第二章 处理器及其相关技术

目录 CONTENT

2.1 CPU组成

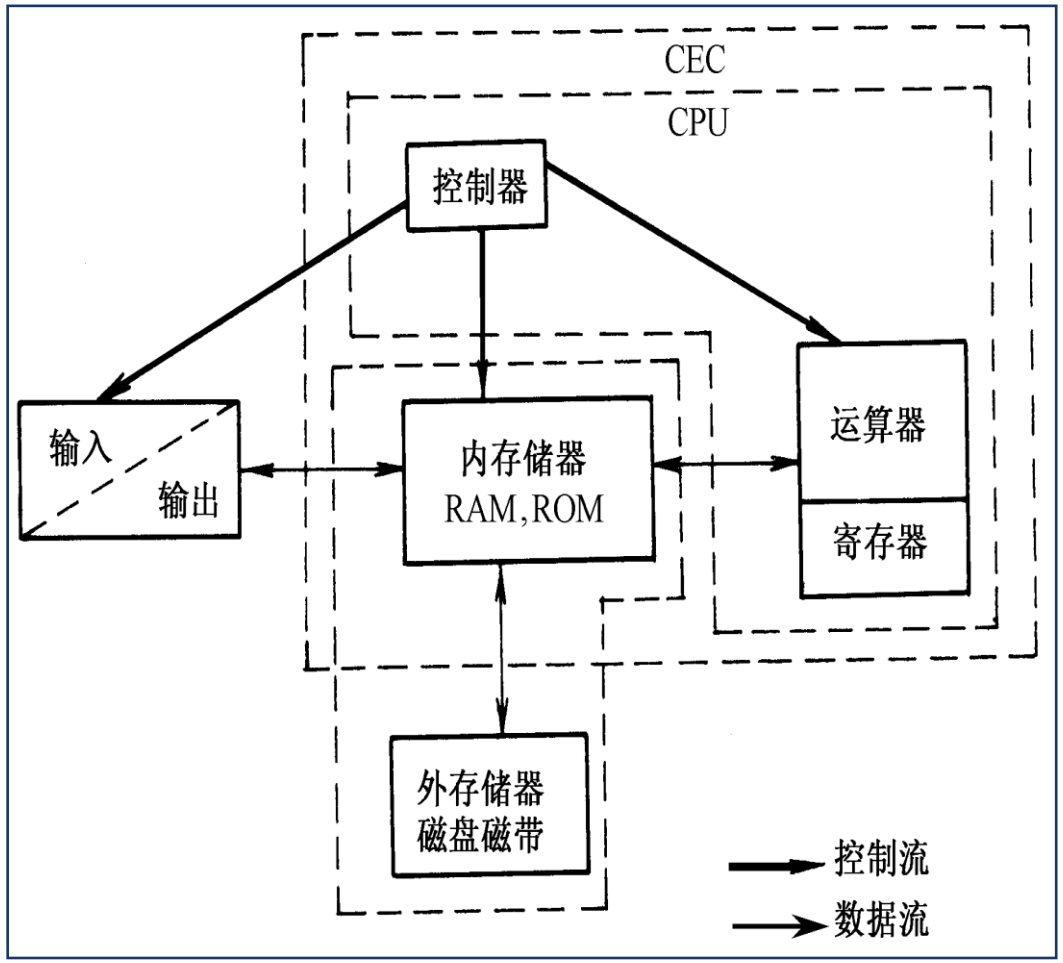


图2-1 CPU组成

中央处理器（CPU）由控制器、运算器和寄存器组成。

（1）控制器：从存储器中取出指令，对指令进行译码，控制整个计算机系统一步一步地完成各种操作

（2）运算器：提供计算与逻辑功能

（3）寄存器（Register）：处理器内部的存储单元

- 控制器中的寄存器，保存程序运行中的状态，存储当前指令信息和将要执行的下一条指令的地址等
- 运算器中的寄存器，存储进行运算与比较的数据及其结果

2.2 指令系统

- 在机器上直接运行的程序是由指令组成的
- 指令系统是软件和硬件之间的一个主要分界面，也是它们之间相互沟通的一座桥梁

  硬件设计人员采用各种手段实现指令系统，而软件设计人员则使用这些指令系统编写系统软件和应用软件，用这些软件来填补指令系统与人们习惯的使用方式之间的语义差距
- 指令系统的设计必须由软件设计人员和硬件设计人员共同完成
- 指令系统发展相当缓慢，需要用软件来填补的东西就越来越多

2.3 数据表示

2.1 数据表示与数据类型

- 数据类型：文件、图、表、树、队列、阵列、链表、栈、向量、串、整数、布尔数、字符等
- 计算机系统结构研究的首要问题：哪些数据类型用硬件实现，哪些数据类型用软件实现及其实现方法
- 数据表示的定义：

  数据表示是指计算机硬件能够直接识别，可以被指令系统直接调用的那些数据类型

  例如：定点、逻辑、浮点、十进制、字符、字符串、堆栈和向量等

数据结构：

研究面向系统软件和应用领域所需处理的数据类型，研究这些数据类型的逻辑结构和物理结构之间的关系，并给出相应的算法

- 除了数据表示之外的所有数据类型，都是数据结构要研究的内容
- 确定哪些数据类型用数据表示实现，哪些数据类型用数据结构实现，是软件与硬件的取舍问题

- 确定数据表示是计算机系统设计人员要解决的难题之一。
- 从原理上讲，计算机系统结构只要有了最简单的数据表示，就能够用软件实现其他各种数据类型
- 确定数据表示的原则
  - 缩短程序的运行时间
  - 减少CPU与主存储器之间的通信量
  - 数据表示的通用性和利用率
- 数据表示在不断发展
  - 例如：矩阵、树、图、表等已经开始用于数据表示中
- 将复杂的数据类型用数据表示实现，系统的硬件成本较高

■ 例1：计算C = A + B，A、B、C均为200×200的矩阵，分析在一般的计算机上和在向量计算机上运算的区别

解：如果在没有向量数据表示的计算机上实现，一般需要6条指令，其中有4条指令要循环4万次，因此，CPU与主存的通信量：

- 取指令：2+4×40,000条
- 读或写数据：3×40,000条

共要访问主存：7×40,000次以上

- 如果有向量数据表示，只需一条指令

减少访问主存（取指令）次数4×40,000次

■复杂数据类型：

- ➢硬件实现，代价大
- ➢软件实现，效率低
- ➢软硬件结合方式效果好

例如：用字节编址和字节运算指令支持字符串数据表示；用变址寻址方式支持向量数据表示

■设计计算机系统时，对于数据类型：

- ➢确定哪些数据类型全部用硬件实现，即数据表示
- ➢确定哪些数据类型用软件实现，即数据结构
- ➢确定哪些数据类型由软硬件共同实现，并确定软硬比例关系

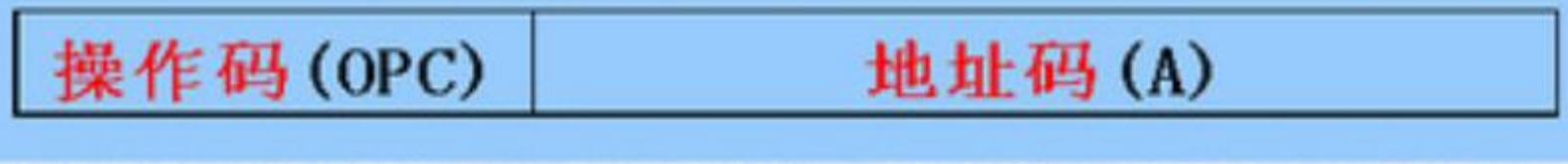
2.3 指令格式设计

- 主要目标
  - 节省程序的存储空间
  - 指令格式尽量规整，便于译码

操作码的三种编码方法
- 固定长度
- Huffman编码
- 扩展编码

- 一般的指令主要由两部分组成：

  操作码和地址码



- 地址码通常包括三部分内容：
  - 地址：地址码、立即数、寄存器、变址寄存器
  - 地址的附加信息：偏移量、块长度、跳距
  - 寻址方式：直接寻址、间接寻址、立即数寻址、变址寻址、相对寻址、寄存器寻址

1. 固定长度编码

- 定长定域：
  - n种指令，操作码位数 $\lceil \log_2 n \rceil$
  - IBM公司的大中型机：最左边8位为操作码
  - Intel公司的Intanium处理机：14位定长操作码
- 主要优点：
  - 规整
  - 译码简单
- 主要缺点：
  - 浪费信息量（操作码的总长位数增加）

- 优化操作码编码的目的：节省程序存储空间

  例如：Burroughs公司的B-1700机

操作码编码方式	整个操作系统所用指令的操作码总位数	改进的百分比
8 位固定长编码	301,248	0
4-6-10 扩展编码	184,966	39%
Huffman 编码	172,346	43%

2. Huffman编码

- 1952年，哈夫曼提出电报报文编码方式，减少报文长度，缩短报文传送时间。
- 将出现概率最大的事件用最少的位来表示，而概率较小的事件用较多的位表示，达到平均编码位数缩短的目的。
- 哈夫曼压缩可用于程序、存储空间、图像、声音等压缩。

- 操作码的最短平均长度可通过如下公式计算：

$$H = -\sum_{i=1}^{n} p_i \cdot \log_2 p_i$$

p_i表示第i种操作码在程序中出现的概率

- 固定长编码相对于Huffman编码的信息冗余量：

$$R = 1 - \frac{-\sum_{i=1}^{n} p_i \cdot \log_2 p_i}{\lceil \log_2 n \rceil}$$

- 必须知道每种操作码在程序中出现的概率

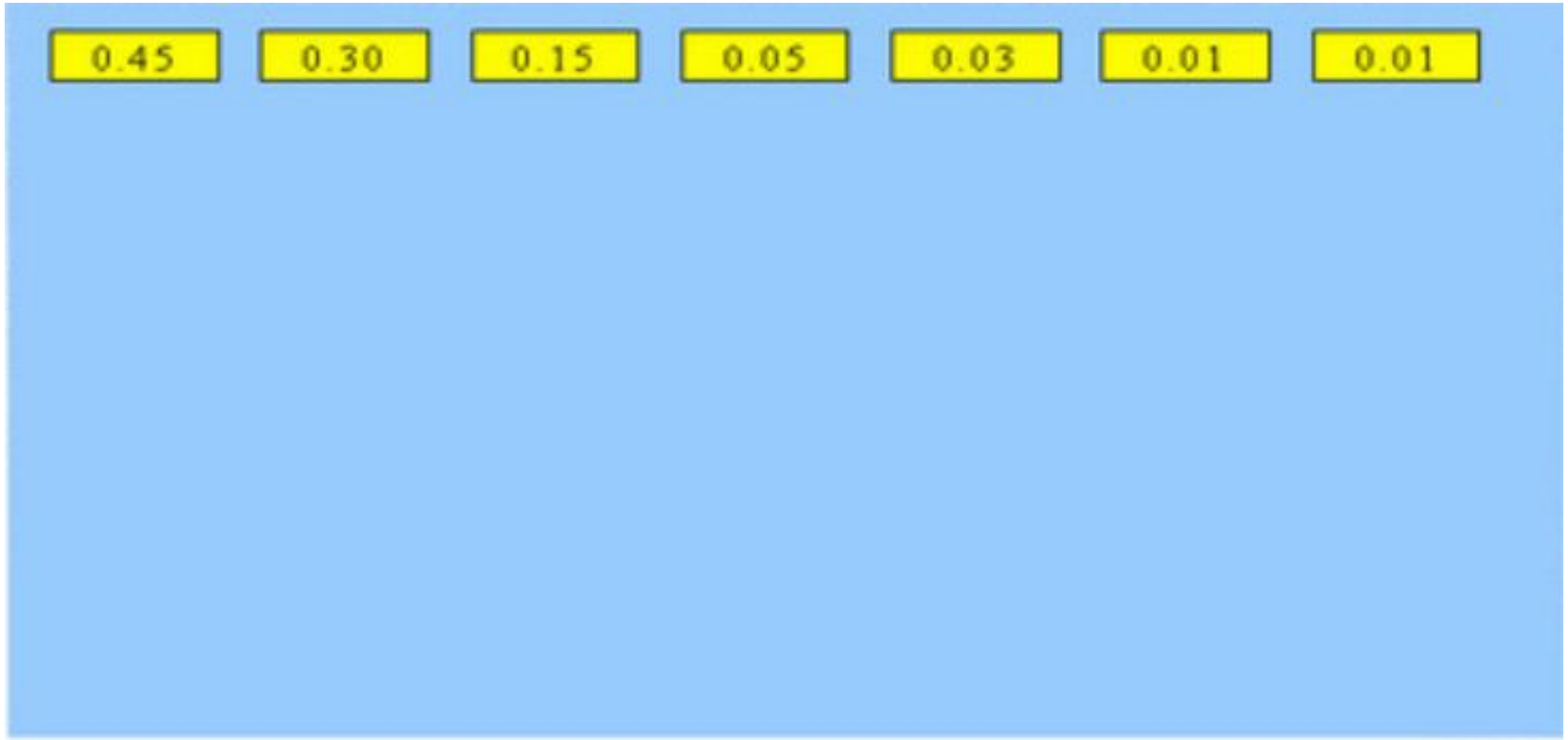
- 例. 假设一台模型计算机共有7种不同的操作码，如果采用固定长操作码需要3位。已知各种操作码在程序中出现的概率如下表，计算采用Huffman编码法的操作码平均长度，并计算固定长操作码和Huffman操作码的信息冗余量

指令序号	I_1	I_2	I_3	I_4	I_5	I_6	I_7
出现的概率	0.45	0.30	0.15	0.05	0.03	0.01	0.01

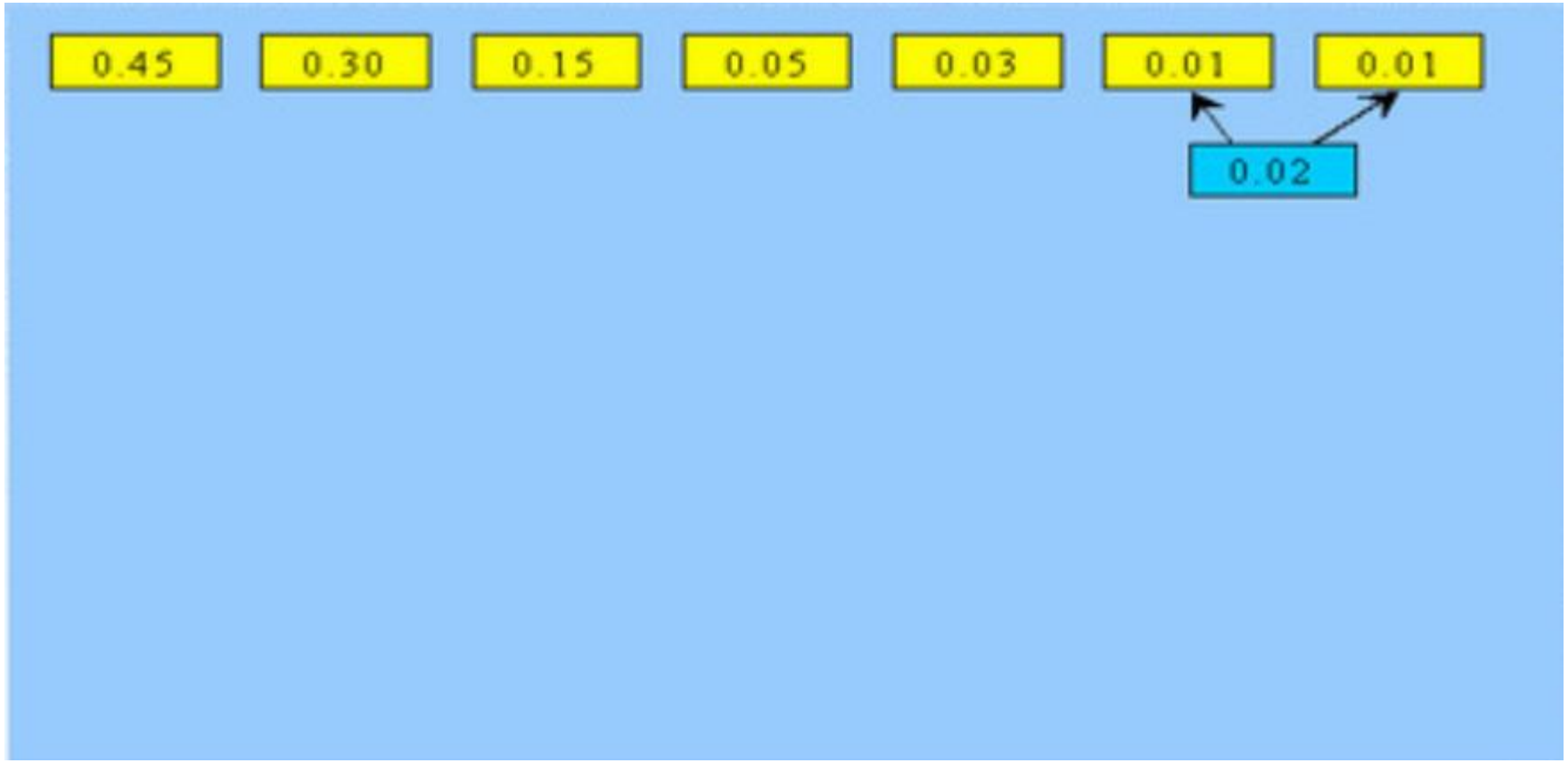
- 利用Huffman树进行操作码编码
  1. 把所有指令按照操作码在程序中出现的概率大小，自左向右顺序排列
  2. 选取两个概率最小的节点合并成一个概率值是二者之和的新节点，并把这个新节点与其他还没有合并的节点一起形成一个新的节点集合
  3. 在新的节点集合中选取两个概率最小的节点进行合并，如此继续执行下去，直至全部节点合并完毕

