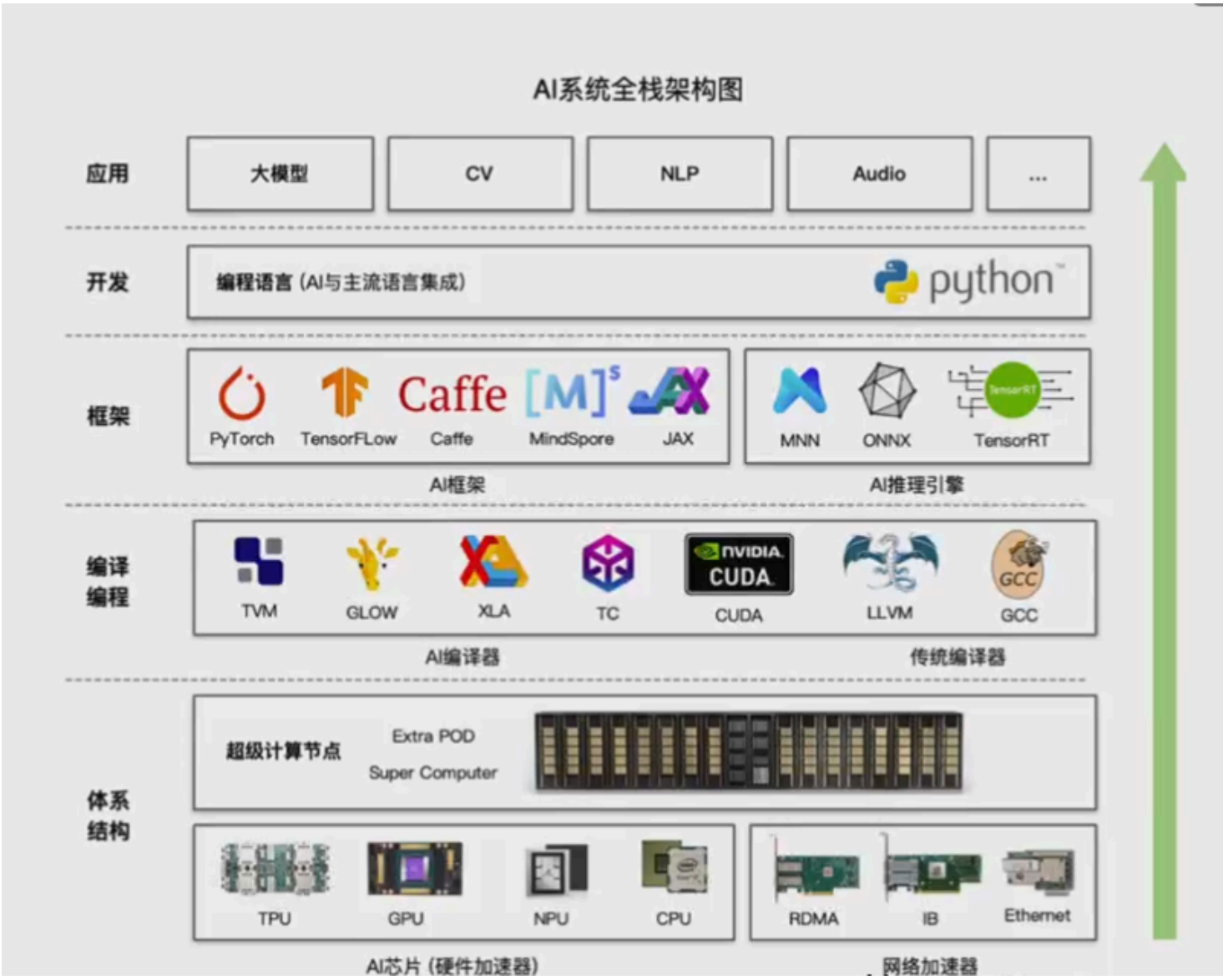
AISystem复习——PPT

总论

AISystem全栈架构图：



GPU：Nvidia，摩尔线程 ...

TOPIC I 深度神经网络与计算框架基础

神经网络基础

深度学习方法基本流程：

1. 定义带参数的函数（神经网络）
2. 定义优化目标（损失函数）
3. 计算梯度并更新参数（梯度下降训练）

$$\begin{aligned} \hat{y}_i &= f(w, x_i) \\ \min_w L(w) &= Loss(\hat{y}_i, y_i) \\ w_j &\leftarrow w_i - \eta \nabla_{w_i} L(w) \end{aligned} \tag{1}$$

神经网络的基本理论在深度学习前已基本奠定

三次浪潮，两次寒冬

2010年后：三阶段（弱人工智能，通用人工智能（具有部分逻辑推理，决策能力），强人工智能）

人工智能成功发展原因：

1. 数据：大量，高质量数据集
2. 算法：更好的激活函数和结构，更复杂网络，更大的参数，更好的训练方法。
3. 框架&算力：体系结构与框架优化，硬件发展。

AISystem=算法+算力+框架+数据

AI框架试图回答的问题：

1. 前端（面向用户）：如何灵活表达?
2. 算子（执行计算）：如何保证算子计算性能（算子级优化）
3. 求导（更新参数）：如何实现后向（求导运算）
4. 后端（系统相关）：如何将同一个算子泡在不同设备上？（GPU）
5. 运行：如何优化和调度一个模型的计算？（整体模型级计算优化）

计算方法的两个极端：flexibility <==> efficiency

AI框架目的

提供灵活的编程模型和编程接口（Flexibility）

提供高效和可拓展的计算能力（Efficiency）

AI框架的重要性相当于操作系统

AI框架发展进程

before 2010（2015）：第一代框架（**Caffe**，Meta）

1. 主要解决问题：基于CNN的模型，**Layer-Based**
2. 特点：Layer-Based——提供每一个Layer的前向和梯度（反向）计算实现；支持多设备加速
3. 优点：提供一定程度的可编程性，支持GPU加速。
4. 缺点：

1. 灵活性受限：
    1. 新型的Layers的出现要求对于每一个新的Layer都要重新实现前向和反向，工作量极大
    2. 新的优化器要求对梯度和参数进行过呢个通用复杂的运算。
2. 基于简单的“前向+后向”的训练模式难以满足新的模式（训练模式创新无法支持）
    1. 实例：RNN，GAN，RL ...

after 2015：第二代框架（pytorch，tensorflow，mxnet）

1. 兼顾变成的灵活性和计算有效性
2. **基于数据流图的（DAG）的计算框架**（见下方）
3. 自动求导（AD）

基于数据流图的（DAG）的计算框架

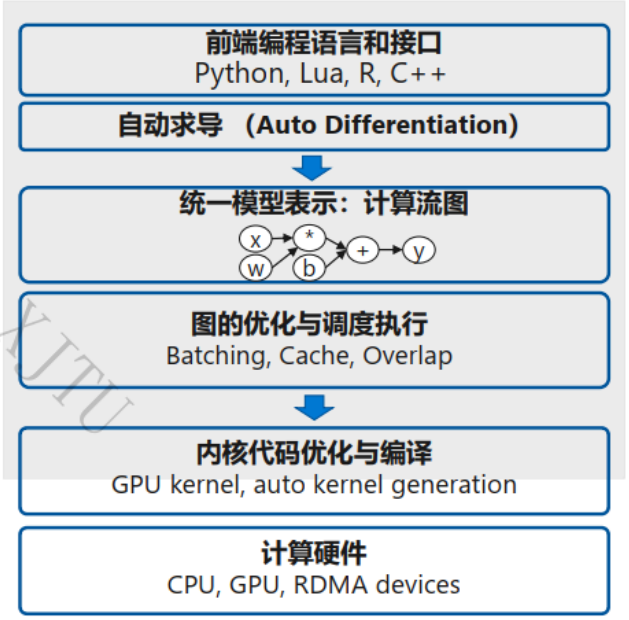
1. 基本数据结构：Tensor（N维数组/张量）
    1. 例：int([2,3,4])
2. 基本运算单元：Operator
    1. 由最基本代数算子组成
    2. 每个Operator接受N个输入Tensor，输出M个输出Tensor
    3. 例：Conv, MatMul, BN, Relu, Loss, Transpose ...
3. 用数据流图表示计算逻辑和状态：
    1. 节点表示Operator，边表示Tensor。
    2. 计算状态也是Operator（如定义变量Variable Operator）
    3. 特殊Operator：Switch，While等构建控制流。
    4. 特殊边：控制边用来表示节点之间的依赖关系
4. 数据流图类型：
    1. 静态数据流图——TensorFlow
        1. **先定义后执行**
        2. 优点：可以进行全局图优化，计算效率高；内存使用效率高。
        3. 缺点：不能实时得到计算结果，难以调试；受限于算子集合，对带有控制流（如GAN）不友好。
        4. 类似于C。
    2. 动态数据流图——PyTorch
        1. **边定义边运行**
        2. 优点：代码简洁，可实时得到计算结果；灵活的可编程性和可调试性。
        3. 缺点：无法进行全局化简或优化，无法精确内存分配管理。

