

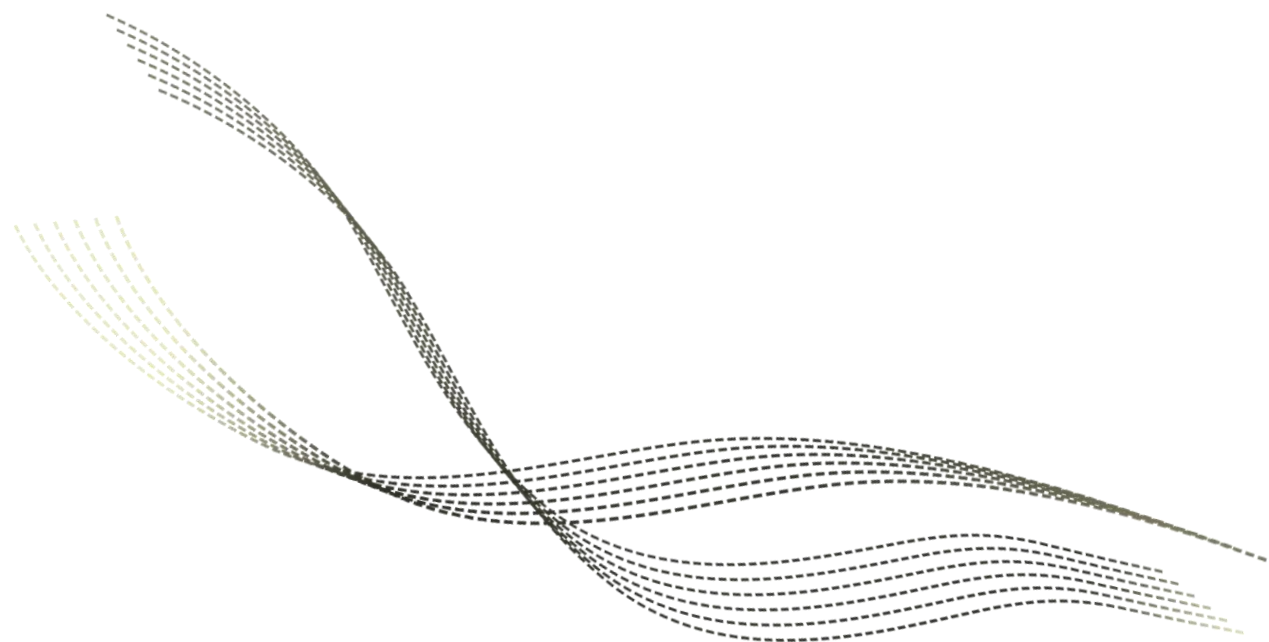
# 高级计算机体系结构

Advanced Computer Architecture

数据并行与GPGPU

---

沈明华



# 目录

CONTENTS

01

数据并行

02

向量处理器

03

GPGPU

# PART 01

## 数据并行

# ■ 背景

---

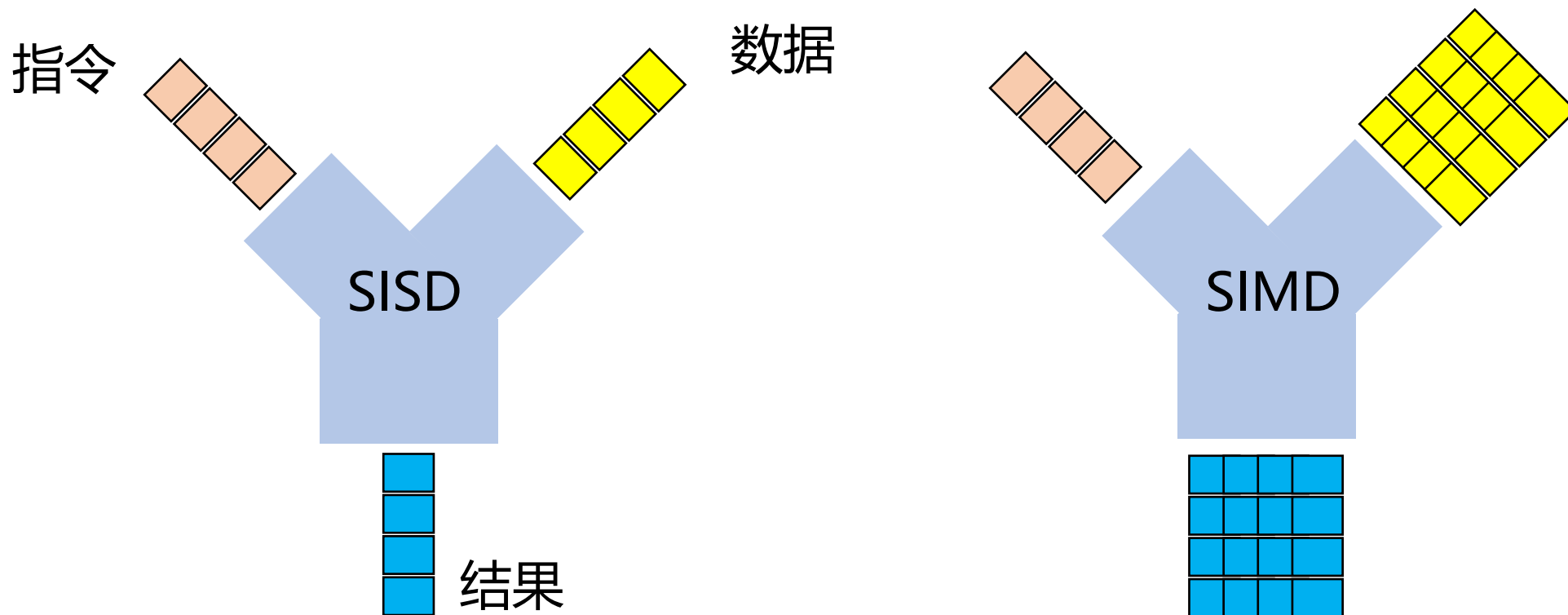
## □ 人们评价一个计算系统的性能有两个角度

- 一是这个计算系统能以多快的速度完成一个任务
  - 如今摩尔定律的困境似乎告诉我们速度已经达到一个上限
- 二是这个计算系统在一定时间内能完成多少个任务
  - 为了进一步提高计算系统的性能，人们开始研究多核处理器
  - GPU是多核处理器的一种极致体现
    - 人们把GPU的核心数量堆到极致
    - 为了在硅片上放下如此之多的核心，人们又把GPU上的核心简化到极致，以至于不能独立执行指令，需要以warp为单位执行

# ■ 数据并行

## □ 数据并行(DLP, data level parallel)

- 并行性来源于对大量数据的同时操作，而不是多线程
- 对大型科学/工程任务十分有用



# ■ 数据并行

---

## □ SIMD优势

- 适合一些数据并行的应用场景
  - 以矩阵计算为主要需求的科学计算
  - 以音视频编码解码为需求的多媒体应用
- 对比MIMD架构在能耗上更有优势
  - 计算多个数据只需取一次指令，比MIMD能效高
- 允许程序员继续以串行思维设计程序

# PART 02

## 向量处理器

# ■ 向量操作

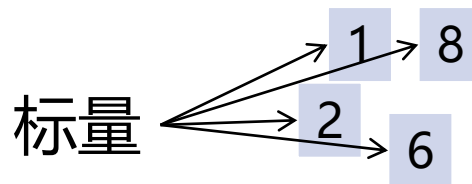
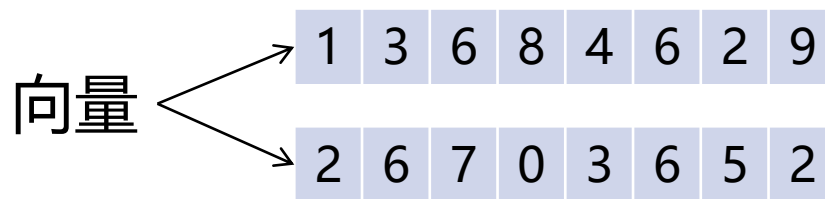
## □ 向量是一个关于数字(numbers)或标量(scalars)的数组

### – 向量处理器需要支持以下操作

- 读取一组数据元素
- 同时操作这一组数据元素

### – 向量指令包含以下关键优势

- 减少指令读取所需带宽
- 减少检测数据冒险所需硬件电路
  - 同一个向量内，数据互相独立
- 降低内存访问延迟
  - 向量指令的内存访问相对整齐、可预测





# ■ 向量处理器架构

---

## □ 向量处理器分为两类主要的架构

### – 内存-内存的向量处理器

- 所有操作是从内存读取，最后直接写入内存
- 例如CDC STAR-100(在当时比CDC 7600快很多)

### – 内存-寄存器的向量处理器

- 使用load-store类型的架构，显式读取和写入内存
- 例如Fujitsu的A64FX处理器
- Cray、Hitachi、NEC提供的超级计算机用处理器

CDC STAR-100



# ■ 向量处理器架构

---

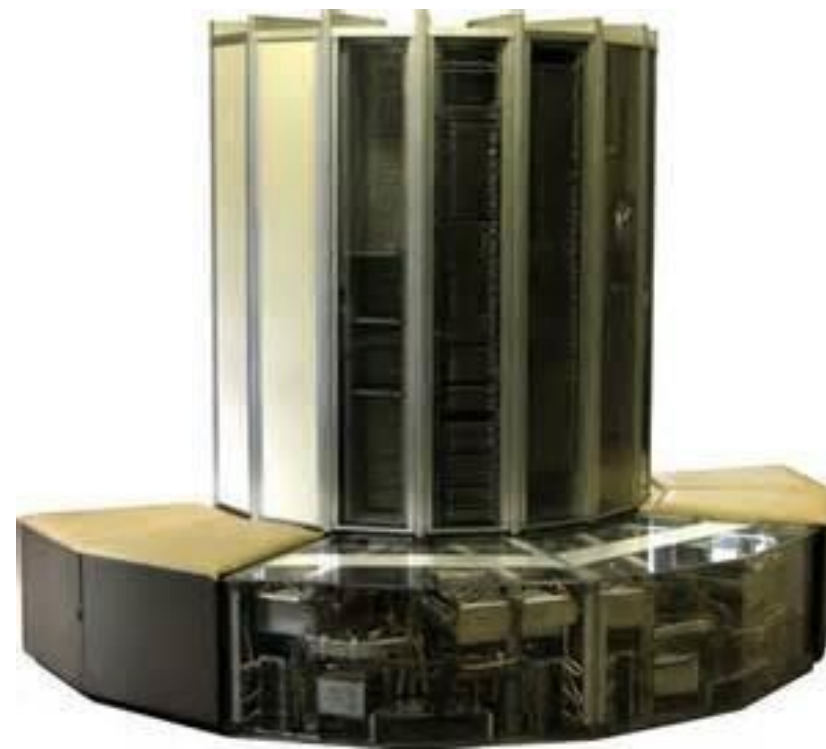
## □ 关键组件

- 向量处理器包含向量组件和普通的标量流水线组件，其中向量组件有
  - 向量寄存器
    - 固定大小，可以存储一个向量
    - 通常是64~128个浮点数
    - 决定向量处理器的最大向量长度
  - 向量寄存器堆
    - 包含8~32个向量寄存器