4. 最后得到的根节点的概率值为1
5. 每个新节点都有两个分支，分别用带有箭头的线表示，并分别用一位代码“0”和“1”标注
6. 从根节点开始，延箭头所指方向寻找到达属于该指令概率节点的最短路径，把沿线所经过的代码排列起来就得到了这条指令的操作码编码

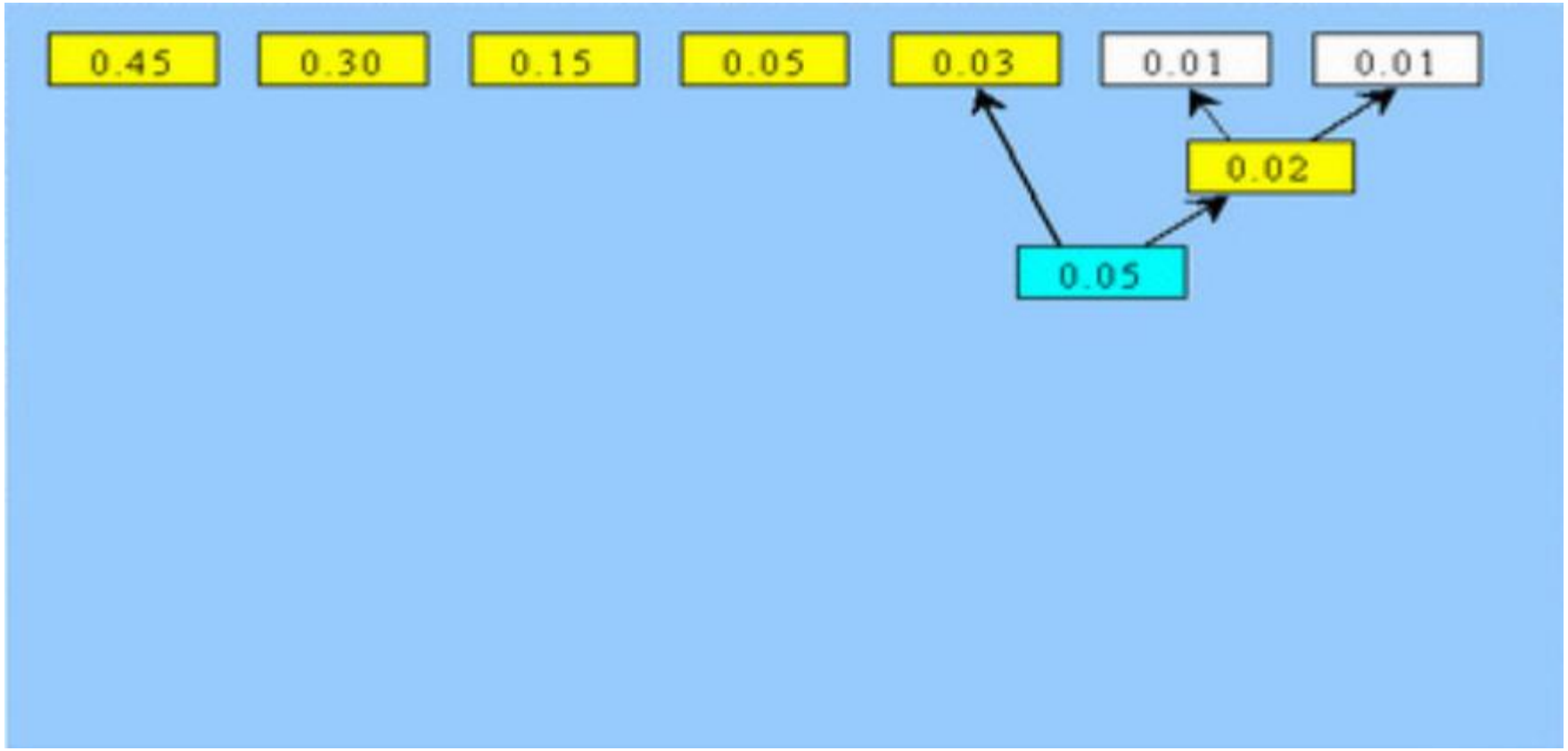
利用Huffman树进行操作码编码



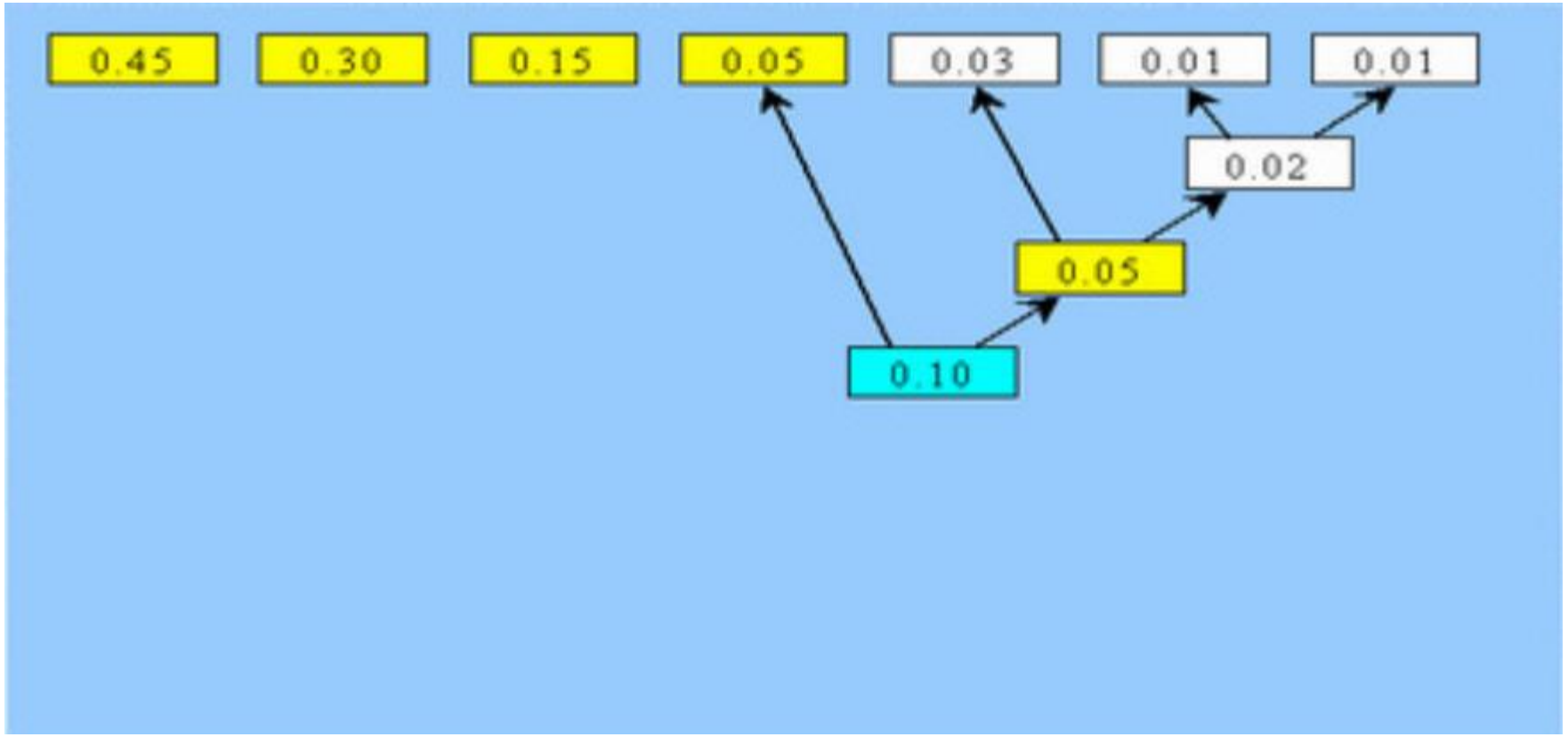
利用Huffman树进行操作码编码



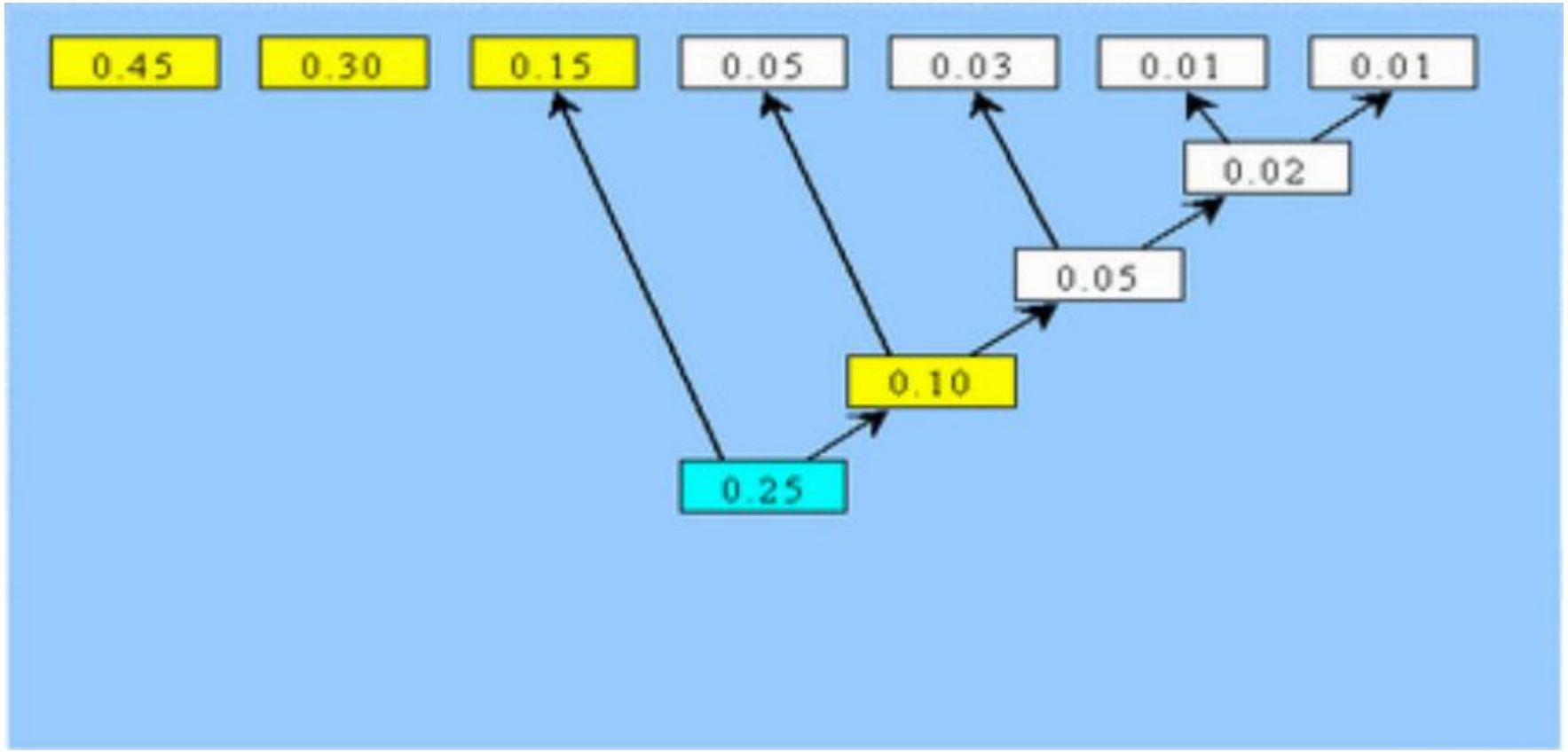
利用Huffman树进行操作码编码



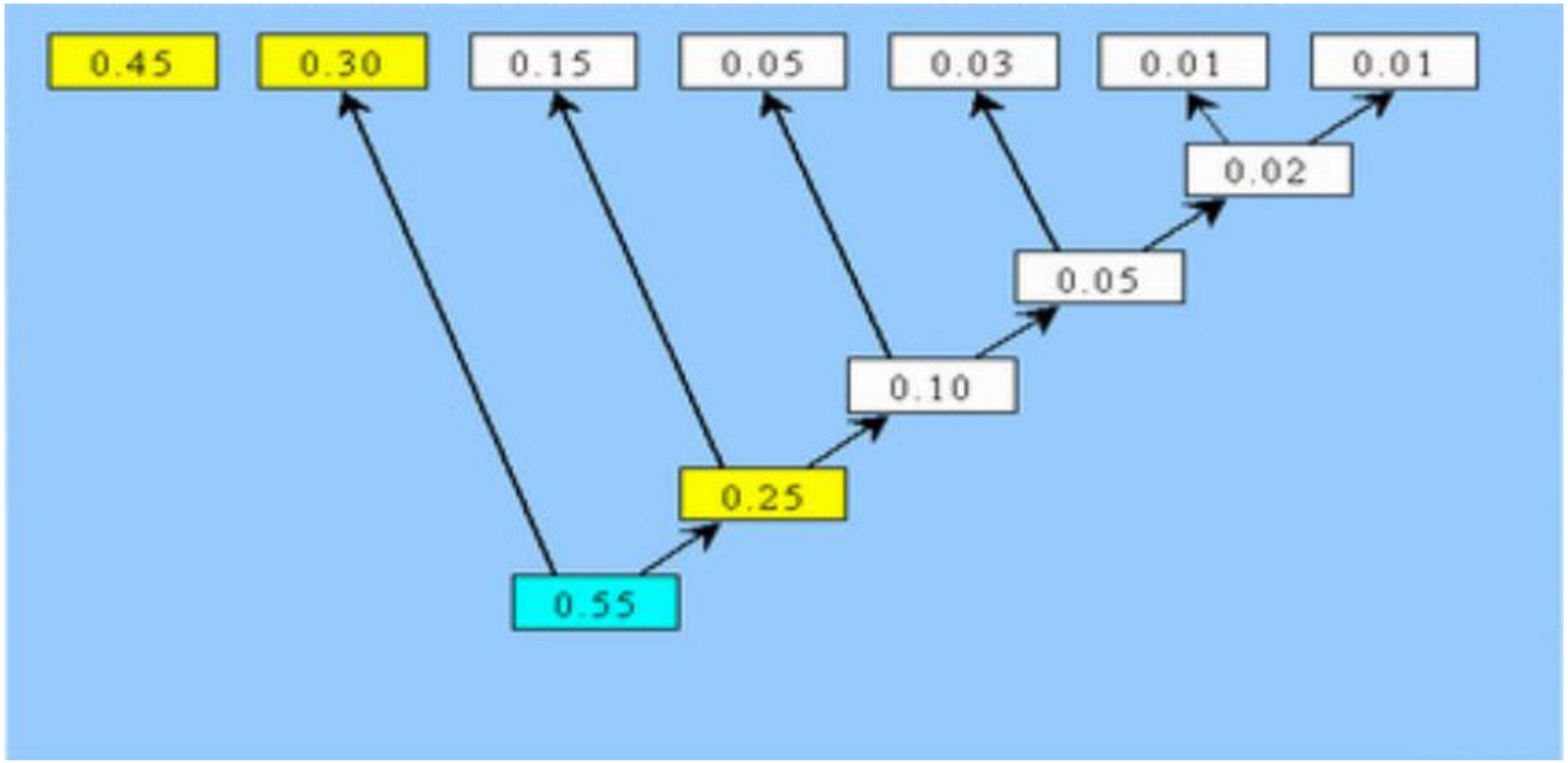
利用Huffman树进行操作码编码



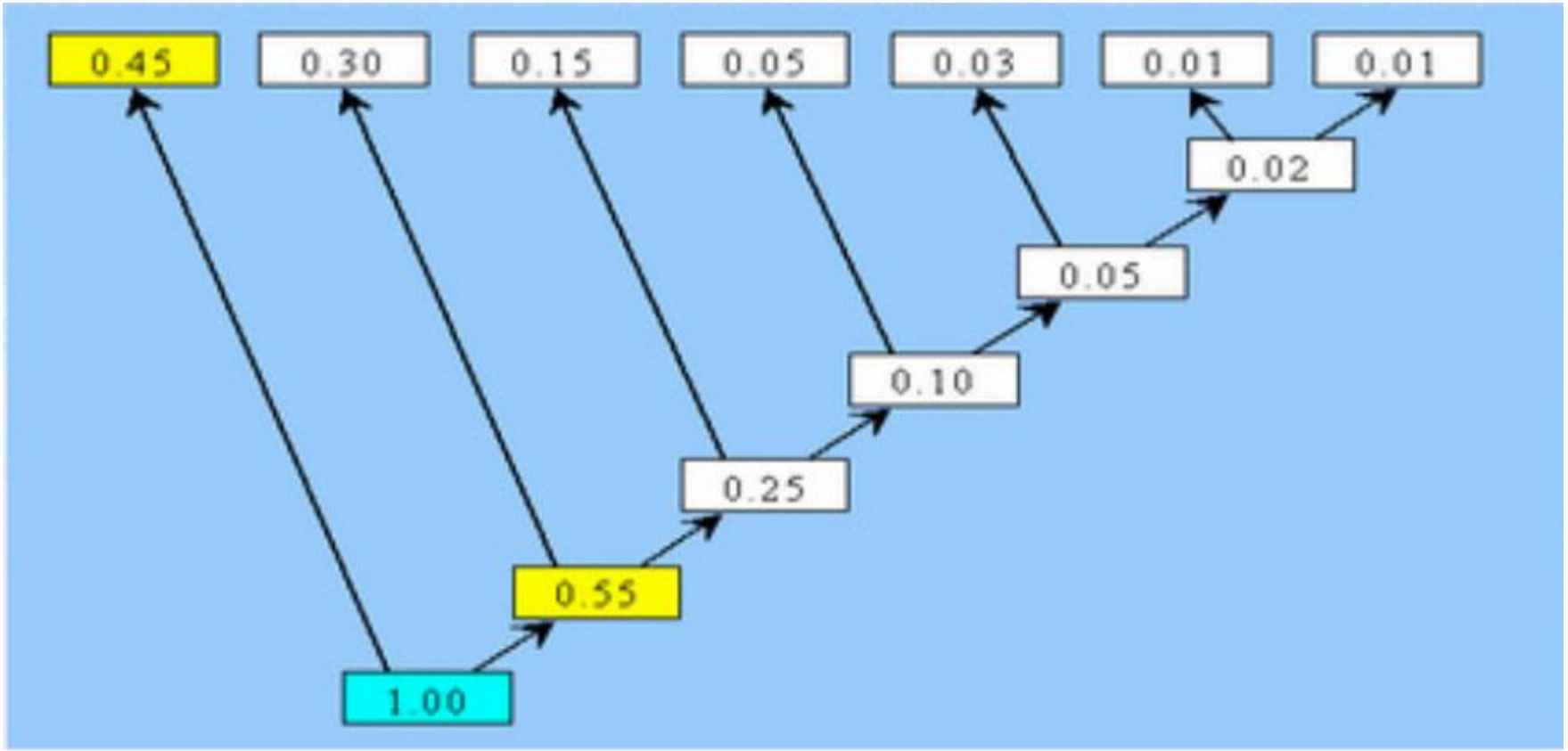
利用Huffman树进行操作码编码



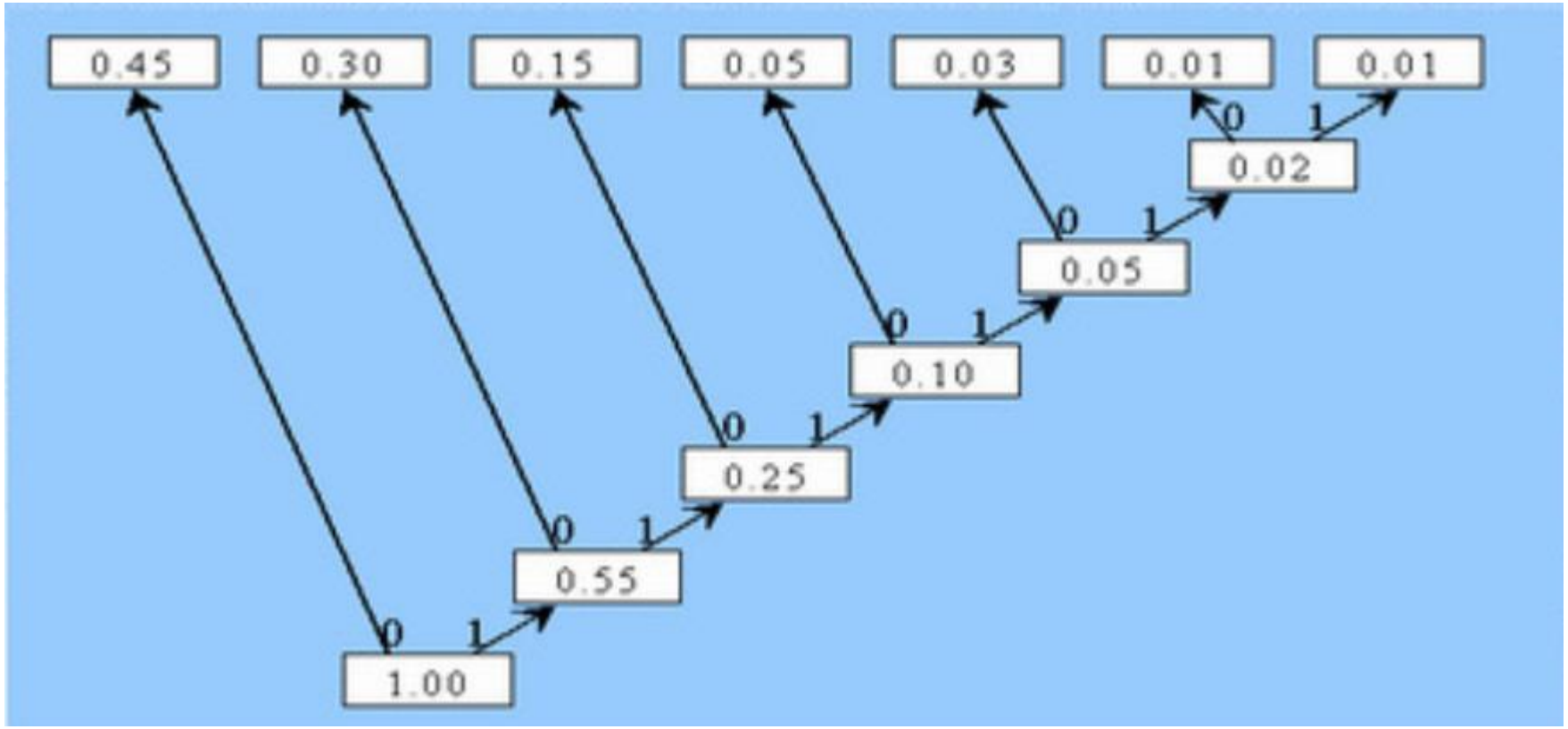
利用Huffman树进行操作码编码



利用Huffman树进行操作码编码



利用Huffman树进行操作码编码



Huffman编码操作码

指令序号	出现的概率	Huffman 编码法	操作码长度
I_1	0.45	0	1 位
I_2	0.30	1 0	2 位
I_3	0.15	1 1 0	3 位
I_4	0.05	1 1 1 0	4 位
I_5	0.03	1 1 1 1 0	5 位
I_6	0.01	1 1 1 1 1 0	6 位
I_7	0.01	1 1 1 1 1 1	6 位

采用*Huffman*编码法的操作码平均长度为：

（实际长度）

$$H=\sum_{i=1}^{7}p_i\times l_i=0.45\times1+0.30\times2+0.15\times3+$$

$$0.05\times4+0.03\times5+0.01\times6+0.01\times6$$

$$=1.97(\text{位})$$

操作码的最短平均长度为：　（理论长度）

$$H=-\sum_{i=1}^{7}p_i\times\log_2 p_i=0.45\times1.152+0.30\times1.737$$

$$+0.15\times2.737+0.05\times4.322+0.03\times5.059$$

$$+0.01\times6.644+0.01\times6.644$$

$$=1.95\ (\text{位})$$

采用3位固定长操作码的信息冗余量为：

$$R = 1 - \frac{H}{\lceil \log_2 7 \rceil} = 1 - \frac{1.97}{3} \approx 35\%$$

*Huffman*编码的信息冗余量仅为：

$$R = 1 - \frac{1.95}{1.97} \approx 1.0\%$$

与3位固定长操作码的信息冗余量35%相比要小得多

3. 扩展编码法

- Huffman操作码的主要缺点：
  - 操作码长度很不规整，硬件译码困难
  - 与地址码共同组成固定长的指令比较困难
- 扩展编码法：由固定长操作码与Huffman编码法相结合形成
- 例. 将前一例改为1-2-3-5扩展编码法及2-4等长扩展编码法，分别求操作码最短平均长度及其信息冗余量

7 条指令的操作码扩展编码法

指令序号	出现的概率	1-2-3-5 扩展编码	2-4 等长扩展编码
I_1	0.45	0	0 0
I_2	0.30	1 0	0 1
I_3	0.15	1 1 0	1 0
I_4	0.05	1 1 1 0 0	1 1 0 0
I_5	0.03	1 1 1 0 1	1 1 0 1
I_6	0.01	1 1 1 1 0	1 1 1 0
I_7	0.01	1 1 1 1 1	1 1 1 1
平均长度		2.0	2.2
信息冗余量		2.5%	11.4%

1-2-3-5编码法操作码平均最短长度：

$$H = 0.45\times1+0.30\times2+0.15\times3+$$

$$(0.05+0.03+0.01+0.01)\times5 = 2.00$$

1-2-3-5编码法信息冗余量：$R = 1-\frac{1.95}{2.00} = 2.5\%$

2-4等长扩展编码法操作码平均最短长度：

$$H = (0.45+0.30+0.15)\times2+$$

$$(0.05+0.03+0.01+0.01)\times4 = 2.20$$

2-4等长扩展编码法信息冗余量为：

$$R = 1-\frac{1.95}{2.20} = 11.4\%$$

操作码等长扩展编码法

操作码编码	说　明
0000 0001 …… 1110	4 位长度的 操作码共 15 种
1111 0000 1111 0001 …… 1111 1110	8 位长度的 操作码共 15 种
1111 1111 0000 1111 1111 0001 …… 1111 1111 1110	12 位长度的 操作码共 16 种

等长 15/15/15……扩展法