    4. 类似于Python
5. 自动反向求导
    1. 问题：
        1. 符号求导：参数量过大，有的无法求梯度
        2. 求极限数值求导：不精确；不适用于无法求导的算子
        3. 链式法则：需要保存大量中间计算结果。
    2. 计算流图求导：
    3. 方便全局图优化，节省内存。
    4. 现代框架提供常见函数自动求导系统；同时提供接口，让用户自己提供反向函数。
6. 图优化
    1. 优化Pass去化简计算流图或提高执行效率。
    2. 常用手段：表达式化简，公共子表达式消除，常熟传播，Operator Batch，表达式替换，算子融合等。
7. 调度与执行&并发：根据依赖关系，依次调度运行代码。找到相互独立的算子进行并发调度，提高计算的并行性。
8. 划分与设备放置：
    1. 显式图划分：`with tf.device('/gpu:0')`
    2. 跨设备的边被自动替换成一组Send/Recv operators
9. 内核与多硬件支持
    1. **内核（Kernel）是定义了一个算子在某种具体设备的计算实现。**

小结：

- **模型表示**：数据流图
- **前端语言**：用来构建数据流图
- **自动求导**：基于backpropagation的原理自动构建求导数据流图

- **图的优化**: 图化简
- **调试执行**: 并发调度
- **设备放置**: 计算图切分
- **算子内核**: 模块化支持多设备



计算框架的发展趋势

1. 基于编译器的算子优化
2. 动态图和静态图的统一
3. 超大规模网络训练范式

TOPIC II Pytorch框架基础与实践

神经网络模型搭建

Lenet

```
class Lenet(nn.module):
    def __init__(self):
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
```

```
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```

训练实例：

```
def train(model, trainloader, device, optimizer, criterion):
    for batch_idx, (inputs, targets) in enumerate(trainloader):
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
    return model
```

torch.nn：Pytorch为神经网络设计的模块化接口，基于AutoGrad实现

Fully Connected NN：全连接网络，参数量极大，难以训练

Locally Connected NN：局部性链接网络，局部性特征链接，参数两显著缩小。

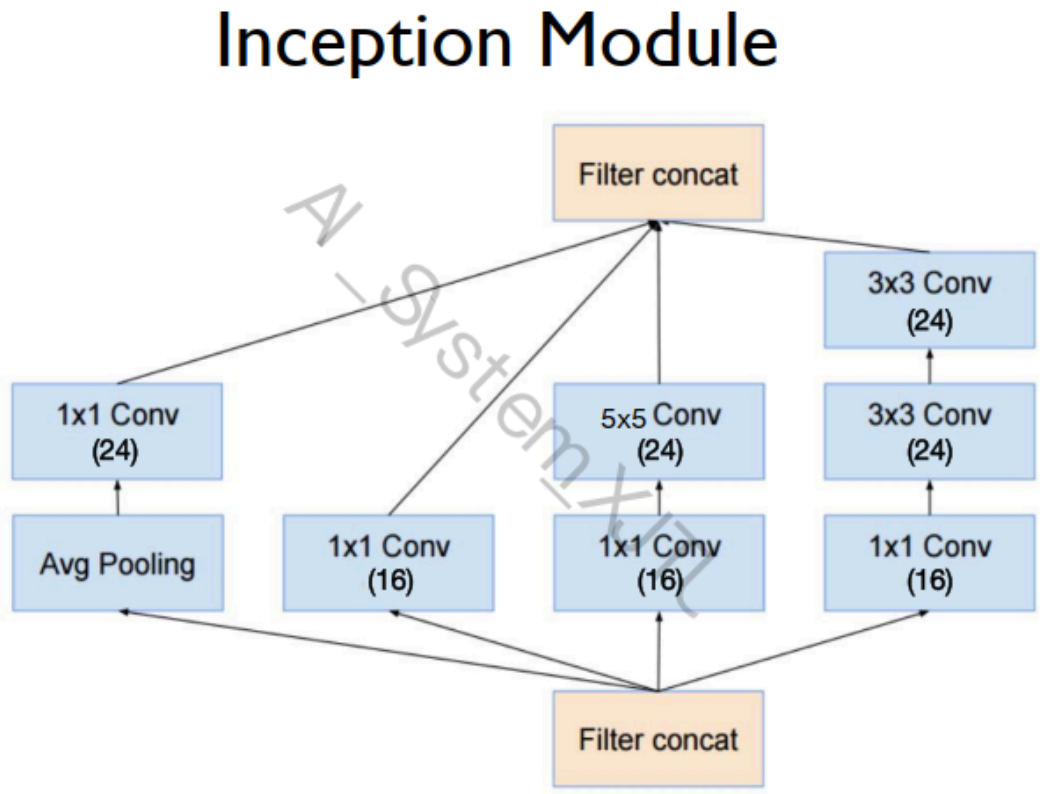
Pytorch语法说明

