

神经网络量化加速

Introduction of Neural Network Quantization

- Sensetime HPC Group



神经网络的计算可以做到多快？

2.1 指令执行过程

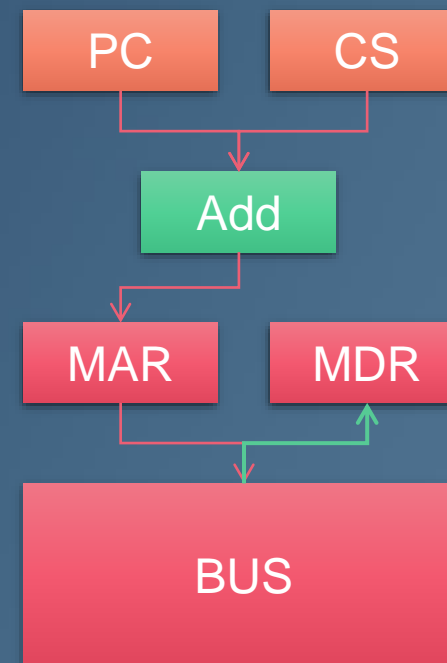
Execution Procedure



2.1.1 取指令 Instruction Fetch



- 在取指令阶段中，CPU从PC寄存器读出地址偏移，再与段寄存器相加后将信号写入地址总线，总线从指定位置读入指令。你需要注意到，在冯诺依曼体系结构中，指令和数据并没有本质的区别，都是在内存中存储的一系列01串。
- 现代计算机中，由CPU和操作系统厂商提供了一系列安全措施来防止用户直接操作内存中保存程序指令的区域。
- 取指令过程需要访存。

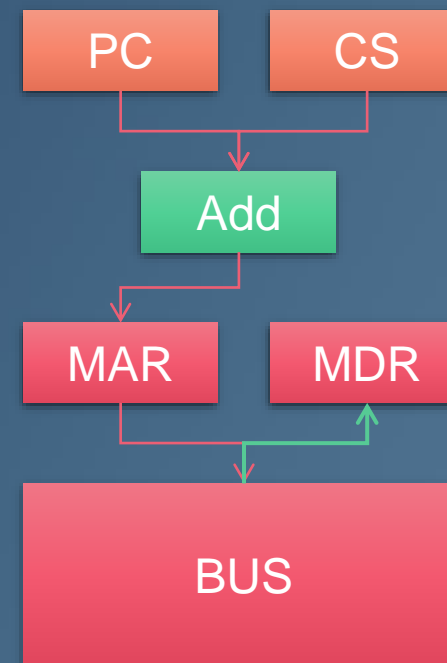


2.1.1 取指令

Instruction Fetch



- PC : 指令计数器
- CS : 代码段寄存器
- MAR : 存储器地址寄存器
- MDR : 存储器数据寄存器
- BUS : 总线 (实际分为地址总线、数据总线、控制总线)
- 由PC到MAR的过程由电路直接实现, 非常快



2.1.2 译码

Decoding



- 在从总线中取回指令后，CPU使用译码电路执行指令译码，并生成相应控制信号准备计算。
- 在这一过程中，被译码的指令可能被拆分成多条微指令，即将一个复杂的操作拆解成一系列简单操作的总和。
- 处理器支持的种类数量越丰富，其指令译码器设计也更加复杂，CPU拥有最为复杂的指令译码电路，相比之下GPU与ASIC的指令译码机构则更简单，为此能够腾出更多的芯片空间来设计计算单元。

2.1.2 ZEN3 架构指令集

Insturction set of AMD ZEN3



指令	微指令个数	指令延迟	指令吞吐量
ADD	1	1	4
ADDSS	1	3	2
PADDDB	1	1	4
MUL (r8)	1	3	1
MUL (r32)	2	3	1
MULPS	1	3	2
DIVSS	1	10.5	0.3

2.1.3 访存

Operand Fetch



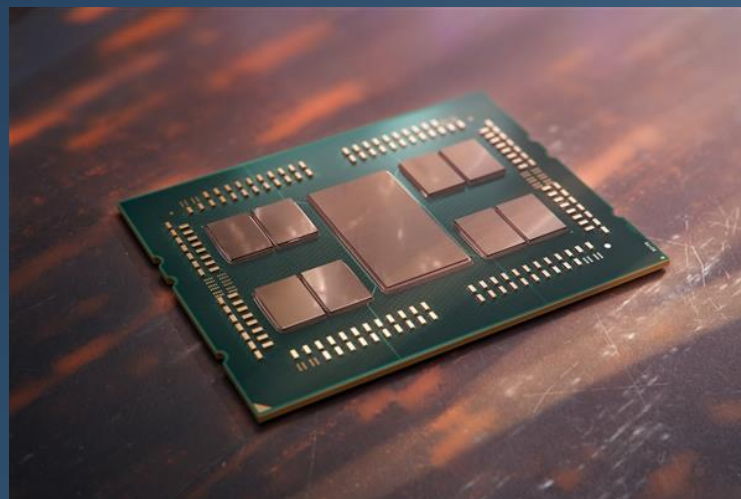
- 对于大部分指令而言，我们总是需要从内存或寄存器上取回一些数据进行计算，这一过程被称为访存。
- 为了掩盖访存延迟，现代计算机引入了**指令流水线**机制，在当前指令处于访存阶段时，计算机可以立即开始下一条指令的取指令或译码工作，从而加快运行效率。
- 对于CPU而言，其访存机制也更加复杂，因此设计了复杂的**内存控制器**。而GPU、FPGA的访存则简单的多，它们往往也能提供更大的**内存带宽**。

2.1.3 内存控制器

Memory Controller



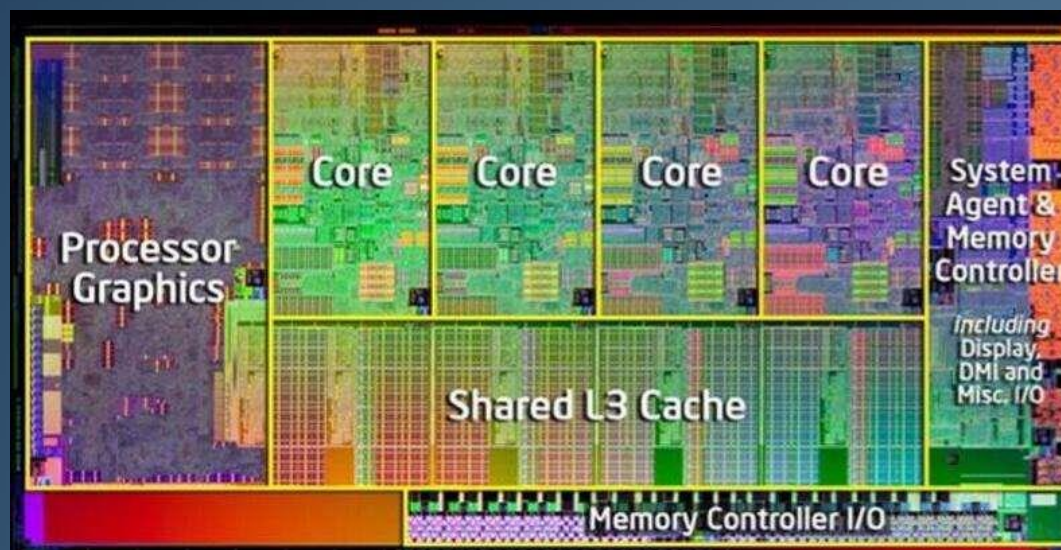
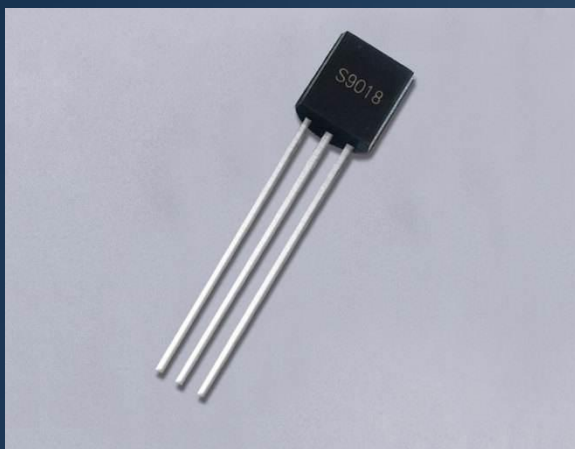
- 在CPU中，内存控制器往往占据了大量芯片面积，如下图所示 AMD ThreadRipper 处理器中，位于中央核心部分的就是该芯片的**内存控制器**。



2.1.4 执行 Execution

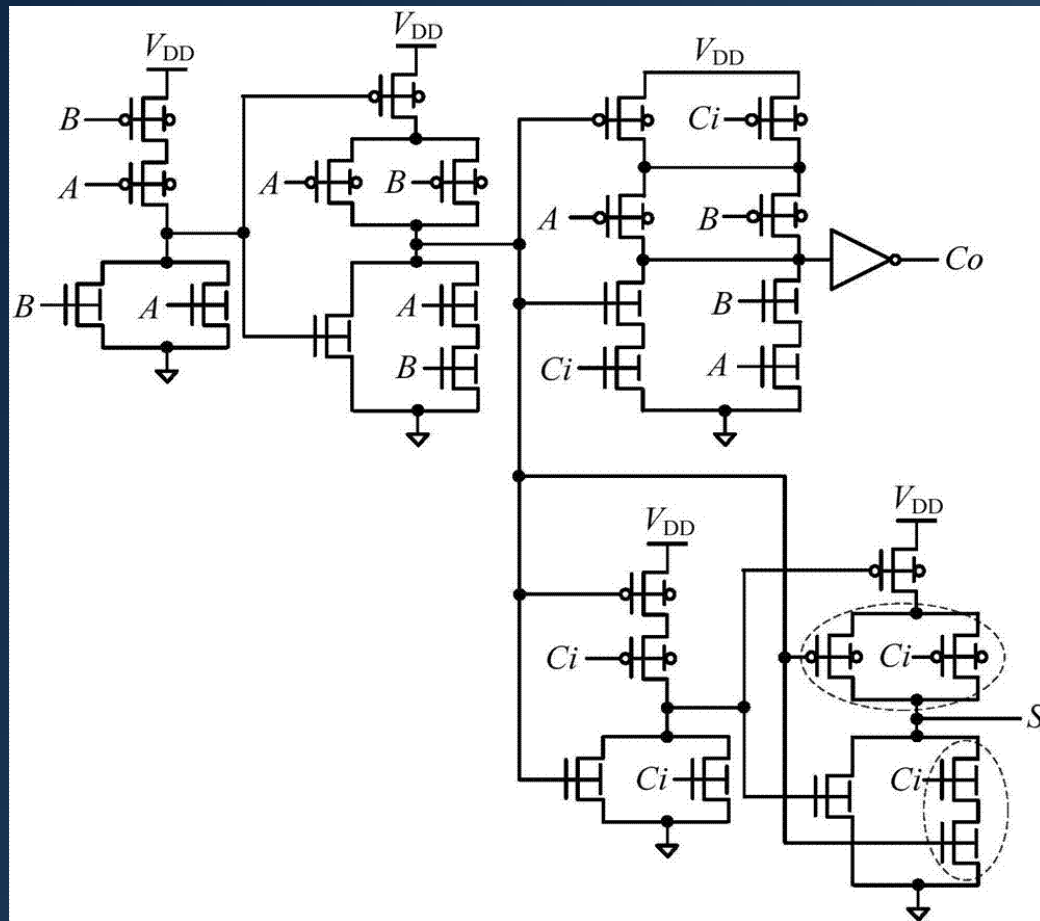


- 对于现代计算机而言，所谓的计算实际上是处理器芯片内的晶体管逐个导通的过程。处理器根据指令译码的结果，将特定信号送入晶体管的输入端，而后在晶体管的输出端接收结果。