操作码编码	说　明
0000 0001 …… 0111	4 位长度的 操作码共 8 种
1000 0000 1000 0001 …… 1111 0111	8 位长度的 操作码共 64 种
1000 1000 0000 1000 1000 0001 …… 1111 1111 0111	12 位长度的操 作码共 512 种

等长 8/64/512……扩展法

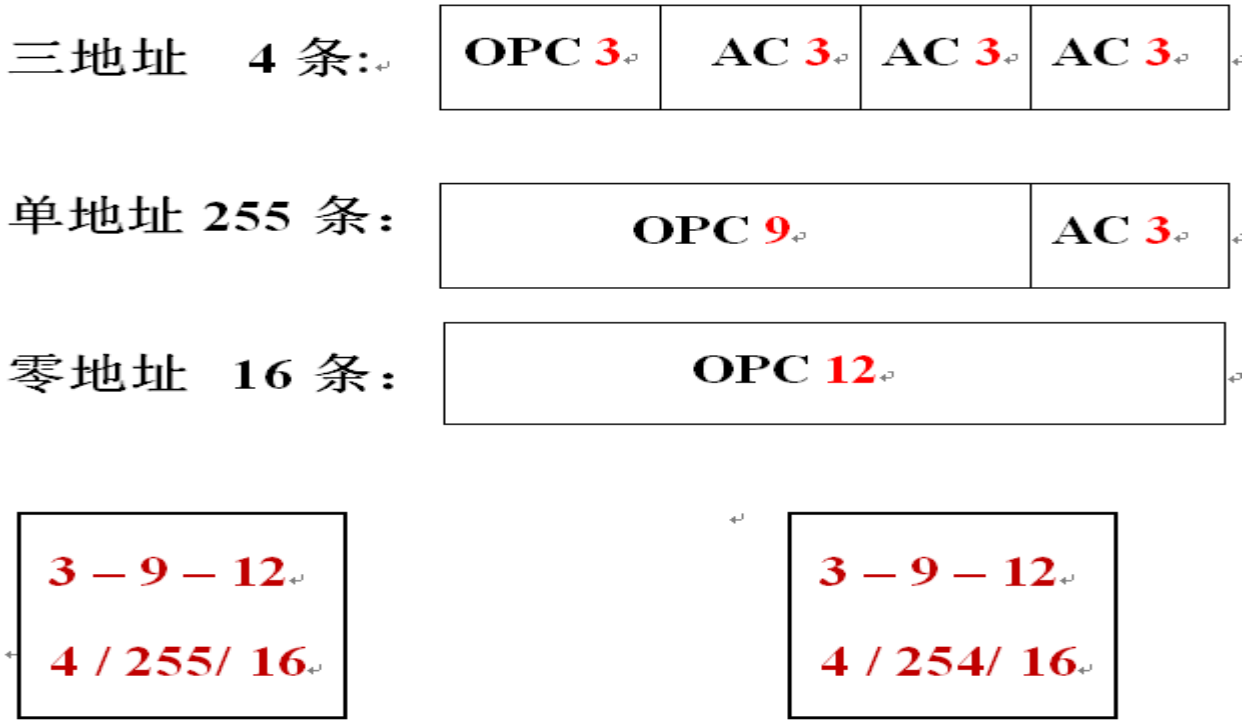
不等长操作码扩展编码法(4-6-10 扩展编码)

编码方法	各种不同长度操作码的指令			指令种类
	4 位操作码	6 位操作码	10 位操作码	
15/3/16	15	3	16	34
8/31/16	8	31	16	55
8/30/32	8	30	32	70
8/16/256	8	16	256	280
4/32/256	4	32	256	292

- 例1. 一个处理机共有10条指令I_1到I_{10}，各指令在程序中出现的概率如下：0.25，0.20，0.15，0.10，0.08，0.08，0.05，0.04，0.03，0.02

1. 采用Huffman编码编写这10条指令的操作码，并计算操作码的平均长度和信息冗余量
2. 分别采用2/8和3/7扩展编码法编写这10条指令的操作码，并计算平均长度和信息冗余量

- 例2. 若某机器指令系统要求：三地址指令有4条，单地址指令有255条，零地址指令有16条。假设指令字长为12位，每个地址码长为3位，能否以扩展操作码的方式为其编码？若其中单地址指令为254条呢？说明其理由



- 例3. 某机器指令字长16位，设有单地址指令和双地址指令两类，若每个地址字段为6位，且双地址指令有x条，问单地址指令最多有多少条？

RISC和CISC ------CPU两种架构

- 指令系统设计的思路
  - 指令系统复杂化，增强指令功能，设置一些功能复杂的指令，把一些原来用软件实现的、常用的功能改用硬件实现，减少用户编程负担 ——复杂指令系统CISC (Complex Instruction Set Computer)
  - 指令系统简单化，只保留功能简单、能在一个时钟周期内完成的指令，而其它复杂功能用一段子程序实现 —— 精简指令系统RISC (Reduced Instruction Set Computer)

RISC和CISC ------CPU两种架构

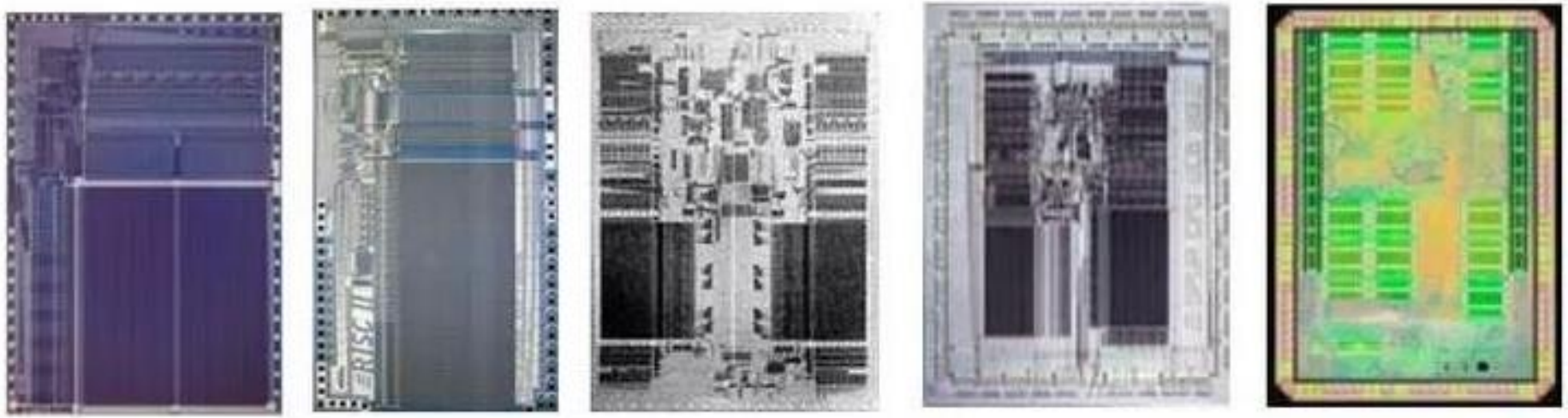
- 早期的CPU全部采用CISC架构
- CISC
  - 通过提供复杂指令来提高性能
  - 所含的指令至少300条，甚至超500条
  - 设计目的是用最少的机器语言指令来完成所需的计算任务
  - 使得硬件越来越复杂，造价也相应提高，日趋庞杂的指令系统不但不易实现，而且还可能降低系统性能
- CISC系统中一个典型程序的运算过程所使用的80%指令只占处理器指令系统所有指令中的20%
- RISC设计思想：指令系统应当只包含那些使用频率很高的少量指令，并提供一些必要的指令以支持操作系统和高级语言

2.4.2 RISC-V

2010年，伯克利研究团队要设计一款CPU。

Intel对x86的授权很严格，ARM的指令集授权费用很高，MIPS，SPARC，Open Power也都需要各自的公司授权。

伯克利研究团队决定设计一套全新的指令集。



2.4.2 RISC-V

- RISC-V 指令集是基于RISC-V原理**完全开源**，设计简单，采用**模块化**设计，易于移植unix系统。
- 在处理器领域，目前主流的架构为x86、ARM架构，但是他们作为商用的架构，需要保持架构的兼容性，需要保持许多过时的定义。RISC-V相对而言不需要。
- RISC-V架构相比于其他成熟商业架构，不同之处在于，**它是一个模块化的架构，因此RISC-V架构不仅短小精悍**，而且其不同的部分还能够以模块化的方式组织在一起。
- **将基准指令和扩展指令分开，可以通过扩展指令做定制化的模块和扩展**，基本的RISC-V指令仅有40多条，加上其他的模块化扩展指令总共不超过100条。**RISC-V的基准指令确定后将不会再有变化，这是RISC-V稳定性的重要保障**

王鹏 集成电路设计

- pengwang@shu.edu.cn

2.5 时钟频率

2.5 时钟频率

- 时钟频率是指同步电路中时钟的基础频率
  单位为hz，常用单位为MHz，GHz，THz
- **完成某条指令的时钟周期数是固定的，因此提高时钟频率通常就意味着CPU性能的提升**
- CPU的时钟频率越来越高，系统中的其他低速部件的工作频率难以跟上CPU的速度从而与之保持同步。故，“倍频”的概念被提出
  - 允许CPU的时钟频率相比外频成倍提高：CPU时钟频率=外频*倍频
- **提升CPU的时钟频率是提升CPU性能最直接的方法**
  - 但时钟频率不可无限提升的，与CPU的架构、制造工艺相关
  - 更高的时钟频率会导致CPU的功耗和发热量呈指数速率增长
- 改进架构和实现各层面上的并行则是现代CPU提升性能的主要方式，
  - 增加CPU的内核数，而主频一般在1～3GHz。