```
import torch
import torch.nn as nn
nn.Conv2d(in_channels = 3, out_channels = 8, kernel_size = (3, 3), stride = 2, padding = 1, bias = false, padding_mode = 'zeros')
# 输入张量[N, in_channels, H, W] -> 输出张量[N, out_channels, (H+2padding-kernel_size[0])/stride + 1, (W+2padding-kernel_size[1])/stride + 1]
# padding_mode: 如何用什么padding，默认为'zeros'
nn.MaxPool2d(kernel_size)
x.view(-1, 8)
# 表示将第1维度设为8，第0维度自动计算。
x = x.view(-1, self.num_flat_features(x))
# self.num_flat_features表示x的特征总数，需要在net中自己实现。


```

nn.module封装网络，Layer定义在`__init__(self)`里。

1*1卷积核作用：调整通道数

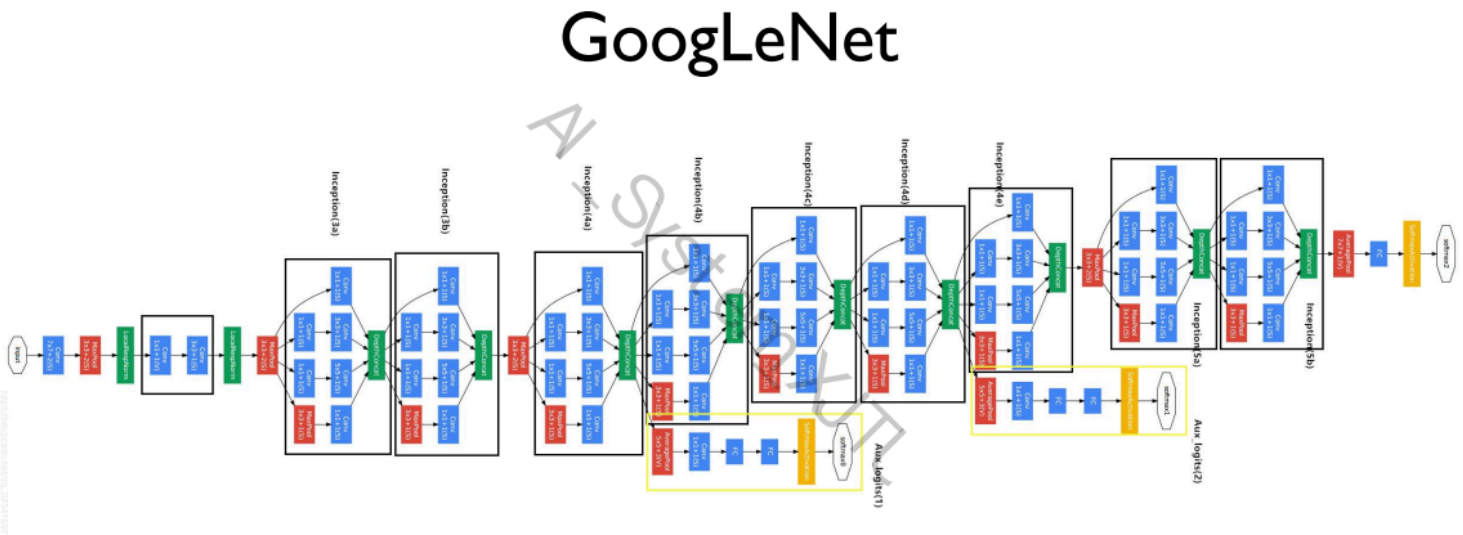
GoogleNet



Filter concat: 对不同卷积层做出来的通道数进行通道维度的concat。

Inception块实现了用不同的大小的kernel对图像特征进行提取，大kernel感受野大，但是对细节掌握较少；小kernel感受野小，但细节信息很多的。

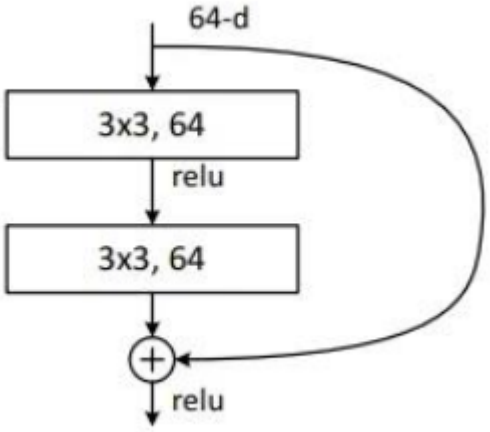
GoogleNet首次引入块概念，实现进一步模块化网络设计。



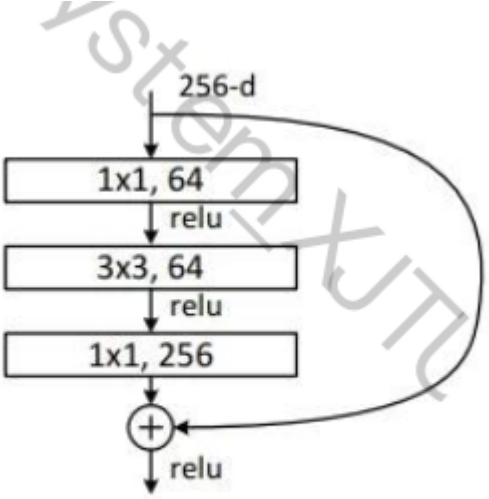
Resnet

Resnet首次引入残差，将`x+G(x)`进行前向传播，极大程度解决了深层网络梯度爆炸/梯度消失的问题
Resnet残差块分为两种模式：

1. BasicBlock（Resnet18&Resnet34）：



2. Bottleneck(Resnet50 or deeper)



Resnet搭建：循环调用残差块，构造残差模块

torchvision Resnet源码中_make_layer函数如下：

```
def _make_layer(
        self,
        block: Type[Union[BasicBlock, Bottleneck]],
        planes: int,
        blocks: int,
        stride: int = 1,
        dilate: bool = False,
    ) -> nn.Sequential:
```

```
        norm_layer = self._norm_layer
        downsample = None
        previous_dilation = self.dilation
        if dilate:
            self.dilation *= stride
            stride = 1
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                conv1x1(self.inplanes, planes * block.expansion, stride),
                norm_layer(planes * block.expansion),
            )

        layers = []
        layers.append(
            block(
                self.inplanes, planes, stride, downsample, self.groups,
self.base_width, previous_dilation, norm_layer
            )
        )
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(
                block(
                    self.inplanes,
                    planes,
                    groups=self.groups,
                    base_width=self.base_width,
                    dilation=self.dilation,
                    norm_layer=norm_layer,
                )
            )

        return nn.Sequential(*layers)
```

其中，for循环实现残差块循环调用，组成残差模块。planes表示输出维度，inplanes表示输入维度。判断逻辑为：若stride！=1，或者输入通道数和输出通道数*block增加倍数（比如bottleneck就是4）不同，则需要下采样模块实现通道数对齐。

layers是一个四元素列表，表示每个残差模块中block的数量，用于区分Resnet-x

定制数据集

torchvision实现了常用的数据加载功能。

使用自带数据集——CIFAR10为例

```
transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ]
)
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)

testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse','ship', 'truck')
```

这里的transform共进行两步操作：将numpy转torch（numpy是H*W*C的，torch是C*H*W）；第二部将torchvision输出的数据集图像（[0,1]范围内的PIL图像）转化为[-1,1]的Tensor数据。可以如下理解

$$X_{new} = a + (b-a)\frac{X - Min}{Max - Min} = \frac{X - Mean}{Std} \quad [a=-1, b=1] \tag{2}$$

使用自定义数据集

数据标注

图像标注可用labelImg .....

安装方式：

