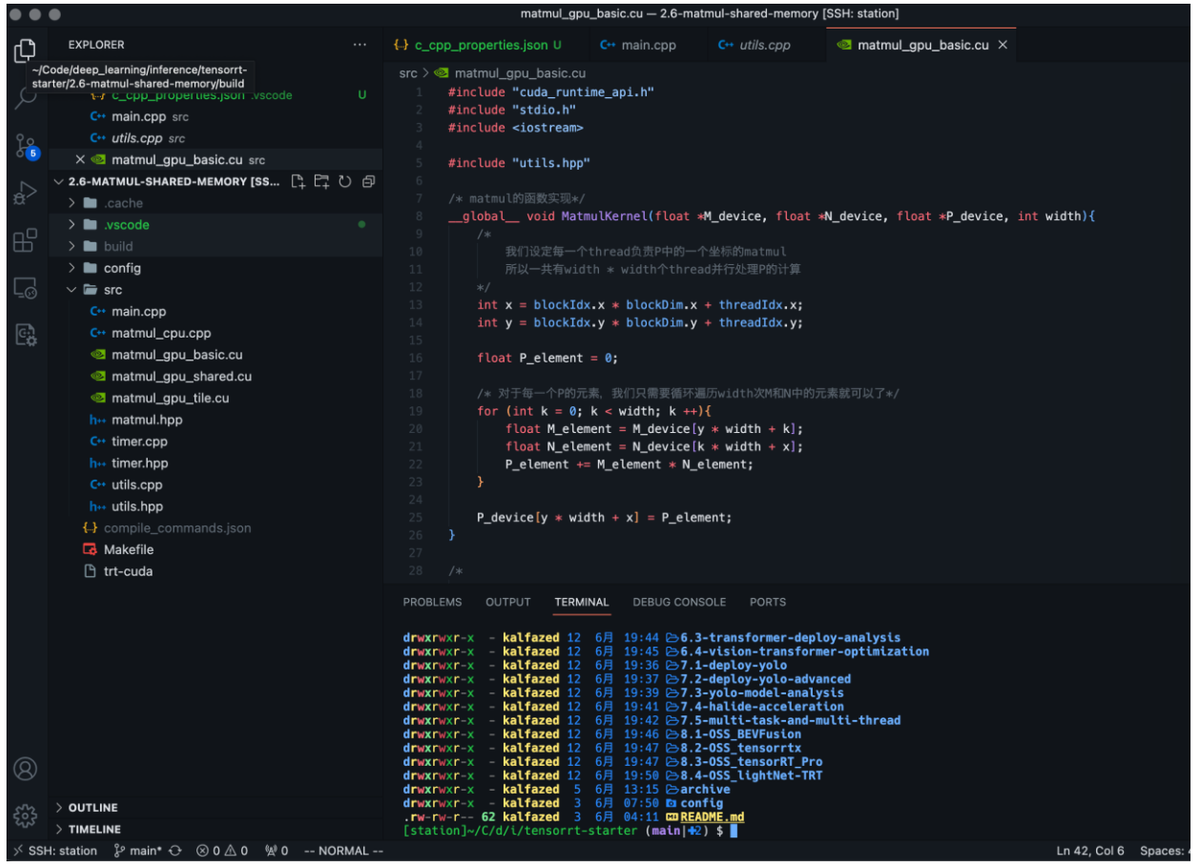


编辑器环境配置(windows/mac/linux host端)

整体的流程(VS Code编辑器篇)

1. 创建compile_commands.json
2. 安装必要的插件
3. 设置c_cpp_properties.json
4. 设置tasks.json
5. 设置launch.json
6. 设置setting.json





编辑器环境配置(linux server端)

neovim

整体的流程(neovim编辑器篇, optional)

1. neovim的安装
2. lunarvim编辑器的安装
3. 使用lsp signature进行neovim补全信息的提示
4. 设置lsp server进行编程语言的自动补全
5. 安装trouble.nvim进行代码的diagnostics
6. 安装markdown preview进行远程实时编辑markdown
7. 开启端口允许远程访问markdown preview
8. 通过DAP在neovim中进行C/C++/Python的Debug
9. 扩展neovim的DAP进行GUI上的Debug操作
10. 可以参考的C/C++的DAP环境配置
11. 设置lsp diagnostics的error/warning信息的提示方式
12. 对python lsp server进行python错误显示的级别设定
13. 设置.clangd进行C/C++, CUDA代码实现/声明/使用跳转