  内核数能无限提升吗？

2.6 并行

- SISD:指令顺序执行，一次一条，每条指令可能需要多个时钟周期。CPU资源无法充分利用。
- 多指令调度，并行运行，提高资源利用率，提高速度。
- CPU层次并行的三种实现方式：指令级并行、线程级并行及数据并行。

2.6.1 指令级并行

Instruction-Level Parallelism，ILP

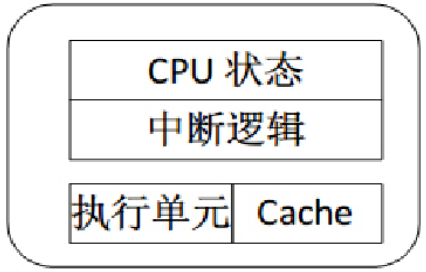
目的：**让cpu在同一时刻并行执行多条指令**

基本原理：指令不相关时，可以重叠起来并行执行

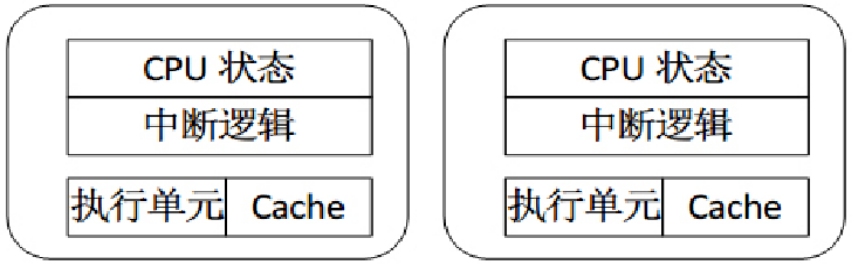
例：Intel Pentium、Pentium Pro和Pentium Ⅱ处理器分别采用了两条和三条独立的指令流水线，每条流水线平均在一个时钟周期内执行一条指令，因而它们平均一个时钟周期可分别执行2条和3条指令。

2.6.2 线程级并行

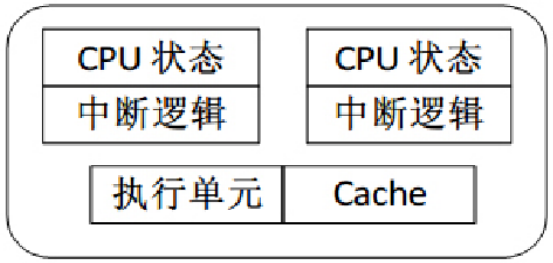
实现CPU 线程级并行的基础是CPU具有多个物理内核，每个核执行一个独立线程，实现多线程并行，被称为多核技术



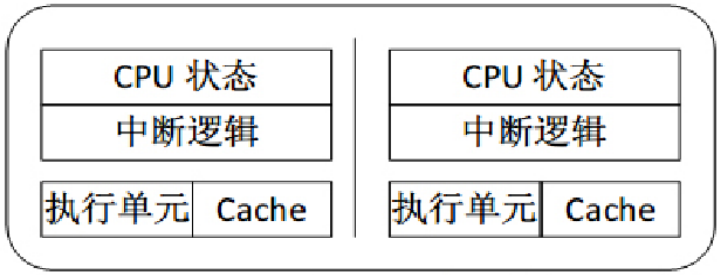
（a）单核处理机结构



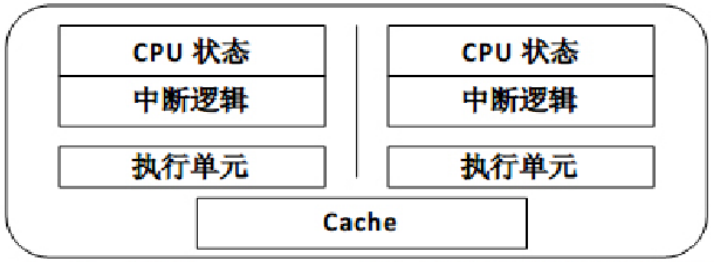
（b）多处理机结构



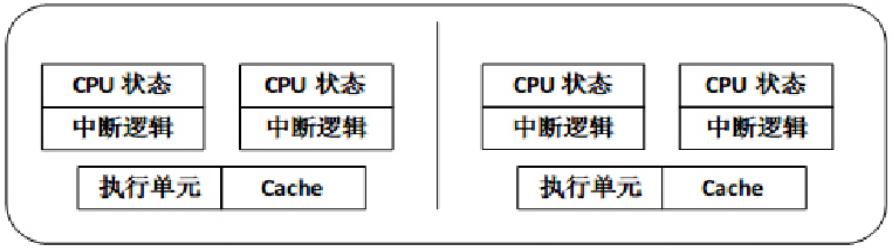
（c）超 线程处理机结构



（d）多核处理机结构

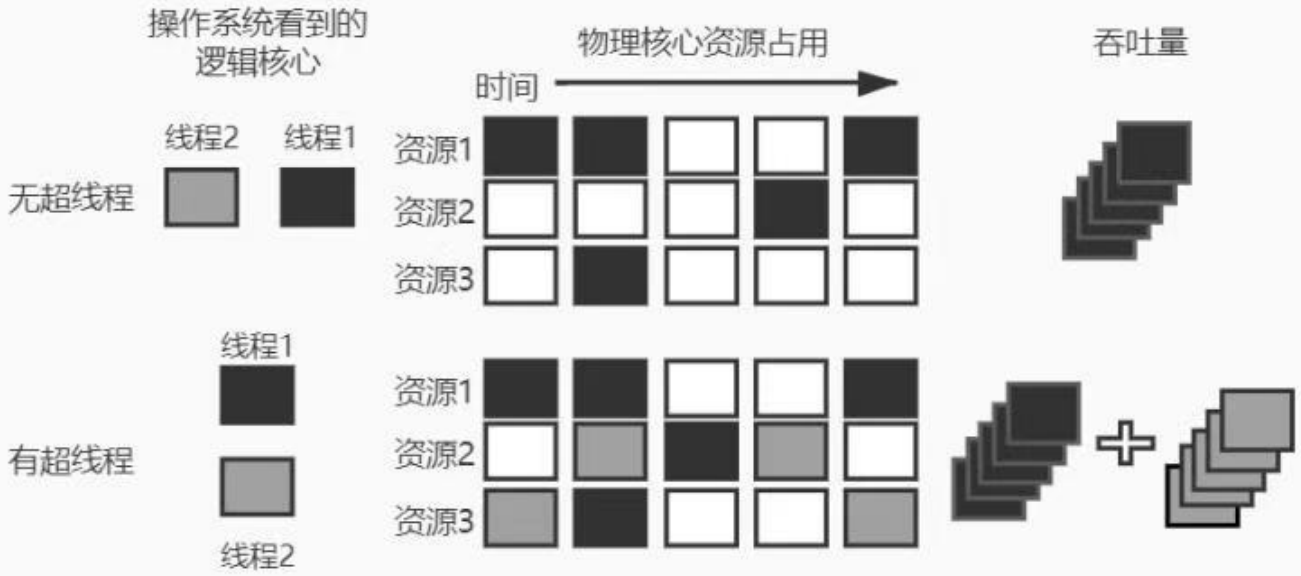


（e）共享 Cache 的多核处理机结构



（f）支持超线程的多核处理机结构

2.6.2 线程级并行



超线程（Hyper-Threading，HT）技术

采用超线程即是在同一时间里，应用程序可以使用芯片的不同部分，提高芯片利用率。单线程芯片在任一时刻只能够对一个线程进行操作，CPU内部资源无法充分利用，而超线程技术可以使芯片同时进行多线程处理，这些线程可以同时使用不同的资源，或者象流水线一样轮流使用相同的资源，从而充分利用CPU内部资源。

例如图中所示，若两个逻辑内核都需要用到处理器的同一个组件，那么一个线程必须要等待。而超线程技术通过并行执行两个线程，从而提升单内核处理器的资源利用效率。

单个核多线程？超线程技术

2.6.3 数据并行和SIMD指令集合

- 数据并行：把需要处理的数据划分成若干部分，分配到不同的处理部件上，并在每个处理部件上运行相同的处理程序/指令对所分派的数据进行处理，即遵循SIMD（单指令多数据流）模型。
  - 在多媒体领域应用广泛。在进行图像处理时，有些格式的图像数据的特点是：
    ① 图像数据的像素数量庞大，如分辨率是1024*768像素的图像；
    ② 每个像素所占的位数往往小于或等于 8 位，如使用32位处理器进行运算，就能同时处理4组不同的数据。
- MMX
- SSE、SSE2、SSE3、SSSE3（Supplemental SSE3）、SSE4
- AVX、AVX-512

2.7 多核技术

2.7 多核技术

现代处理器结构

- 早期，多CPU技术 → 提升性能
  （1）如何管理多个CPU及其附属资源在单个操作系统上的运行?
    - 对称多处理器结构（SMP）
    - 非一致存储访问结构（NUMA）
    - 海量并行处理结构（MPP）；
  （2）多CPU间通信开销，更多CPU不能获得线性的性能提升
  （3）CPU数增多，系统越复杂，设计/制造/运营的费用大幅增加
- 多核技术，多CPU内核 → 单IC
  - 核，独立工作、独立计算单元和高速缓存，并且一般共享末级缓存和内存
  - 主板的单个Socket可适应多核CPU，不需更改主板硬件结构

2.7 多核技术

现代处理器结构

每个内核都具有独立的L1缓存和L2缓存，并且共享末级（L3）缓存和内存控制器。

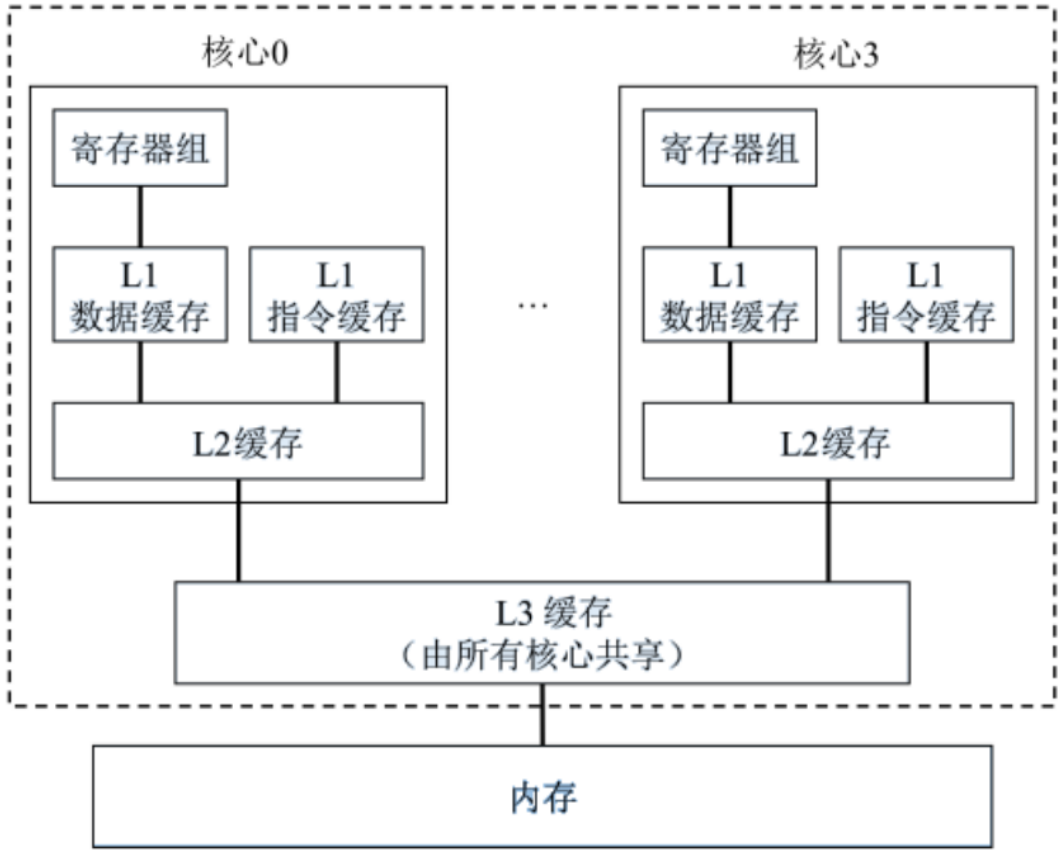


图2-4　Intel Core i7处理器的多内核组织结构

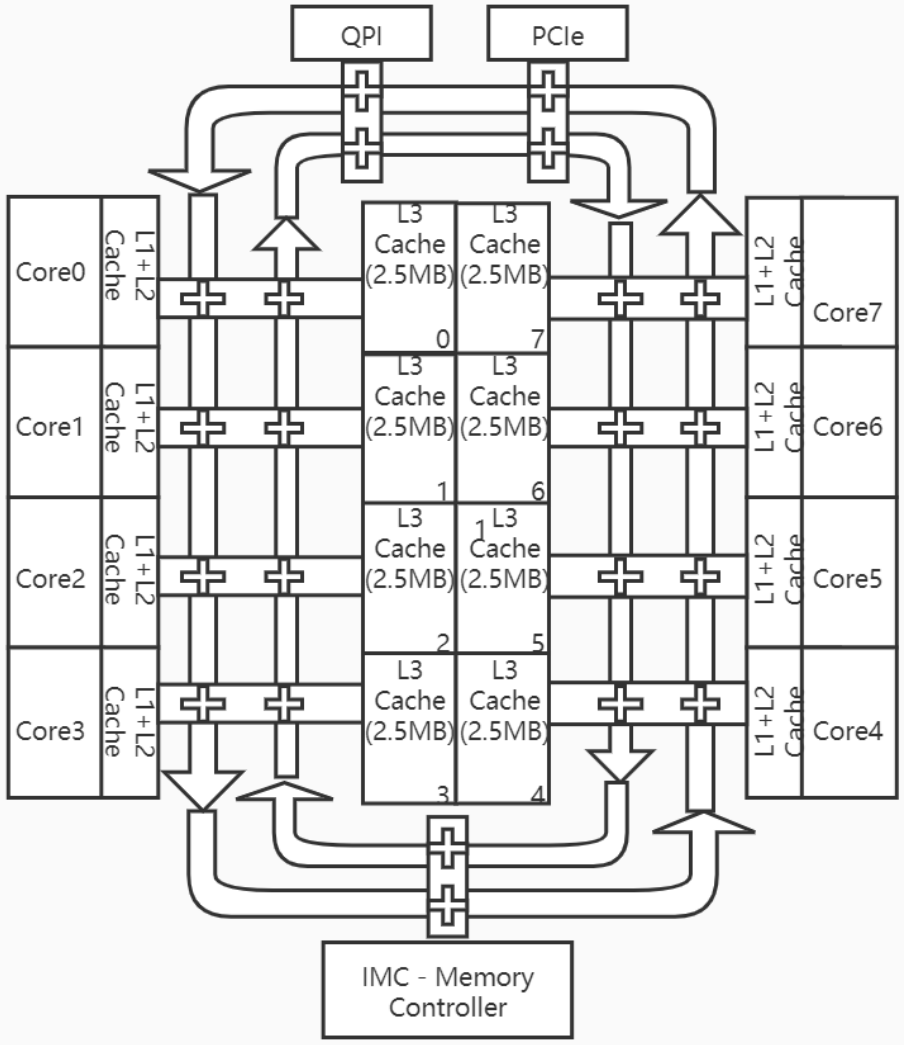
2.8 CPU内部互连

2.8.1 早期的星形总线与跨Socket互连

现代处理器结构

- 早期cpu，核数少，内部模块少，结构单一，星形总线
  - 星形总线中
    - CPU在中心位置，各个模块都与其连接
    - 各模块不直接交互，须通过内核中转
  - 为实现单机扩展，一种简单的方法是在一个主板上放置多个CPU插槽
    - 不同CPU的内核之间的访问需经过主板，延迟较高