```
$ pip install labelimg
$ labelImg
```

定制Dataset

torch.utils.data.Dataset是代表一个数据集的抽象类，要定制自己数据集，需要继承Dataset类并重写下面三个函数：

1. `__len__`:返回dataset的大小
2. `__getitem__`用来支持dataset索引，即dataset[i]可以用来得到第i个样本，用于优化内存，在需要的时候再读取图像。
3. `__init__`：指定csv文件及图像目录
4. 使用可选的transform操作，对样本进行必要的预处理操作。

定制Transforms

用来对样本进行预处理（图像样本大小不一致）

常用预处理操作：

1. Rescale：缩放
2. RandomCrop，随机裁剪，用以数据增强，防止过拟合！
3. ToTensor：Numpy2Torch

为了方便，可以重写成可调用的类，这样就不必每次调用时都要传递参数了。为实现目的，需要重写在transform.compose中，对应类的`__call__`, `__init__`方法。

`__call__`方法的输入参数是sample，即再dataset中`__getitem__`的sample的格式

注意：这里的transform和dataset定义是配套的！

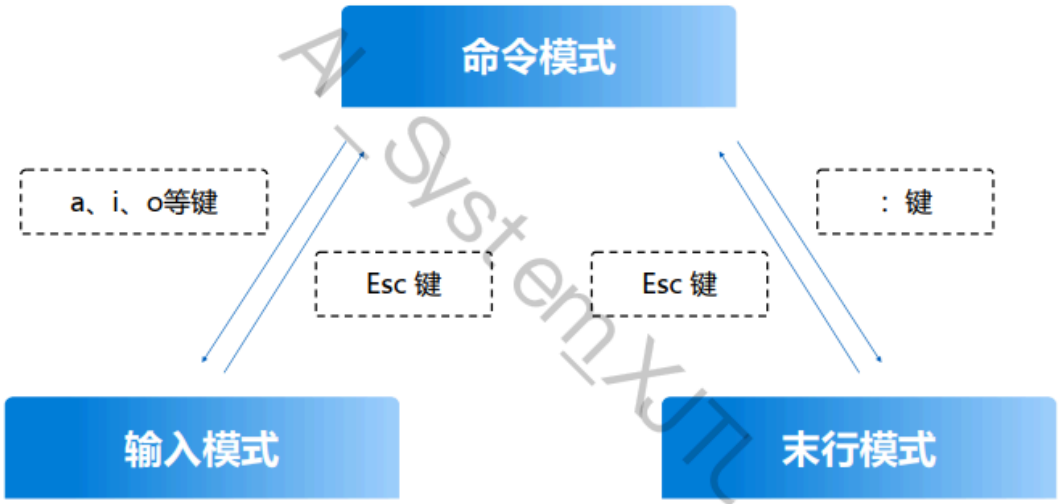
使用服务器

Why Linux？——开源，对开发友好

在Linux系统中，一切都是文件

Vim——默认安装在所有linux系统上！

Vim的模式与模式切换



命令模式常用命令：（感觉不大重要）

命令	作用
dd	删除（剪切）光标所在整行
5dd	删除（剪切）从光标处开始的5行
yy	复制光标所在整行
5yy	复制从光标处开始的5行
n	显示搜索命令定位到的下一个字符串
N	显示搜索命令定位到的上一个字符串
u	撤销上一步的操作
p	将之前删除（dd）或复制（yy）过的数据粘贴到光标后面

末行模式命令（感觉记住:w :q）

命令	作用
:w	保存
:q	退出
:q!	强制退出（放弃对文档的修改内容）
:wq!	强制保存退出
:set nu	显示行号
:set nonu	不显示行号
:命令	执行该命令
:整数	跳转到该行
:s/one/two	将当前光标所在行的第一个one替换成two
:s/one/two/g	将当前光标所在行的所有one替换成two
:%s/one/two/g	将全文中的所有one替换成two
?字符串	在文本中从下至上搜索该字符串
/字符串	在文本中从上至下搜索该字符串

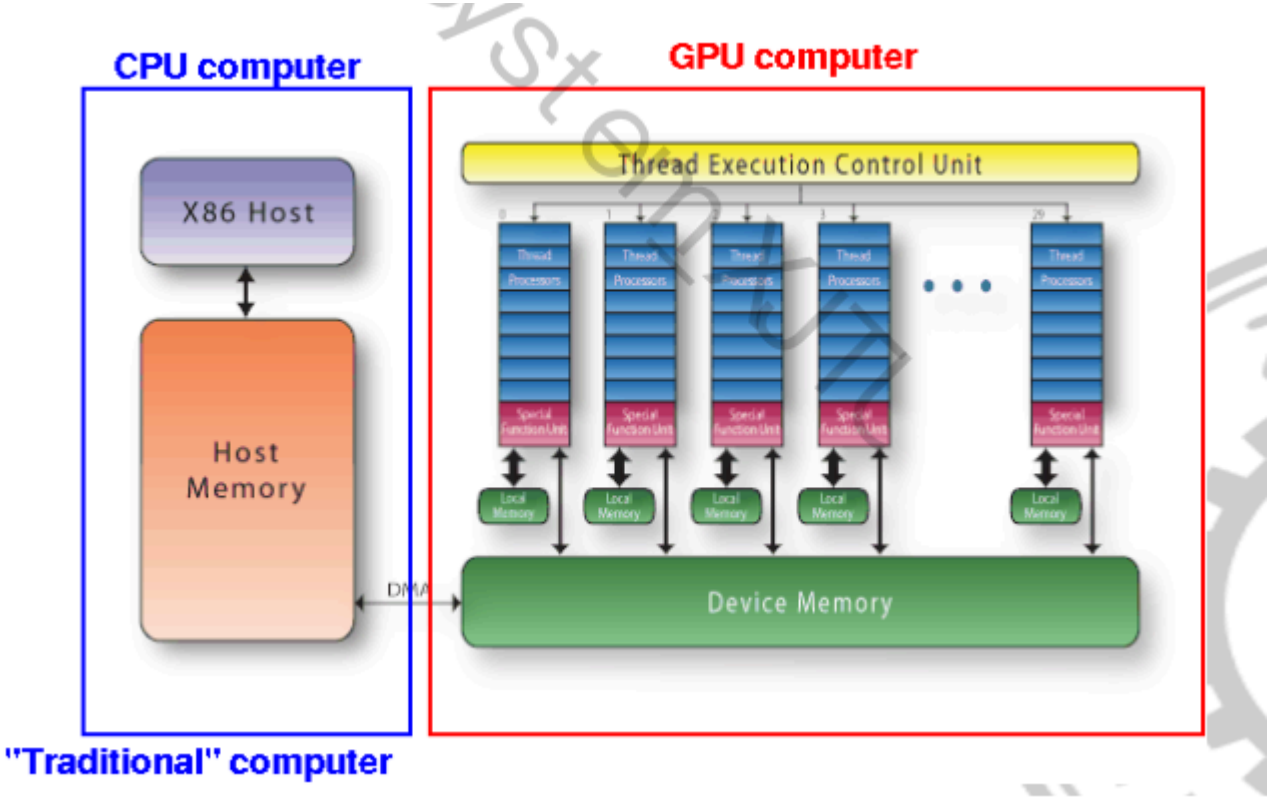
GPU环境配置

Nvidia Driver：对显卡进行监控的必要软件，示例安装方式如下（以410 Turing为例）

```
$ sudo apt-get purge nvidia
$ sudo add-apt-repository ppa:graphics drivers/ppa
$ sudo apt-get update
$ sudo apt-get install nvidia-driver-410
```

安装成功—» nvidia-smi

CUDA：Nvidia推出的通用并行计算架构，使GPU能解决复杂计算问题！他包含了CUDA指令集架构（ISA）以及GPU内部并行计算引擎。可用cpp为CUDA架构编写程序，其程序可在支持CUDA的处理器上超高性能运行



CUDA安装：deb安装。

在~/.bashrc中加入环境变量（以cuda10.0为例）：

```
export PATH = /usr/local/cuda-10.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH = .usr.local.cuda-
10.0/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

验证：nvcc -v

cuDNN：用于深度神经网络的GPU加速库

安装：登录nvidia，找到对应CUDA版本的cudnn，下载后可用tar文件解压并安装，也可直接sudo dkpg -i <三个deb文件>，在/usr/include可以看到cudnn文件。

使用tar安装方式如下：解压后，在解压文件目录下：

```
$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn*/usr/local/cuda/lib64
$ sudo chmod a+R /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

Conda

Anaconda作用：构建虚拟环境，互相隔离，互不冲突。

常用命令：