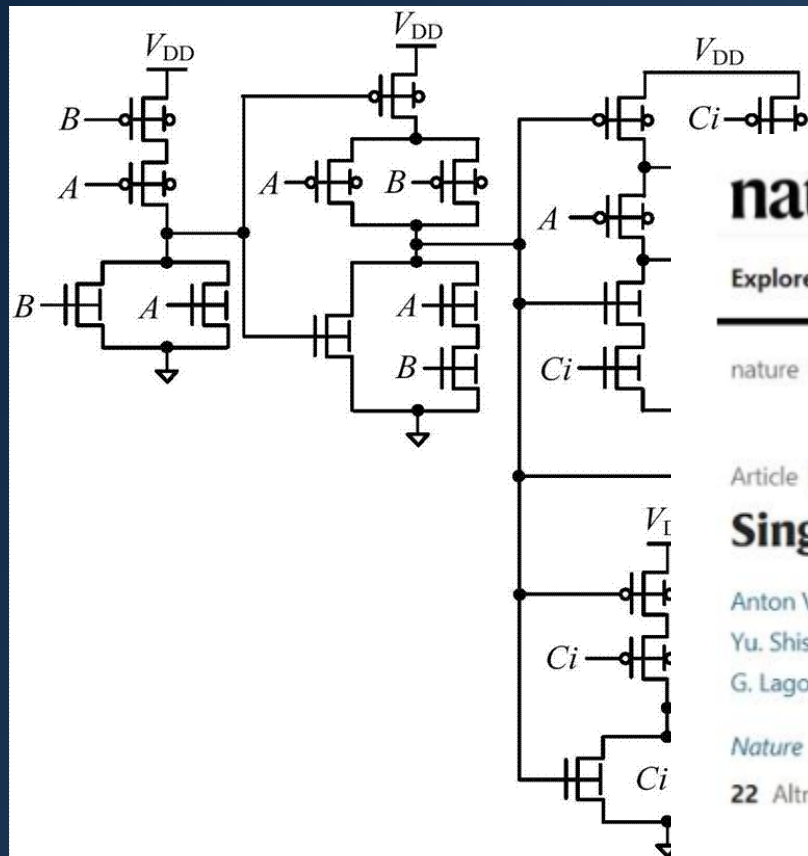
2.1.4 加法器实例电路

Example of a full-adder



- 输入信号 A, B, C_i
- 输出信号 S, C_o
- 试一试理清左图中的电路逻辑

Example of a full - adder



- 输入信号 A, B, Ci
- 输出信号 S, Co



[Explore content](#) ▾ [About the journal](#) ▾ [Publish with us](#) ▾

[nature](#) > [articles](#) > [article](#)

Article | Published: 22 September 2021

Single-photon nonlinearity at room temperature

Anton V. Zasedatelev , Anton V. Baranikov, Denis Sannikov, Darius Urbonas, Fabio Scafirimuto, Vladislav Yu. Shishkov, Evgeny S. Andrianov, Yuri E. Lozovik, Ullrich Scherf, Thilo Stöferle, Rainer F. Mahrt & Pavlos G. Lagoudakis 

Nature **597**, 493–497 (2021) | Cite this article

22 Altmetric | Metrics

01有机分子振动凝聚激子极化子，开关切换速度高达每秒1万亿次！

2.2 执行指令所需时间

Execution Time Fleet



操作	时间代价
取指	100 (访主存) 1 (访缓存)
译码	> 1
访存	100 (访主存) 1 (访缓存)
执行	1~5
写回	100 (访主存) 1 (访缓存)

操作	时间代价
CPU 进程上下文切换	1000周期以上(重建缓存)
CPU 线程上下文切换	10~100周期 (切换寄存器组)
Python 加法	100~1000
访硬盘 (机械)	4ms(磁盘寻道)
访硬盘 (固态)	0.002ms

3.1 现代处理器

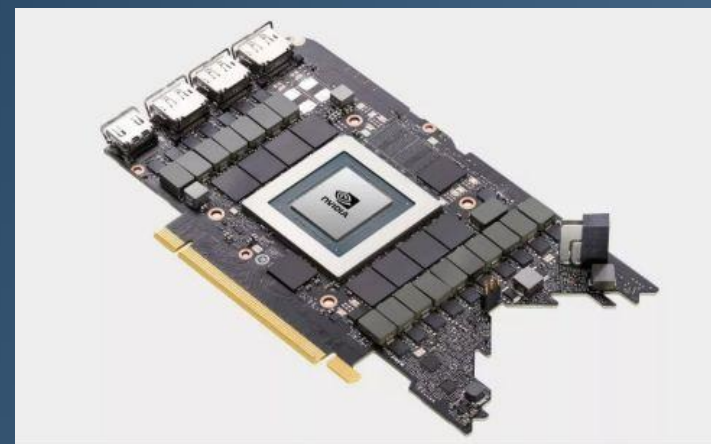
Modern Processors



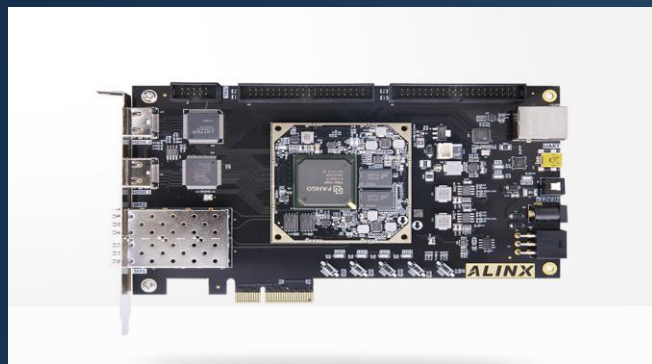
- X86架构



- ARM架构



- CUDA架构



3.1.1 CPU、GPU渲染性能

Blender Performance(CPU&GPU)

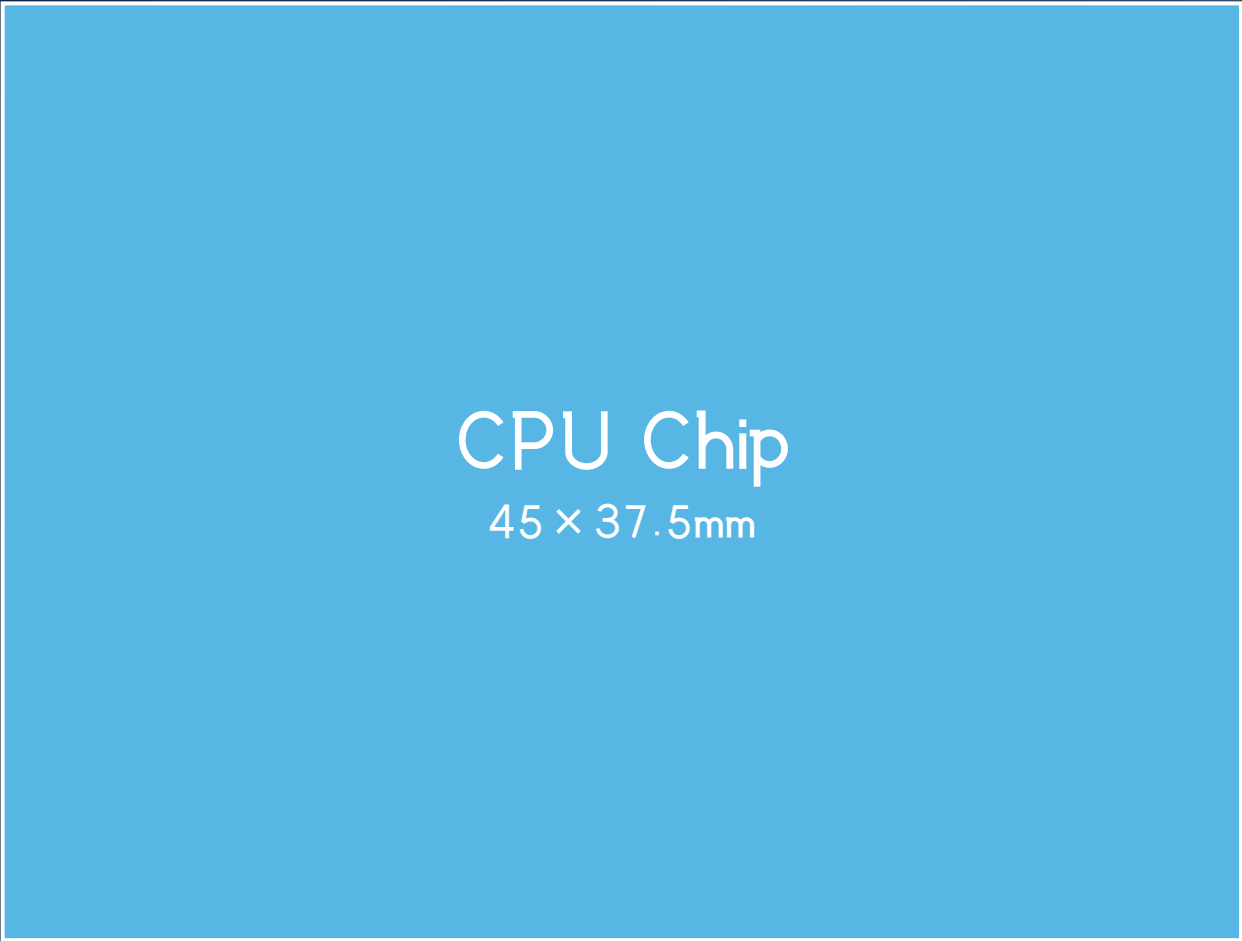


CPU型号	渲染分数
AMD Ryzen Threadripper 2990WX 32 - Core Processor	473.63
AMD Ryzen 9 5950X 16 - Core Processor	460.37
AMD Ryzen 9 3950X 16 - Core Processor	435.6
Intel Core i9 - 7980XE CPU @ 2.60GHz	423.67
AMD Ryzen 9 3950X 16 - Core Processor	387.57
12th Gen Intel Core i9 - 12900K	367.42
12th Gen Intel Core i9 - 12900KF	356.69
12th Gen Intel Core i7 - 12700K	321.1
AMD Ryzen 9 3900X 12 - Core Processor	281.08
AMD Ryzen Threadripper 2950X 16 - Core Processor	271.68
AMD Ryzen Threadripper 1950X 16 - Core Processor	270.69