2.8.2 Intel 环形总线架构



环形总线优点

① 双环，任两个接入点之间的距离不超过接入点总数的一半，延迟较低。

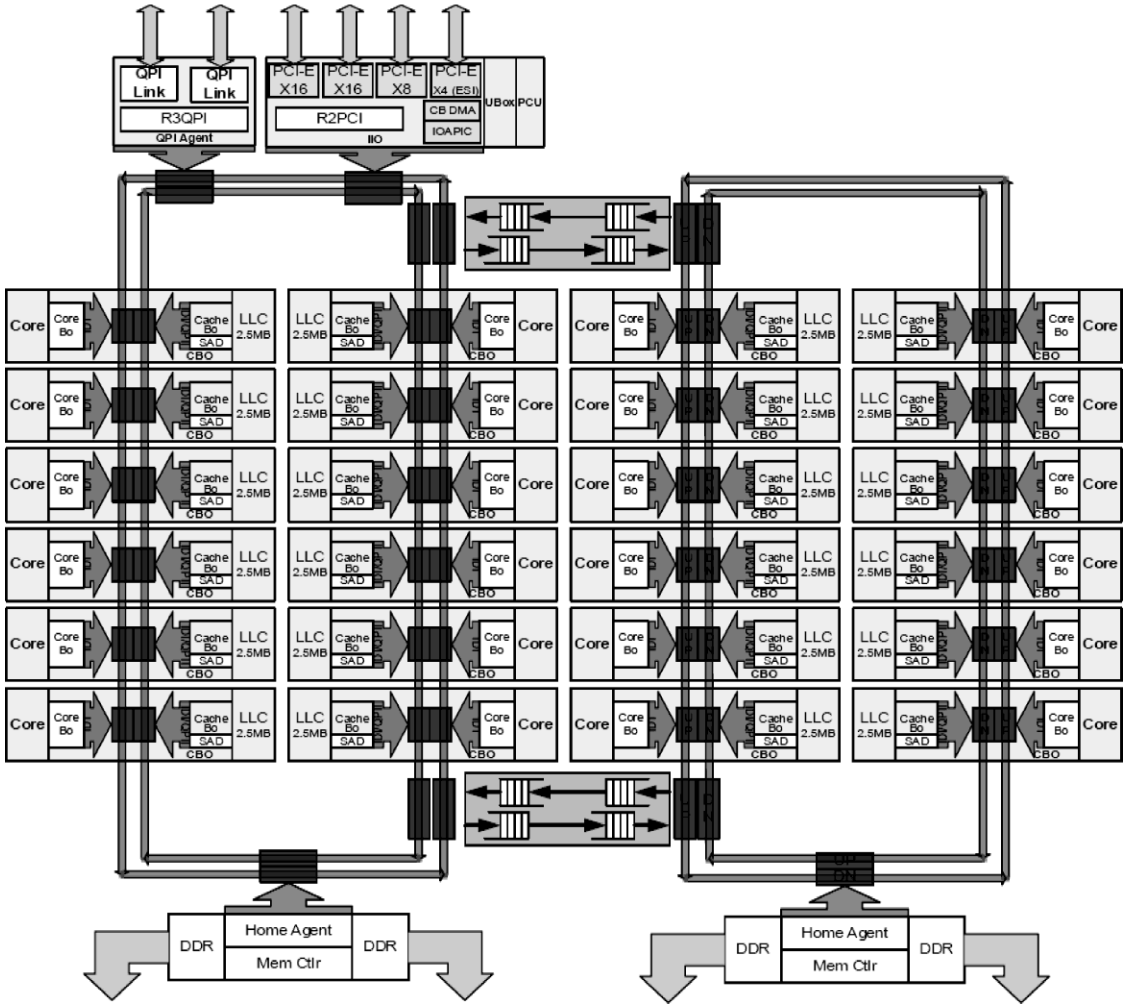
② 各模块间交互方便，不需通过核中转，使得一些高级加速技术如直接缓存访问（Direct Cache Access，DCA）等应运而生。

③ 高速的环形总线保证了高性能

在Nehalem架构中，核到核的访问延迟只有60 ns左右，带宽达192GB/s。

④ 方便灵活。增加一个核，只要在总线上加一个新的接入点。

2.8.2 Intel 环形总线架构

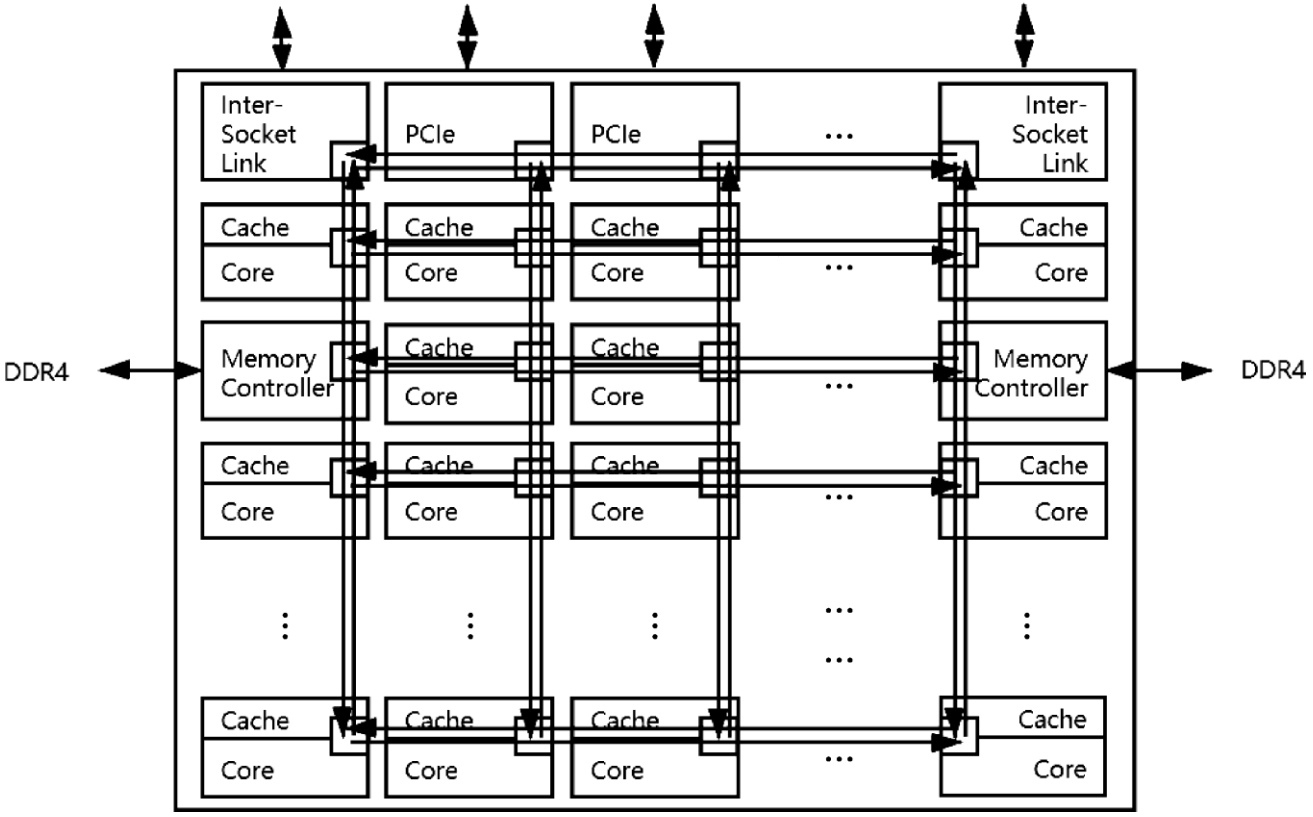


环形总线缺点及改进

核数的进一步增加，环形总线长度越来越长，跨核访问的延迟越高。

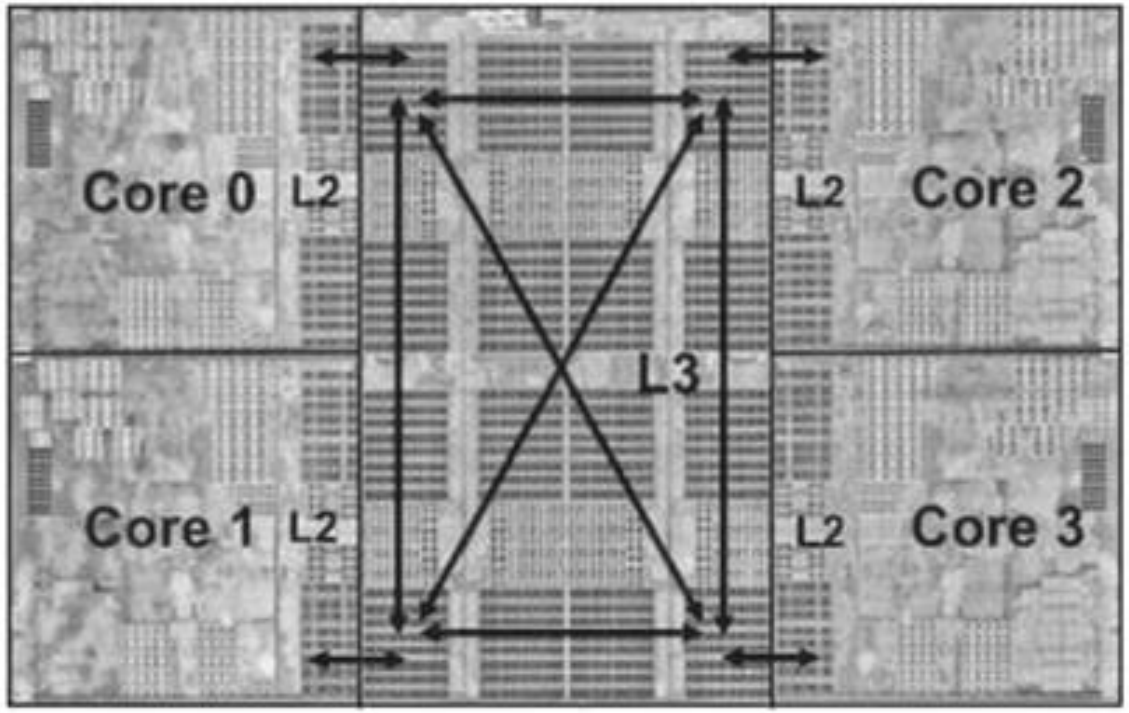
Intel研究发现，在核数>12时，系统延迟变得不可接受、缓存间难一致性。
故，引入一种变种，如左图：
Intel在Xeon E5 v4 HCC（High Core Count，高内核数产品）中采用的双环形总线互连结构。

2.8.3 Intel Mesh总线架构



- 比环形总线，Mesh有更强的可扩展性
  - 不增加平均访问延迟，连接更多核
- 位于每个总线连接点处的控制器提供了不同核之间最佳路径的路由功能
- 避免了环形总线必须顺序访问的问题，访问延迟大幅降低
- I/O操作的初始化延迟也得到了改善

2.8.4 AMD CCX架构和Infinity Fabric总线



AMD CCX架构

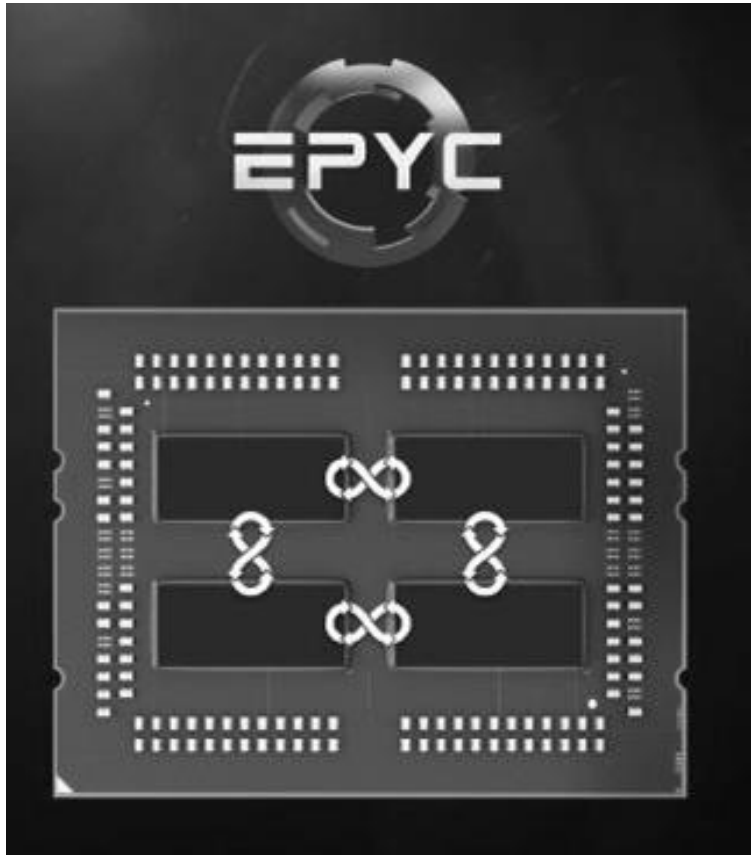
AMD 的模块化思想体现为：

① 由若干核组成的CCX（Core Complex，内核复合体）

② 实现CCX之间互连的Infinity Fabric总线。

核，位于CCX的四角，L2缓存和L3缓存则被排布到4个内核的正中间，L3缓存由4个内核内部共享，速度同步于最高的内核。同时，4个内核之间采用全连接的方式，访问延迟极低。

2.8.4 AMD CCX架构和Infinity Fabric总线



Infinity Fabric 支持4个CCX

Infinity Fabric与ccx模块化的高可扩展性优势

1. 不更改架构设计的情况下，可以通过增加CCX的数量来达到更多的内核数
   - 第二代EPYC处理器最多64核
2. 不制造1个32核的CPU Die，而是4个8内核的Die，后者工艺要求更低
3. 芯片上的某个CCX无法使用时，可以屏蔽，其他CCX仍能正常工作，并仍能通过Infinity Fabric共享被屏蔽的CCX的L3缓存