# ■ 向量处理器设备

## □ 早期使用向量处理器的计算机：Cray-1

- 1975年发布
  - 最早拿出符合超级计算机定义的实物，建立Cray公司的Seymour Cray因此被称为超级计算机之父
- 机器重达5.5吨
- 耗电115KW，提供160MFLOPS的算力
- 8个64元素的向量寄存器
  - 每个元素64bit，每个寄存器容量达4096个bit
  - 向量寄存器堆存储容量为4KB
- 支持向量操作
  - 向量+向量，向量+标量

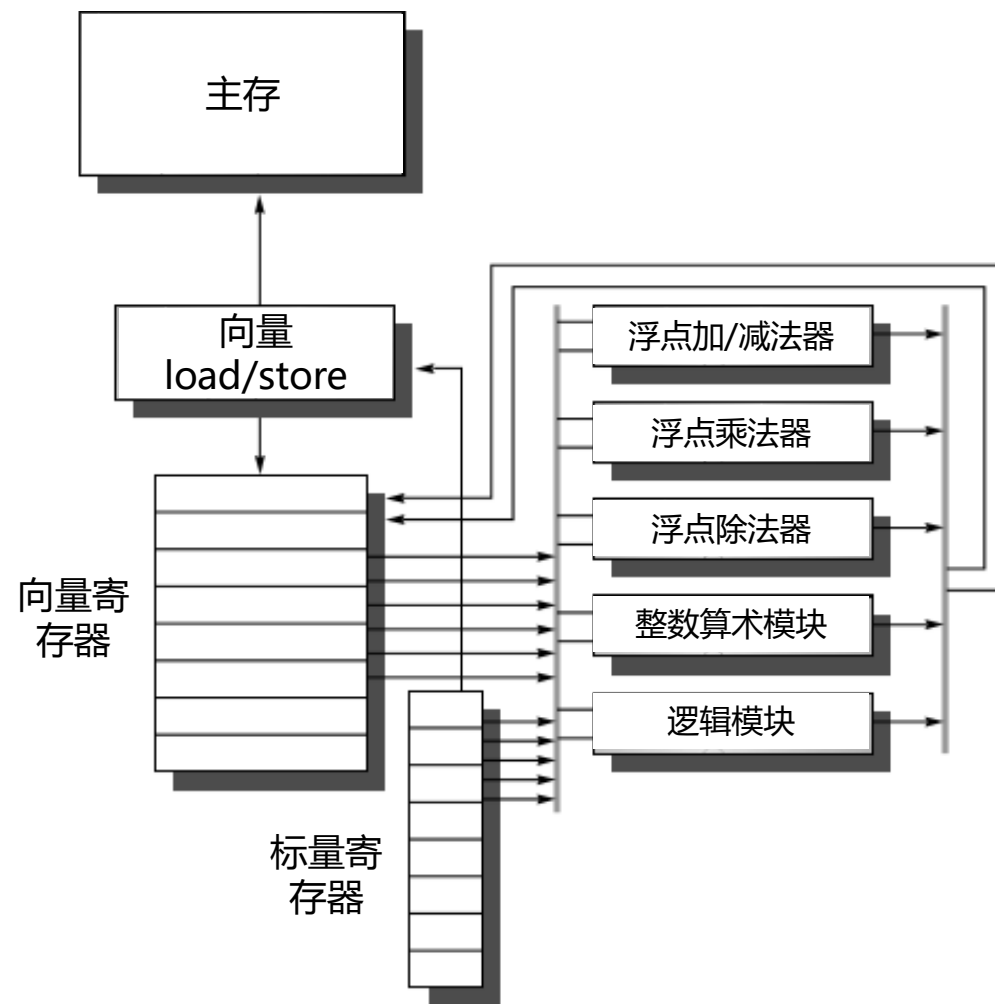


Cray-1

# ■ 向量处理器学习案例

## □ 以VMIPS指令集为例

- 向量寄存器
  - 每个寄存器可存储64个元素
- 向量操作元件(FU)
  - 5个流水线化的FU，带冒险检测
- 向量加载元件(load-store unit)
  - 加载/存储一个向量
  - 每周期读写一个字(word)的数据



# ■ 向量处理器学习案例

---

## □ VMIPS为例

- 两个向量相加指令：ADDVV.D V1, V2, V3
  - 把V2和V3两个寄存器的向量相加，结果存放到V1寄存器
- 向量和标量相加指令：ADDVS.D V1, V2, F0
  - 把标量F0加到V2，结果存放到V1寄存器
- 加载一个向量：LV V1, R1
  - 从地址R1开始读取数据，存放到V1寄存器
- 存储一个向量：SV V1, R1
  - 从地址R1开始把V1寄存器的数据写入

# ■ 向量处理器学习案例

## □ VMIPS v.s. MIPS

– 以**DAXPY**循环为例：双精度计算  $Y = a \times X + Y$

➤ 假设X和Y的起始地址为Rx和Ry，计算64个双精度的数据

### MIPS指令

L.D	F0, a	;加载标量a
DADDIU	R4, Rx, #512	;加载地址
Loop:		
L.D	F2, 0(Rx)	;加载X[i]
MUL.D	F2, F2, F0	;计算 $a \times X[i]$
L.D	F4, 0(Ry)	;加载Y[i]
ADD.D	F4, F4, F2	;计算 $a \times X + Y$
S.D	F4, 0(Ry)	;保存结果到Y[i]
DADDIU	Rx, Rx, #8	;自增x地址
DADDIU	Ry, Ry, #8	;自增y地址
DSUBU	R20, R4, Rx	;计算循环次数
BNEZ	R20, Loop	;是否退出循环

### VMIPS指令

L.D	F0, a	;加载标量a
LV	V1, Rx	;加载向量X
MULVS.D	V2, V1, F0	;向量-标量乘法指令
LV	V3, Ry	;加载向量Y
ADDV.D	V4, V2, V3	;向量-向量加法指令
SV	V4, Ry	;保存计算结果

需要大约600条指令

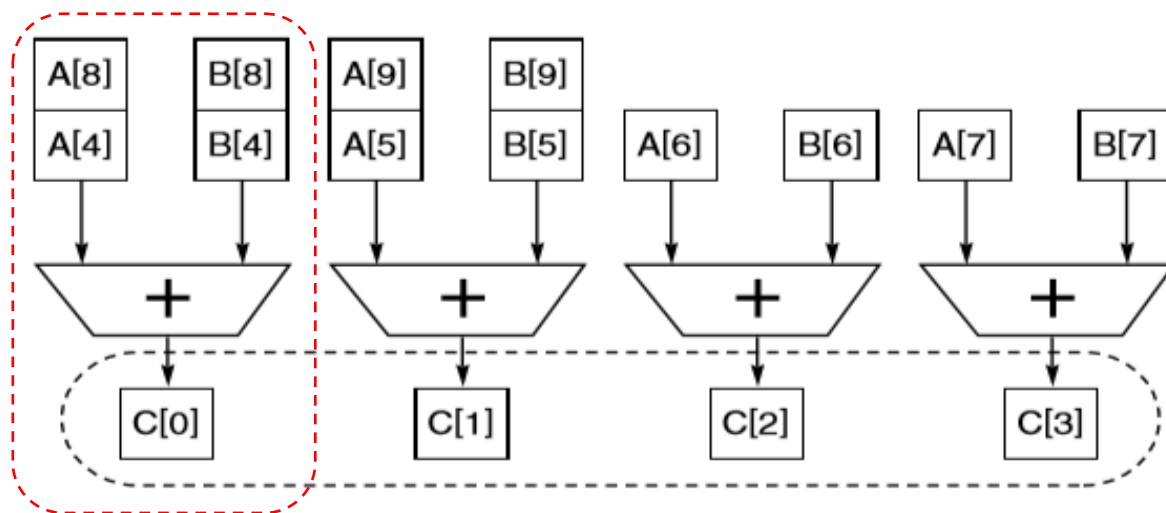
仅需6条指令  
向量操作能同时  
作用于64个元素

# ■ 向量处理器执行

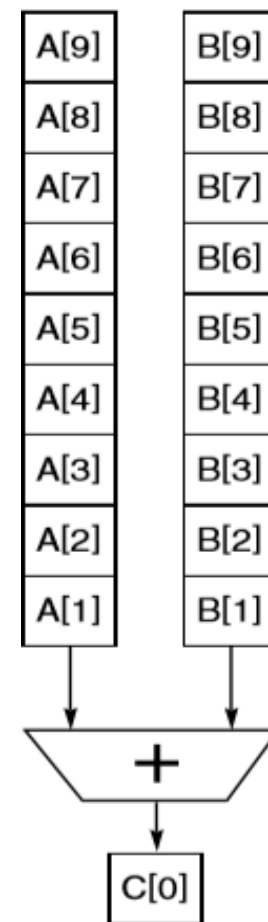
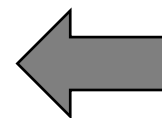
## □ 多车道(multiple lanes)执行

– 执行 ADDV C, A, B

- 向量A中的元素与向量B中对应元素 “硬连接”
- 通过多个并行车道(lanes)流水线, 在一个周期内计算多个数据



车道(lanes)



# ■ 向量处理器执行

---

## □ 向量指令耗时计算

- 向量指令的执行部分时间主要受以下条件影响
  - 操作的向量长度
  - 结构冒险
  - 数据依赖
- 其他开销
  - 流水线启动开销：取决于流水线深度
  - VMIPS的FU计算一个元素所需周期
    - 向量的执行时间正比于向量的长度

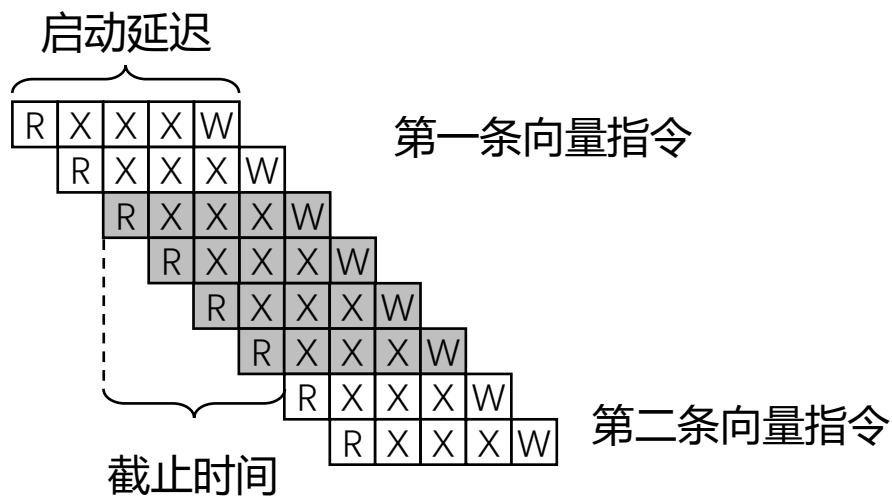


# ■ 向量处理器执行

## □ 向量指令耗时计算

### – 启动开销由两部分构成

- 启动延迟：是一个数据完全通过一个FU的时间
- 截止时间或恢复时间(dead time or recovery time)：运行下一条向量指令的间隔时间
  - 因为不同向量指令不能重叠执行