```
$ conda init #写入.bashrc
$ conda -V #
$ conda info --envs #查看所有可用环境
$ conda env list# 这个也可以查环境
$ conda create -n name python=3.10 #写入.bashrc
$ conda activate name #写入.bashrc
```

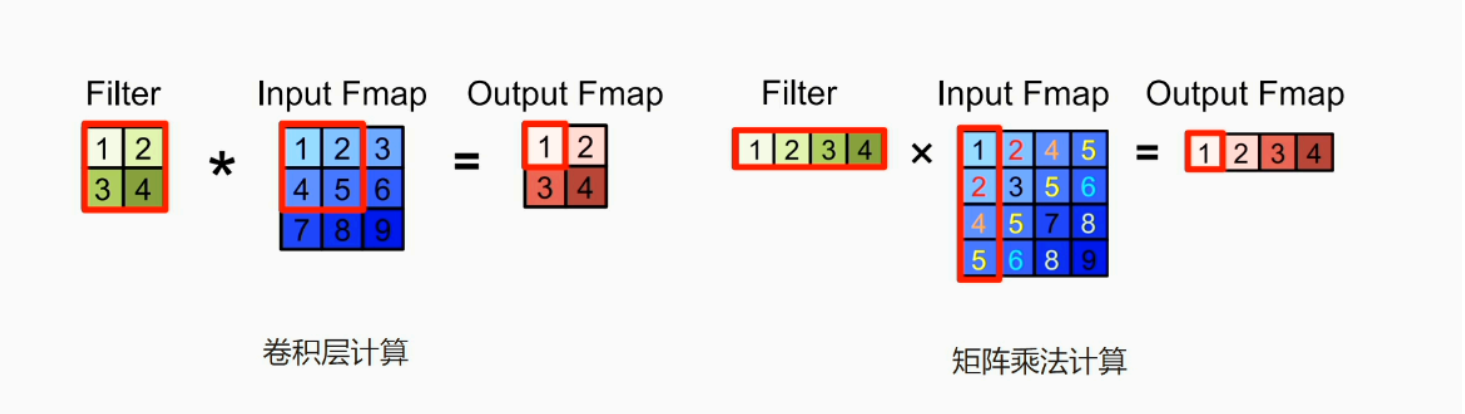
训练过程可视化——TensorBoard

见实验部分

Computer Architecture for Matrix computation

常见层都可以映射到矩阵运算

卷积层：



优点：体系结构对矩阵计算的优化；缺点：数据有重复/冗余。

主流网络结构的计算均可表擦成矩阵运算

Why？1. 矩阵计算表达出好的计算并行性；2. 有成熟的矩阵计算加速硬件和软件库

CPU

体系结构

核心由**复杂的控制单元**和**少量计算单元**组成

特点：计算密度较低——ALU较少；控制逻辑复杂——较大的控制单元。

主要面向顺序指令执行

1. 成熟的调度技术：如分支预测，推理执行，乱序执行等
2. 擅长处理单线程，控制密集型的计算任务。
3. 缺点：低计算吞吐。

内存架构

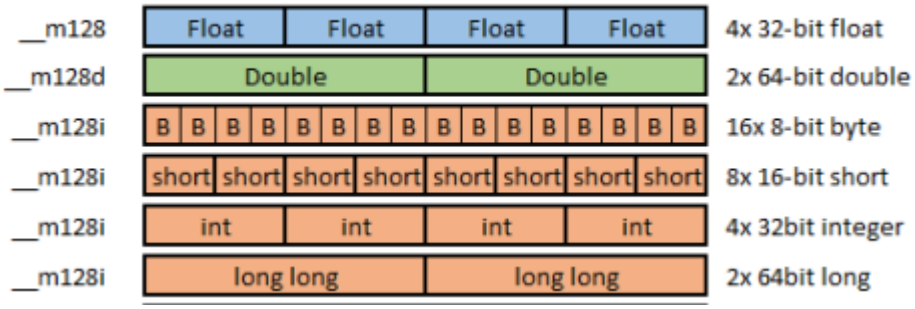
较深的内存架构——多层cache来隐藏访存延时；调度上采用**访存预取，预取预测**机制。

CPU性能增长方向

增加更多的计算核+给单核增加向量化功能

CPU慢的主要原因：内存（memory）的存取太慢

解决方案：SIMD（单指令多数据流，一条指令同时处理一组多个数据）



并行处理硬件架构

SISD（单核CPU）：无法并行计算，一次译码一条指令，一次只提供一份数据，只处理一个数据流。

SIMD：一个控制器控制多个处理器，同时对一组数据中的每个分别执行相同操作。主要执行向量，矩阵等数组运算，适用于科学计算。处理单元数量多，但受到通讯带宽传递速率的限制。

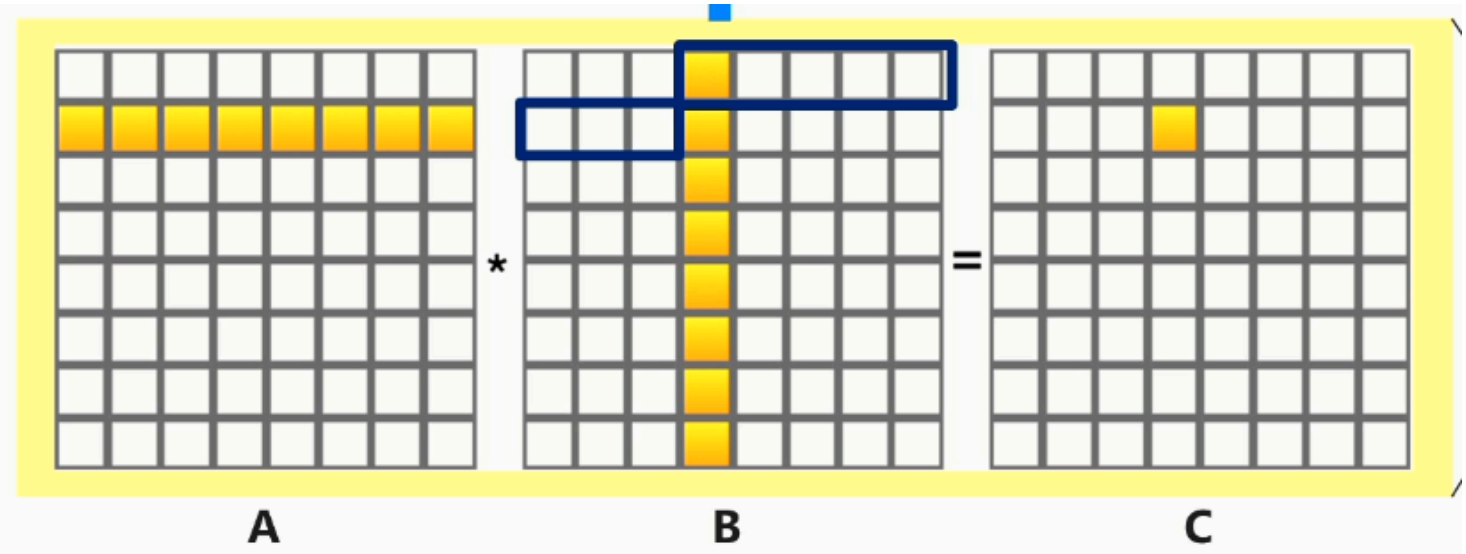
MISD：多个指令处理单个数据流。**理论模型，没有实际应用。**

MIMD（多核CPU）

		Data stream	
		Single	Multiple
Instruction stream	Single	SISD $a_1 + b_1$	SIMD $a_1 + b_1$ $a_2 + b_2$ $a_3 + b_3$
	Multiple	MISD $a_1 + b_1$ $a_1 - b_1$ $a_1 * b_1$	MIMD $a_1 + b_1$ $a_2 - b_2$ $a_3 * b_3$