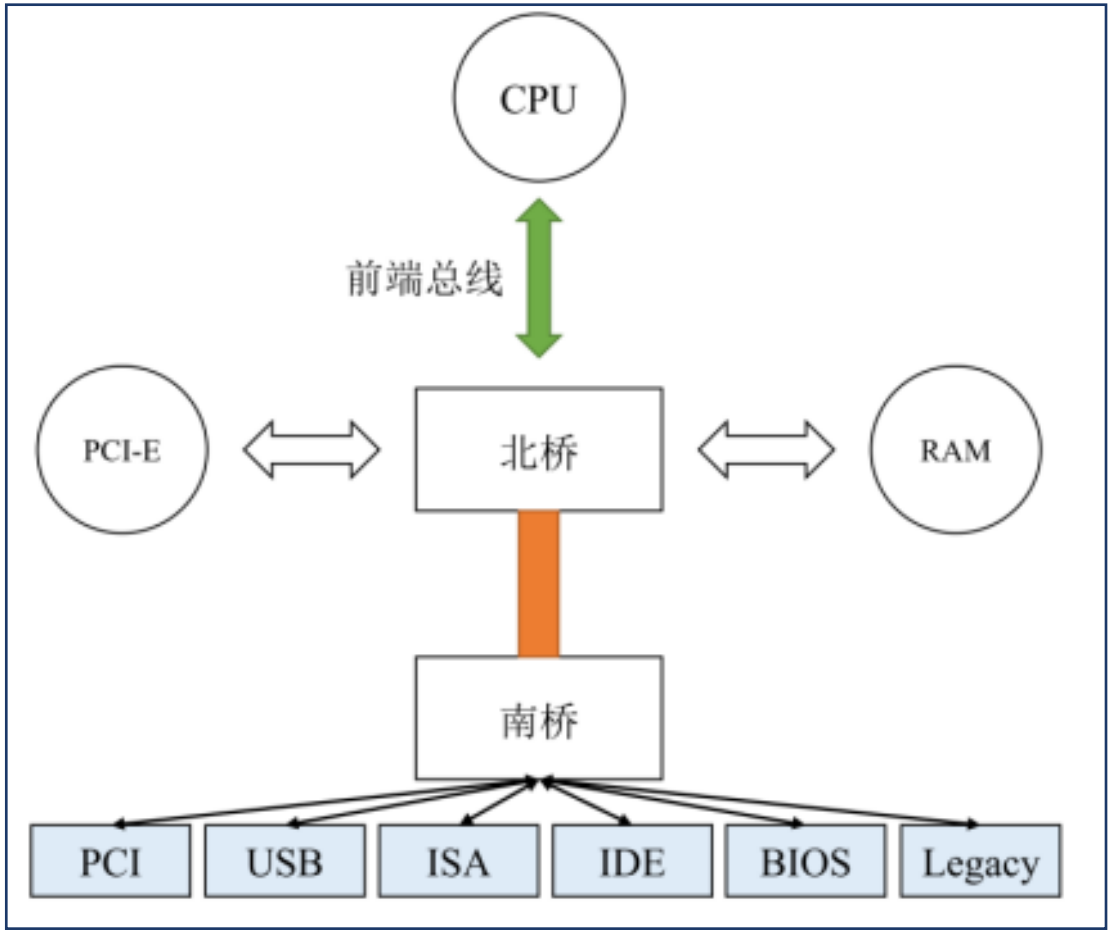
2.9 CPU外部互连

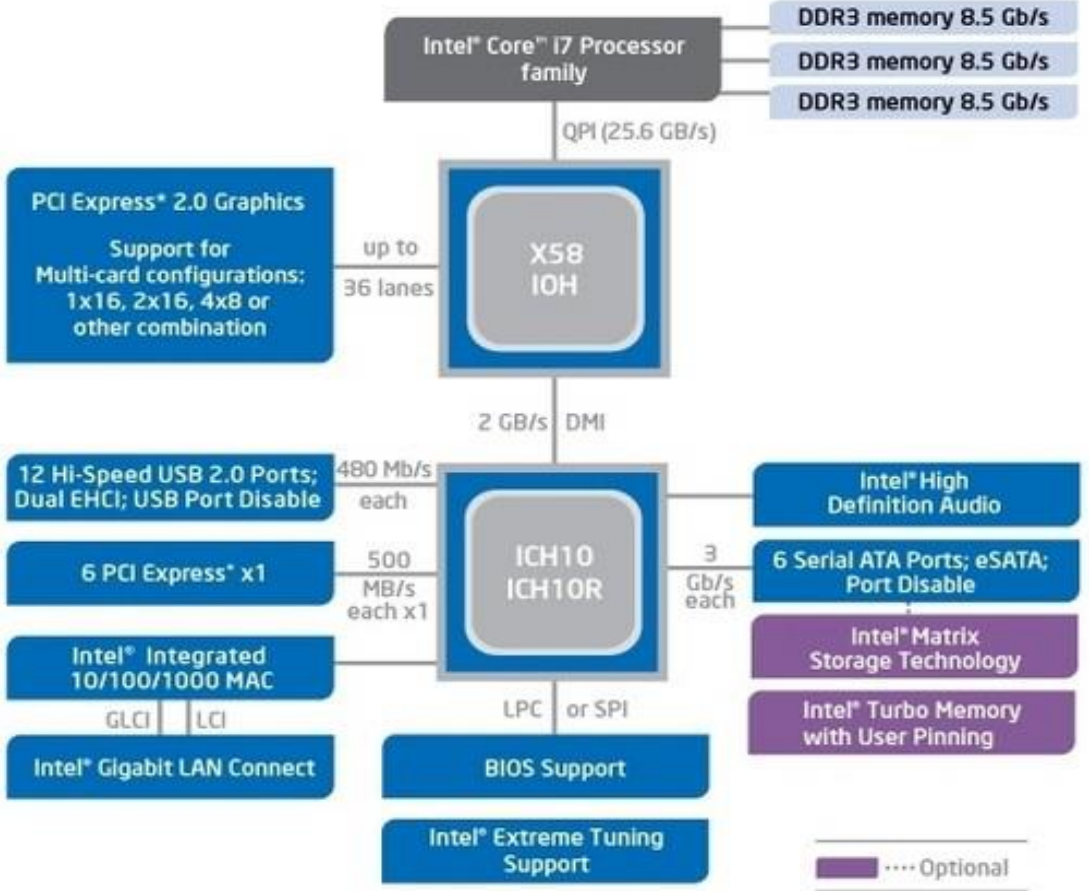
2.9 CPU外部互连

由北桥和南桥组成的芯片组

早期计算机的芯片组由两个部分组成：北桥和南桥。

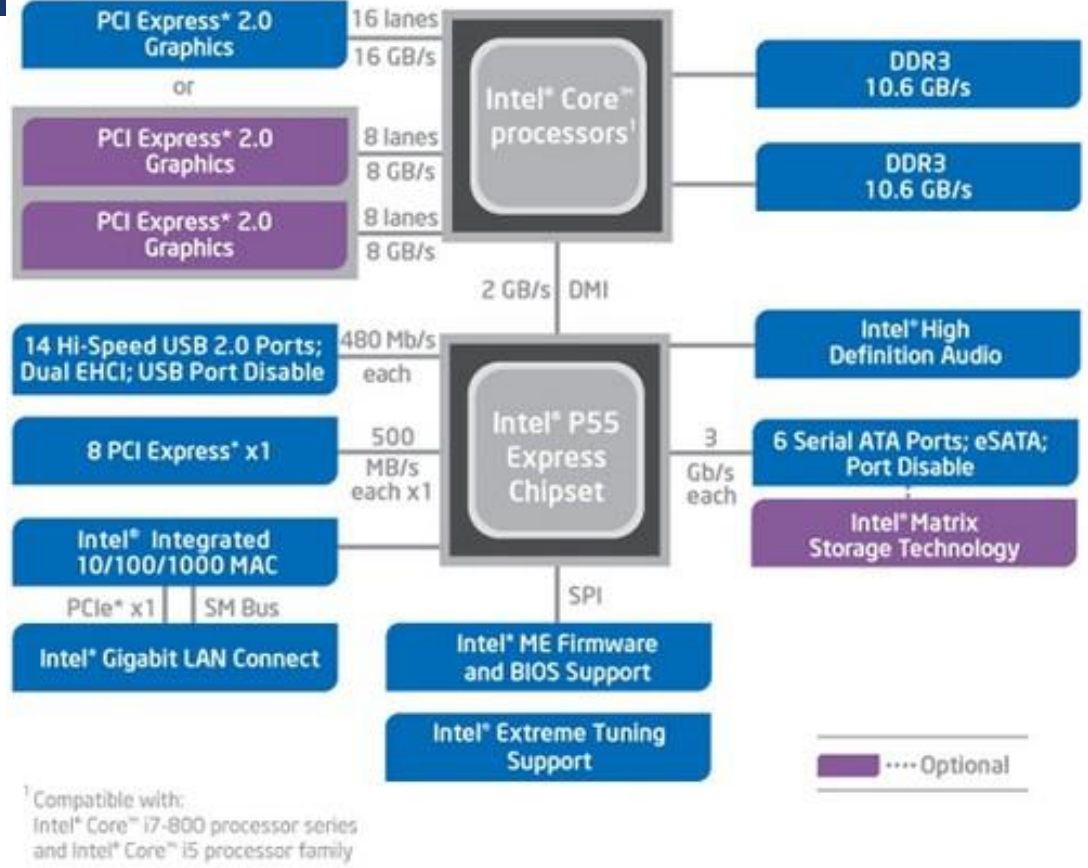
- 北桥
  - 直接通过外部数据总线（又称前端总线，FSB）与CPU相连，负责CPU与内存和高速PCI-E设备（如显卡等）的通讯
- 南桥芯片
  - 通过北桥间接与CPU相连
  - 并控制声卡、网卡、硬盘、USB等低速外设





Bloomfield平台架构

集成内存控制器，CPU直接与内存相连，大大降低了内存访问延迟，同时Intel采用了新的QPI总线连接北桥取代之前的FSB。



Lynnfield平台架构

Intel将高速PCI-E总线控制器也集成到了CPU中，从而彻底在主板上移除了北桥芯片，并继续保留南桥芯片作为新的主板芯片组，它通过低速DMI总线与CPU直接相连。

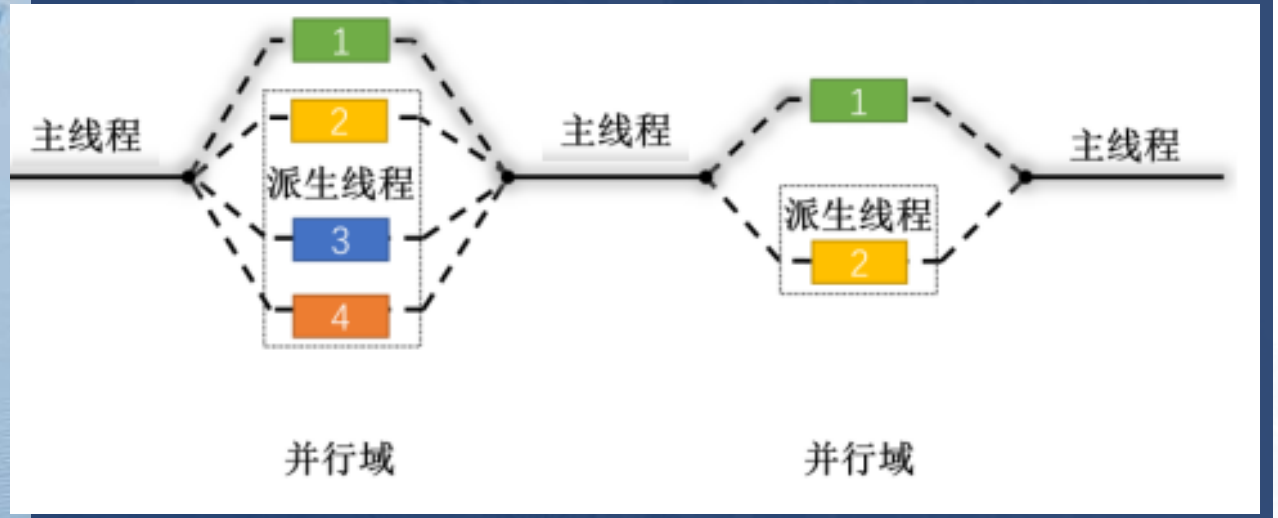
2.10 OpenMP多线程并行编程

2.10 OpenMP多线程并行编程

- 进程是资源分配的最小单位，线程是CPU调度的最小单位
- OpenMP的编程模型以线程为基础，通过编译制导指令来显式地指导并行化
- OpenMP的执行模型采用Fork-Join的形式
- OpenMP采用共享内存模型

OpenMP的编程模型介绍

OpenMP的Fork-Join执行模式



- 开始时，只有一个“主线程”
- 主线程遇到并行域，派生出线程来并行即Fork阶段
- 主线程和派生线程被分配到空闲的CPU核上并行执行
- 当线程组执行完成后，退出并释放CPU核即Join阶段
- 主线程又开始执行串行部分
- Fork-Join过程，可以有若干个，还可以嵌套

2.10.1 编译制导语句

- 编译制导语句

指编译器编译程序的时候会识别的特定的注释语句，这些注释语句包含OpenMP指令。

例如，在C/C++程序中，OpenMP以#pragma omp开始，后面跟具体的功能指令。

```
#pragma omp <directive> [clause [[,]clause]…]
```

其中的directive部分就包含了具体的编译制导语句，这些编译制导语句用来分配并行任务或者实现并行线程间的同步。
程序中循环的计算量很大，因此需要加速循环的计算。

2.10.1 编译制导语句

常用的功能指令如下：

- parallel：用在一个结构块之前，表示这段代码将被多个线程并行执行。
- parallel for：parallel和for指令的结合，也用于for循环语句之前，表示for循环体的代码将被多个线程并行执行，同时具有并行域的产生和任务分配两个功能。
- reduction：执行指定的归约运算。
- single：用在并行域内，表示一段只被单个线程执行的代码。
- critical：用在一段代码临界区之前，保证每次只有一个OpenMP线程进入。
- barrier：用于并行域内代码的线程同步，线程执行到barrier时要停下等待，直到所有线程都执行到barrier时才继续往下执行。

2.10.2 API 函数

除了上述编译制导指令，openmp还提供了一组API函数用于控制并发线程的某些行为，下面是一些常用的OpenMP API 函数以及说明:

- omp_get_thread_num：返回线程号。
- omp_set_num_threads：设置后续并行域中的线程个数。
- omp_get_num_threads：返回当前并行域中的线程数。
- omp_get_num_procs：返回系统中处理器的个数。

例题

例 2-3

编写完整的OpenMP程序：要求定义数组A，实现初始化整数数组A[1000][1000]，对二维数组A的每一行从下标为1的元素开始进行求和，并将求和结果保存在对应行的第0号元素。要求程序中可以根据CPU核数设置线程数。

分析

二维数组的运算一般采用双层for循环。本例题很简单，显然数组计算的行和行之间没有数据相关。可以使用2种方法实现：

1、将计算以行为单位逐行分配到各个线程进行并行计算。

2、将数组按行平均分给各线程，但是这些行是连续的。这样做的好处是使并行计算的粒度比较大，可以减少调度开销。

下面的例程就是使用的方法2。

例程代码

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include "omp.h"
#define N 1000
#define OMP_THREADS 4 // CPU OpenMP 线程数

int main(void) {
    int LINE_FOR_CPU_THREADS = (N + OMP_THREADS - 1) / OMP_THREADS; // 定义每个CPU
线程计算的行数
        int A[N][N]; // 定义数组A
        int i, j; // 随机初始化数组A，每行从下标1开始，第0个元素用于记录该行最大值
        for (i = 0; i < N; i++){
            A[i][0] = 0;
            for (j = 1; j < N; j++){
                A[i][j] = rand()%10;  }  }
```

例程代码