# ■ 向量处理器执行

---

## □ convoy

- 定义：可以一同发射执行的一组向量指令，指令间没有数据冒险和结构冒险

## □ initiation rate

- 定义：FU每周期消耗向量元素的速率

## □ chime

- 定义：执行一个convoy花费的近似时间
- $m$ 个convoy花费 $m$ 个Chime

# ■ 向量处理器执行

## □ 流水线启动开销

– 从指令发射到产生第一个计算结果所需时间，假定所有initiation rate都是1

- Vector load/store 12
- Vector multiply 7
- Vector add 6

L.D	F0, a	;加载标量a
LV	V1, Rx	;加载向量X
MULVS.D	V2, V1, F0	;向量-标量乘法指令
LV	V3, Ry	;加载向量Y
ADDVV.D	V4, V2, V3	;向量-向量加法指令
SV	V4, Ry	;保存计算结果

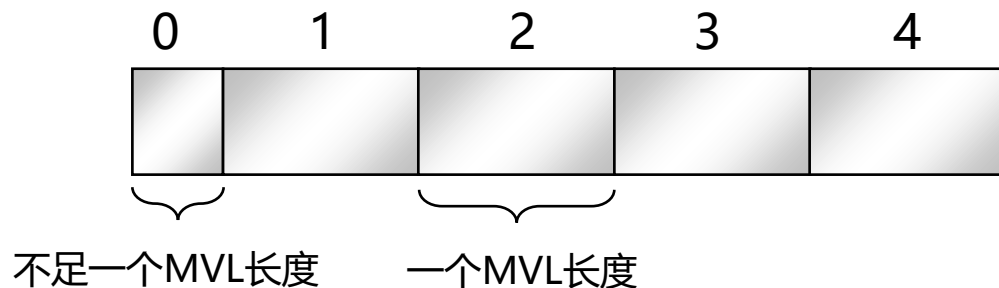
Convoy	开始周期	第一个结果	最后一个结果	备注
1. LV	0	12	11+n	加载向量X
2. MULVS	12+n	12+n+7	18+2n	计算a*X
2. LV	12+n	12+n+12	23+2n	加载向量Y
3. ADDVV	24+2n	24+2n+6	29+3n	计算X+Y
4. SV	30+3n	30+3n+12	41+4n	保存结果

这两个指令之间无依赖关系，组成一个convoy

# ■ 向量处理器执行

## □ 若向量长度超过寄存器长度

- 计算向量  $Y = a \times X + Y$  时，向量长度  $n$  在执行时才确定
  - MVL(maximum vector length)定义一次能操作的最长操作长度
- 解决方案1：用向量长度寄存器(VLR, vector length register)
  - 控制向量操作的操作长度
  - VLR大小不会超过MVL
- 解决方案2：拆分向量，碎片化计算(strip mining)
  - 第一个区间较小采用  $n \text{ MOD } \text{MVL}$ ，剩余计算按照MVL的长度操作



# ■ 向量处理器执行

---

## □ 试计算 $A = B \times s$ 所需执行周期数

- 其中A, B向量长度200, s是一个标量
- 向量寄存器长度64
- 各个部件启动周期 $T_{start}$ 
  - Vector load/store 12
  - Vector multiply 7
- 假设 $T_{loop} = 15$ 是分片操作所需周期数
- 假设 $T_{chime} = 3$ 是一个元素计算所需时间, 且只有一个车道L(Lane)
- 已知:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime} \div L$$

# ■ 向量处理器执行

---

## □ 试计算 $A = B \times s$ 所需执行周期数

– 总执行周期为784，每个元素需要 $784/200 \approx 3.9$ 个周期

- 计算过程分为三步：加载向量B，引入12周期启动延迟
- 执行 $B \times s$ 的计算，引入7周期启动延迟
- 保存向量A，引入12周期启动延迟

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime} \div L$$

$$T_{start} = 12 + 7 + 12$$

$$T_{200} = \left\lceil \frac{200}{64} \right\rceil \times (15 + 31) + 200 \times 3 \div 1 = 784$$

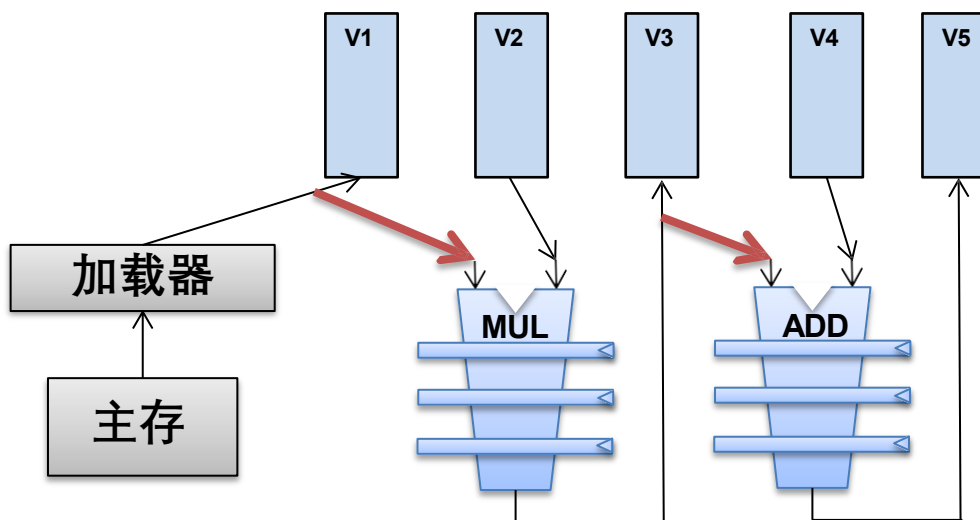
# ■ 向量处理器执行

## □ 向量链(vector chaining)

– 是向量版本的数据旁路

- 一旦所需操作数就绪，通过旁路转发数据，允许向量操作立即执行。免去访问存储器的开销

LV  
MULV    v1 → v3, v1, v2  
ADDV    v5 → v3, v4



# ■ 向量处理器执行

## □ 向量掩码寄存器(VMR, vector mask register)

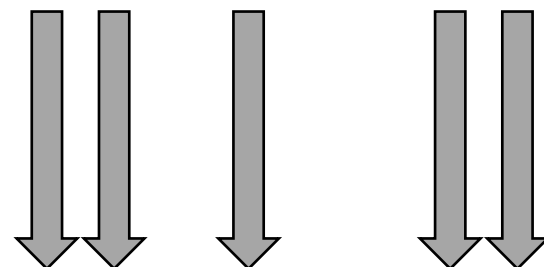
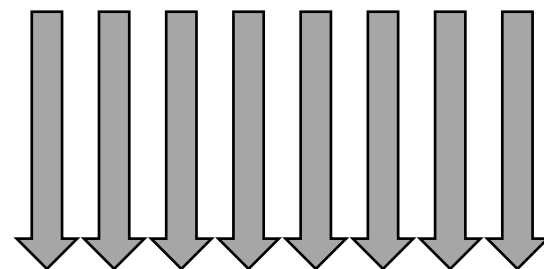
- 基于一个布尔向量
- 若向量处理过程中存在条件
- 只有对应位置是1的向量元素才会执行向量操作

```
for (int i = 0; i < 64; i++)  
    if (X[i] != 0)  
        X[i] = X[i]+Y[i]  
    else  
        ...
```

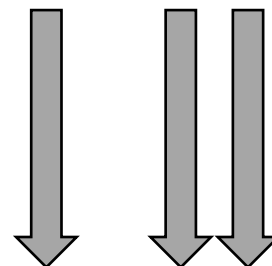


if

VMR



else





# **PART 03**

## **GPGPU**

# ■ 背景

---

## □ GPU(graphic processing unit)

- 第一个GPU于1999年诞生
- GPU天生是为重度并行化的任务
- 如今GPU不仅仅是图像处理器，还有一种通用并行处理单元
  - GPGPU(General-Purpose Graphic Processing Unit)
  - 大约在2001年前后出现，允许调用GPU进行一些计算加速
  - 操作GPU的语言由GPU厂商提供，例如NVIDIA的cuda、AMD的ROCm、华为昇腾的CANN、摩尔线程的musa等等



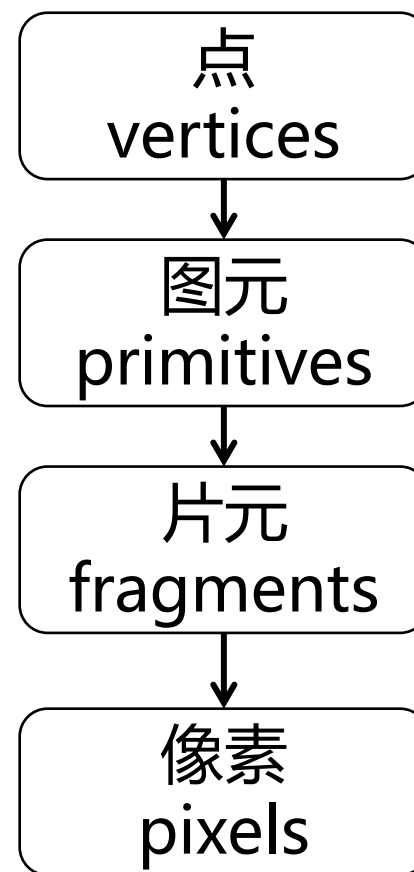
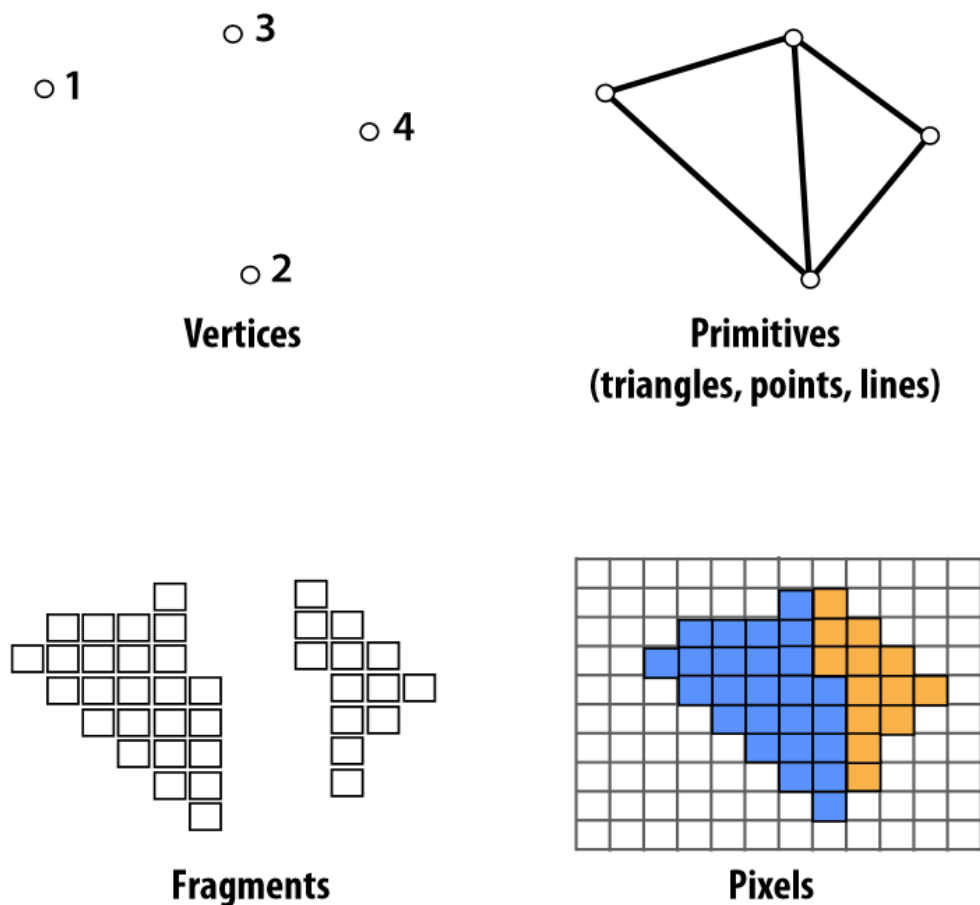
CANN 5.0