GPU型号	渲染分数
NVIDIA GeForce RTX 3090	5588.12
NVIDIA GeForce RTX 3080 Ti	5358.97
NVIDIA GeForce RTX 3080	5130.48
NVIDIA GeForce RTX 3070	3419.82
NVIDIA GeForce RTX 2080	2754.35
NVIDIA GeForce RTX 3060	2445.18
NVIDIA GeForce RTX 2070	2170.22
NVIDIA GeForce GTX 1080 Ti	863.46
NVIDIA GeForce GTX 1660 Ti	820.82
Apple M1 Max (GPU)	702.87

3.2.1 CPU体系结构 (X86)

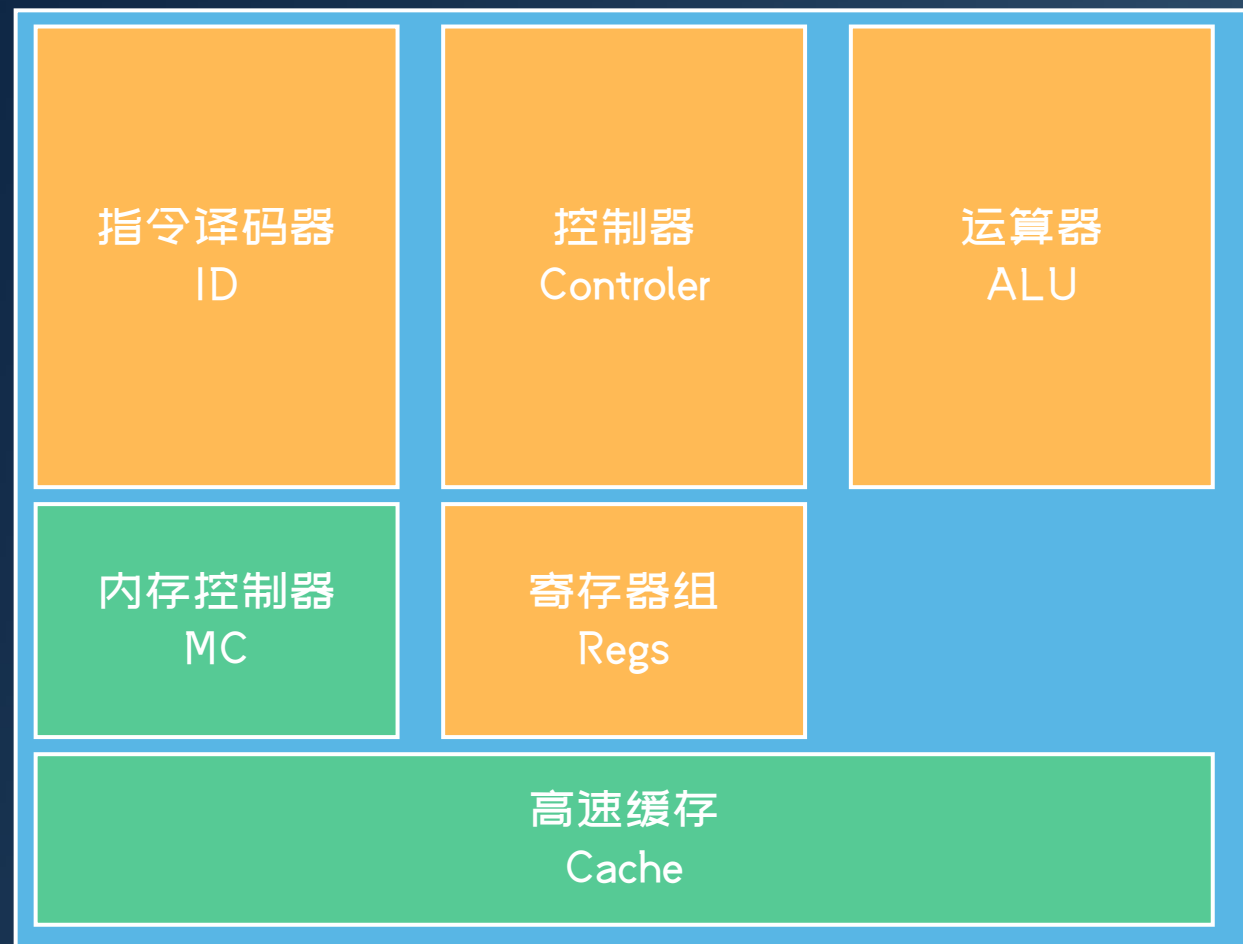
X86 Archticture



CPU Chip
45 × 37.5mm

3.2.1 CPU体系结构 (X86)

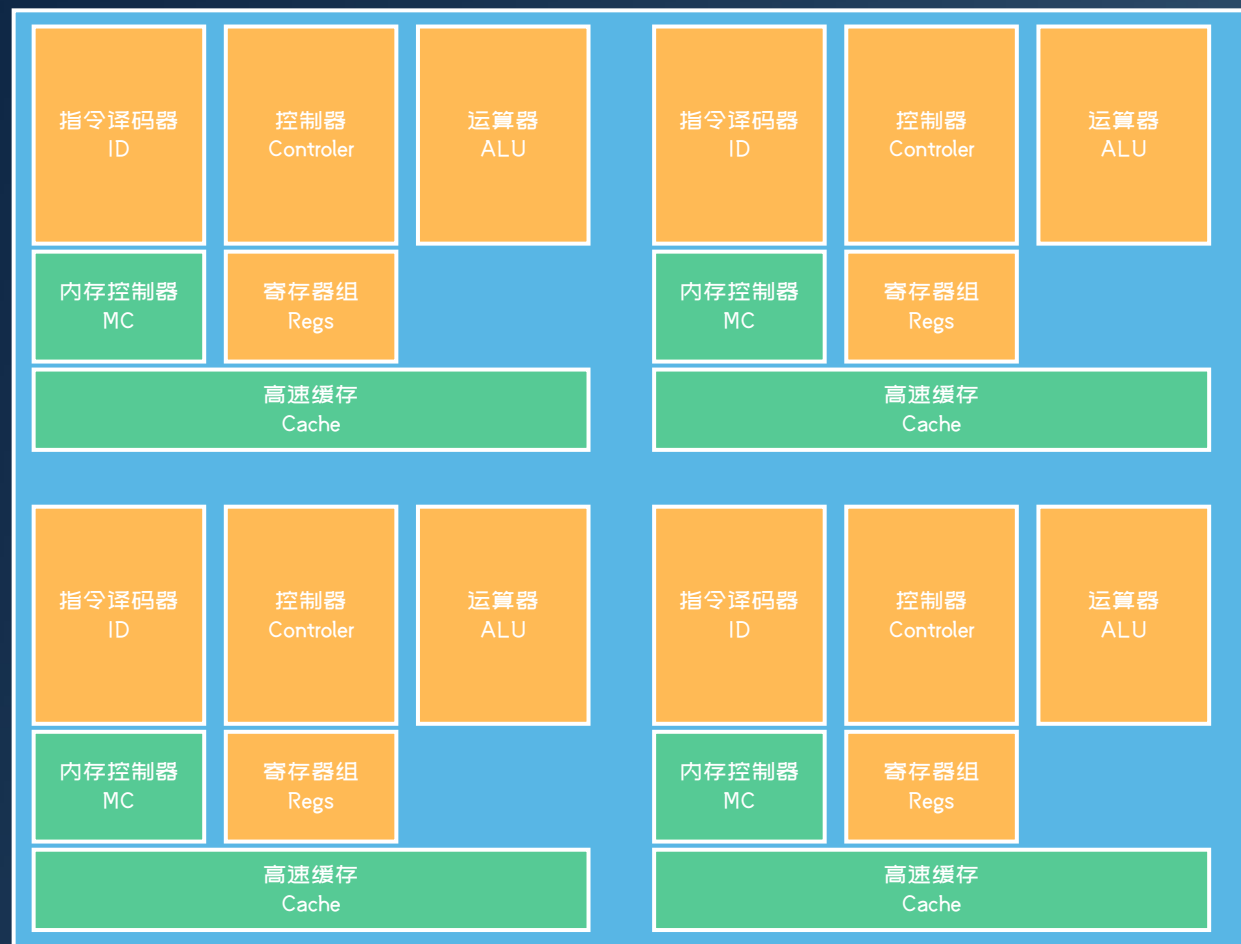
X86 Archticture



- 随着工艺制程的提高，在有限的芯片面积下我们可以放入越来越多的晶体管，但芯片的频率已经无法提升。
- 在这种情况下要如何提升芯片的计算效率？

3.2.1 CPU体系结构 (X86)

X86 Archticture



- CPU核心中的内存控制器与高速缓存，可以设计为公共组件，从而进一步提高集成度，缩小芯片面积，提升运行效率。

3.2.1 CPU体系结构 (X86)

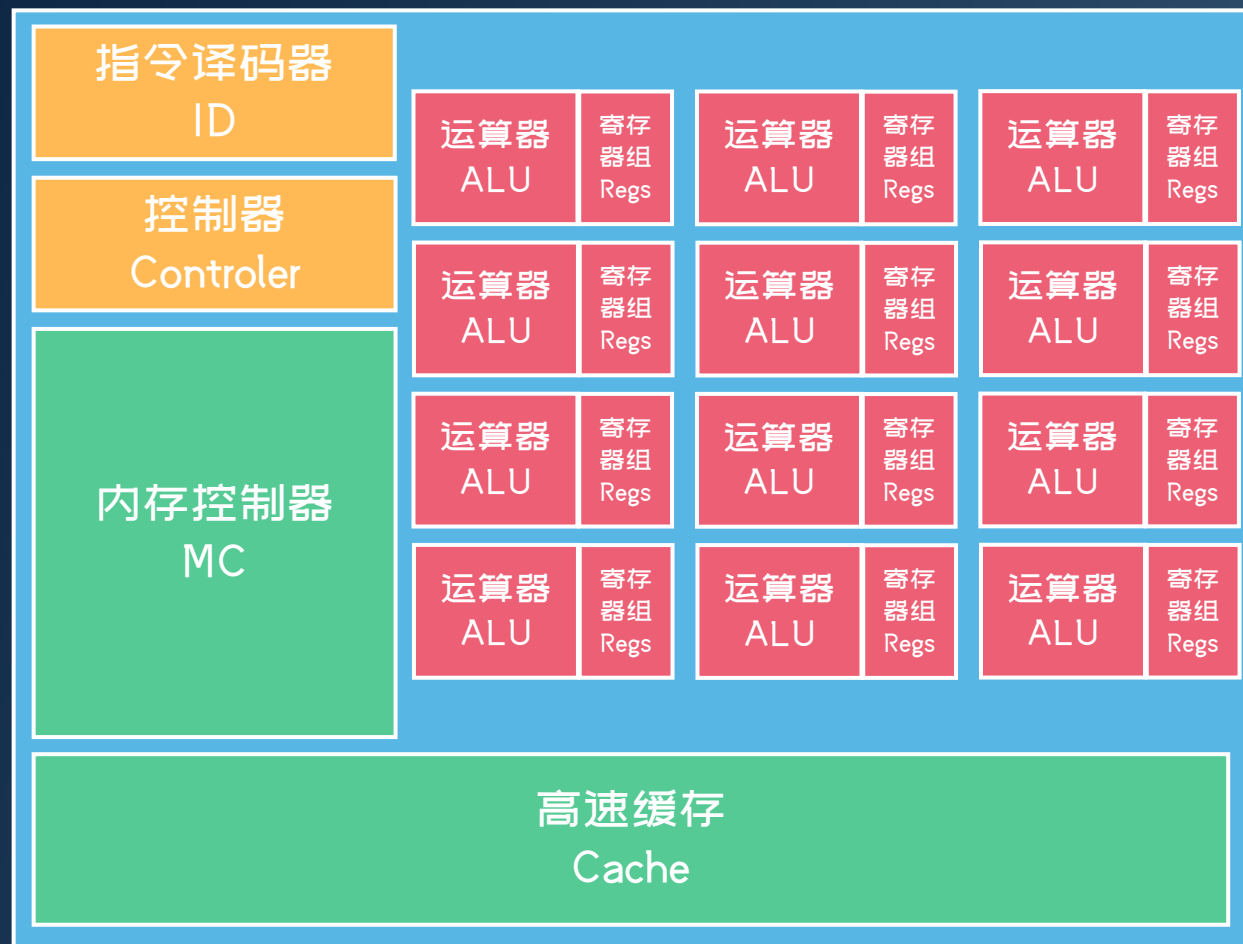
X86 Archticture



- CPU核心中的译码器、控制器、运算器需要支持太多指令集，因此其设计十分复杂，能否设计一些专用核心来降低芯片面积？
- 能否依据此想法进一步提升运行效率？

3.2.2 GPU体系结构 (CUDA)

CUDA Archticture



- GPU无需支持过多指令集，因此其指令译码器与控制器可以设计的**相对简单**，同时GPU体系中要求运算器共享指令译码与控制，因此**只需一套指令译码与控制器件**即可同时操作多个运算器。
- 优点：芯片支持更多核心。
- 缺点：ALU必须运行相同的指令。

3.2.2 GPU体系结构 (CUDA)

CUDA Archticture



