

体系结构复习

chap1 Intro - 计算机性能度量

1. 利用 Amdahl 定律计算**加速比** (t_{old} / t_{new})

f : 加速部分占比; s : 改进倍数;

$$Speed-up = 1 / (1 - f + f/s)$$

Amdahl 定律阐述了一个回报递减规律: 如果仅仅改进一部分计算的性能, 在增加改进时, 所获得的加速比增量会减小;

推论: 如果只针对整个任务的一部分进行优化, 那么获得的加速比不大于 $1/(1-f)$

2. 微处理器内部的能耗和功率:

逻辑脉冲 (0-1) 的动态能耗 = $(1/2) \times \text{容性负载} \times \text{电压}^2$

每个晶体管需要的功率就是一次转换的能耗与转换频率的乘积:

- 动态功率 = $(1/2) \times \text{容性负载} \times \text{电压}^2 \times \text{开关频率}$

Q: 电压和频率降低对能耗和功率的影响?

A: 通过计算比值体现: 能耗新/能耗旧 = 电压变化的平方; 功率还要乘上频率的变化

3. CPU 性能公式

CPU 时间: 程序 CPU 时钟周期数 \times 时钟周期时间 = CPI (组成与指令集体系结构) \times 指令数 (指令集体系结构和编译器技术) \times 时钟周期 (硬件技术与组成)

CPI 平均每条指令的执行时间: 程序执行时间 / 指令条数 = 各种指令在程序中所占的比例 \times 他们自己的 CPI

Q: 针对原有程序的指令, 做出一些加速 (降低部分 CPI) 比较两种方案的性能。

A:

- 计算原有 CPI
- 改进后的 CPI = 原有 CPI - 节省的时钟周期数
- 计算加速比 = 原来CPU时间 / 改进后的 CPU 时间

4. 吞吐率: 单位时间内完成的任务数量

缩短任务的执行时间可以提高吞吐率; 硬件并行可以提高吞吐率和响应时间

5. benchmarks: **一组用于测试的程序**

比较计算机系统的性能

SPEC benchmark: 针对一组应用综合性能值采用 SPEC ratios 的几何平均

6. MIPS: 每秒百万条指令数 = f / CPI

f : 频率, 单位为 MHZ (百万个时钟周期每秒)

chap2 ISA 指令集架构

1. 字节地址映射到字地址 (尾端问题)

小尾端: 0号字节放尾部;

大尾端: 0号字节放头部;

2. 一个字是否可以存放在任何字节边界上 (对齐问题)

对s字节的对象访问地址为A, 如果 $A \bmod s = 0$ 称为边界对齐

边界对齐的原因是存储器本身读写的要求, 存储器本身读写通常就是边界对齐的, 对于不是边界对齐的对象的访问可能要导致存储器的两次访问, 然后再拼接出所需要的数。(或发生异常)

3. 偏移字段的大小在 12-16bit (75%-99%)

4. 立即数字段的大小在 8-16bit (50%-80%)

CISC 指令集:

- 目标: 强化指令功能, 减少指令条数, 以提高系统性能。需要更复杂的硬件设计支持。
- 基本方法: 面向目标程序的优化 (用优化后的指令代替使用频率高的指令串); 面向高级语言和编译器的优化。

RISC 指令集

- 目标: 通过简化指令系统, 用最高效的方法实现最常用的指令
- 思想精华: 减少指令平均执行周期数 (CPI)

硬件: 硬布线控制逻辑; 减少指令和寻址方式的种类; 使用固定格式; 设置多级流水线;

软件: 强调优化编译的作用 (需要复杂的编译支持)

MPIS 指令集 (属于 RISC 指令集架构)

主要特征:

- Load/Store型结构, 专门的指令完成存储器与寄存器之间的传送
- ALU类指令的操作数来源于寄存器或立即数 (指令中的特定区域)
- 降低了指令集和硬件的复杂性, 依赖于优化编译技术, 方便了简单流水线的实现

主要缺点:

- 针对特定的微体系架构的实现方式 (5级流水、单发射、顺序流水线) 进行过度的优化设计
- 16位位宽立即数消耗了大量编码空间, 只有少量的编码空间可供扩展指令
- 乘除指令使用了特殊的寄存器 (HI,LO), 导致上下文切换内容、指令条数、代码尺寸增加, 微架构实现复杂
- ISA对位置无关的代码 (position-independent code, PIC)支持不足。