矩阵计算案例

CPU如何进行矩阵乘法?



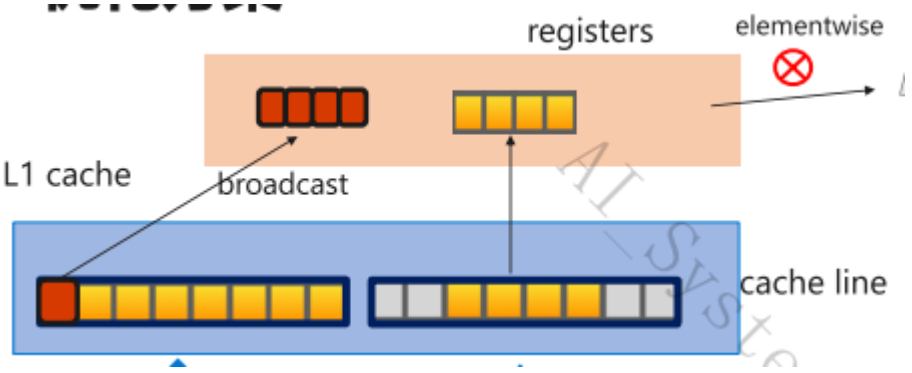
Method 1：三层FOR循环。

1. A的计算效率低：每次读一行，但用完就得重新读，耗时间。可使用向量化指令增加计算吞吐。
2. B的访存低效：cache line每次智能读一行，为了做一个矩阵运算，需要把整个B的每一行一次读进cache，很慢。可采取手段，增加cache利用效率。

Method 2：改进方案1——转置B

1. 理论上可以把B的访存从整个B变成某一行
2. 需要高效的转置，否则转置需要的内存时间消耗也很大。

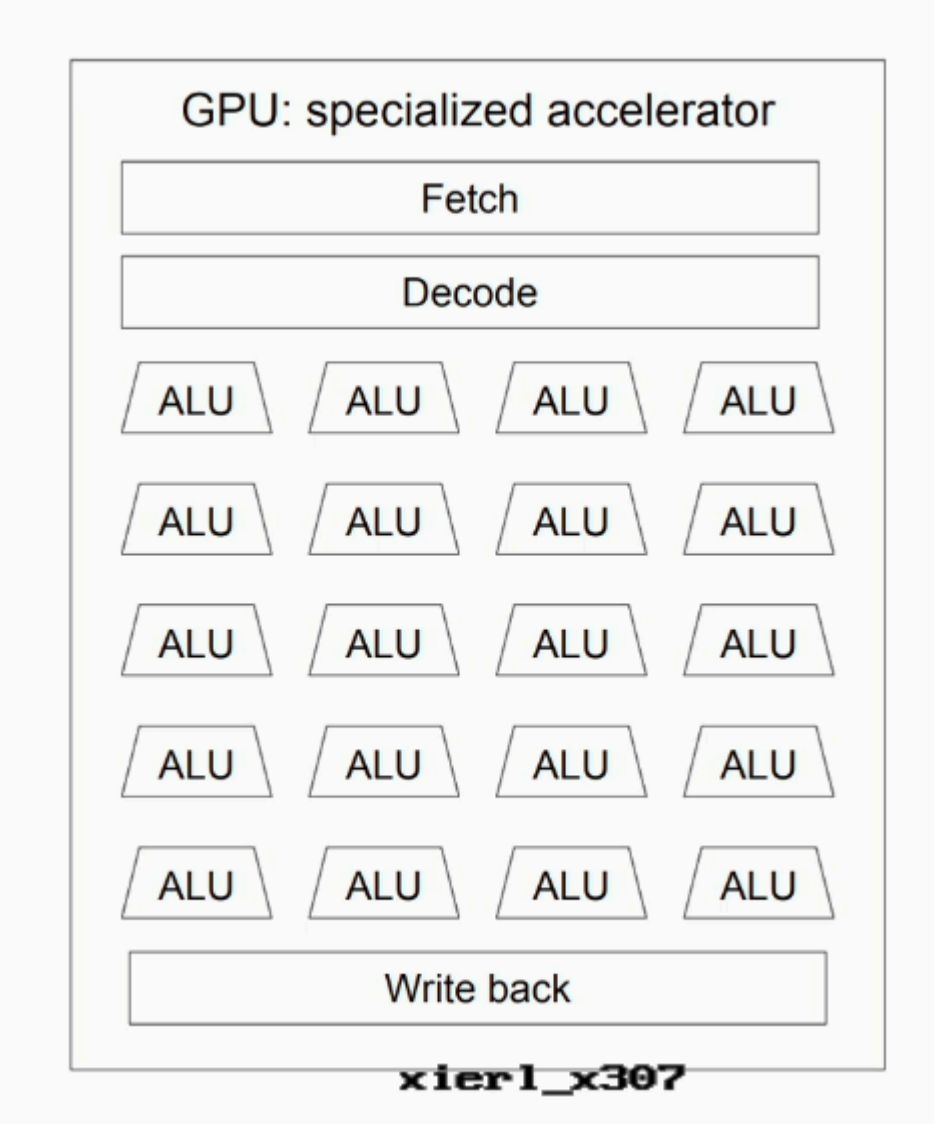
Method3：广播



把A cacheline的第一个读出来，广播成4个，和B的cacheline中的4个进行向量化计算。计算结果放进C，然后累加。

考虑到register的重用，往往需要将A和B的矩阵划分成合适大小的块，使最终访问性价比最高。

GPU



GPU与CPU比，多了大量的ALU。

由上千个简单core组成，每个core非常简单，没用控制能力，只进行简单计算。

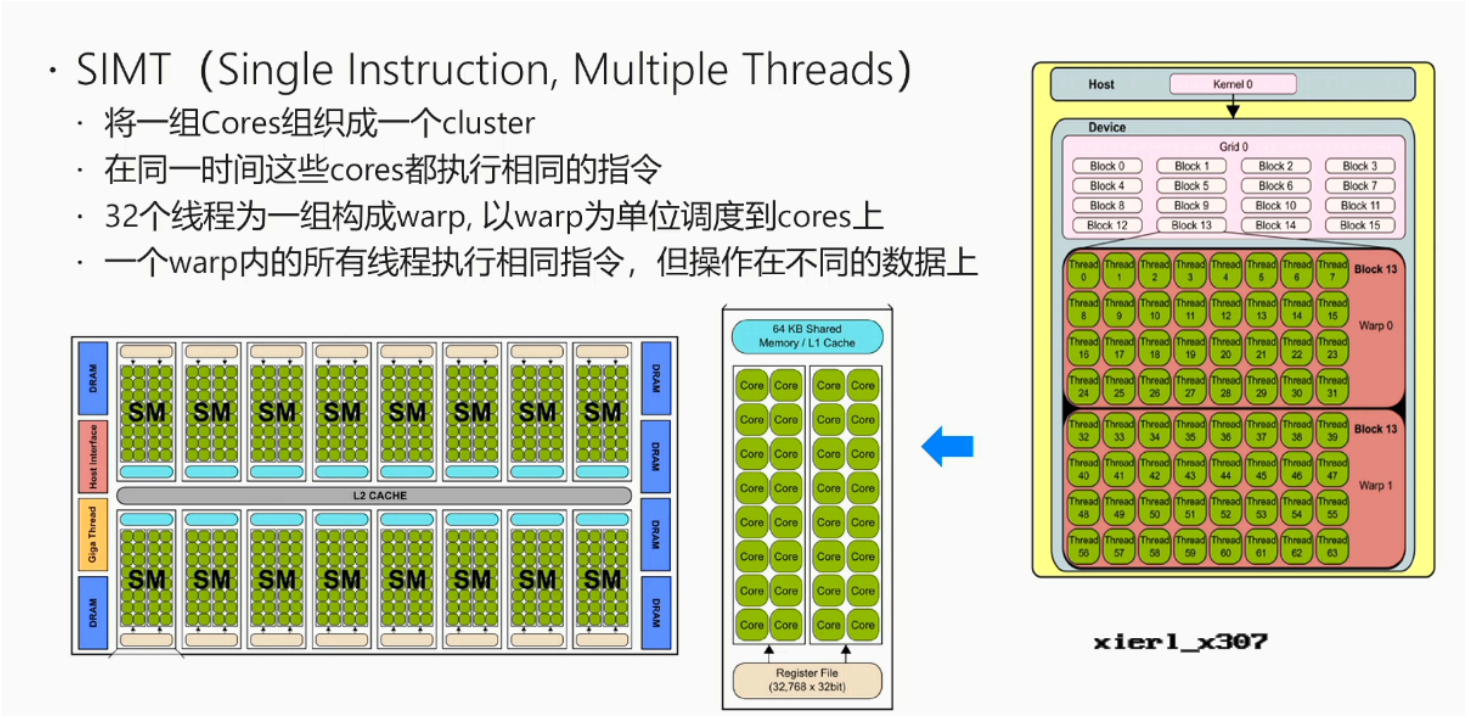
GPU有较高的计算密度，计算与访存比 较高，擅长高度并行的计算（如图像处理，矩阵计算），早期主要用于显示处理。

GPU缺点：不擅长控制逻辑复杂的程序。

GPU执行模型——SIMT

GPU架构解释：**硬件层面**：一个GPU里有多个SM（streaming multiple processor），每个SM包含多个streaming processor（core/SP）。**软件层面**：一个core对应一个thread，一个thread调用一个core执行任务。一组thread打包成block，多组block打包成一个grid，一个核函数kernel（任务）对应一个grid。

比如，对于有32个SP的SM，如果一个block有64个thread，则需要分成两个warp，以warp为单位，每个warp依次调度到core上进行运行。对于一个warp内的所有thread执行相同的指令，由此也就是对一个SM里的SP执行相同的指令。



GPU编程

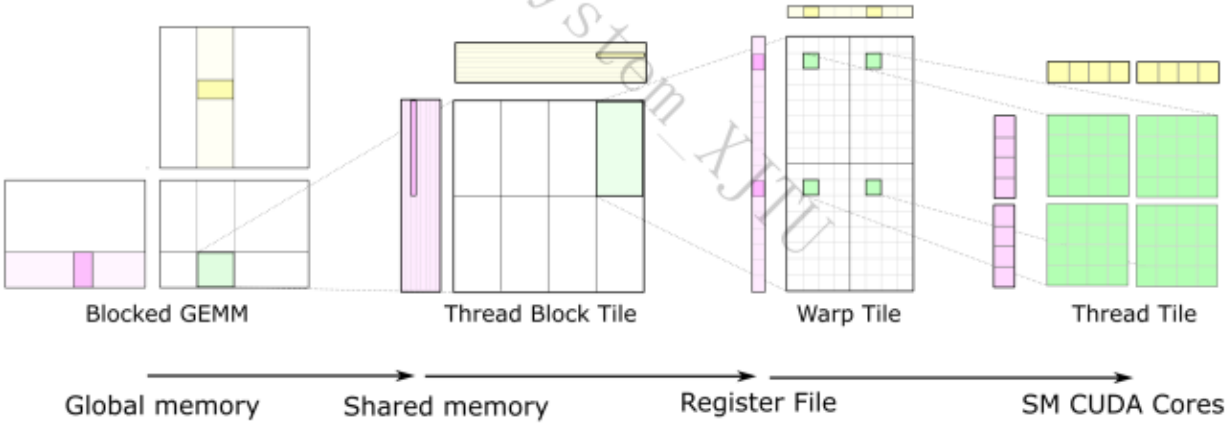
```
__global__ void VecAdd(float* A, float* B, float* C){
    int i = threadIdx.x;//依据线程进行索引
    C[i] = A[i] + B[i];
}