```
#pragma omp parallel for num_threads(OMP_THREADS) // 启动多线程计算
    for (i = 0; i < OMP_THREADS; i++) {
        printf("ThreadIdx: %d\n", i);
        int k, lineID, sum;
        int line_for_cpu_idx = LINE_FOR_CPU_THREADS * i;// 计算该线程的起始计算行号
        for (lineID = line_for_cpu_idx; lineID < line_for_cpu_idx + LINE_FOR_CPU_THREADS; lineID++) {
            if (lineID >= 0 && lineID < N) {
                sum = 0;
                for (k = 1; k < N; k++) {
                    sum += A[lineID][k];
                }
                A[lineID][0] = sum;  }  }  }
    for (i = 0; i < N; i++) { // 回到主线程，输出计算结果
            for (j = 0; j < N; j++)
            printf("%d ", A[i][j]);
            printf("\n");  }
    }
```

2.10.2 API 函数

在使用OpenMP编程注意点：

- 创建线程时的开销。例如，若并行循环体内的计算量很小，即使使用OpenMP也并不一定能够获得性能的提升。
- 读写连续存储的数据访问开销比较小。

并行程序分析方法：

① 计算规模。从很小开始，直到计算规模大于内存大小；

② 从一个线程到线程数大于超线程数量。

观察性能变化，想想和存储、调度、通信等系统结构的关系，运用一些工具查看原因，并进行性能优化。上述测试对于理解并行计算特点和性能优化很有帮助。

2.11 GPU

2.11.1 GPU简介

GPU是Nvidia公司首先提出的概念，由图形芯片来负责几何转换和光照等图形处理操作

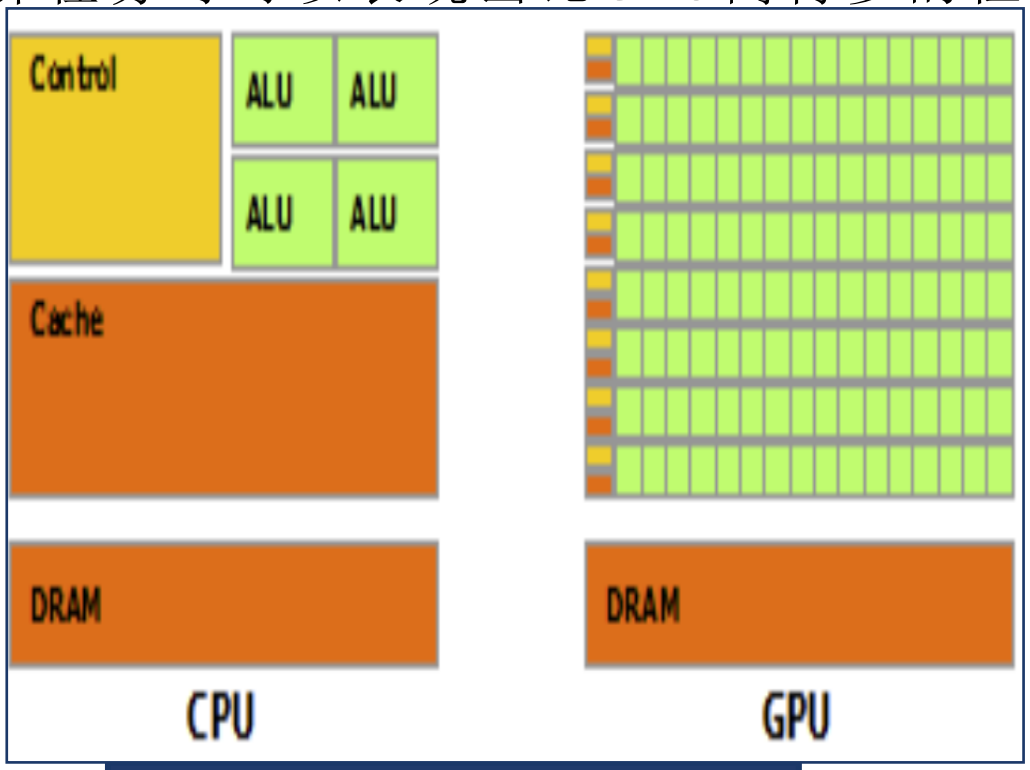
CPU不擅长这些操作，因为它的浮点计算性能不强

另外CPU很忙！

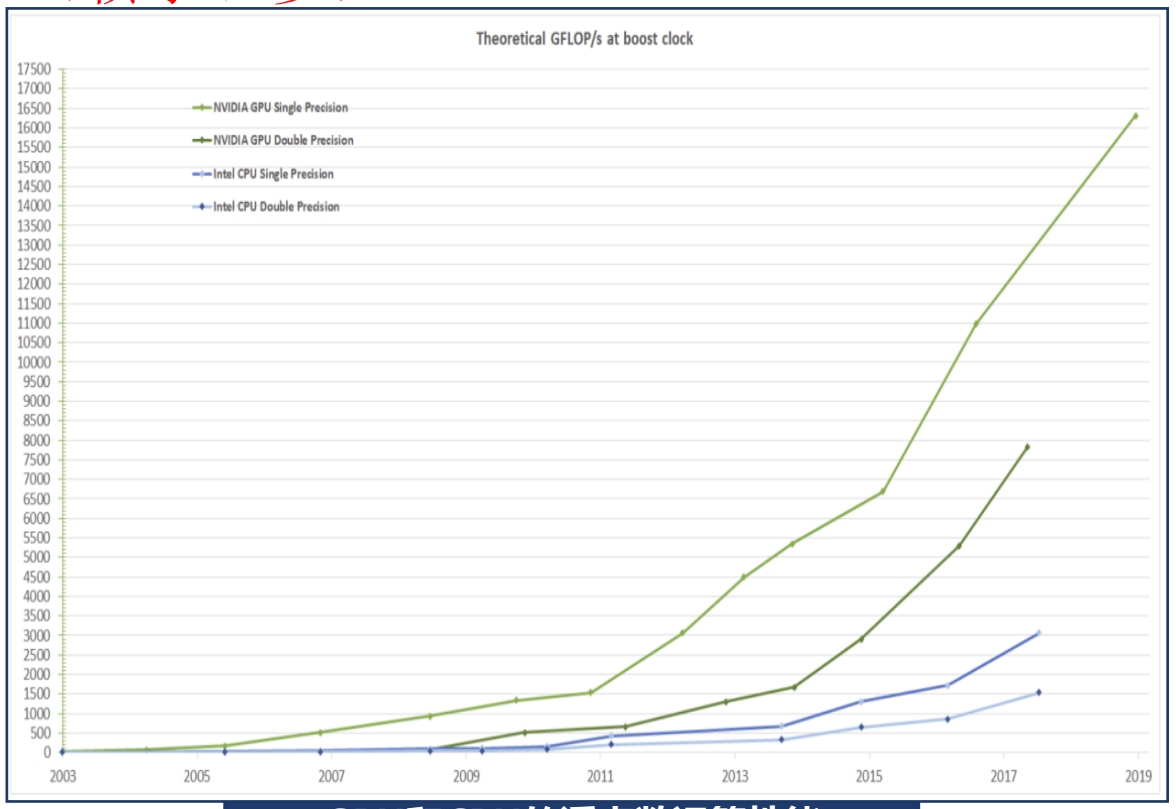
2.11.2 GPU硬件结构

从硬件设计上，CPU 是面对通用计算需求，需要处理各种任务，因此CPU的设计复杂，目前每个CPU的核数仅能达到几十个；（核大、少）

GPU专门面对高并发的计算任务，设计较CPU相对简单，因此每个核比CPU“小”多了，每块GPU的核达到上千个。由于在硬件上具有比CPU多得多的运算核心，GPU在进行浮点数运算任务时可以表现出比CPU高得多的性能。（核小、多）



CPU和GPU的硬件结构区别



GPU和CPU的浮点数运算性能

2.11.2 GPU硬件结构

基本的运算单元被称为流处理器（SP）。多个流处理器、寄存器、缓存和指令控制单元组成流多处理器（SM）。SM是GPU执行指令的基本单位，在同一SM内的SP具有共享的高速一级缓存。整个GPU则是由多组SM、共享二级缓存以及显存控制器等组成。

- SP
- SM
- Tensor Core

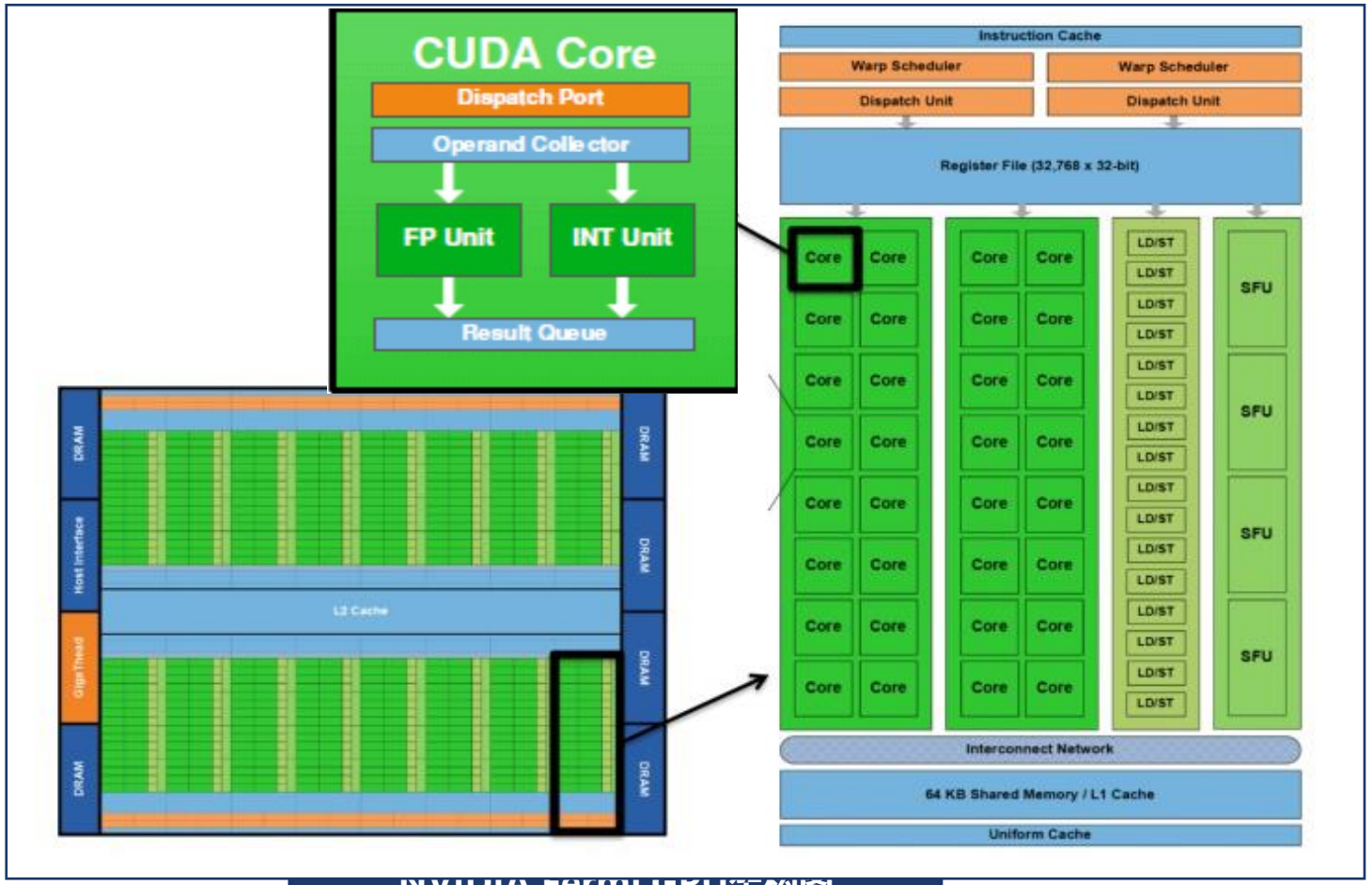
系统结构是最具革命性的：

GPU-->GPGPU

单精度-->双精度　　支持HPC计算

双精度-->半精度　支持AI计算

Tensor Core出现　支持AI计算



NVIDIA Fermi GPU架构图

2.11.3 GPU的存储层次

可由上至下分为三级，速度依次降低。

①寄存器(Register)位于GPU的每个SM中，访问速度最快，用于存放线程所需要的变量，具有线程私有的性质。

②共享内存(Shared Memory)位于每个SM中，访问速度仅次于寄存器，由一个SM中的所有线程共享。共享内存主要存放频繁修改的变量，可以为局部多线程建立通信。；

③全局内存（GlobalMemory）位于GPU片外存储体，即显存。全局内存容量最大，但访问延迟高、速度较慢。所有GPU需要处理的数据都必须先传入全局内存，最后在程序执行完毕后从全局内存传给主机的内存。

2.12 CUDA

2.12.1 CUDA简介

基于C语言，定义了“host-device”（主机-设备）的GPU编程模式。CUDA编程基于SIMD编程模型

- GPU上执行的代码称为核函数（kernel）
- 执行核函数之前，用户需要（手动）将数据通过PCI-E通道由内存复制到显存中
- 核函数执行完毕后，用户再将运算后的数据由显存复制回主机的内存中。

CUDA程序的一般执行流程为：

①调用cudaMalloc()函数，在显存中开辟数据存储空间；

②调用cudaMemcpy()函数，将数据由主机内存复制到GPU显存；

③启动kernel函数进行相关高并行度计算；

④调用cudaMemcpy()函数，将数据由显存取回内存中；

⑤调用cudaFree()函数，释放显存占用空间。

2.12.2 线程管理

CUDA线程的组织，分为**线程、线程块和网格**三个层次：

① 线程（Thread）是CUDA程序的基本执行单元，每个线程内的指令都会顺序执行。遵循SIMD编程模型，所有的线程都会执行相同的代码或相同代码的不同分支。理论上，所有的线程都是并行执行的，没有先后之分。

②线程块（Block）是由一组线程组成的。每个线程块内部的线程之间可以进行协作，有可以共同访问的共享内存。在CUDA程序执行时，每个线程块会在GPU上的同一个流多处理器（SM）中执行，而每个线程块内的线程又会以线程束（Warp）为单位，分组在SM中执行，**线程束是GPU在执行时调度的最小线程单位。线程束的大小固定为32**。

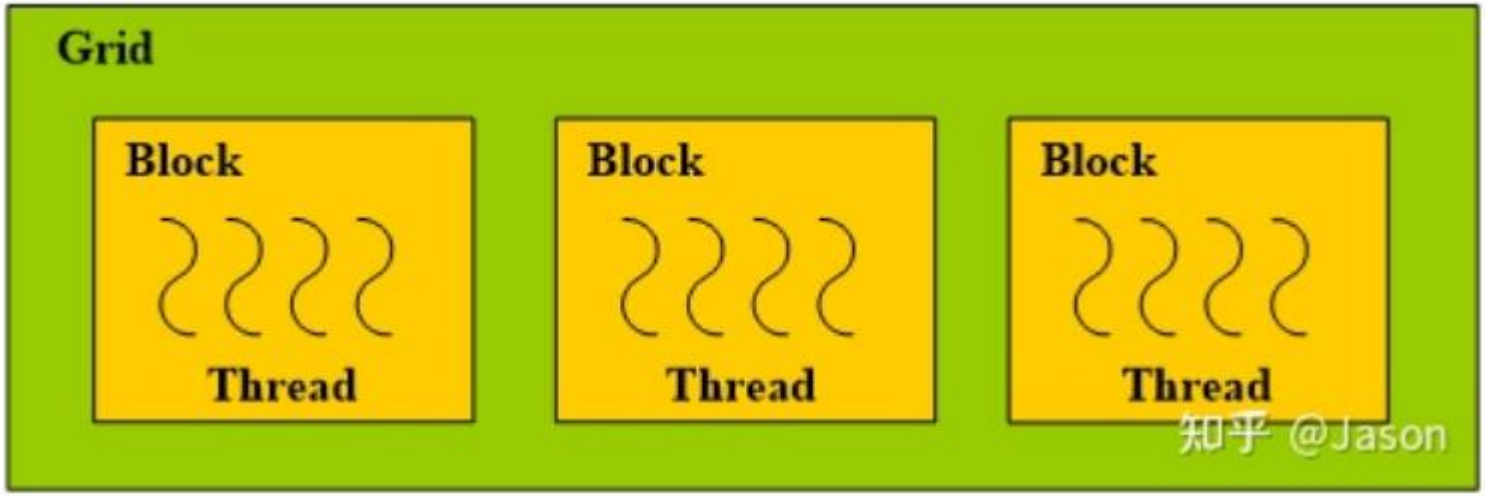
③网格（Grid）是一组线程块的集合。网格里的线程块会被调度到GPU的多个SM上去执行。线程块之间并没有同步机制，线程块被执行的先后顺序是不确定的。

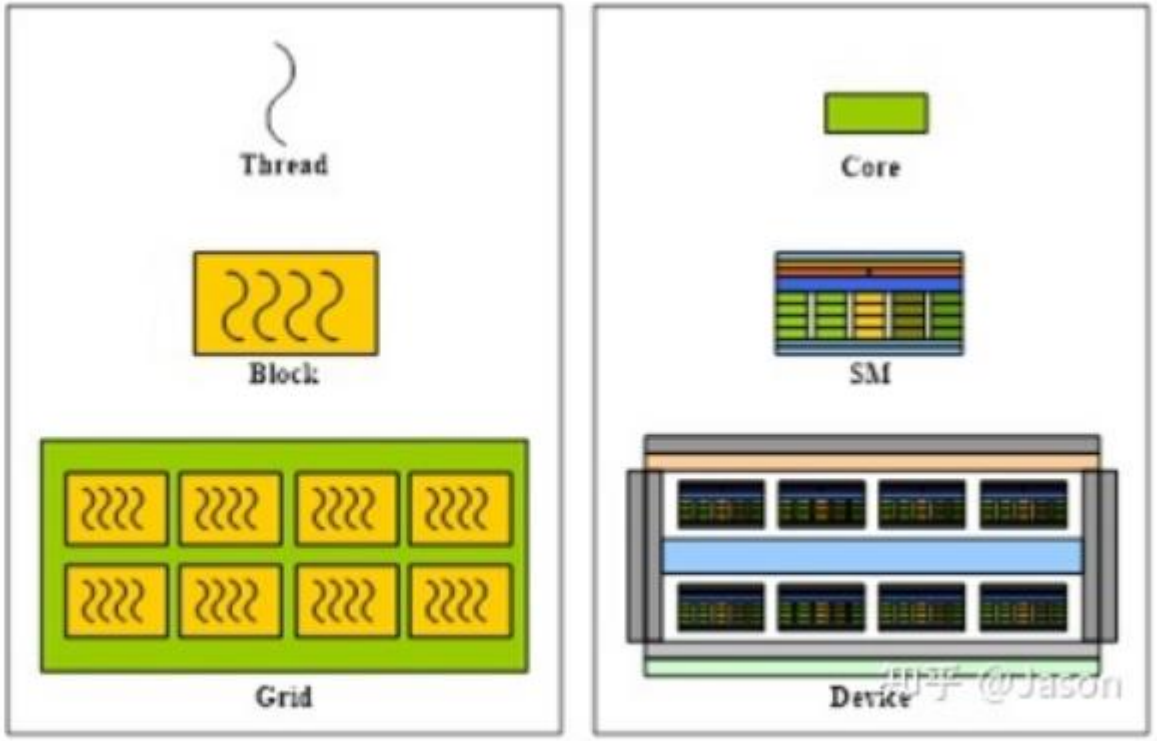
CUDA核函数

- 核函数是GPU端运行的代码，规定GPU的各个线程访问哪些数据执行什么计算
- 编写核函数必须遵循的GUDA规范
  - 必须写在*.cu文件中
  - 必须以_global_限定符声明定义
  - 返回类型必须是void
  - 不支持可变数量的参数
  - 核函数内部只能访问设备内存
  - 核函数内部不能使用静态变量

核函数如何使用线程

- CUDA从逻辑上将GPU线程分为三个层次：网格Grid, 线程块block，和线程thread
- 每个核函数对应一个Grid，一个Grid中有一个或多个block, 一个block中有一个或多个thread





CUDA核函数与GPU硬件的对应关系

GPU硬件的三个层次，类似集群：

- Core(SP),
- SM（流多处理器）
- Device

任务从高层到底层传递

- **Grid分配到Device上多个SM上运行**
- **Block分配到一个SM上运行；**
- **Thread分配到Core上运行。**

核函数的调用方式

当使用int类型时，表示一维排布，比如：

kernel_name<<<5,8>>>(...);

表示一个Grid中有5个Block，在(x,y,z)三个方向上的排布

方式分别是5、1、1；

一个Block中有8个Thread，在(x,y,z)三个方向上的排布方

式分别是8、1、1。

dim3 grid(3,2,1), block(4,3,1);

kernel_name<<<grid, block>>>(...);

表示一个Grid中有3x2x1=6个Block，在(x,y,z)三个方向上的排布方式分别是3、2、1；

一个Block中有4x3x1=12个Thread，在(x,y,z)三个方向上的排布方式分别是4、3、1。

四个重要的内置变量

- blockIdx：索引到线程块
- threadIdx：索引到某个块内的线程
- blockDim：得到一个块内线程总数
- gridDim：得到一个格内块总数
- 在一维的情况下，计算线程全局id公式为
  - 线程全局id = blockIdex.x * blockDim.x + threadIdx.x
- 在一维的情况下，核函数内的线程总数为
  - 核函数的线程总数 = gridDim.x * blockDim.x

2.12.3 CUDA编程

例 2-4

编写完整的CUDA程序：要求定义数组A，实现初始化整数数组A[1000][1000]，对二维数组A的每一行从下标为1的元素开始进行求和，并将求和结果保存在对应行的第0号元素。

分析

CUDA编程中最重要的是定义CUDA核函数。根据题意，核函数需要完成的任务是：根据传入的A数组，计算对应行从下标1至999号元素的和，并将结果存储在第0号元素。**首先根据块索引（blockIdx.x）和块内线程索引（threadIdx.x）确定需要处理的行号（lineID）**，然后再实现计算功能。接着根据CUDA编程的一般流程，编写主函数功能。

例程代码---核函数

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#define N 1000
#define N_THREADS_PER_BLOCK 32
__global__ void kernel(int *A) {
    int row = blockIdx.x;
    int col = threadIdx.x;
    int height = blockDim.x;
    int lineID = col + height * row; // 根据blockID和threadID确定需要操作的行ID
    if (lineID >= 0 && lineID < N) { // 若行ID有效，则执行计算最大值操作
        int i, sum = 0;
        for(i = 1; i < N; i++) {
            sum += A[lineID * N + i];  }
        A[lineID * N] = sum;  }  }
```

例程代码---主程序

```
int main(void) { // 定义数组A
        int A[N][N];
        int i, j;// 随机初始化数组A，每行从下标1开始，第0个元素用于记录该行最大值
        for (i = 0; i < N; i++){
    A[i][0] = 0;
                for (j = 1; j < N; j++){
                        A[i][j] = rand()%10; } }
    int *dev_A; // 定义数组A在GPU端的指针dev_A
  cudaSetDevice(0);
  cudaMalloc((void**)&dev_A, N * N * sizeof(int)); //在GPU显存中开辟用于存放数组A的空间
  cudaMemcpy(dev_A, A, N * N * sizeof(int), cudaMemcpyHostToDevice);// 将数组A由主机端复制到显存端
  kernel << <(N+N_THREADS_PER_BLOCK-1)/N_THREADS_PER_BLOCK, N_THREADS_PER_BLOCK  >> >
(dev_A);// 执行核函数，传入数组A指针，并设置线程块数和线程数
  cudaMemcpy(A, dev_A, N * N * sizeof(int), cudaMemcpyDeviceToHost); // 将计算后的数组A由显存复制回
主机端
  cudaFree(dev_A);// 释放显存空间
  for (i = 0; i < N; i++) {// 输出计算结果
        for (j = 0; j < N; j++)
        printf("%d ", A[i][j]);
        printf("\n");  } }
```

2.13 OpenMP-CUDA混合编程

2.13 OpenMP-CUDA 混合编程

CPU和GPU都有很多内核，都能用来进行并行计算。

GPU由上千个内核（SP），计算能力强，适合处理大规模的数据并行计算（SIMD）任务。

CPU也有几十个内核，计算能力较弱，但单个核的计算能力比SP强，且这些核可以处理MIMD类型的任务，因此多核CPU适用范围比GPU广，能在GPU上计算的SIMD任务在CPU上也能计算。

一种并行程序设计：

① 先基于OpenMP的程序并行，并验证程序正确；

② 在OpenMP并行计算中把计算量大的SIMD计算分配给GPU，这部分计算用CUDA编程实现。不要把计算量小的任务交给GPU，可能导致GPU的计算速度不如一个CPU内核。

例 2-5

编写完整的OpenMP-CUDA程序：要求定义数组A，实现初始化二维整数数组A[1000][1000]，对数组A的每一行从下标为1的元素开始进行求和，并将求和结果保存在对应行的第0号元素。假设系统可用资源有4个CPU核和1块GPU卡。