直接跳转没有提供PC相对寻址, 需要通过间接跳转方式实现PIC, 增加了代码尺寸, 降低了性能

- ISA假设浮点操作部件是一个独立的协处理器, 使得单芯片实现无法最优

例如, 浮点数转换为整型数结果写到浮点数寄存器, 使用结果时, 需要额外的mov指令, 更糟糕的是浮点数寄存器文件与整型数寄存器文件之间的mov指令, 有显式的延迟槽

- 除了技术方面, MIPS是非开放的专属指令集, 不能自由使用

RISC-V ISA

指令编码:

- 支持边长指令 (16; 32; >32) 【基础指令集固定32位编码长度】
- 所有的条件转移和无条件转移的转移地址 16-bit 对齐【基础指令集要求字对齐, RISC-V 每条指令2字节对齐就可以了】

chap 3 流水线性能, 冒险, 异常处理

流水线性能分析

1. 加速比:

理想状态 (没有冲突), 完成 n 个任务的时间: $k + n - 1$;

潜在加速比等于流水线级数: $S_k = nk / (k + n - 1)$; $S_k \rightarrow k$ for large n

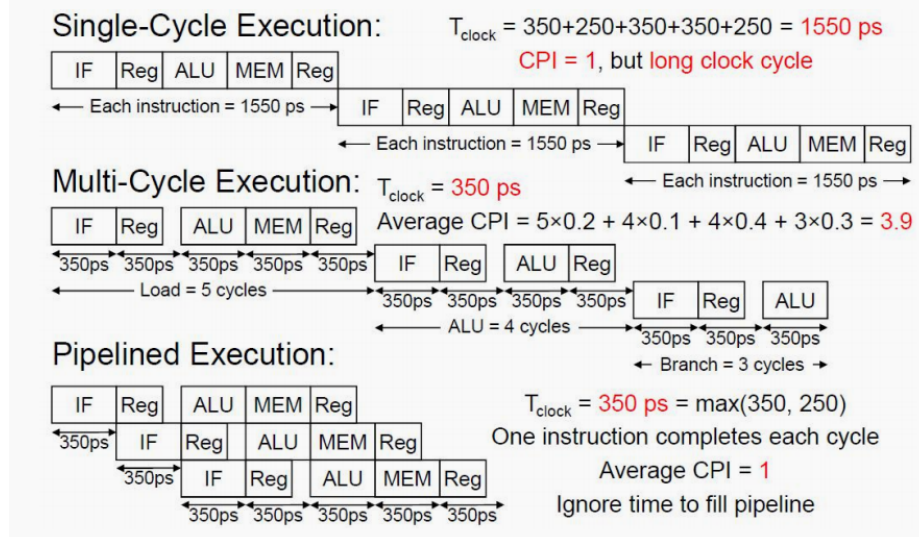
注意: 流水段不是越多越好, 当流水线段数增多时, hazards增多, 流水线的优势也会随之减小。

Q: 比较单周期、多周期、流水线的性能。

A: 分别计算每种情况下的 CPI (执行每条指令需要的时钟周期个数)



单周期、多周期、流水线控制性能比较



在有分支指令的情况下：若理想 $\text{CPI} = 1$

加速比 = 流水线深度 / (1【理想 CPI】+ 分支导致的流水线停顿周期【分支频率 × 分支代价】)

$$\text{CPI}_{\text{pipeline}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst pipelined}$$

$$\text{Speedup} = \text{Ideal CPI} * \text{PipelineDepth} * \text{CycleTime}_{\text{unpipelined}} / (\text{Ideal CPI} + \text{Pipeline stall CPI}) * \text{CycleTime}_{\text{pipelined}}$$