# ■ 渲染管线

## □ 图形渲染管线(graphics pipeline)

– 现代GPU的数据计算流程十分类似

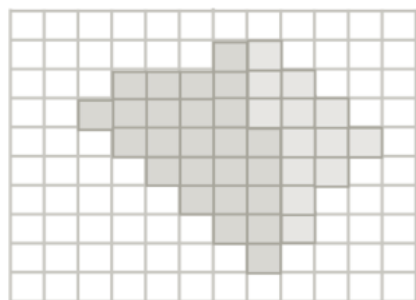


# ■ 渲染管线

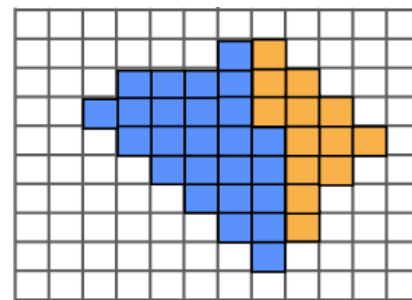
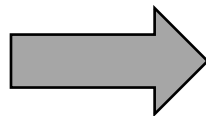
---

## □ 对像素进行相同的流式计算

- 像素着色，只是输入数据略有不同，执行完全相同的着色程序
- 非常适合大规模并行



Pixels



Pixels

## ■ 数据并行工作负载

---

- 同样是数据并行，不同架构的并行负载并不一致
  - 以对数据进行相同操作的并行计算为例

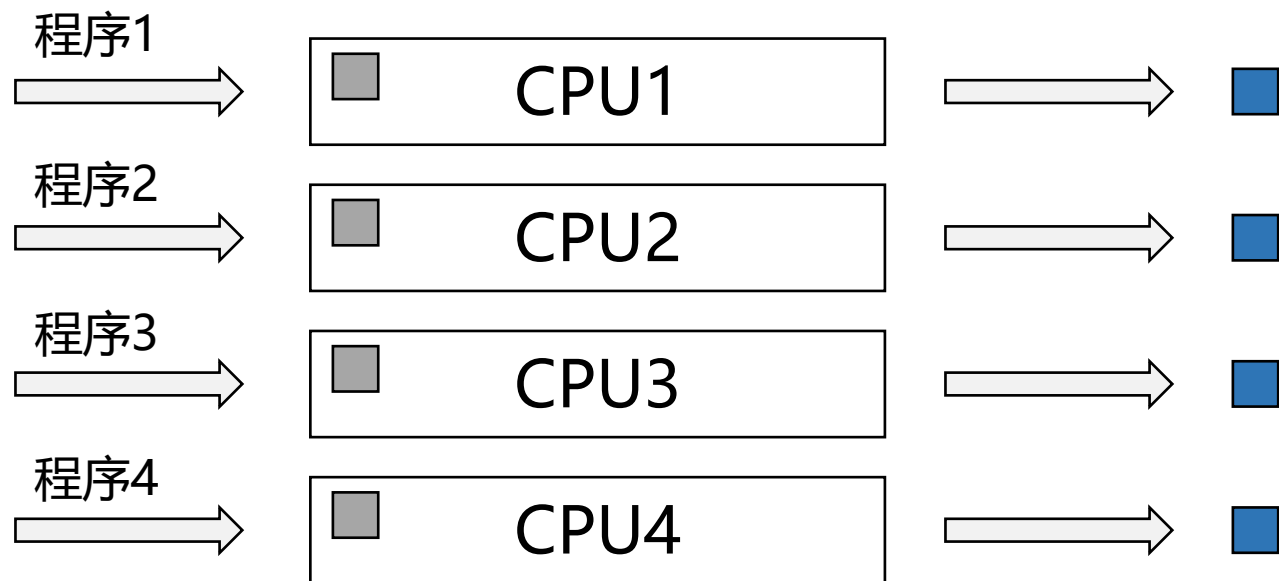


# ■ 数据并行工作负载

---

## □ MIMD系统

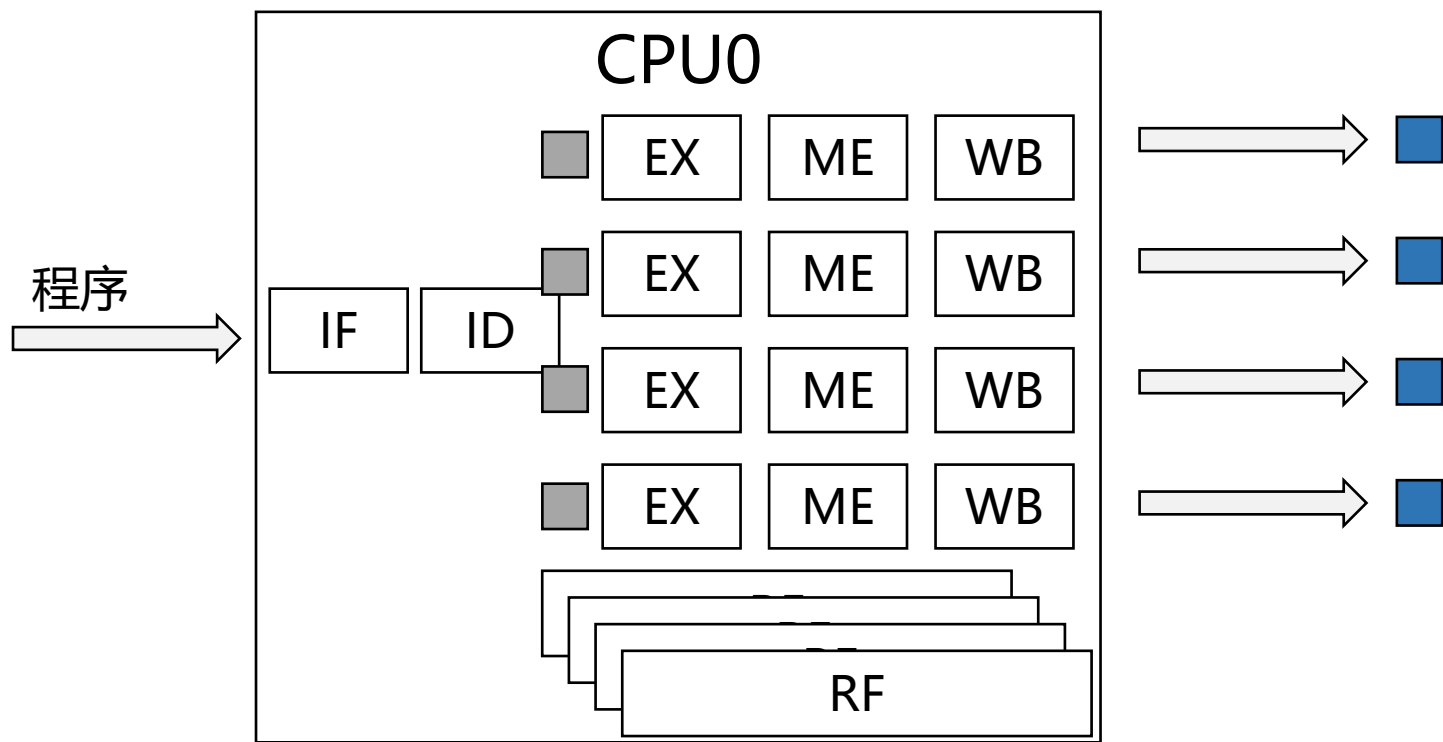
- 为每个数据创建一个任务
- 任务划分到多个不同的CPU上执行



# ■ 数据并行工作负载

## □ SIMD系统

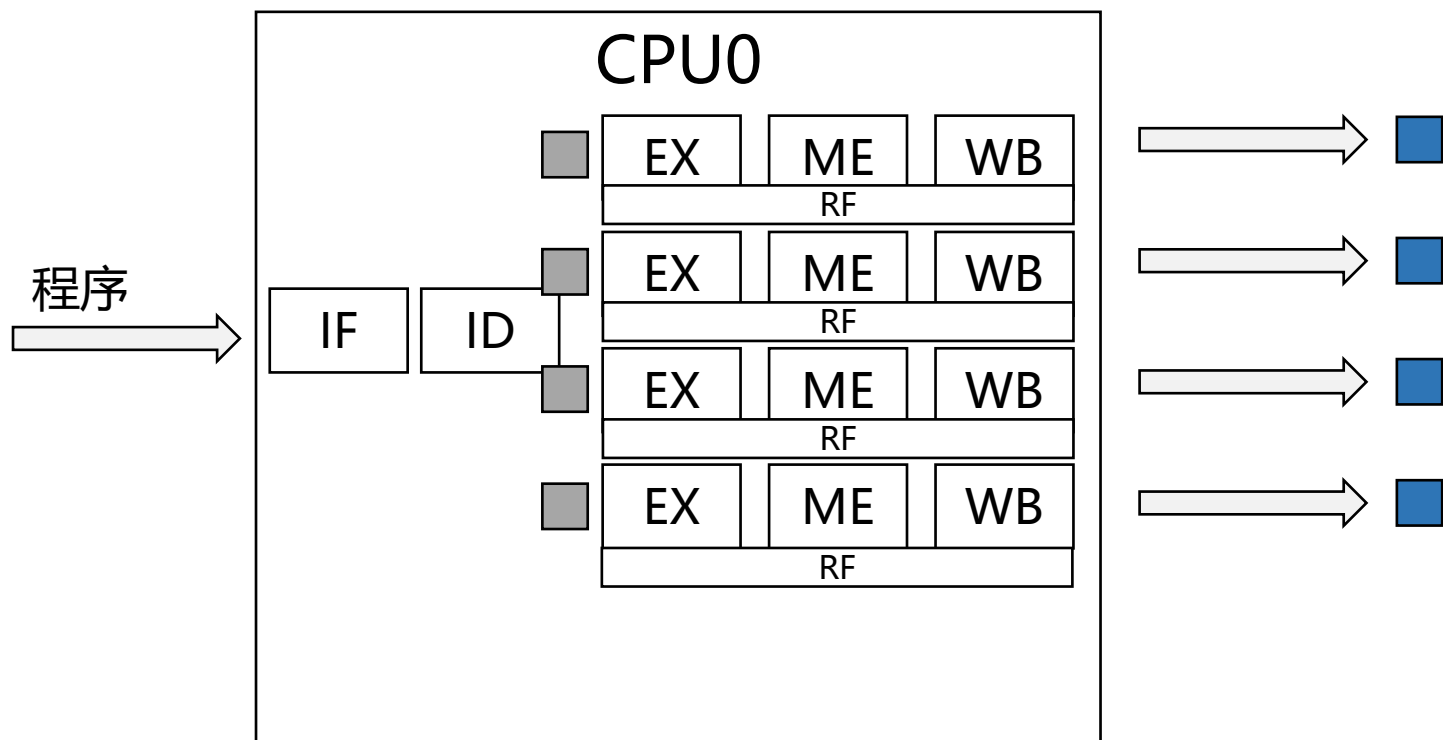
- 任务划分到多个不同的执行流，不同执行流共享寄存器堆



# ■ 数据并行工作负载

## □ SIMT系统(single instruction multiple thread)

- 每个数据由单独的线程执行操作，不共享寄存器堆，比SIMD更加灵活





# ■ GPU编程模型

---

## □ GPU进一步发展SIMT编程模型

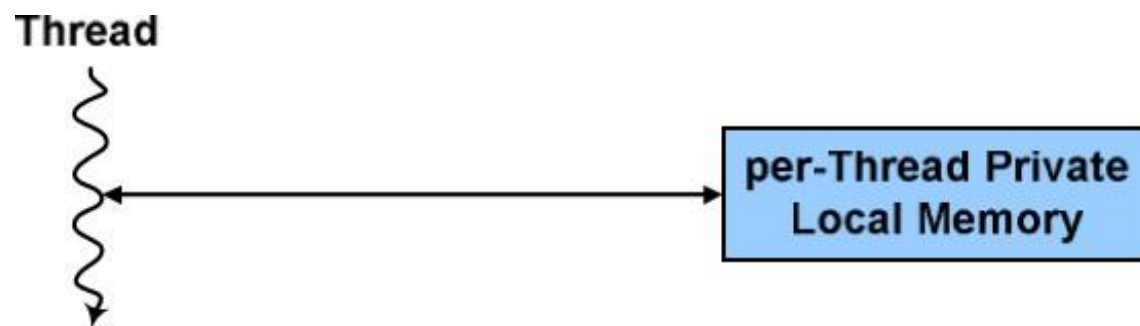
- 程序员编写一系列任务定义渲染管道的操作逻辑(称为着色器程序shader program), GPU用不同微处理器执行程序