int main(){
    ......
    VecAdd<<<1, N>>>(A, B, C);//启用一个Block，一个Block含有N个线程。
}
```

GPU擅长在规则，稠密的数据上的高数据并行计算任务，与CPU上比能获得更高的能效比。因为有大量的算术运算单元，有大量的Core共享指令解码单元和控制单元。

异构计算

GPU如何高效计算矩阵乘法

提高访存计算比：从global memory到register的每层经可能多的复用数据。（分块思想）



此处说明，GPU也是层级结构。数据读到GLobal memory上，然后每个block可以用自己的Shared memory，之后，每个thread可以用自己的Register File，最后每个SP进行计算得到结果。

分块vs不分块

过程简图如上所示

不分块——读取需要$2K*MN$次访存

分块后：考虑到sharedmemory大小有限，假设每次只能从A读m*k的块，B中读k*n，则读取放存量$(mk*kn)\frac{K}{k}*\frac{M}{m}*\frac{N}{n}=\frac{m+n}{mn}KMN$

block层面，每次计算上面所说的块大小的数据后，滑动，将结果压进C中进行累加文件中进行累加。

每个warp tile负责A和B中一个分片的计算，每个一个warp tile中的warp分时复用。在nvidia的设计中，一个warptile有4个warp。

对于warp中由于线程之间无法相互访问寄存器文件，因此每个线程应尽可能将寄存器的数据进行重复使用。在nvidia的设计中，一个thread进行4*4的矩阵计算，再结合刚刚设计的一个warptile有4个warp，一个SP最终会完成8*8的矩阵计算。

整体而言，核心思想就是分块，把邻域的数据尽可能高效率利用。

软件流水线

由于大量使用寄存器使得并发运行的blocks较少，故采用软件流水线的方式隐藏方寸延时。

综上，最终可以得到90%的cublas性能。对于自定义算子，可以用nvidia开源API实现

ASICs

- 观察：当前深度学习模型的计算特点：
  - 深度学习的计算本身是一种近似求解方法 → 可容忍较低计算精度
  - 当前深度学习模型的计算大量基于矩阵乘法运算 →可使用较简单的计算指令
  - 矩阵乘法在现有体系结构上都是访存瓶颈 → 增加片上内存，减少访存
  - 负载中有较大量的浮点运算→ 增加算术运算密度
- 设计思路
  - 使用低精度计算，如16bit 浮点 或8bit 定点
  - 使用简化的指令集描述关键计算指令
  - 使用流水线计算模型来减少数据读取，如脉动阵列
  - 使用比向量并行指令计算密度更高的并行方法

以Google TPU为例

采用8bit计算数据类型（只可用于模型推理）

采用简化的指令集

高度并行的矩阵处理单元（MXU）

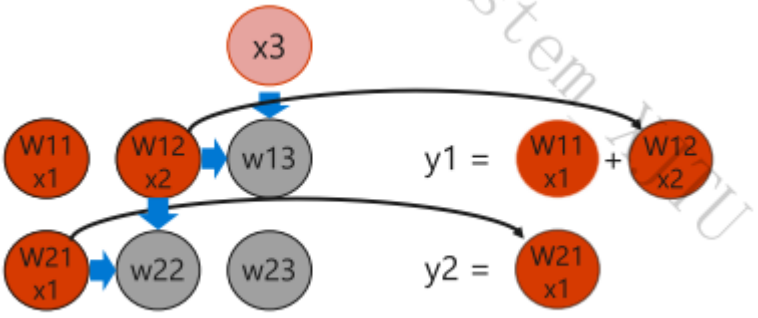
脉动阵列（TPU与CPU&GPU最大的区别）

CPU和GPU通常需要花费大量的功耗去读取寄存器的值

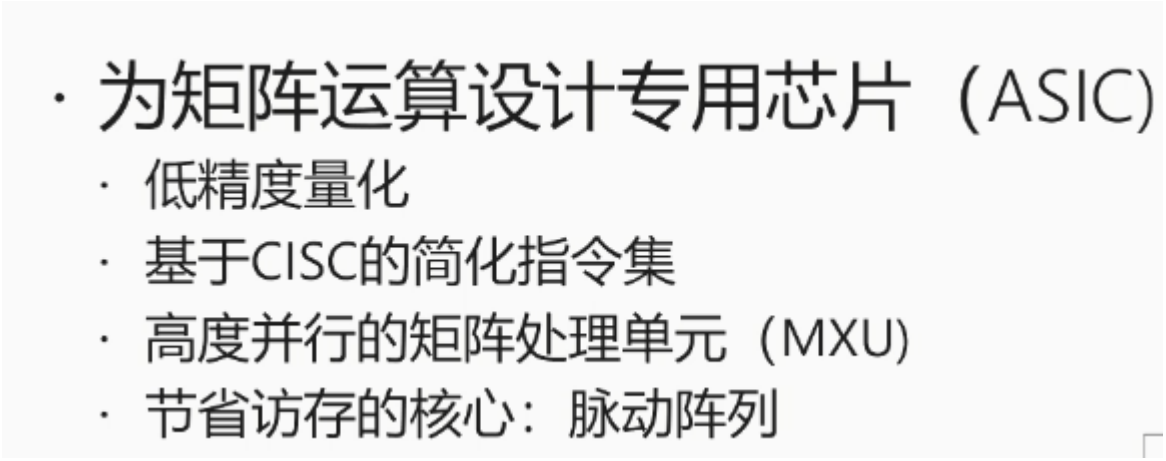
脉动阵列的设计思想使将多个ALU运算单元串联起来，从而避免每次计算都读取寄存器。

缺点：要求计算符合特定的规则。

例：矩阵*向量



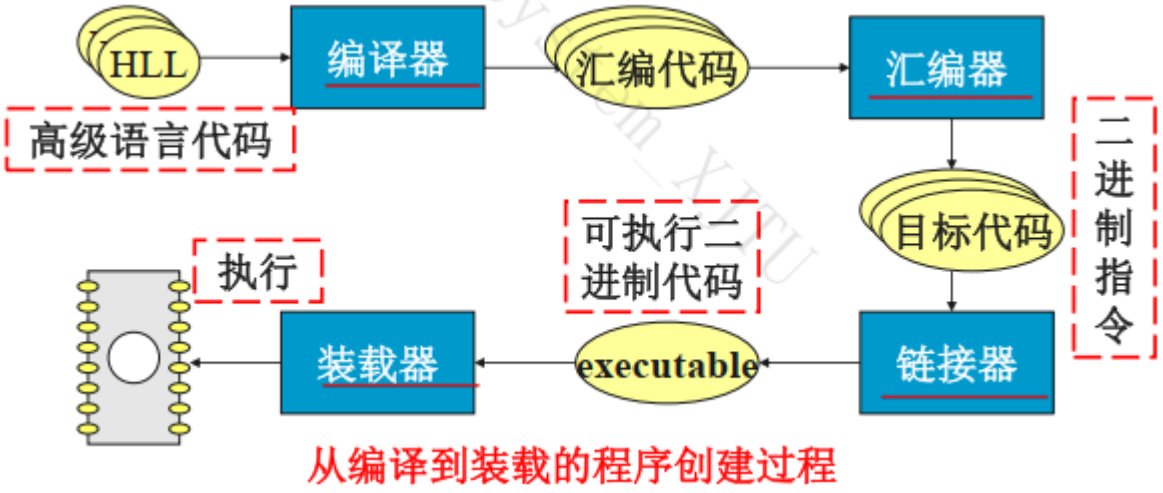
计算完后，不存回寄存器，而是将结果传给相邻的节点。周期数->4



AI编译器与编译优化

传统编译器

编译器过程：



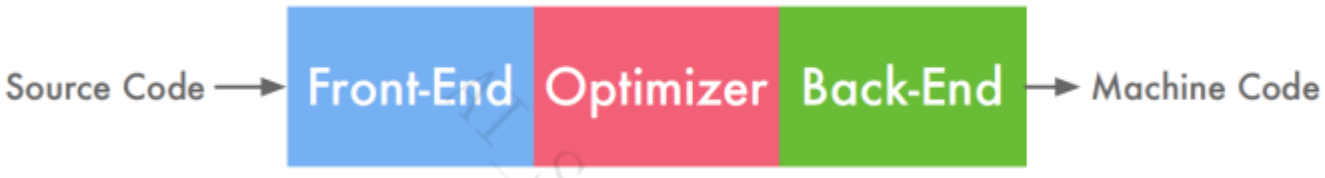
编译器是一种计算器程序，将高级程序语言转化成计算机能理解的机器语言代码。

编译器：翻译+优化

汇编器：汇编转二进制指令，但只是对文件进行翻译，不知道文件之间的关系

链接器：把二进制指令之间的关系进行联系，得到可执行二进制代码。

编译器基本构成（范式）：



1. Front-End：主要负责词法和语法分析，检查源代码是否正确，然后转化成抽象语法树。

2. Optimizer：对中间代码进行优化。
3. Back-End：将优化过的中间代码转化成针对各自平台的机器代码。

传统编译器示例——GCC与LLVM

GCC：虽然三段式，但是没有较好的前后端解耦性，类似于“全连接”，即“对每种高级语言和后端，都有一套体系”。

LLVM：鲜明三段式。有IR（中间表示），模块化好，解耦性强。前端开发者不需要了充分了解后端，后端不需要充分了解前端，IR是前后端的解耦工具，便于开发。在LLVM的Middle-end，有Pass，代表对源程序的一次完整扫描与优化处理。

AI编译器

AI编译器的挑战：

1. 越来越多的新算子被提出，算子库的开发、维护、优化和测试工作量越来越大。（Nvidia算子开发库：CuDNN）