1. 吞吐率： $TP = n/T_k$

n ：任务数量； T_k ：完成 n 个任务用的时间； δt ：流水线的瓶颈段（时间最长的段）

$$TP = n / (k + n - 1) \delta t$$

$$TP_{\text{max}} = 1 / \delta t$$

$$TP = n TP_{\text{max}} / (k + n - 1)$$

解决流水线瓶颈段的方法：

- 细分瓶颈段（串行）
- 重复设置瓶颈段（并行）

2. 效率：流水线中的设备实际使用时间与整个运行时间的壁纸，即流水线设备的利用率

各流水段时间相等： $E = n / (k + n - 1)$

各段时间不等： n 个任务占用的时空面积（阶梯状）和 k 个段总的时空面积（长方形）比

流水线相关

结构相关：一条指令需要另一条指令使用的资源

数据相关：一条指令需要前一条指令产生的结果

- RAW：写后读相关（load + R 只能停顿，其余可以添加旁路转发）
- WAR：读后写相关：寄存器重命名解决
- WAW：写后写相关：寄存器重命名解决

解决方法：联锁机制；旁路定向路径设置；投机，猜测一个值，猜错了再改；

控制相关：需要前面的指令给出下一条指令的地址

- 软件填充延迟槽（delay slots）：若添加 NOP 会增加 I-cache 的失效率
- 分支预测技术

流水线异常 - 陷阱和中断

异常：由程序执行过程中引起的异常内部事件

陷阱：因异常情况而**被迫**将控制权移交给监控程序

中断：**运行程序之外的外部事件**，导致控制权转移到监控程序

精确中断：

如果流水线可以控制使得引起异常的指令前序指令都执行完，故障后的指令可以重新执行，则称该流水线支持精确中断。【引起故障的指令没有改变机器的状态】

- **停止当前指令(li)的执行，执行完当前指令前面的指令（执行完 li-1）**
- 将 li 指令的 pc 值保存到专门的 寄存器（EPC）中
- 关中断，将程序控制转移到以监控模式运行的指定的中断处理程序

中断恢复：

- 开中断
- 将处理器恢复到用户模式
- 恢复硬件状态和控制状态

Q：如何处理不同流水段的多个并发异常？

A：统一在 write back 之前提交错误【缓存操作结果，乱序到达，顺序写回】；针对某一给定的指令，早期流水阶段中的异常覆盖

该指令的后续异常；

Q2：如何以及在哪里处理外部异步中断？

A：在提交阶段并入异步中断请求（覆盖其他中断）；

如果提交时检测到异常：（1）更新异常原因以及 EPC 寄存器；（2）终止所有流水段；（3）将异常处理程序的地址送入 PC 寄存器，以跳转到处理程序中执行；

MIPS 支持浮点数操作

问题：

结构冲突增多（WAW相关）；指令乱序完成；指令延迟加大导致 RAW 相关的 stall 数增多；需要更多的代价增加定向路径；

解决方法：

- 在 ID 段跟踪写端口的使用情况，以便发生冲突时暂停指令发射；
- 进入 MEM 和 WB 段时，暂停冲突的指令，让有较长延时的指令先做（假设他们会更容易引起其他的 RAW 相关）

chap4 Cache 原理及性能分析

存储层次工作原理：局部性（locality）

- 时间局部性：保持最近访问的数据项最接近微处理器；
- 空间局部性：以**地址连续**的若干个字构成的块为单位，从底层赋值到上一层

平均访存时间 = 命中时间 + 失效率 × 失效开销

Cache 结构

全相联方式：即所调入的块可以放在cache中的任何位置

直接映象方式：主存中每一块只能存放在cache中的唯一位置

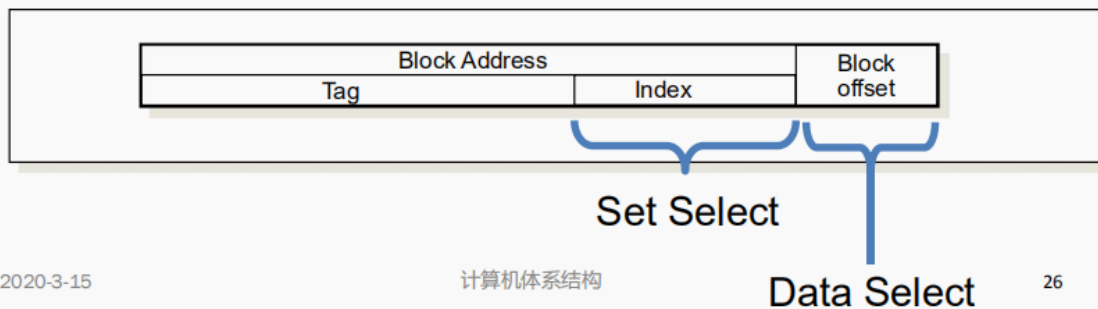
N路组相连：Cache -> 若干组（set）-> 组里有 N 块（block）-> 每一个块里有一些地址tag+数据（通过地址 tag 确定是否是需要的块）

