**一. 填空题**

1. 控制器、运算器、寄存器；
2. 微处理器；
3. 操作码、地址码、操作码；
4. RISC、CISC；
5. 星形总线、环形总线、Mesh总线；
6. 高并发；
7. GPU；
8. 静态调度、动态调度、动静态调度；
9. 负载可预测应用、负载不可预测应用；
10. 硬件；
11. 共享内存并行系统。

**二. 选择题**

1.A 2.B 3.C 4.A 5.B 6.D 7.A 8.B 9.C 10.D

**三. 问答题**

1.答：RISC和CISC是CPU的两种架构。RISC克服了CISC系统庞杂的缺陷，其指令系统只包含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。RISC-V指令集是基于RISC原理建立的开放指令集架构，完全开源，设计简单，采用模块化设计，易于移植UNIX系统，可以根据具体场景进行选择。

2.答：中央处理器（CPU）由控制器、运算器和寄存器组成，三个部件的功能如下：

（1）控制器。其任务是负责从存储器中取出指令，确定指令的类型，并对指令进行译码，控制整个计算机系统一步一步地完成各种操作。

（2）运算器。为计算机提供计算与逻辑功能，控制器把数据带给ALU后，就根据指令完成算术运算或逻辑判断及逻辑运算。

（3）寄存器。它是处理器内部的存储单元，主要存储当前指令信息和将要执行的下一条指令的地址等。

3.答：CPU并行运算有三种实现方式：指令级并行、线程级并行以及数据并行。各自特点如下：

（1）指令级并行是一种指令层面的并行技术，目的是使CPU能够在同一时刻并行执行多条指令，其基本原理是：当指令之间不相关时，它们可以重叠起来并行执行。

（2）线程级并行是多处理器支持多个线程同时并行执行。实现CPU的线程级并行的基础是CPU具有多个物理内核，能够在每个内核上执行一个独立线程。

（3）数据并行是指把需要处理的数据划分成若干个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对所分派的数据进行处理，即遵循SIMD（单指令多数据流）模型。

4.答：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：位于GPU上的每个SM中，访问速度最快，用于存放线程执行时所需要的变量，具有线程私有的性质。

（2）共享内存：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：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。

5.答：CUDA程序的一般执行流程为：

（1）调用cudaMalloc()函数，在显存中开辟空间用于存储需要由GPU计算的数据。

（2）调用cudaMemcpy()函数，将数据由主机内存复制到GPU显存中。

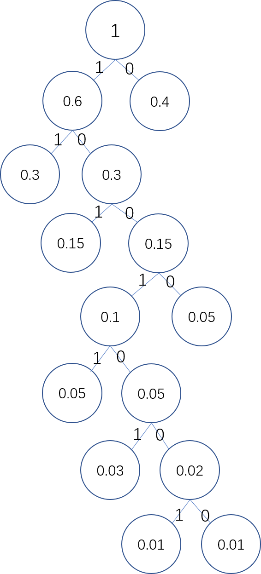
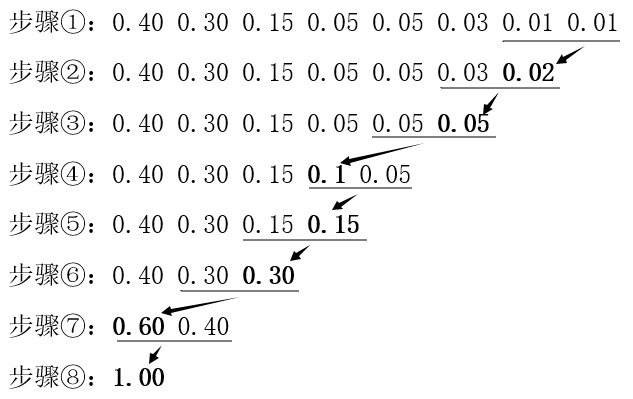
（3）启动核函数，进行相关高并行度计算。

（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。

**四. 计算题**

**1.解：**请具体计算步骤如下图所示：



最后得到哈夫曼编码如下表所示：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 |
| 编码 | 0 | 1000 | 101 | 100101 | 10011 | 11 | 1001001 | 1001000 |
| 码长 | 1 | 4 | 3 | 6 | 5 | 2 | 7 | 7 |

最短编码长度为：

H= 0.40\*1+0.05\*4+0.15\*3+0.03\*6+0.05\*5+0.30\*2+0.01\*7+0.01\*7 = 2.22

**五. 编程题：**

1.答：方法a实现代码如下：

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main(){

int i,n;

float a[100],b[100],result;

//initializations

n=100;

result=0.0;

for(i=0;i<n;i++){

a[i]=rand()%5;

b[i]=rand()%9;

}

#pragma omp parallel for

for(i=0;i<n;i++){

#pragma omp critical

result=result+(a[i]\*b[i]);

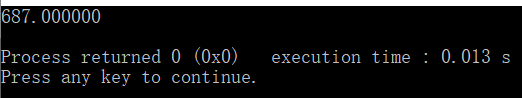
}

printf("%f\n",result);

return 0;

}

结果输出：



方法b实现代码如下：

#include <stdio.h>

#include <stdlib.h>

#include <stdio.h>

#include <omp.h>

//reduction

int main(){

int i,n,chunk;

float a[100],b[100],result;

//initializations

n=100;

chunk=10;

result=0.0;

for(i=0;i<n;i++){

a[i]=rand()%5;

b[i]=rand()%9;

}

#pragma omp parallel for default(shared)private(i)schedule(static,chunk)reduction(+:result)

for(i=0;i<n;i++)

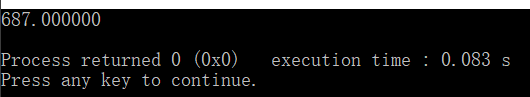
result=result+(a[i]\*b[i]);

printf("%f\n",result);

return 0;

}

结果输出：



2.答：实现代码如下：

#include <stdio.h>

\_\_global\_\_ void helloFromGPU(void){

printf("hello world from GPU!\n");

}

int main(void){

printf("hello world from CPU!\n");

helloFromGPU <<<1,10>>>();

cudaDeviceReset();

return 0;

}