- GPU无需支持过多指令集，因此其指令译码器与控制器可以设计的**相对简单**，同时GPU体系中要求运算器共享指令译码与控制，因此**只需一套指令译码与控制器件**即可同时操作多个运算器。
- 优点：芯片支持更多核心。
- 缺点：ALU必须运行相同的指令。

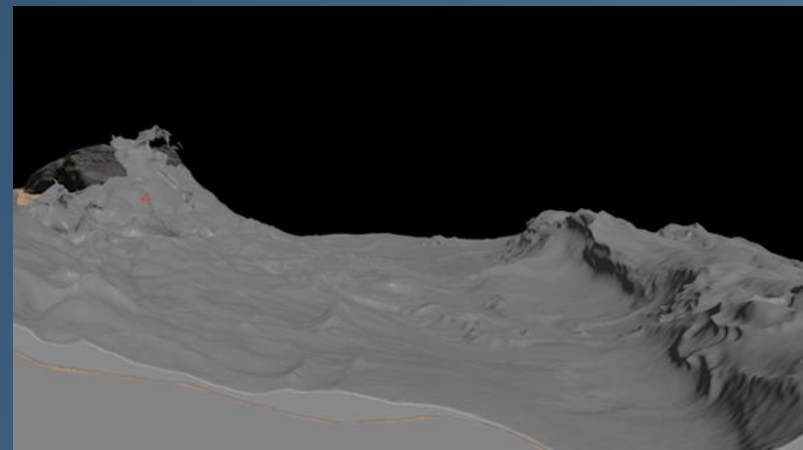
3.2.2 SPH方法求解纳维 - 斯托克斯方程

CUDA example: Solving NS Equation by SPH method.



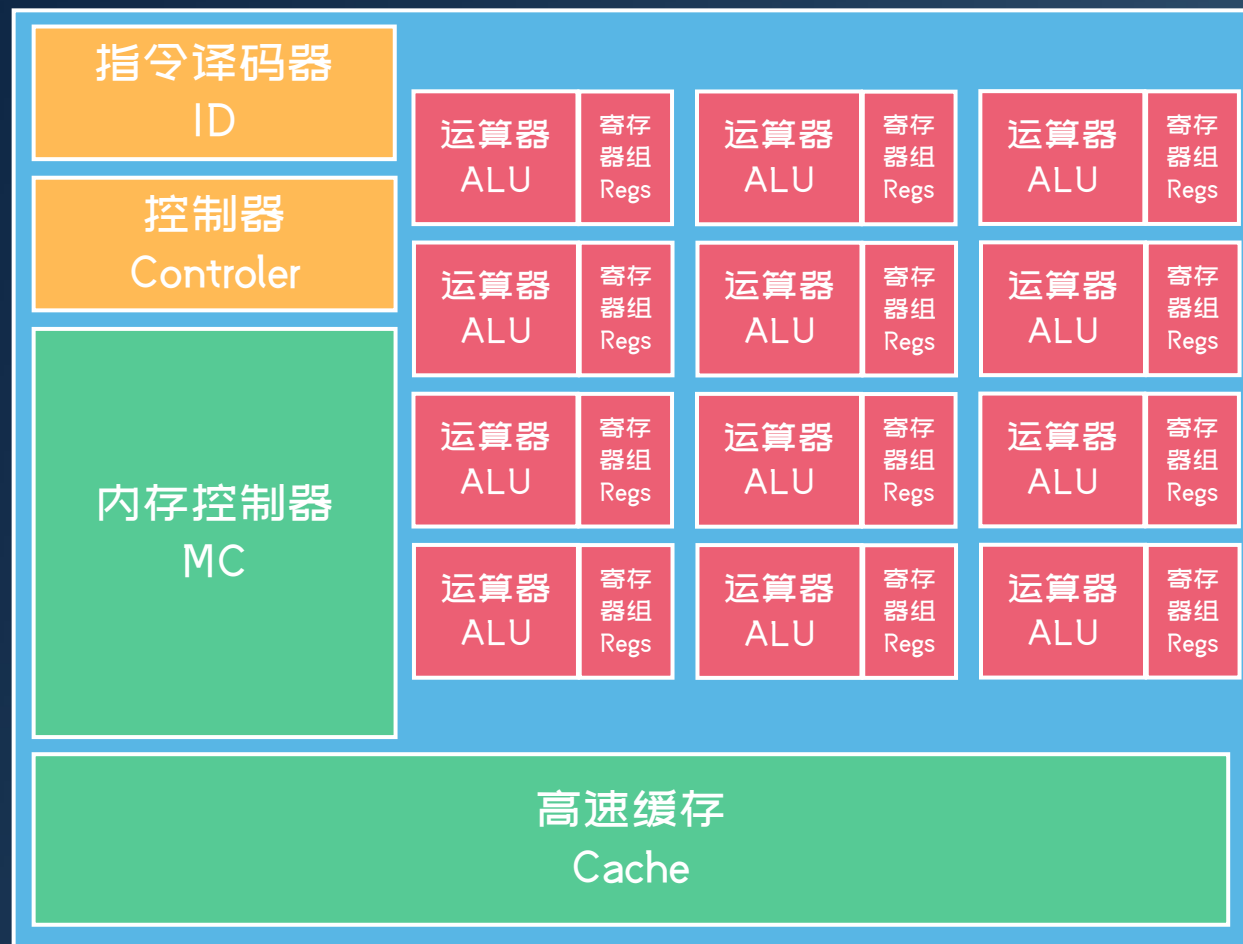
$$\rho \frac{Dv}{Dt} = \rho g - \nabla p + \nabla^2 v$$

- 方程描述了流体粒子加速度 $\frac{Dv}{Dt}$ ，与密度 ρ 、重力 g 、压力 p 之间的关系
- 在实际计算中，常用SPH对其进行模拟，算法流程：
 1. 初始化粒子集合 S ，其中包含100万个粒子
 2. 对于每一个粒子 $s_i \in S$ ，计算其附近密度 ρ_{s_i} 与压强 p_{s_i}
 3. 将 p_{s_i} ρ_{s_i} 代入方程，求解 $\frac{Dv}{Dt} |_{s_i}$
 4. 更新粒子 s_i 的位置与速度。
 5. 更新完所有粒子位置后，返回第二步进行循环。



3.2.3 专用芯片架构

ASIC Archticture



- 在GPU的基础上，我们可以更快吗？
- 虽然GPU已经移除了大量指令集系统，但对于特定应用而言，仍然有很多指令是不必须的。
- GPU为了图形运算，还有一些特殊器件包括texture memeory, ray tracing core 等等，这些东西也可以被移除。
- GPU的吞吐大，但延迟通常较高，功耗也不占优势能否从芯片设计的角度解决这些问题？

3.2.3 专用芯片架构

ASIC Archticture



- 移除浮点运算器，添加更多整数运算器。
- 移除图形计算设备，不支持图形相关指令。
- 更大的缓存，更大的显存。
- 如右图所示，MLU370-X4智能加速卡，峰值性能为：256TOPS(INT8)，24TFLOPS(FP32)，但功耗仅为150W
- 还可以通过改变架构做到更低延迟。

3.3 异构计算与主从设备交互

ASIC Architecture



Jetson NX核心模块

CPU: NVIDIA Carmel ARMv8.2 (6-core) @ 1.4GHz(6MB L2 + 4MB L3)