相连度越高，cache空间利用率越高，块冲突就越小，失效率越低；但 cache 的实现会更复杂，查询代价越大，一般用直接相连或者两路，四路组相连。



Q2(1/2): 查找方法

- 在CACHE中每一block都带有tag域（标记域），标记分为两类
 - Address Tags：标记所访问的单元在哪一块中，这样物理地址就分为三部分：Address Tags ## Block index## block Offset
 - 全相联映象时，没有Block Index
 - 显然 Address tag越短，查找所需代价就越小
 - Status Tags：标记该块的状态，如Valid, Dirty等



查找流程：

1. 看 Index 找到是第几组（不用比较，直接定位是第几组）
2. 比较 tag 【检查是否命中】（找到是这一组里地址对应的块）
3. 根据 block offset 选择期望拷贝的数据。

替换算法

- 随机法：随机选择一块替换
简单，易于实现；没有考虑 cache 块的使用历史，反映程序的局部性较差，失效率高
- FIFO：选择最早调入的块
简单；反映程序局部性不够，因为最先进入的块，很可能是经常使用的块。
- LRU 最近最少使用法：
较好的利用了程序的局部性，失效率低。
比较复杂，硬件实现较困难。

观察失效率结果：

相连度高，cache容量大，失效率降低；

LRU在cache容量较小时失效率降低；

随着 cache 容量增大，random 的失效率在降低；

写策略

数据项写入缓存时的策略：

- 写直达法 - 不按写分配

信息被写入缓存中的块和低一级存储器的块。

- 写回法 - 对应写时取

信息仅被写入缓存中的块，修改后的缓存块仅在被替换时才被写到主存储器。

发生写失效时：

- 按写分配法（写时取）：写失效时，先把块调入 cache，再写入；
- 不按写分配（绕写法）：写失效时，直接写入下一级存储器，不调入 cache

性能分析

CPU 执行时间 = (CPU时钟周期+存储器停顿周期) × 时钟周期时间

存储器停顿周期 = 缺失数量 × 缺失代价

= (读指令数量 × 读 miss rate × 读 miss penalty) + (写指令数量 × 写 miss rate × 写 miss penalty)

平均每条指令的存储器停顿周期 = 平均每条指令的缺失数 × 缺失代价 = 缺失率 × 平均每条指令的访存数 × 缺失代价

平均每条指令的缺失数 = 缺失率 × 平均每条指令的访存数

平均内存访问时间 AMAT = $\sum \text{指令占比} \times \text{该指令平均访存时间}$ = $\sum \text{指令占比} \times (\text{命中时间} + \text{Miss rate} \times \text{Miss penalty})$

访存缺失率 = 缺失次数 / 总访存次数 = 平均每条指令的缺失次数 / 平均每条指令的访存次数

降低失效率

- 强制性失效：第一次访问，只能从下一级 load
- 容量失效：容量不足，需要替换
- 冲突失效：多个块映射到同一组（块）中

相连度越高，冲突失效越小。但提高相连度，会增加命中时间

2:1 cache 经验规则：即大小为N的直接映象Cache的失效率约等于大小为N/2的两路组相联的Cache失效率。

- Victim cache（全相连）存放由于（冲突）失效而被丢弃的那些块



多级cache的性能分析

- **局部失效率**：该级Cache的失效次数 / 到达该级Cache的访存次数
 - Miss rateL1 for L1 cache
 - Miss rateL2 for L2 cache
- **全局失效率**：该级Cache的失效次数 / CPU发出的访存总次数
 - Miss rateL1 for L1 cache
 - Miss rateL1 × Miss rateL2 for L2 cache
 - 全局失效率是度量L2 cache性能的更好方法
- **性能参数**
 - $AMAT = \text{Hit TimeL1} + \text{Miss rateL1} \times \text{Miss penaltyL1}$
 - $\text{Miss penaltyL1} = \text{Hit TimeL2} + \text{Miss rateL2} \times \text{Miss penaltyL2}$
 - $AMAT = \text{Hit TimeL1} + \text{Miss rateL1} \times (\text{Hit TimeL2} + \text{Miss rateL2} \times \text{Miss penaltyL2})$