分析

CUDA核函数部分与例2-4相同；在主函数中，由OpenMP开启4个CPU线程，并根据线程号执行不同的操作：由0号CPU线程管理GPU实现计算；1-3号CPU线程直接进行计算。为了方便GPU在多线程环境中使用cudaMemcpy将结果回写至数组A，我们规定GPU所处理的行号为从第0行开始的连续行，CPU计算线程处理GPU之后的行。

例程代码

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include "omp.h"
#define N 1000
#define N_THREADS_PER_BLOCK 32
#define OMP_THREADS 4 // GPU卡数和CPU核数
#define GPU_COUNT 1
#define SPEED_GPU 7// GPU和CPU的性能比，用于调度
#define SPEED_CPU 3
__global__ void kernel(int *A) {
    int row = blockIdx.x;
    int col = threadIdx.x;
    int height = blockDim.x;
    int lineID = col + height * row;// 确定需要操作的行ID
```

例程代码

```
    if (lineID >= 0 && lineID < N) {// 若行ID有效，则执行计算最大值操作
       int i, sum = 0;
       for(i = 1; i < N; i++) {
          sum += A[lineID * N + i];  }
       A[lineID * N] = sum;  }  }
int main(void) {
   int LINE_FOR_GPU = N*SPEED_GPU/(SPEED_CPU+SPEED_GPU);// 计算分别由GPU、CPU处理的行数
   int LINE_FOR_CPU = N-LINE_FOR_GPU;
   int CPU_COMPUTE_THREADS = OMP_THREADS - GPU_COUNT;// 定义有几个CPU线程用于计算
   int LINE_FOR_CPU_THREADS = (LINE_FOR_CPU + CPU_COMPUTE_THREADS - 1) /
CPU_COMPUTE_THREADS;// 定义每个CPU线程计算的行数
          int A[N][N];// 定义数组A
          int i, j;// 随机初始化数组A，每行从下标1开始，第0个元素用于记录该行最大值
          for (i = 0; i < N; i++){
       A[i][0] = 0;
                    for (j = 1; j < N; j++){
                              A[i][j] = rand()%10;  }  }
```

例程代码

```
#pragma omp parallel for num_threads(OMP_THREADS)  // 启动多线程计算
   for (i = 0; i < OMP_THREADS; i++) {
      printf("ThreadIdx: %d\n", i);
      if (i == 0) { // 如果是0号线程，则执行CUDA程序，处理从第0行开始的LINE_FOR_GPU行
         int *dev_A;
         cudaSetDevice(0);
         cudaMalloc((void**)&dev_A, LINE_FOR_GPU * N * sizeof(int));
         cudaMemcpy(dev_A, A, LINE_FOR_GPU * N * sizeof(int), cudaMemcpyHostToDevice);
         kernel << <(LINE_FOR_GPU+N_THREADS_PER_BLOCK-1)/N_THREADS_PER_BLOCK,
N_THREADS_PER_BLOCK  >> > (dev_A);
         cudaMemcpy(A, dev_A, LINE_FOR_GPU * N * sizeof(int), cudaMemcpyDeviceToHost);
         cudaFree(dev_A);
      } else {// 否则，根据线程号分配给CPU线程计算
         int k, lineID, sum;
         //计算该线程的起始行号，注意CPU线程计算的起始行就是CUDA处理的最后一行，即从LINE_FOR_GPU开始
         int line_for_cpu_idx = LINE_FOR_GPU + LINE_FOR_CPU_THREADS * (i - 1);
```

例程代码

```
for (lineID = line_for_cpu_idx; lineID < line_for_cpu_idx + LINE_FOR_CPU_THREADS;
lineID++) {
            if (lineID >= 0 && lineID < N) {
                sum = 0;
                for (k = 1; k < N; k++) {
                    sum += A[lineID][k];
                }
                A[lineID][0] = sum; }
        }   }   }

#pragma omp barrier
            // 回到主线程，输出计算结果
    for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++)
            printf("%d ", A[i][j]);
            printf("\n");
    }
}
```

2.14 多核CPU-GPU计算平台任务调度

2.14 多核CPU-GPU计算平台任务调度

- 静态调度

  不需要竞争资源，调度开销小，但容易导致负载不均衡和计算资源利用率低

- 动态调度

  在程序执行时根据负载 动态进行任务分配和资源的竞争，能实现负载均衡，但会产生额外的调度开销

- GPU基于SIMT，适合做大规模并行计算，计算开销包括：

  内存与GPU之间的数据通信时间 + 计算时间

  GPU适合做计算粒度大，通信量小的计算。

  - 任务分配时应该根据CPU和GPU的架构特点将计算粒度大的任务分配给GPU计算，粒度小的任务分配给CPU计算。
  - 通过静态调度和动态调度的组合可以大大减少调度次数，即减少动态调度的开销，又可以通过动态调度实现负载均衡

2.14 多核CPU-GPU计算平台任务调度

并行计算应用可分为：

① 负载可预测应用，可以运用负载预测调度算法实现高效调度

② 负载不可预测应用。

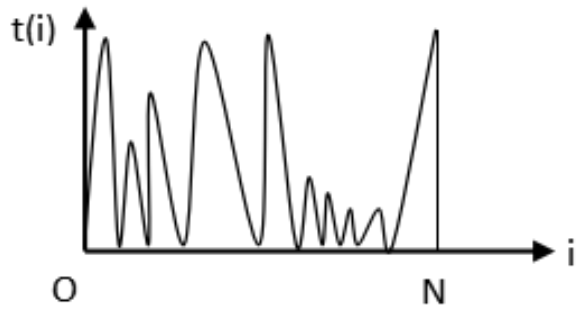
负载预测调度算法的主要思想是：

1、通过负载预测将计算量大的计算任务分配给GPU、计算量小的分配给CPU计算，充分发挥GPU的计算性能。

2、测试CPU和GPU的计算性能比，根据性能计算静态和动态的比例，尽量减少动态调度的次数，减少调度开销。

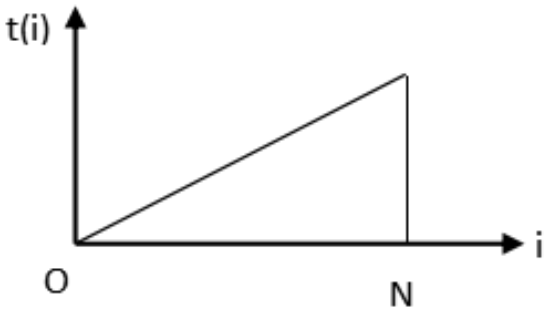
1

随机型



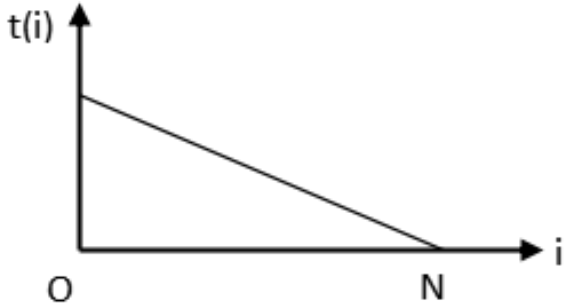
2

递增型



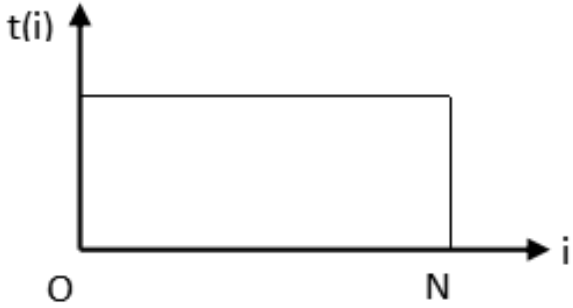
3

递减型



4

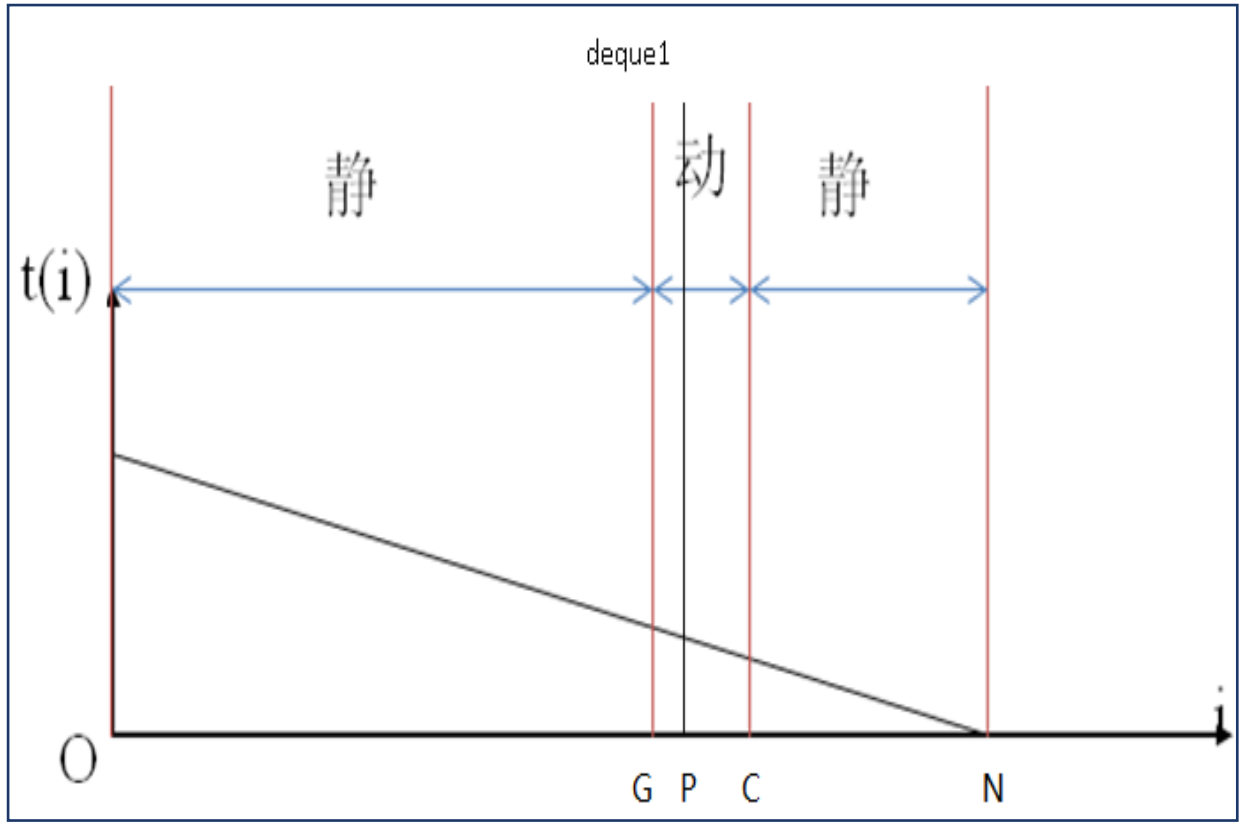
常数型



并行循环的4种类型

负载预测调度算法

以非常数型并行循环任务为例介绍负载预测调度算法的具体实现方法

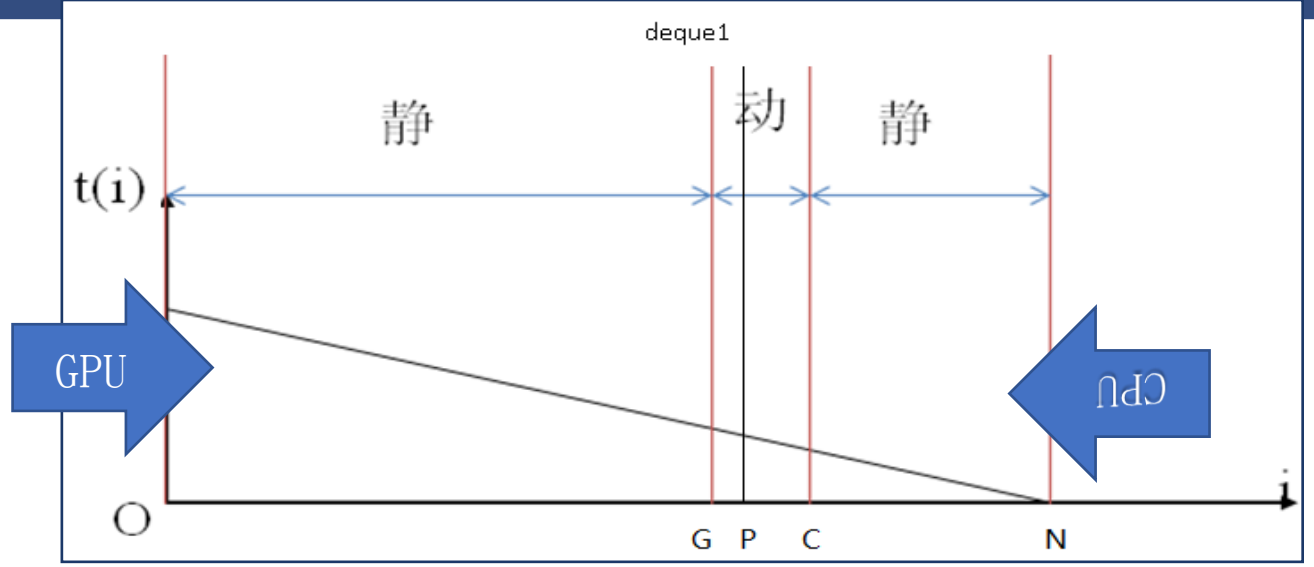


1 预处理

首先预测每次循环迭代的计算量，并将每次迭代的计算量按从大到小排序，建立左图队列。

x轴：循环的迭代序号
y轴：每次迭代的计算量（或计算时间）

2 使用负载预测调度方法进行调度



根据预测获得的总的计算量和CPU、GPU的计算性能，就能估算出纯CPU线程和GPU加速线程能完成的循环迭代数，即图中P的位置。

根据P、线程总数、CPU、GPU的计算速度就可以确定动态调度窗口[G,C]，即图中间标"动"字的区域。

第一级调度是负载预测静态调度：

GPU加速的线程从队首开始逐个处理计算量大的任务，就是图中左边那个标记"静"的区域；

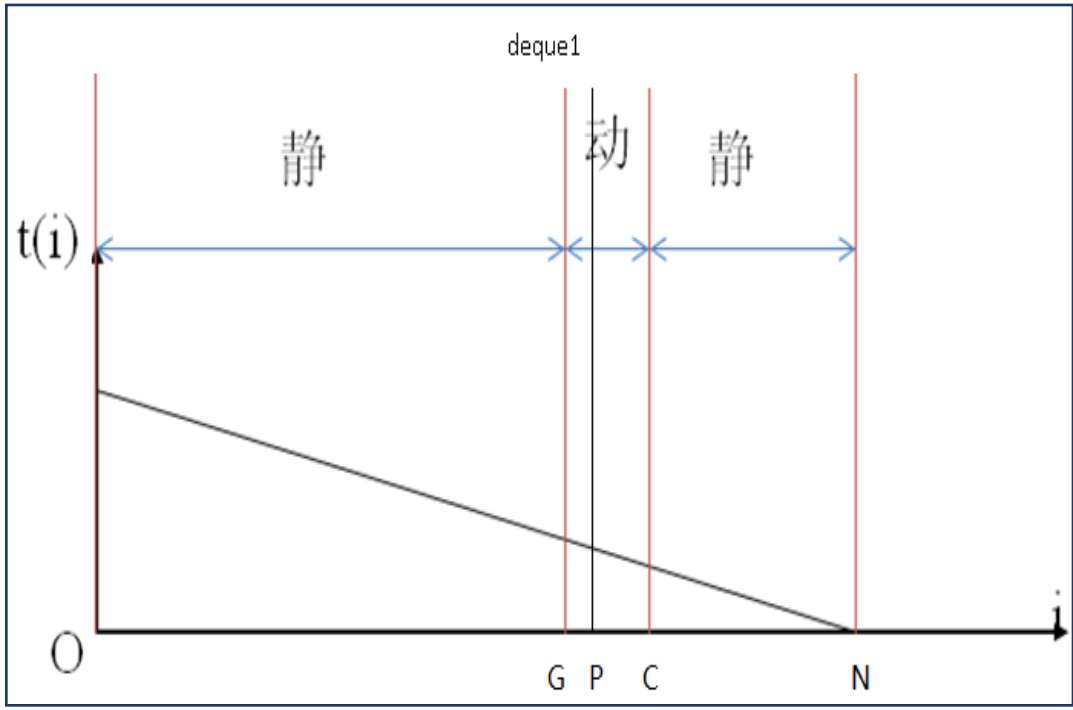
其它CPU线程从队尾开始逐个处理计算量小的任务，就是图中右边那个"静"的区域。

当左右2个静态调度中任意一个部分计算完毕后，该线程直接进行第二级负载预测动态调度。

例如当GPU加速的线程完成了自己的静态调度任务，即左边标记“静”的迭代任务后从左边进入动态调度窗口获取计算任务，直到动态调度窗口内的任务全部完成为止；同理当任意一个纯CPU线程完成右边标记“静”的迭代任务后从右边进入动态调度窗口获取计算任务，直到动态调度窗口内的任务全部完成为止。

显然任务调度会在P点附近结束，GPU加速的线程和纯CPU的线程将在完成所有的计算任务后在这里“会师”。

二级调度的任务分配方法与一级调度一样。



GPU选做实验

- **Linux/Windows下 CUDA安装及矩阵乘法编程实现**
- **主要内容：**
  - **简洁列出CUDA安装步骤和要点。**
  - **实现实验报告2中的矩阵乘法，可以仅CUDA实现或OpenMP-CUDA混合编程，基于个人机器配置用多种角度对比加速比，附上源码。**