  - vertex processors
    - 运行顶点处理程序
    - 操作点、线和三角形图元
  - fragment processors
    - 运行片元处理程序
    - 在光栅上处理填充图元纹理信息
- 可以使用cuda或OpenCL编写GPGPU可执行程序
  - 是C/C++程序一个子集
  - 告诉GPGPU在大量数据元素上如何执行操作

# ■ CUDA并行计算模型

---

## □ CUDA(compute unified device architecture)

- 是一种 HW/SW 架构, 允许NVIDIA的GPU执行C/C++编写的程序
- GPU实例化一个核心程序, 这个程序由一大堆CUDA线程组成



CUDA线程与一般POSIX线程不同  
拥有私有内存空间

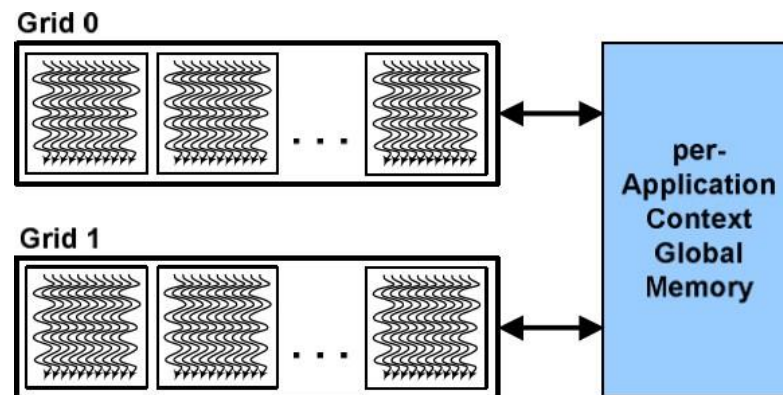
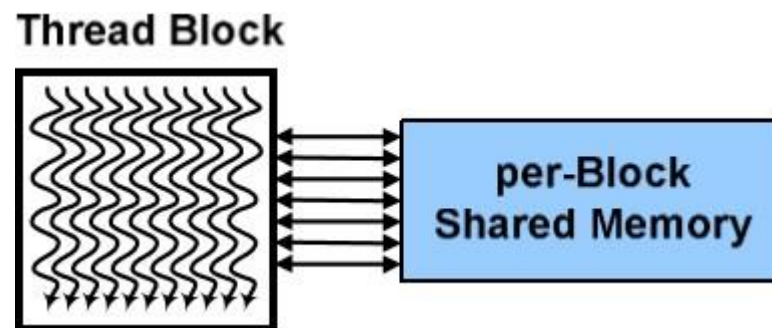
# ■ CUDA并行计算模型

## □ 线程块(thread block)

- 一个线程块是一组互相协作的线程组
  - 可以包含1~512个线程
  - 每个线程有自己独一无二的ID

## □ Grid

- 一组执行相同任务的线程块
  - 可读写全局内存



# ■ CUDA硬件模型

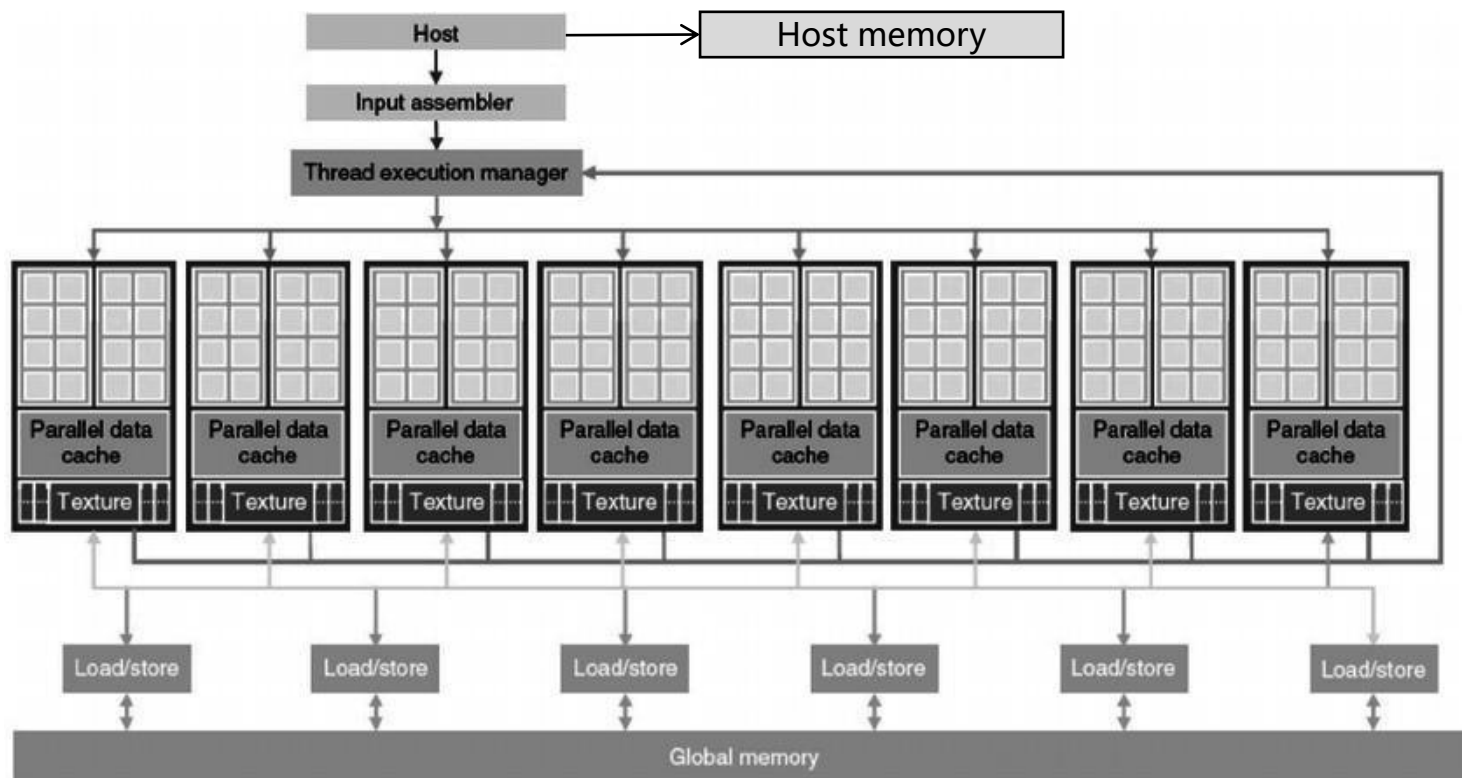
---

- GPU硬件包含一组可以执行Grid的多线程SIMD处理器
  - 有block级的调度器
    - 把block分配给一个多线程SIMD处理器执行
  - 有SIMD级调度器
    - SIMD处理器决定SIMD的指令何时运行
  - SIMD处理器的FU必须支持并行
    - 存在多条SIMD的车道(SIMD lanes)