减小失效开销

1. 多级cache技术：减小失效开销，缩短平均访存时间，较大的二级cache可以捕捉L1 cache的失效，降低全局失效率

多级包容：

L1 cache 的块总是在 L2 中

L1 miss 但在 L2 命中，则从 L2 拷贝相应的块到 L1

L1 L2 都 miss，从低级拷贝相应的块到 L1 L2

对 L1 的写操作将数据同时写到 L1 L2 (WriteThrough策略)

Writeback策略可用于L2到更低级存储器，以降低存储总线的数据传输压力

L2 中的一块被替换出去，L1相应块也要被替换

多级不包容：

L1 中的块不会在 L2 中出现，以避免浪费

L1 miss L2 命中，cache间互换

都 miss 仅从更低块拷贝到 L1

L1 被替换的块移到 L2 (以备后续还要使用)

L1->L2 -> 更低级 cache (writeback)

性能参数：


$AMAT = hit\ time\ 1 + miss\ rate\ 1 \times miss\ penalty\ 1$

$= hit\ time\ 1 + miss1 \times (hit\ time\ 2 + miss\ rate\ 2 \times miss\ penalty\ 2)$

Q: L1 分为 I-cache 和 D-cache 如何计算 miss rate 1?

A:

- 计算平均每条指令访存次数 = 1 (访问 I-cache) + k (访问数据的概率, LoadStore 指令频度)
- 计算平均每条指令的失效次数 = m (访问 I-cache的失效率) + n (访问 D-cache) 的失效率
- $miss\ rate\ 1 = (2) / (1)$



两级Cache的性能

- Problem: 程序运行产生1000个存储器访问**
 - I-Cache misses = 5, D-Cache misses = 35, L2 Cache misses = 8
 - L1 Hit = 1 cycle, L2 Hit = 8 cycles, L2 Miss penalty = 80 cycles
 - Load + Store frequency = 25%, CPIexecution = 1.1 (perfect cache)
 - 计算memory stall cycles per instruction 和有效的CPI
 - 如果没有L2 cache, 有效的CPI是多少?
- Solution:**
 - L1 Miss Rate = $(5 + 35) / 1000 = 0.04$ (or 4% per access)
 - L1 misses per Instruction = $0.04 \times (1 + 0.25) = 0.05$
 - L2 misses per Instruction = $(8 / 1000) \times 1.25 = 0.01$
 - Memory stall cycles per Instruction = $0.05 \times 8 + 0.01 \times 80 = 1.2$
 - $CPI_{L1+L2} = 1.1 + 1.2 = 2.3$, $CPI/CPI_{execution} = 2.3/1.1 = 2.1x$ slower
 - $CPI_{L1only} = 1.1 + 0.05 \times 80 = 5.1$ (worse)

2020-3-15

计算机体系结构

40

2. 读优先于写

写直达 -> write buffer 不必等待写操作完成，将要写的数据和地址送到 write buffer 中后，cpu 继续其他操作

写回法 -> victim buffer 被替换的脏块放到 victim buffer，在脏块写回前，要先处理读失效

Q: vb 中可能含有读失效要读的块

A: 查找 vb，若命中就直接将该块调入 cache


scoreboard

1. IF 发射：检测结构相关（顺序发射，乱序执行）

如果当前指令所使用的功能部件空闲，并且没有其他活动的指令使用相同的寄存器 **WAW**，记分牌发射该指令到功能部件，并更新记分牌内部数据。

若有结构相关或 WAW，则暂停当前及后续指令的发射，知道相关解除。

2. ID 读取操作数：要求没有数据相关（**RAW**）时再读取
3. EX 接收到操作数时，开始执行，结束后告诉记分牌。
4. WB 记分牌得到执行完毕的信息，如果没有检测到 **WAR** 相关，则写结果，如果有，暂停整条指令。（顺序写回）



记分牌的结构

1. **Instruction status**—记录正在执行的各条指令所处的状态步
2. **Functional unit status**—记录功能部件(FU)的状态。用9个域记录每个功能部件的9个参量：
 - Busy**—指示该部件是否空闲
 - Op**—该部件所完成的操作
 - Fi**—其目的寄存器编号
 - Fj, Fk**—源寄存器编号
 - Qj, Qk**—产生源操作数 Fj, Fk 的功能部件
 - Rj, Rk**—标识源操作数 Fj, Fk 是否就绪的标志
3. **Register result status**—如果存在功能部件对某一寄存器进行写操作，指示具体是哪个功能部件对该寄存器进行写操作。如果没有指令对该寄存器进行写操作，则该域为 Blank