GPU: 384-core Volta @ 1100MHz + 48 Tensor Cores

DL加速: dual NVIDIA Deep Learning Accelerators

内存: 8GB 128-bit LPDDR4x @ 1600MHz | 51.2GB/s

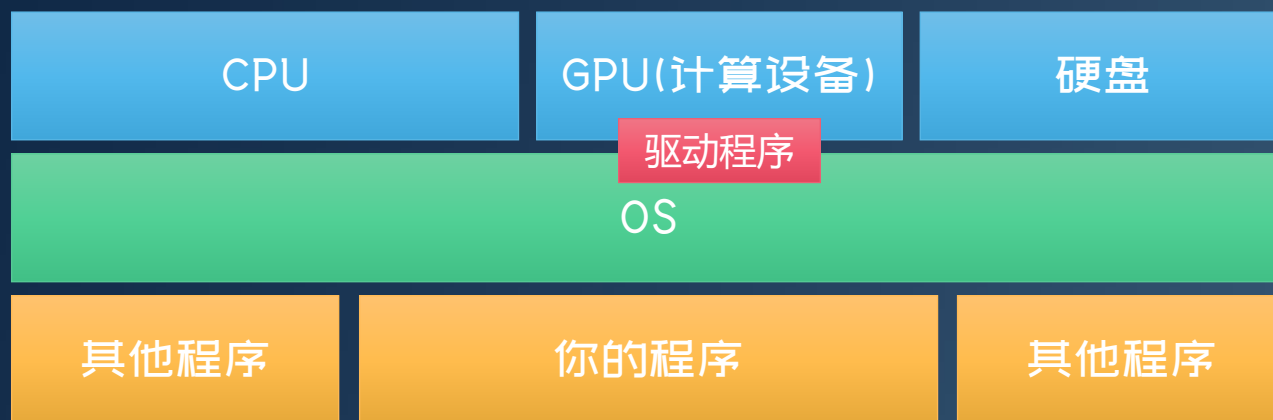
储存: 16GB eMMC 5.1

- 问题：为什么在开发板上有了GPU，还要有CPU？为什么有内存，还要有显存？什么是统一内存架构？
- CPU与GPU之间如何协作完成任务？

3.3.1 主从设备

Host & Device

逻辑视图：



物理视图：



- 对于现代计算机而言，为了运行你的应用程序，首先需要建立一个操作系统或类似的中间组件。
- 对计算设备的任何请求，都将通过 OS -> CPU -> OS -> GPU 的路径传输请求信号。
- 在物理层面，CPU与GPU通过PCIE总线互联，构成了通信基础。在软件层面，GPU驱动使得操作系统知道如何操作GPU进行计算。

3.3.1 主从设备交互

Host & Device communication



```
While(!Shutdown):
```

```
    # 从外部设备接收消息，代指一切输入，包括鼠标键盘  
    输入，网络信号，电信号等。
```

```
    message = receive_message()
```

```
    if message.type == "CALL GPU" :  
        sys.cuda.send_message(message)
```

```
    elif message.type == "KEYBOARD" :  
        sys.screen.print(message)
```

```
    else:...
```

消息循环

3.3.1 主从设备交互

Host & Device communication

```
While(!Shutdown):
```

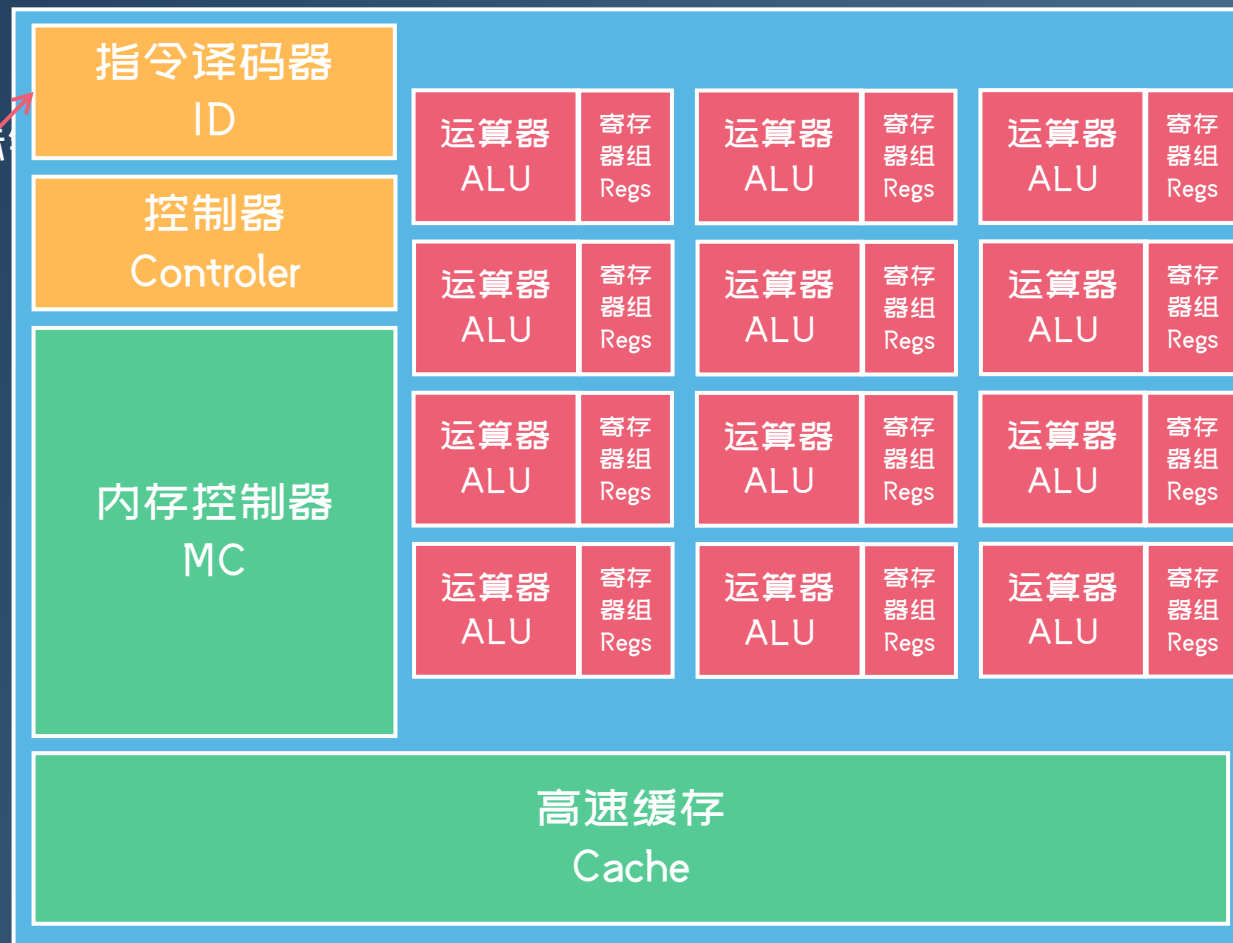
```
    # 从外部设备接收消息，代指一切输入，包括鼠标  
    输入，网络信号，电信号等。
```

```
    message = receive_message()
```

```
    if message.type == "CALL GPU" :  
        sys.cuda.send_message(message)
```

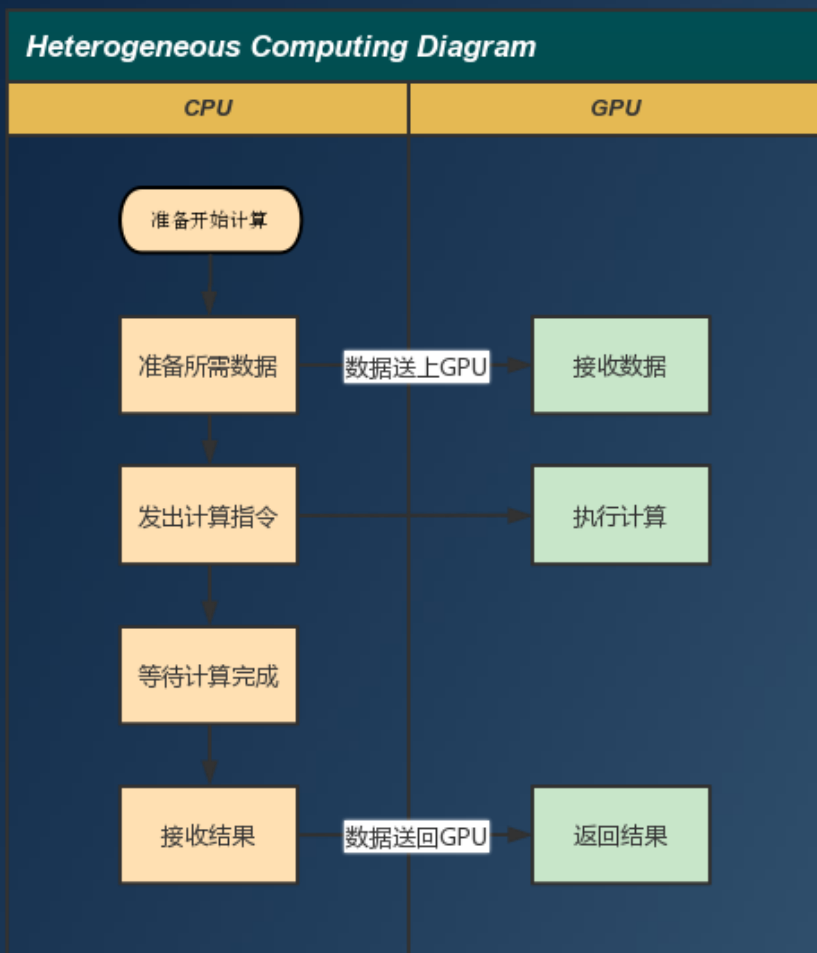
```
    elif message.type == "KEYBOARD" :  
        sys.screen.print(message)
```

```
    else:...
```



3.3.2 主从设备交互流水线

Host & Device pipeline



- CPU 与 GPU 之间总是异步的，这使得CPU可以在等待GPU完成运算的时候转手去做一些其他的事情，比如准备下一批需要送上数据。
- 实际上，在GPU上有所谓的指令流水线，GPU可以缓存接受到的任务并逐个执行，CPU完全无需等待GPU的执行结果就可以继续它其他的工作内容。
- 由于硬件特性以及设备流水线的存在，神经网络的延迟与吞吐并不是完全反比的关系：

$$Latency \neq \frac{1}{Throughput}$$

3.4 找出你的性能瓶颈

Performance bottleneck



- `import torch`
- `import torchvision`
- `from tqdm import tqdm`
- `DEVICE = 'cuda'`
- `model = torchvision.models.mobilenet.mobilenet_v2(pretrained=True)`
- `model = model.to(DEVICE)`
- `with torch.no_grad():`
- `data = torch.rand(size=[1, 3, 224, 224])` 
`272 Batch/Sec`
- `for i in tqdm(range(1024)):`
- `o = model.forward(data.to(DEVICE))`
- `data = torch.rand(size=[128, 3, 224, 224])` 
`22.4 Batch/Sec`
- `for i in tqdm(range(128)):`
- `o = model.forward(data.to(DEVICE))`