# ■ GPU样例

## □ NVIDIA的Tesla架构

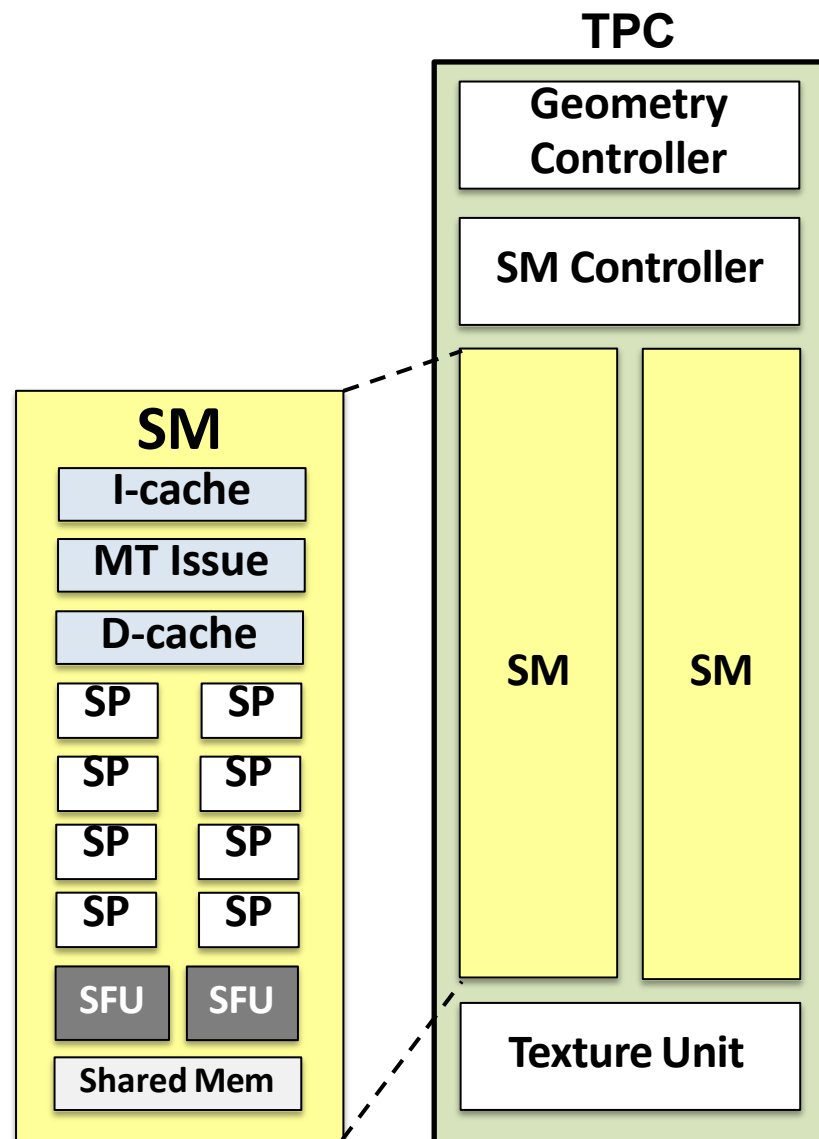
- 最大的更新是统一着色器处理器，提高硬件利用率，是NVIDIA首个GPGPU
  - 在此之前的GPU中，点计算和片元计算使用单独的处理器
  - 导致一些重点计算或重片元计算的任务不能充分利用GPU的硬件资源



# ■ GPU样例

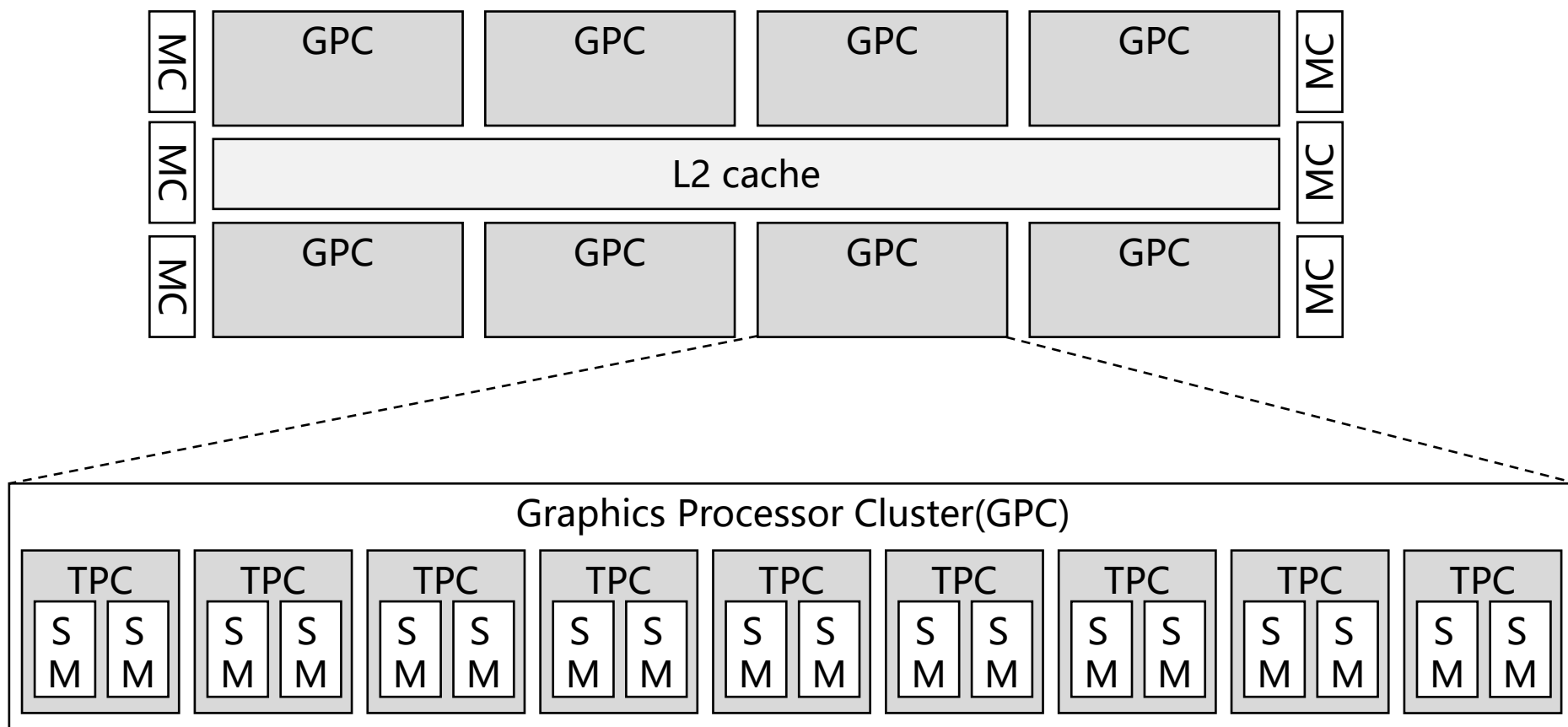
## □ NVIDIA的Tesla架构

- 8个纹理处理器集群(TPC, texture processor clusters)
- 每个TPC包含2个流式多处理器(SM, streaming multiprocessors)
  - 一个统一的图形和计算引擎，负责线程块的实际执行
- 每个SM包含8个流处理器(SP, streaming processor)核
  - 为每个CUDA线程提供一个标量ALU
  - 但SP核本身支持SIMD，是GPU最基本的处理单元



# ■ GPU样例

## □ NVIDIA的Hopper架构总览

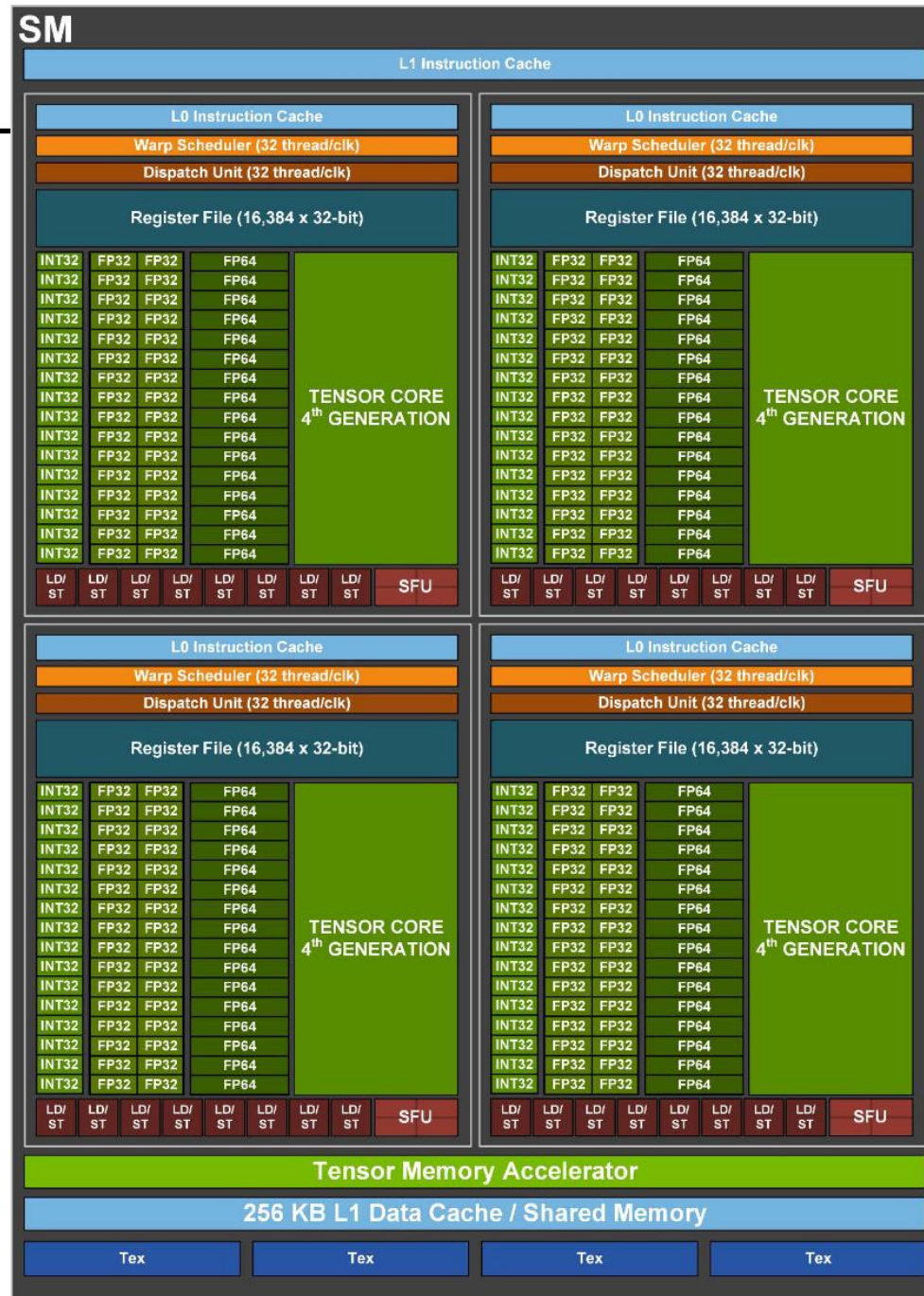
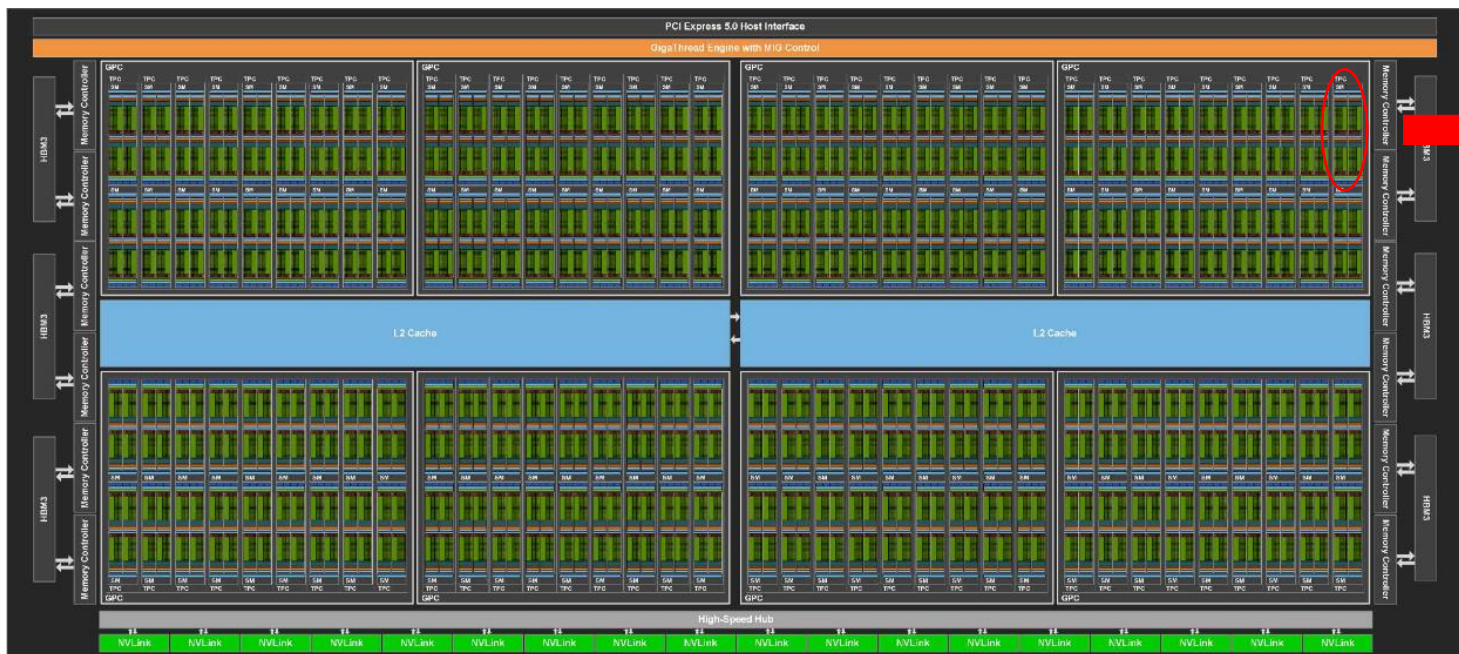