2. 专用加速芯片爆发导致性能可移植性成为一种刚需。不同厂家的ISA多种多样，缺少通用的编译工具链，导致已经存在的CPU和GPU的优化算子库很难短期一直到NPU上，难以复用。

AI编译器和传统编译器对比：

1. 理念类似：都力求通过一种更通用，自动化的方式尽进行程序优化和代码生成，从而降低对不同硬件的手工优化。
2. 优化方式类似：通过统一IR执行不同的Pass进行优化，从而减少内存占用，提高执行性能；
3. 软件结构栈类似：分成前端，优化，后端三段式。
4. AI编译器依赖传统编译器：AI编译器对Graph IR进行优化后，将优化后的IR转换成传统编译器IR，最后依赖传统编译器进行机器码生成。

AI编译器主要目的是优化程序性能，其次是降低变成难度。传统编译器的主要目的是降低编程难度，其次是优化程序的性能。

具体而言：

1. IR表达层级差异：AI编译器一般会有high-level IR，用来抽象描述深度学习模型中的运算。传统编译器相对而言low-level IR，用于描述基本指令运算。
2. 优化策略差异：AI编译器面向AI领域，优化时引入更多领域特定只是，从而进行high-level的优化手段。如：
    1. 在high-level的IR上做operator fusion等
    2. 降低计算精度，如int8， fp16等。传统编译器一般不执行改变变量类型和精度优化。

神经网络编译器

前端：基于Python的DSL语言

后端：神经网络加速器

中间表达：计算流图，算子表达式等

优化过程：*前端优化，后端优化*。

前端优化

通过图的等价变换化简计算图，从而降低计算复杂度或内存开销。与硬件无关。

算术表达式化简

a*0 -> 0

减少乘法（减少一个kernel调用）！

公共子表达式消除

目标：快！

如果消除后，变慢了，就不消除

死代码消除

移除对程序执行结果没有任何影响的代码

1. 避免执行不必要的操作，提高运行效率，减少运行时间
2. 节省不必要的资源分配，优化空间
3. 减少代码的长度，增加可读性。

实例：在googlenet中，有Pass，在训练时计算，推理时不计算。对于推理过程而言，属于死代码。

常量折叠

如果有的节点输出值确定了，可换成常量。

在编译期计算并化简

常熟传播的优化在DL，尤其是模型推理的时候非常有用（不更新参数）。

实例：BN折叠

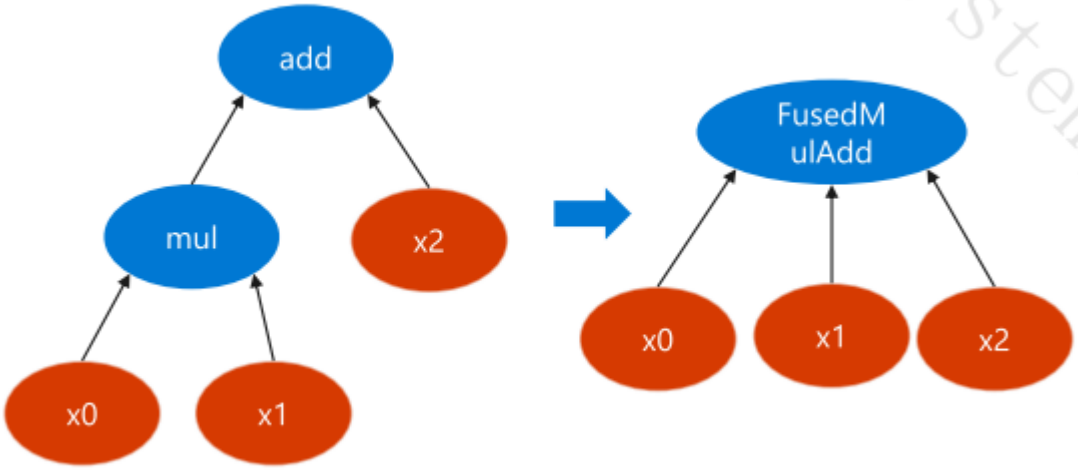
训练时，$\gamma, \beta, \mu, \sigma$一直在更新；在推理时，这四个值全部固定，BN就变成了对上一层结果进行简单的线性变换。

卷积也是一个线性变换，因此这两个操作可以合并成一个单一的线性变换，这将减少推理时要执行的操作数量，提升推理速度。（减少了一次储存进memory和从memory中读的时间）

$$z = W * x + b$$

$$out = \gamma \cdot \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

算子融合（operator fusion）



GEMM自动融合

通过将输入张量合并成一个大的张量来实现将相同的算子合并成一个更大的算子，更好的利用硬件的并行度。

布局转换

行优先

列优先

卷积网络特征图：4D（N, H, W, C）或（N, C, H, W）

NCHW适合需要对同一个通道的数据单独运算的操作（如MP），计算式需要的储存更多，更适合GPU运算。利用GPU带宽较大，并行性强的特点，其访存与计算的控制逻辑相对简单。

NHWC适合对不同通道的同一位置元素进行某种运算（如conv1*1）

事实和多喝CPU运算。CPU内存带宽相对小，每个数据计算的时延较低，优势计算机采用异步方式边读边算减少访存时间，因此计算控制灵活且复杂

以NPU/GPU为基础的Pytorch和MindSpore框架默认采用NCHW格式

Tensorflow采用NHWC。

内存分配

如何节省内存?

1. 内存复用
2. 替代操作

```
#DEFINE A
B = sigmoid(A);
C = sigmoid(B);
E = sigmoid(C);
F = sigmoid(B);
G = E*F;
```

3. 内存共享：两个数据共享同一块内存空间，如果有一个计算后不再需要，后一个数据可以覆盖前一个数据。

前端优化时独立于机器的优化，与硬件无关！

后端优化

优化算子的内部具体实现

循环优化

循环转开，减少循环的开销，使能后续优化，比如指令并行。

循环分块：数据量过大的时候，无法一次性将数据加载到设备内存。将数据分块储存在Cache中，经可能直接访问缓存中的数据块。**避免需要从主存中读取数据带来的访问延迟**

循环融合：将相邻或紧密间隔的循环融合在一起，减少循环开销和增加的计算密度可改善软件流水线，数据结构的缓存局部性增加。

循环拆分：将循环拆分成多个循环。

可以将控制和计算分开！分别用cpu，gpu处理。