3.4 找出你的性能瓶颈

Performance bottleneck



- `import torch`
- `import torchvision`
- `from tqdm import tqdm`
- `DEVICE = 'cuda'`
- `model = torchvision.models.mobilenet.mobilenet_v2(pretrained=True)`
- `model = model.to(DEVICE)`
- `with torch.no_grad():`
- `data = torch.rand(size=[1, 3, 224, 224]).to(DEVICE)`
- `for i in tqdm(range(1024)):`
- `o = model.forward(data.to(DEVICE))`
- `data = torch.rand(size=[128, 3, 224, 224]).to(DEVICE)`
- `for i in tqdm(range(128)):`
- `o = model.forward(data.to(DEVICE))`

272 Batch/Sec

28.1 Batch/Sec

3.4 使用Torch.Profiler 来确定你的性能瓶颈

Introduction of torch.profiler



4.1 人工神经网络

ASIC Architecture



计算密集型算子

Conv, MatMul, BatchedMatMul, Gemm, LSTM,
ConvTranspose

运算复杂度 $O(n^2) \sim O(n^3)$

访存密集型算子

Relu, Relu6, Tanh, Sigmoid, Softmax, Swish,
Add, Mul, Sub, Div, CumSum, Depthwise Conv,
Slice, Transpose, Permute, Concat, Shuffle, ...

运算复杂度 $O(n^1) \sim O(n^2)$

4.1 算子

ASIC Architecture



计算密集型算子

Conv, MatMul, BatchedMatMul, Gemm, LSTM,
ConvTranspose

运算复杂度 $O(n^2) \sim O(n^3)$

访存密集型算子

Relu, Relu6, Tanh, Sigmoid, Softmax, Swish,
Add, Mul, Sub, Div, CumSum, Depthwise Conv,
Slice, Transpose, Permute, Concat, Shuffle, ...

运算复杂度 $O(n^1) \sim O(n^2)$

4.1 两类算子

ASIC Architecture



计算密集型算子

Conv, MatMul, BatchedMatMul, Gemm, LSTM,
ConvTranspose

运算复杂度 $O(n^2) \sim O(n^3)$

访存密集型算子

Relu, Relu6, Tanh, Sigmoid, Softmax, Swish, Add,
Mul, Sub, Div, CumSum, Depthwise Conv, Slice,
Transpose, Permute, Concat, Shuffle, ...

运算复杂度 $O(n^1) \sim O(n^2)$

扩展知识：内存带宽

计算机执行过程

执行：对于大部分计算，我们总是需要将计算结果保存起来（这些数据此时还在CPU的寄存器上），因此需要额外的访存指令来将它们写入内存。相比于执行而言，访存过程仍然将是缓慢而低效的。



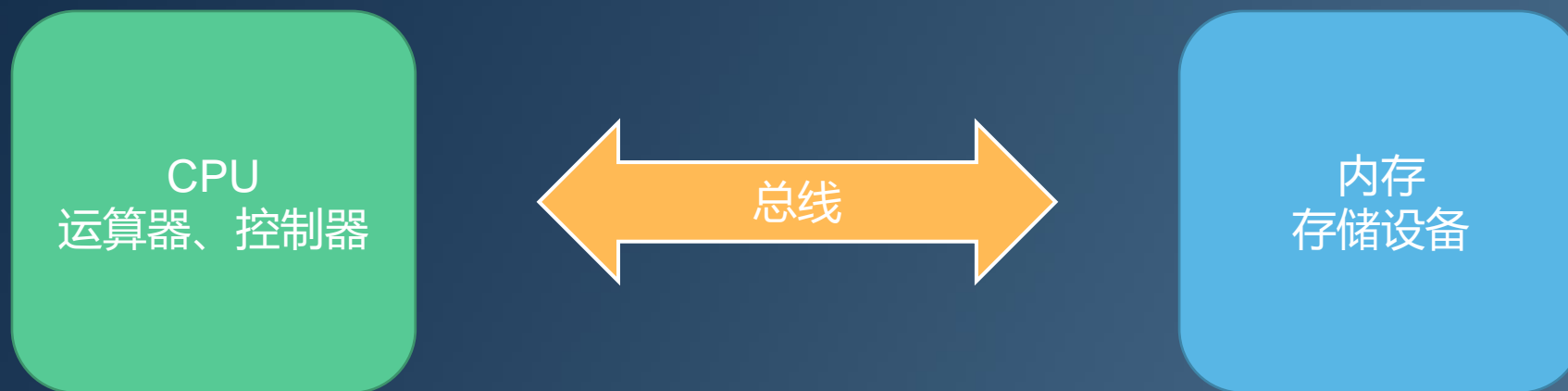
取指令

译码

访存

执行

冯诺依曼体系与现代计算机



冯诺依曼体系与现代计算机



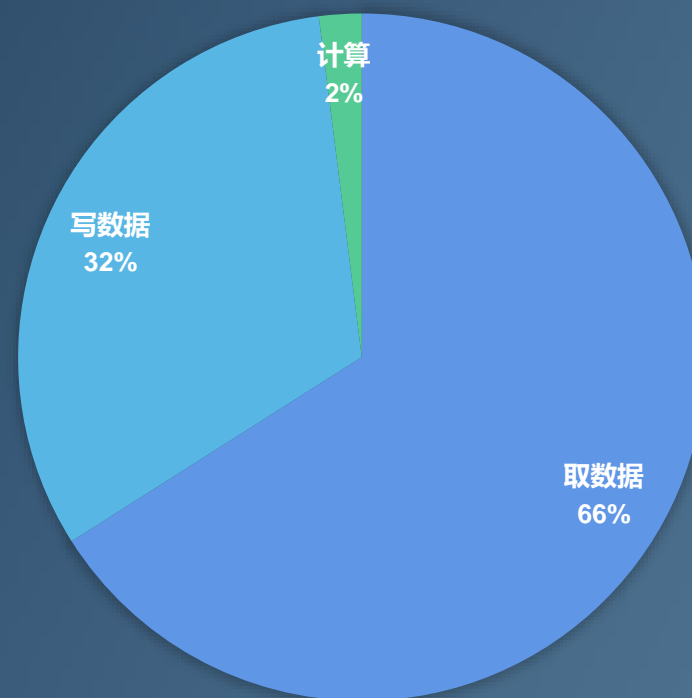
冯诺依曼体系与现代计算机



指令执行阶段所需时间

阶段	所需周期
取指	100
译码	>1
访存	100
执行	1~5
写回	100

你的计算机在干什么



扩展知识：线程切换

阶段	所需周期
取指	100
译码	>1
访存	100
执行	1~5
写回	100

阶段	所需周期
CPU 进程上下文切换	20ms(时间片切换) 1000周期(重建缓存)
CPU 线程上下文切换	10~100周期 (切换寄存器组)
Python 加法	100~1000
访硬盘（机械）	4ms(磁盘寻道)
访硬盘（固态）	0.002ms

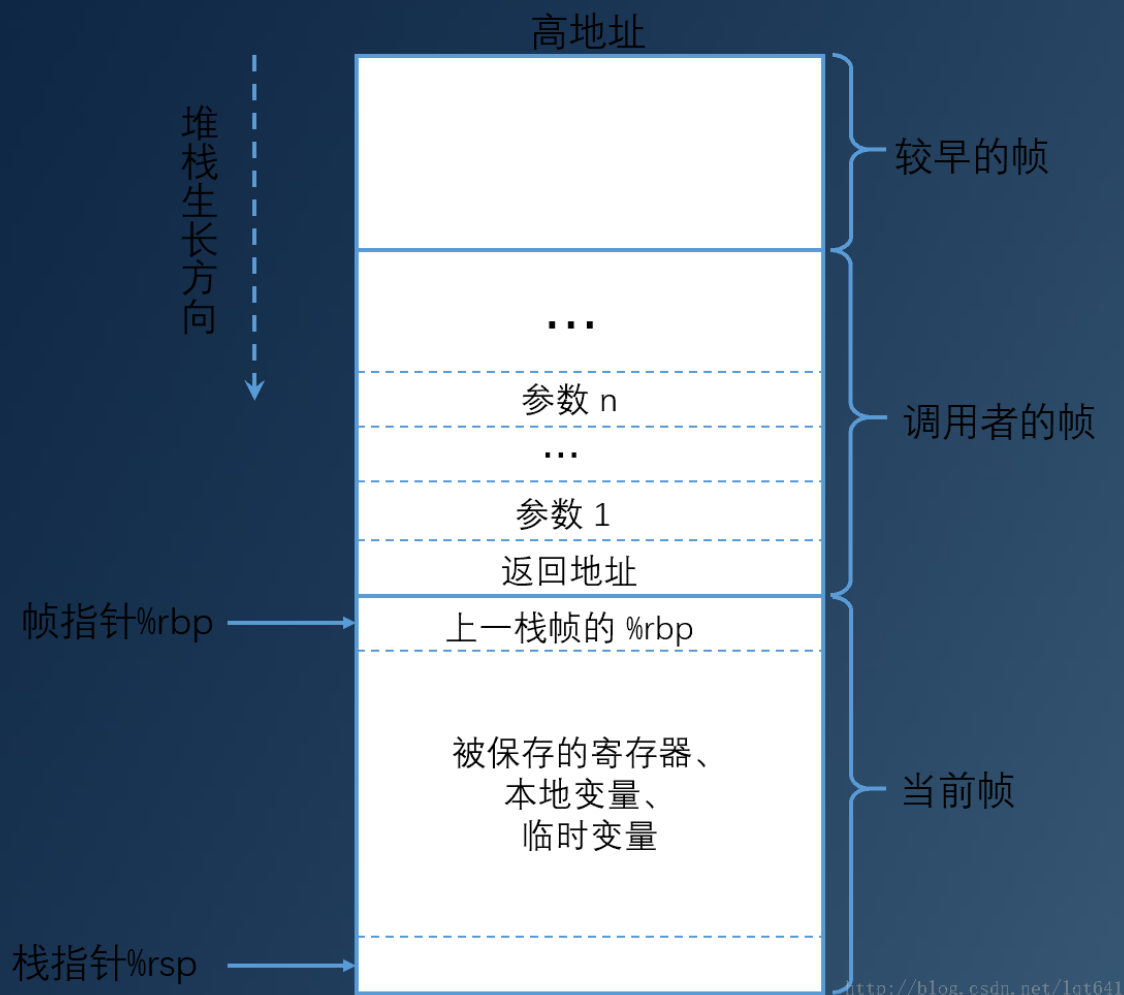
1ms = 100000 周期

扩展知识：汇编

```
int square(int num) {  
    int product = num * num;  
    return product;  
}
```

```
square(int):  
    push    rbp  
    mov     rbp, rsp  
    mov     DWORD PTR [rbp-20], edi  
    mov     eax, DWORD PTR [rbp-20]  
    imul    eax, eax  
    mov     DWORD PTR [rbp-4], eax  
    mov     eax, DWORD PTR [rbp-4]  
    pop     rbp  
    ret
```