# GPU样例

## □ NVIDIA的Hopper架构——SM细节

- SM包含4个SP
- SP有16个INT32、32个FP32、16个FP64的Cuda核心。以及1个第四代Tensor core
- 以及一个拥有16384个寄存器的寄存器堆





# ■ GPU线程调度

---

## □ Warps: GPU上的线程按组调度

- GPU依赖大规模的硬件多线程来保持算术单元的利用率
- SM调度执行线程组
  - NVIDIA称这样一个线程组为 “warps”
  - AMD称之为 “wavefront”
- 线程组由32个执行相同指令的并行CUDA线程组成
  - 在相同的指令地址开始执行
  - warp内所有CUDA线程共享程序计数器(program counter)
  - warp是SM调度的最小单元

# ■ GPU线程调度

---

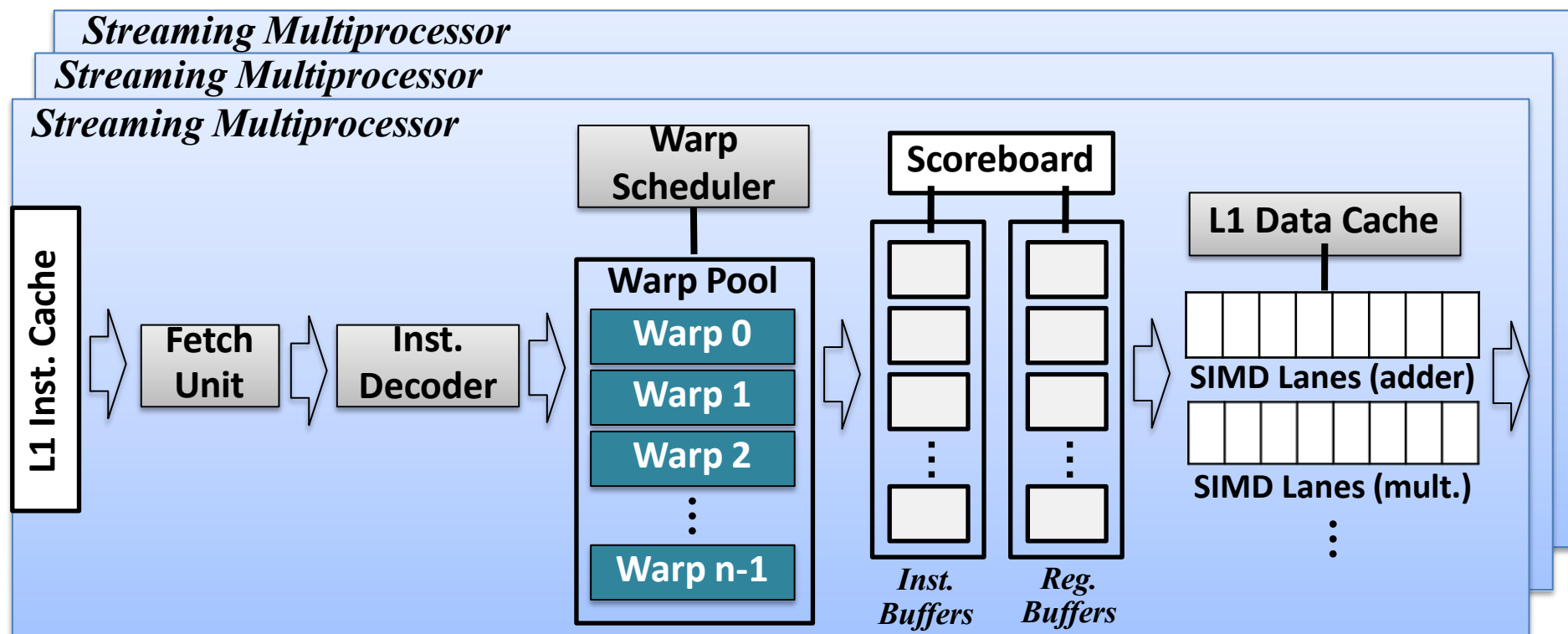
## □ Warp调度

- 调度器从线程池中选择一个warp执行
  - 调度时需要考虑指令类型及公平性等要求
- 最好占用SM上所有的SIMD lanes
  - 一个warp内的32个线程执行路径相同，没有if分支
- GPGPU采用一些实现隐藏加载延迟
  - 快速的上下文切换，当前warp访问内存时，快速切换到另一个warp继续执行
  - warp的数量巨大，有充足的待执行warp满足切换需求
  - warp之间允许乱序执行

# ■ GPU线程调度

## □ 降低warp执行速度的一些因素

- 指令/数据的缓存失效
- 结构/数据/控制冒险
- 同步原语
- 调度策略



# ■ GPU线程调度

---

## □ Branch divergence

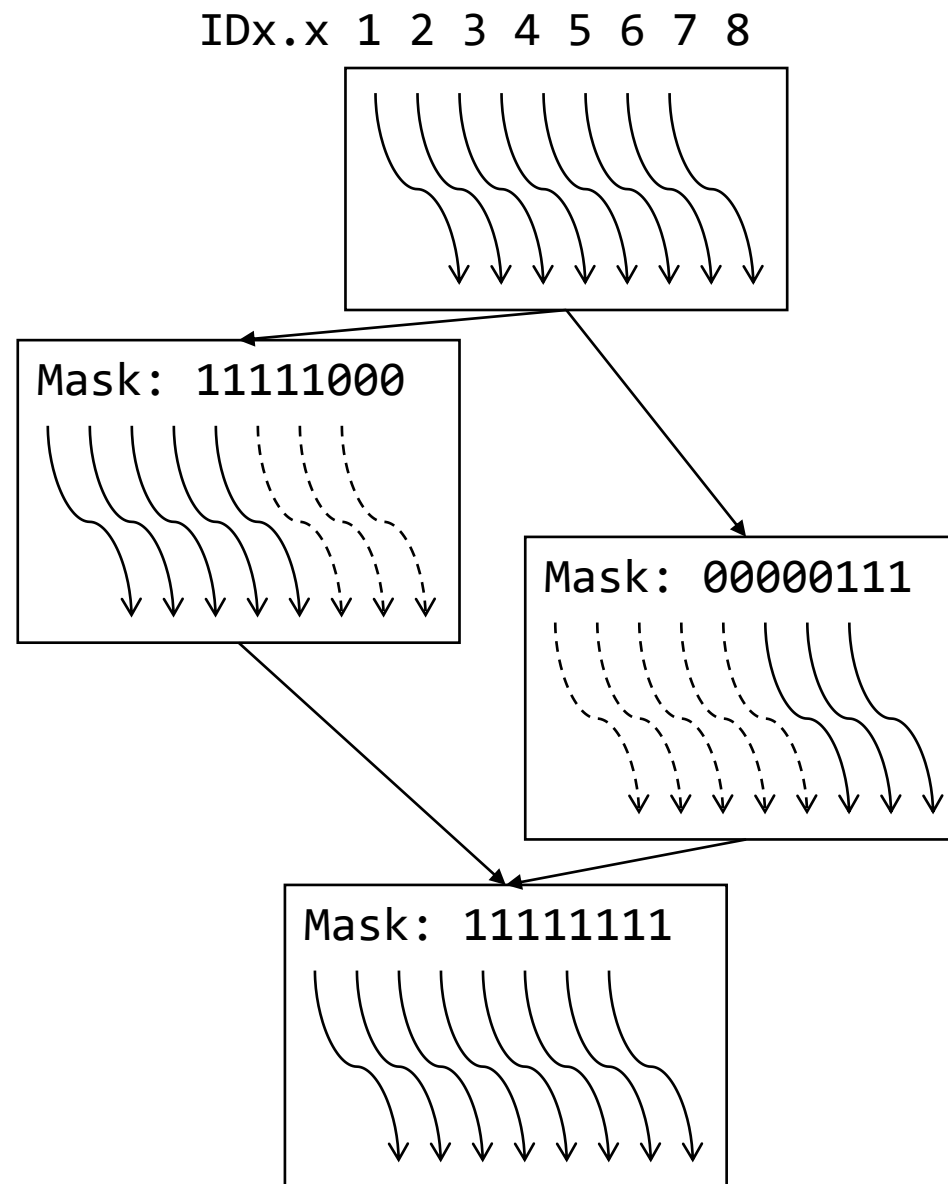
- Branch divergence: 分支指令可能导致部分线程跳转，而其余线程仍然顺序执行的现象
- 因为同一warp内线程共享程序计数器
- 若程序存在if分支，warp内的线程会分化
  - 一部分需要执行if，另一部分需要执行else
  - 宏观上warp需要串行执行if和else内所有代码
  - 在一定时刻内只有部分线程活跃，不能充分利用GPU硬件资源