2020-4-3计算机体系结构13

tomasulo

1. IF 发射：从 FP 操作队列中取指令

如果 RS 空闲，则控制发射指令和操作数，消除 WAR，WAW 相关

2. EX 当**两个操作数就绪**后即可执行，没有准备好，则检测 common data bus 获取结果，通过推迟指令执行避免 RAW 相关。
3. WB 通过 common data bus 传给所有等待该结果的部件，表示 RS 可用。

控制和缓存分布在各部件中；

指令中的寄存器在 RS 中用寄存器值或指向 RS 的指针代替（寄存器重命名）

传给 FU 的结果从 RS 来，FU 的计算结果通过 CDB 以广播的方式发向所有功能部件



Reservation Station 结构

- Op:** 部件所进行的操作
- Vj, Vk:** 源操作数的值。Store 缓冲区有Vk域，用于存放要写入存储器的值
- A:** 存放存储器地址。开始存立即数，计算出有效地址后，存放有效地址
- Qj, Qk:** 产生源操作数的RS
注：没有记分牌中的准备就绪标志， $Qj, Qk=0 \Rightarrow \text{ready}$
Store 缓存区中Qk表示产生结果的RS
- Busy:** 标识RS或FU是否空闲
- Register result status**—如果存在对寄存器的写操作，指示对该寄存器进行写操作的部件。
- Qi:** 保留站的编号

chap 6 vector



Summary: 向量体系结构

- **向量处理机基本概念**
 - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- **向量处理机基本特征**
 - VSIW—一条指令包含多个操作
 - 单条向量指令内所包含的操作相互独立
 - 以已知模式访问存储器-多体交叉存储系统
 - 控制相关少
- **向量处理机基本结构**
 - 向量指令并行执行
 - 向量运算部件的执行方式-流水线方式
 - 向量部件结构-多“道”结构-多条运算流水线
- **向量处理机性能评估**
 - 向量指令流执行时间: Convey, Chimes, Start-up time
 - 其他指标: R_{∞} , $N_{1/2}$, N_V
- **向量处理机性能优化**
 - 链接技术
 - 条件执行
 - 稀疏矩阵



并行的类型

- **指令级并行(ILP)**
 - 以并行方式执行某个指令流中的独立无关的指令 (pipelining, superscalar, VLIW)
- **数据级并行(DLP)**
 - 以并行方式执行多个相同类型的操作 (vector/SIMD execution)
 - Array Processor 、 Vector Processor
- **线程级并行 (TLP)**
 - 以并行方式执行多个独立的指令流 (multithreading, multiple cores)
- **Which is easiest to program?**
- **Which is most flexible form of parallelism?**
 - i.e., can be used in more situations
- **Which is most efficient?**
 - i.e., greatest tasks/second/area, lowest energy/task

GPU弱控制，强计算

GPU 使用 SIMT（单指令多线程）模型，每个CUDA现成的标量指令流汇聚在硬件上以 SIMD（硬件）方式执行。

SPMD 编程模型（每个单元处理同样的过程，不同的数据）可以在 SIMT 机器上运行

warp：一组执行相同指令的线程由硬件动态组织成 warp，是由硬件形成的 SIMD 操作

大量线程组织成很多线程块 block，许多线程块组成 grid

warp：SIMD 线程；线程调度的基本单位；



SIMD vs. SIMT Execution Model

- **SIMD: 一条指令流（一串顺序的SIMD指令），每条指令对应多个数据输入（向量指令）**
- **SIMT: 多个指令流（标量指令）构成线程，这些线程动态构成Warp。一个Warp处理多个数据元素**
- **SIMT 主要优点:**
 - 可以独立地处理线程,即每个线程可以在任何标量流水线上单独执行（MIMD 处理模式）
 - 可以将线程组织成warp, 即将执行相同指令流的线程构成warp, 形成SIMD 处理模式, 以充分发挥SIMD处理的优势

5/8/2020

中国科学技术大学

82

CPU 和 GPU 的异同

不同点:

CPU 强控制，弱计算；GPU 弱控制，强计算；GPU有更多的寄存器和车道；

GPU cache 占用的空间比 CPU 小；且使用 SIMD 的执行模型；计算量可以很大，但支持的计算方式比较简单；多寄存器，支持很多 threads，适合计算密集型、容易并行化的程序。