扩展知识：汇编



square(int):

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-20], edi
mov     eax, DWORD PTR [rbp-20]
imul    eax, eax
mov     DWORD PTR [rbp-4], eax
mov     eax, DWORD PTR [rbp-4]
pop     rbp
ret
```

扩展知识：编译优化 -O3

```
int square(int num) {  
    int product = num * num;  
    return product;  
}
```

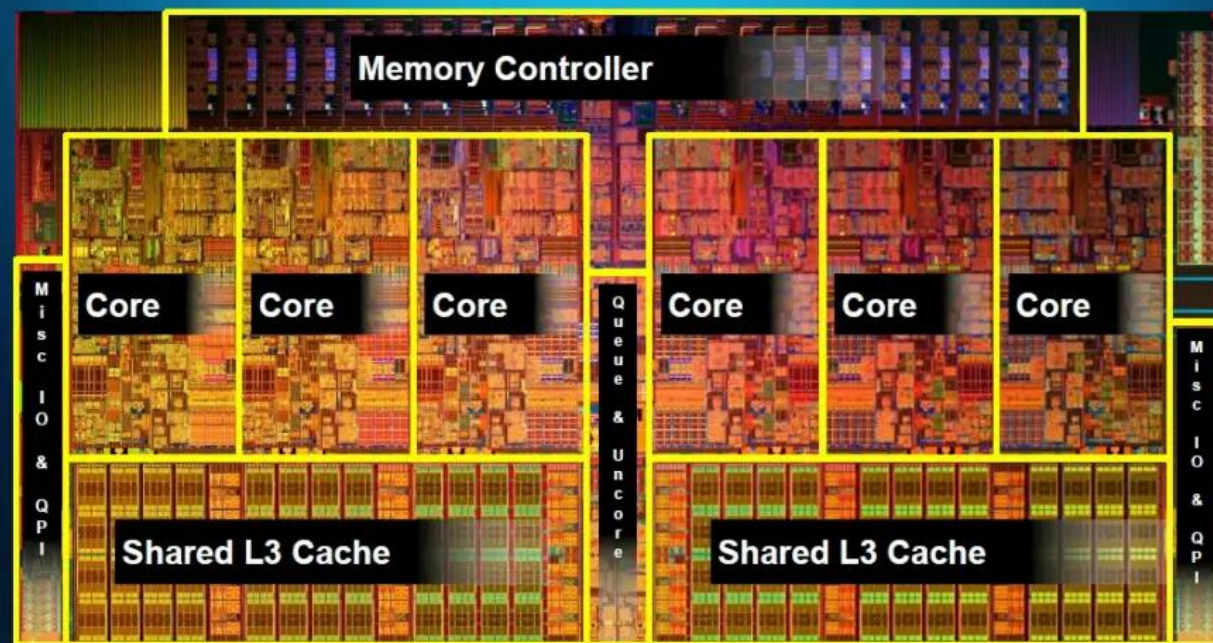
```
square(int):  
    mov     eax, edi  
    imul    eax, edi  
    ret
```

CPU、GPU与FPGA(ASIC)

CPU型号	渲染分数	GPU型号	渲染分数
AMD Ryzen Threadripper 2990WX 32-Core Processor	473.63	NVIDIA GeForce RTX 3090	5588.12
AMD Ryzen 9 5950X 16-Core Processor	460.37	NVIDIA GeForce RTX 3080 Ti	5358.97
AMD Ryzen 9 3950X 16-Core Processor	435.6	NVIDIA GeForce RTX 3080	5130.48
Intel Core i9-7980XE CPU @ 2.60GHz	423.67	NVIDIA GeForce RTX 3070	3419.82
AMD Ryzen 9 3950X 16-Core Processor	387.57	NVIDIA GeForce RTX 2080	2754.35
12th Gen Intel Core i9-12900K	367.42	NVIDIA GeForce RTX 3060	2445.18
12th Gen Intel Core i9-12900KF	356.69	NVIDIA GeForce RTX 2070	2170.22
12th Gen Intel Core i7-12700K	321.1	NVIDIA GeForce GTX 1080 Ti	863.46
AMD Ryzen 9 3900X 12-Core Processor	281.08	NVIDIA GeForce GTX 1660 Ti	820.82
AMD Ryzen Threadripper 2950X 16-Core Processor	271.68	Apple M1 Max (GPU)	702.87
AMD Ryzen Threadripper 1950X 16-Core Processor	270.69		

专用芯片加速你的计算

Intel® Core™ i7-980X Processor Die Map *32nm Westmere High-k + Metal Gate Transistors*



Transistor count: 1.17B
Die size: 248mm²



Copyright © 2010, Intel Corporation. All rights reserved. INTEL CONFIDENTIAL

谁的执行速度更快？

例程1（访存密集型）：

```
int* a = new int[1000000];
```

```
int* b = new int[1000000];
```

```
int* sum = new int[1000000];
```

```
for (int i = 0; i < 1000000; i++)
```

```
    sum[i] = a[i] + b[i];
```

例程2（计算密集型）：

```
int* m1 = new int[100];
```

```
int* m2 = new int[100];
```

```
int* m3 = new int[100];
```

```
for (int i = 0; i < 100; i++)
```

```
for (int j = 0; j < 100; j++)
```

```
for (int k = 0; k < 100; k++)
```

```
    m3[k] += (m1[i] + m2[k]);
```


神经网络里的算子

计算密集型算子

Conv, MatMul, BatchedMatMul, Gemm, LSTM,
ConvTranspose

运算复杂度 $O(n^2) \sim O(n^3)$

访存密集型算子

Relu, Relu6, Tanh, Sigmoid, Softmax, Swish, Add,
Mul, Sub, Div, CumSum, Depthwise Conv, Slice,
Transpose, Permute, Concat, Shuffle, ...

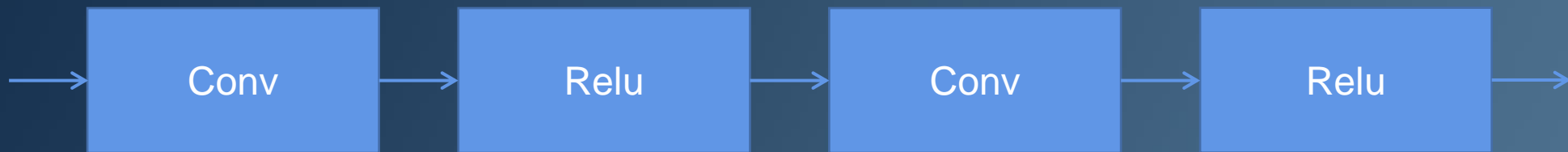
运算复杂度 $O(n^1) \sim O(n^2)$

算子：神经网络的最小组成单元

算子是一个函数空间到函数空间上的映射 $O: X \rightarrow X$ 。

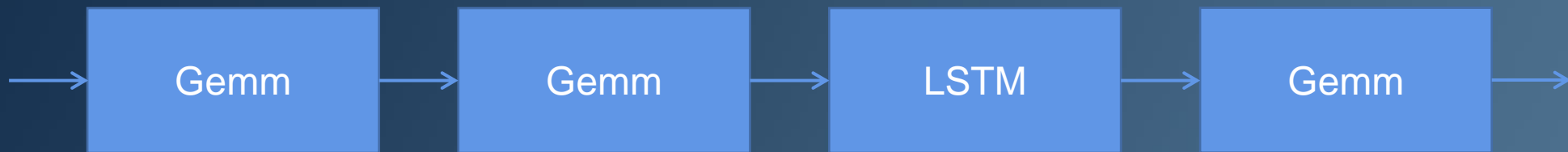
广义上的算子可以推广到任何空间，如内积空间等。

在神经网络中，算子表示一个独立的计算过程，**同时算子也是调度和运算的最小独立单位**。一个网络总是可以被表示成由算子构成的**有向无环图**。



扩展知识：LSTM算子

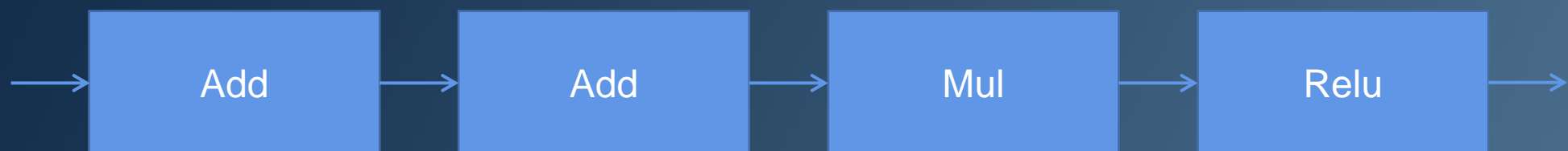
LSTM算子的计算过程中带有循环，因此将其引入到计算图中时，将破坏有向无环图的定义，为此我们总是将其视作一个完整的算子，从而避开了将循环引入计算图的窘境。



融合算子以加速访存

假定要求的计算过程为： $y = \text{Relu}((x + 1 + 2) * 2)$

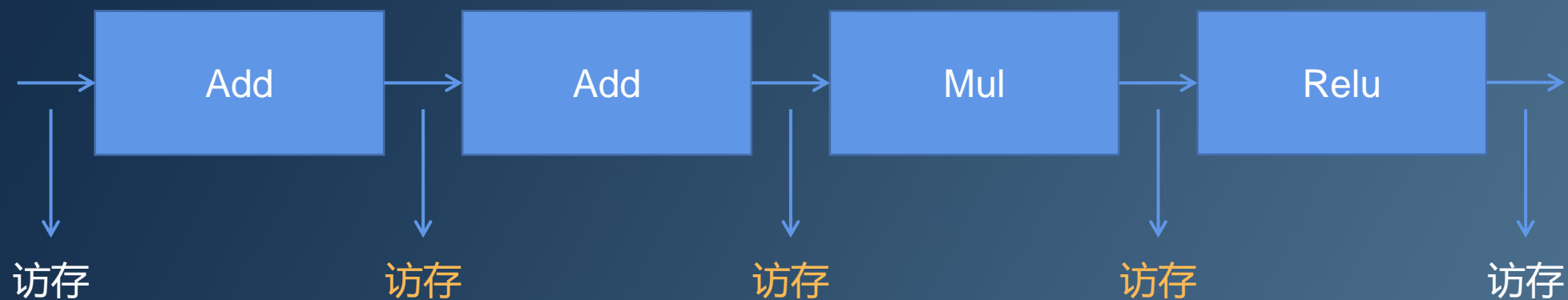
生成的神经网络为：



融合算子以加速访存

假定要求的计算过程为： $y = \text{Relu}((x + 1 + 2) * 2)$

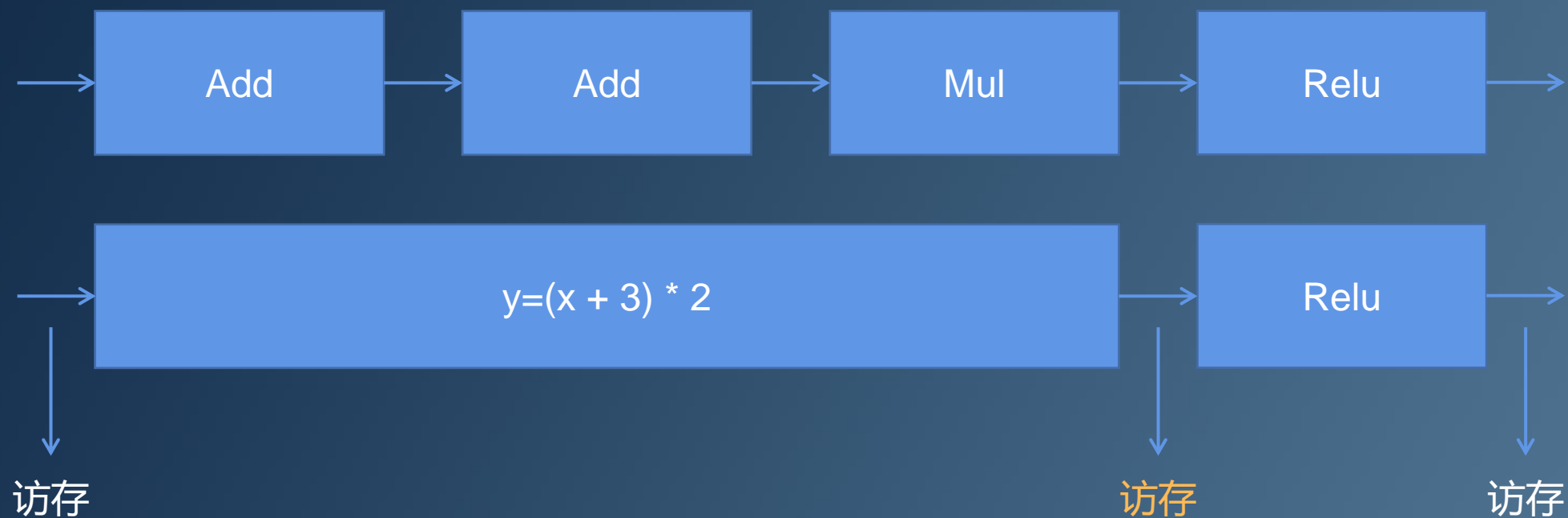
生成的神经网络为：



融合算子以加速访存

假定要求的计算过程为： $y = \text{Relu}((x + 1 + 2) * 2)$

融合算子，将减少中间结果的访存



SIMD加速

我们总是可以将操作数强制转换为整数，从而降低访存需求，甚至提升计算效率。

使用量化加速你的计算

神经网络部署加速

例程1 (矩阵相乘) :