# ■ GPU线程调度

## □ Branch divergence

– 思考更极端的情况会如何

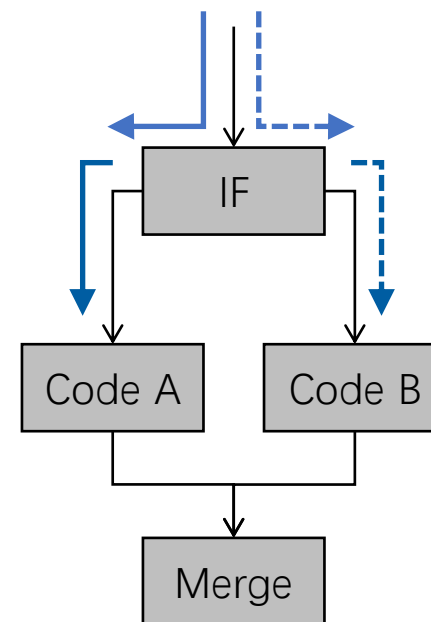
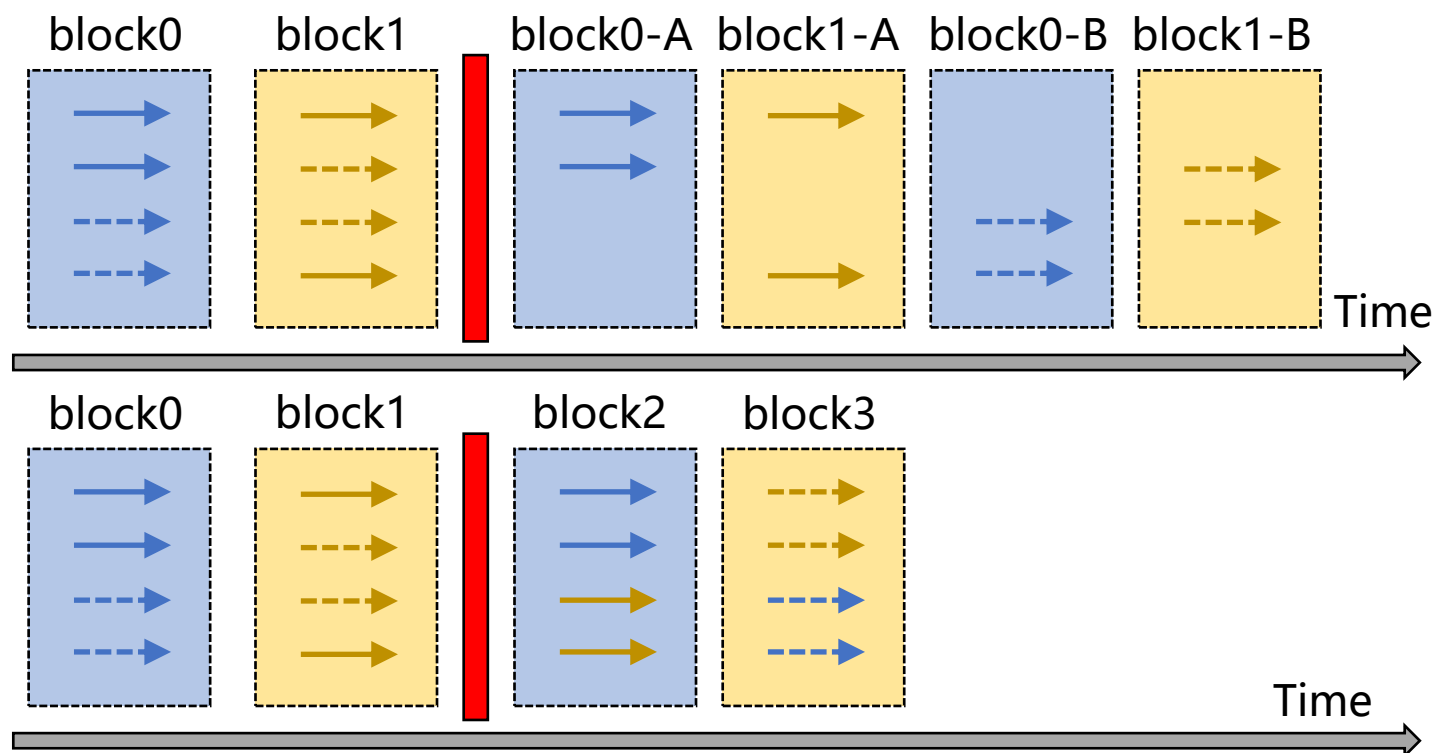
```
if (IDx.x <= 5)
    my_data[IDx.x] += 1;
else
    my_data[IDx.x] -= 1;
```



# ■ GPU线程调度

## □ warp动态重组

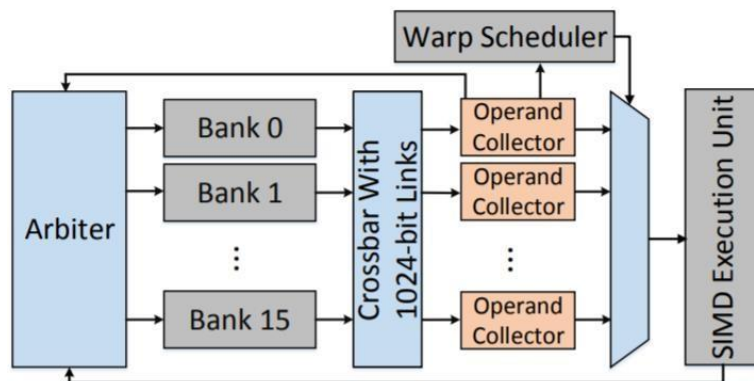
- 执行过程中重组那些程序计数器在相同位置的CUDA线程



# ■ GPU资源

## □ GPU资源限制

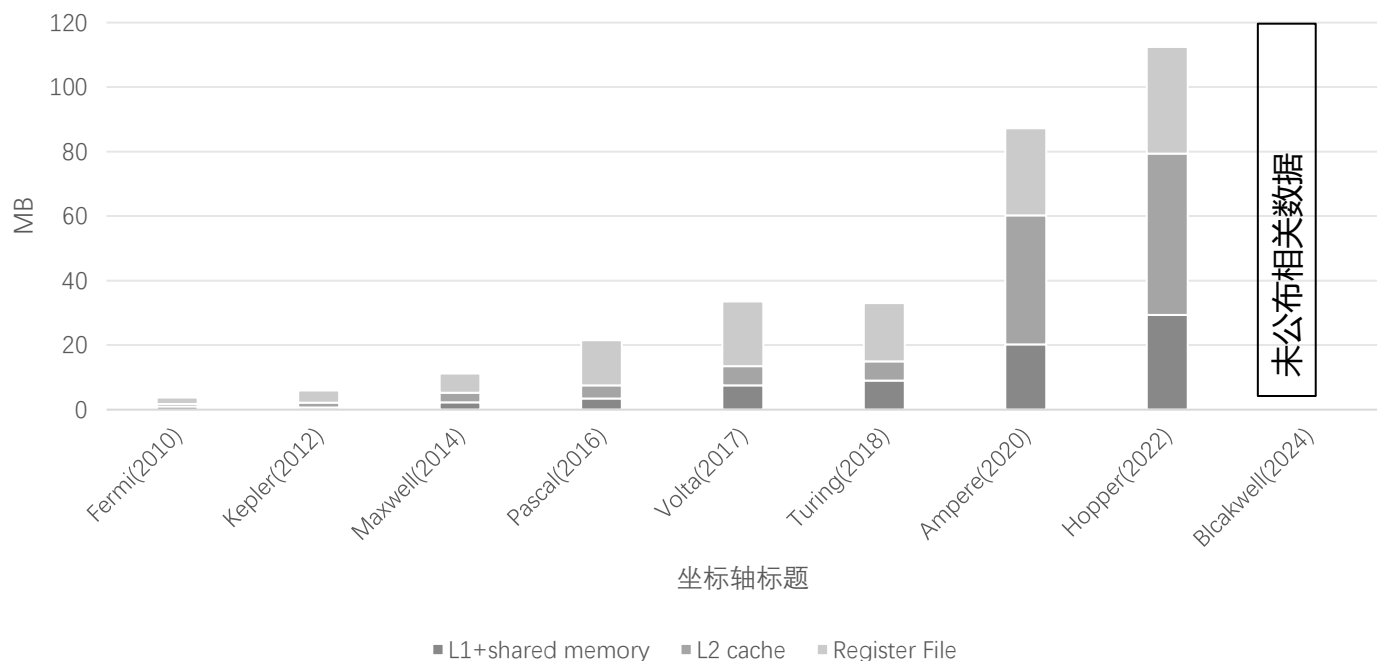
- GPU的最大并行性由寄存器堆(register file)的容量决定
  - 具有高线程级并行的应用程序容易创建非常多的warp
  - 但GPU不能无限制的创建线程块
- 寄存器堆是一个大的SRAM
  - 是处理器能访问到的最快的内存块
  - 用于存放算术单元(例如ALU)的中间计算结果
  - 功耗较高



# ■ GPU资源

## □ 历代GPU资源的变化

- 在GPU中寄存器堆的规模通常大于L2/L1缓存
  - 比较特别2020年的A100，L2缓存首次大于寄存器堆规模，增加L2有利于加速深度学习
    - 2018年GPT-1发布，但GPT-1的论文因为架构和算法缺乏创新，几经拒稿，最后挂在OpenAI官网
    - 也许当时NVIDIA有所察觉，随即在下一代架构中显著增加L2缓存规模





# ■ GPU资源

---

## □ GPU资源设计考虑平衡点

### – 更大寄存器堆容量

#### ➤ 优点

- 能容纳更多的活跃线程
- 允许更快速的上下文切换，降低切换开销
- GPU可以充分利用显存带宽

#### ➤ 缺点

- 更高的功耗，以及更大的硅片面积，增加制造成本

### – 潜在优化方向

- 缩小寄存器堆规模？
- 采用更新的存储技术？

**感谢！**

---