```
void Matmul(const float** A, const float** B,
float **C, const int num_of_element) {
for (int i = 0; i < num_of_element; i++)
for (int j = 0; j < num_of_element; j++)
for (int k = 0; k < num_of_element; k++)
    C[i][j] += A[i][k] * B[k][j];
}
```

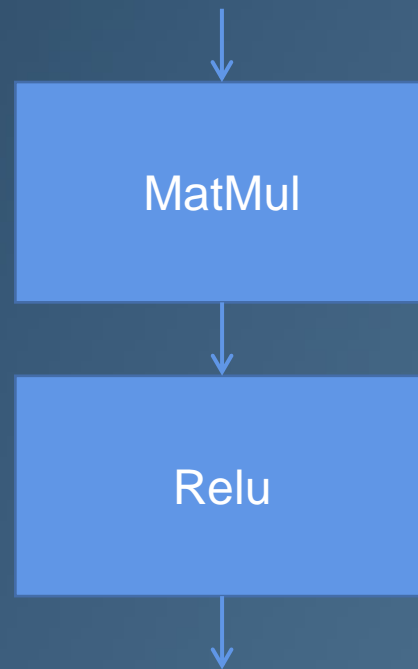
例程2 (Relu 激活) :

```
void Relu(float** C, const int num_of_element) {
for (int i = 0; i < num_of_element; i++)
for (int j = 0; j < num_of_element; j++)
    C[i][j] = 0 ? C[i][j] < 0 : C[i][j];
}
```

图融合加速

例程1 (矩阵相乘 + Relu 合体) :

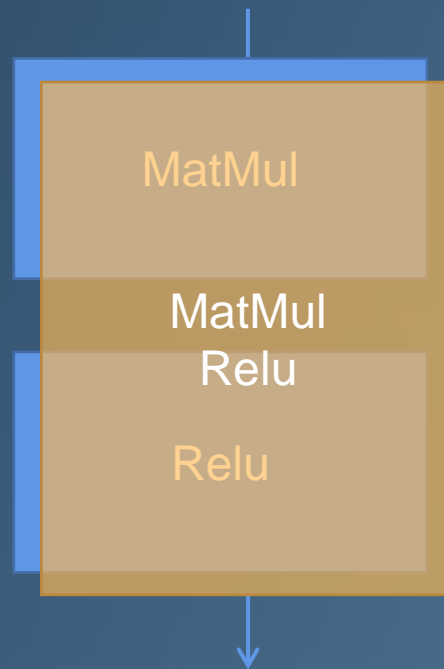
```
void Matmul(const float** A, const float** B,  
float **C, const int num_of_element) {  
    for (int i = 0; i < num_of_element; i++){  
        for (int j = 0; j < num_of_element; j++){  
            for (int k = 0; k < num_of_element; k++){  
                C[i][j] += A[i][k] * B[k][j];  
            }  
            C[i][j] = 0 ? C[i][j] < 0 : C[i][j];  
        }  
    }  
}
```



图融合加速

例程1 (矩阵相乘 + Relu 合体) :

```
void Matmul(const float** A, const float** B,
float **C, const int num_of_element) {
for (int i = 0; i < num_of_element; i++){
    for (int j = 0; j < num_of_element; j++){
        for (int k = 0; k < num_of_element; k++){
            C[i][j] += A[i][k] * B[k][j]; //中间结果可以
            保存在缓存当中
        }
        C[i][j] = 0 ? C[i][j] < 0 : C[i][j];
    }
}
```



FP16, INT8 加速

例程1 (int8 矩阵相乘 + Relu 合体) :

```
void Matmul(const char** A, const char** B, char
**C, const int num_of_element) {
    for (int i = 0; i < num_of_element; i++){
        for (int j = 0; j < num_of_element; j++){
            for (int k = 0; k < num_of_element; k++){
                C[i][j] += A[i][k] * B[k][j];
            }
            C[i][j] = 0 ? C[i][j] < 0 : C[i][j];
        }
    }
}
```

INT8 量化

例程1 (int8 矩阵相乘 + Relu 合体) :

```
void Matmul(const char** A, const char** B, char
**C, const int num_of_element) {
    for (int i = 0; i < num_of_element; i++){
        for (int j = 0; j < num_of_element; j++){
            for (int k = 0; k < num_of_element; k++){
                C[i][j] += A[i][k] * B[k][j]; // 可能溢出
                // 并且只能表达 [-128, 127]
            }
            C[i][j] = 0 ? C[i][j] < 0 : C[i][j];
        }
    }
}
```

INT8 量化POLICY

模式1 (对称量化) :

```
float value = 1.3; float scale = 0.1;  
int qt32 = round(value / scale);  
char qt8 = clip(qt32, -128, 127);
```

模式3 (Power of 2) :

```
float value = 1.3; int shift = 4;  
int qt32 = round(value << shift);  
char qt8 = clip(qt32, -128, 127);
```

模式2 (非对称量化) :

```
float value = 1.3; float scale = 0.1;  
int qt32 = round(value / scale) - offset;  
unsigned char qt8 = clip(qt32, 0, 255);
```

模式4 (指数量化) :

欢迎自行了解!

INT8 术语

```
float value = 1.3; float scale = 0.1;  
int qt32 = round(value / scale) - offset;  
char qt8 = clip(qt32, MIN, MAX);  
float dequant = (qt8 + offset) * scale;
```

scale - 尺度缩放因子

offset - 偏移量

MIN, **MAX** - 截断值

qt8 - 量化值

dequant - 反量化值

shfit - 定点位

INT8 量化POLICY 2

模式1 (PerTensor 量化) :

FP	SCALE	OFFSET	QT8
-0.3	0.1	0	-3
2.1			21
13.2			127
15.7			127

模式2 (PerChannel 量化) :

FP	SCALE	OFFSET	QT8
-0.3	0.1	0	-3
2.1			21
13.2	1	0	13
15.7			15

问题：

1. 设 n 阶方阵 $M = \begin{bmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{n1} & \cdots & M_{nn} \end{bmatrix}$, M 中元素独立同

分布且 $M_{ij} \sim N(0,1)$, 设 M 矩阵中绝对值最大的元素为

α , 以 $scale = \frac{|\alpha|}{128}$, $offset = 0$ 对方阵 M 进行量化与

反量化, 记反量化后的矩阵 M 为 M' , 计算量化噪声能

量与信号能量的比值的数学期望:

$$E_M\left(\frac{\sum_{i,j}(M'_{ij} - M_{ij})^2}{\sum_{i,j} M_{ij}^2}\right)$$

问题：

2. 设 n 阶方阵 $M = \begin{bmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{n1} & \cdots & M_{nn} \end{bmatrix}$, M 中元素独立同

分布且 $M_{ij} \sim N(0,1)$, 将所有 M 中的负数元素截断到0,

设 M 矩阵中绝对值最大的元素为 α , 以 $scale =$

$\frac{\alpha}{128}$, $offset = 0$ 对方阵 M 进行量化与反量化, 记反量

化后的矩阵 M 为 M' , 计算量化噪声能量与信号能量的

比值的数学期望:

$$E_M\left(\frac{\sum_{i,j}(M'_{ij} - M_{ij})^2}{\sum_{i,j} M_{ij}^2}\right)$$