**中山大学本科生期末考试**

**考试科目：《并行程序设计》（C卷）**

学年学期：**2022**学年第**2**学期 姓 名：

开课单位：中山大学计算机学院 学 号：

考试方式：闭卷 年 级：

考试时长：**120**分钟 院 系：

警示 《中山大学授予学士学位工作细则》第八条：“考试作弊者，不授予学士学位。”

------------以下为试题区域，共###道大题，总分100分,考生请在答题纸上作答------------

1. **简答题（共 8 小题，每小题 5 分，共 40 分）**
2. [5分] 简要说明单指令多数据流（SIMD）和单程序多数据流（SPMD）的区别与联系。

区别：

SIMD：单指令多数据流（硬件级并行），一条指令同时处理多个数据。

SPMD：单程序多数据流（软件模型），多个进程/线程独立执行相同程序但处理不同数据（如MPI程序）。

联系：

两者均通过数据并行提升性能，SPMD可在SIMD硬件上实现（如MPI+向量化）。

（SIMD VS SIMT

目标：均通过单指令控制多数据来提高计算吞吐量

适用于 数据并行 任务（如大规模数组运算）

SIMT可视为SIMD的扩展

SIMD 是严格的同步执行（所有数据通道必须执行相同操作）

SIMT 在SIMD基础上增加了 线程级灵活性：

**同一warp内的线程可执行不同分支**

支持线程间通信（共享内存、同步）

）

1. [5分] 什么是阿姆达尔定律（Amdahl’s law）？它告诉了我们什么？

并行化所产生的加速比存在上限

1. [5分] 如何理解多线程编程中的线程安全性？如何解决线程安全性问题？

线程安全（Thread Safety） 指在多线程环境下，程序或数据结构能够正确、一致地工作，而不会因并发访问导致数据损坏、逻辑错误或不可预测的行为。

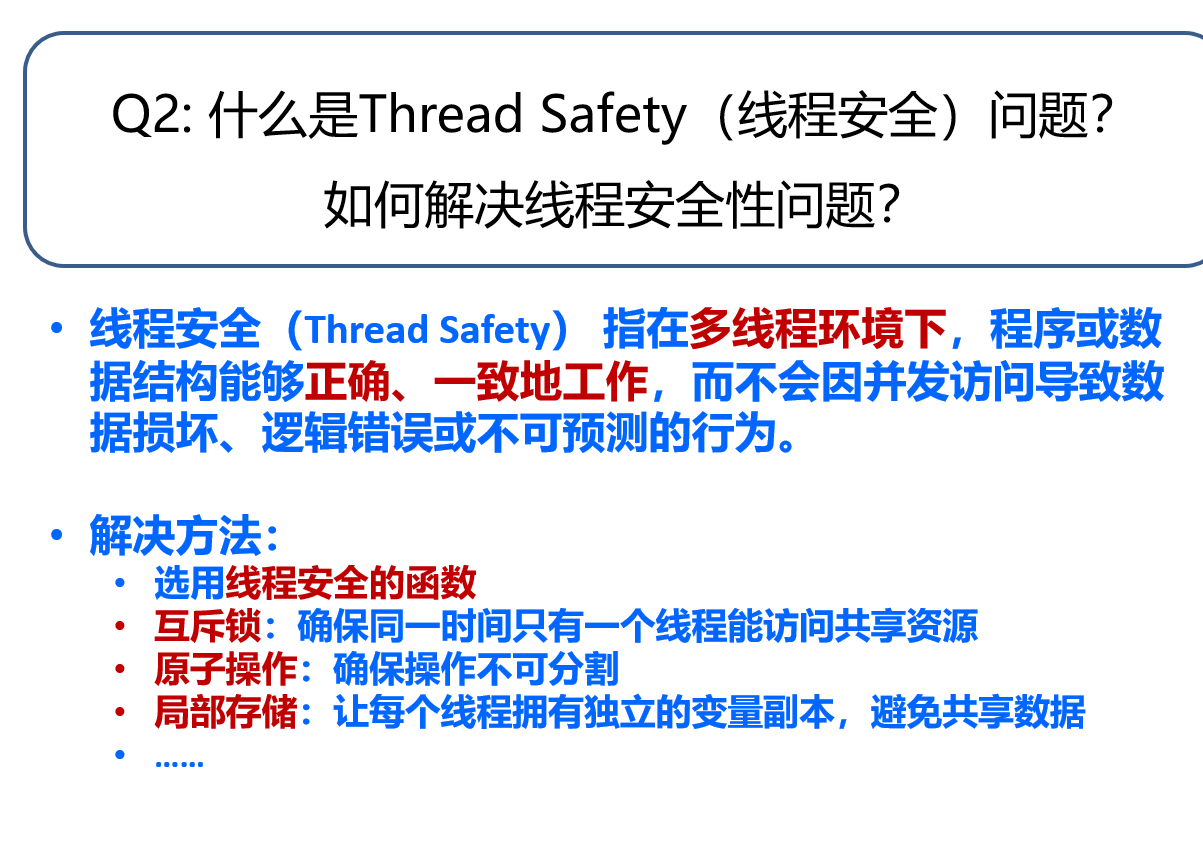
解决方法：

选用线程安全的函数

互斥锁：确保同一时间只有一个线程能访问共享资源

原子操作：确保操作不可分割

局部存储：让每个线程拥有独立的变量副本，避免共享数据



1. [5分] 简要说明线程间同步的方式（两种即可）。

互斥锁、信号量、屏障、条件变量

互斥锁在访问共享资源前对互斥量进行加锁，访问完成后释放互斥量。任何其他试图再次对互斥量加锁的线程都会被阻塞，直到当前线程释放该互斥量‌。

条件变量允许线程睡眠，直到满足某种条件。当满足条件时，可以向该线程发送信号，通知并唤醒该线程。条件变量通常与互斥量配合使用‌。

1. [5分] CUDA编程中，可用于同步线程的机制有哪些？试举例说明其中三种。

1.\_\_syncthreads()：

同步同一线程块内所有线程（用于共享内存操作后）。

2.原子操作：

如 atomicAdd(&sum, value) 保证全局累加安全。

3.流（Stream）：

异步操作序列，通过 cudaStreamSynchronize(stream) 同步。

1. [5分] 试说明原子操作的概念，并以一个CUDA中的原子操作函数为例，说明其在实际应用中的作用。

概念：不可中断的操作（读-修改-写），避免数据竞争。

atomicAdd(&counter, 1); // 原子加1

作用：多线程安全累加（如统计像素值）。

1. [5分] 简述在CUDA编程中，主机内存、设备内存、及统一内存寻址的概念，并说明统一内存寻址的作用。

·主机内存（Host Memory）：CPU管理的RAM（malloc分配）。指运行CUDA程序的主机计算机的内存

·设备内存（Device Memory）：GPU显存（cudaMalloc分配）。指GPU的专用内存

·统一内存（Unified Memory）：通过 cudaMallocManaged 分配，CPU/GPU共享同一地址空间，自动迁移数据。

作用：简化编程，避免手动拷贝（如 cudaMemcpy）。

1. [5分] 简述存储体冲突的概念，结合具体例子说明如何消除存储体冲突。

·概念：当多个线程同时访问同一存储体（Bank） 时（如GPU共享内存的32个Bank），导致串行访问。

·示例：矩阵转置中，若线程按列写入共享内存，相邻线程访问同一Bank（冲突）。

·消除方法：地址偏移（Padding）：

\_\_shared\_\_ float tile[32][32 + 1]; // 增加1列避免Bank冲突

1. **应用题（共 3 小题，每小题 5 分，共 15 分）**

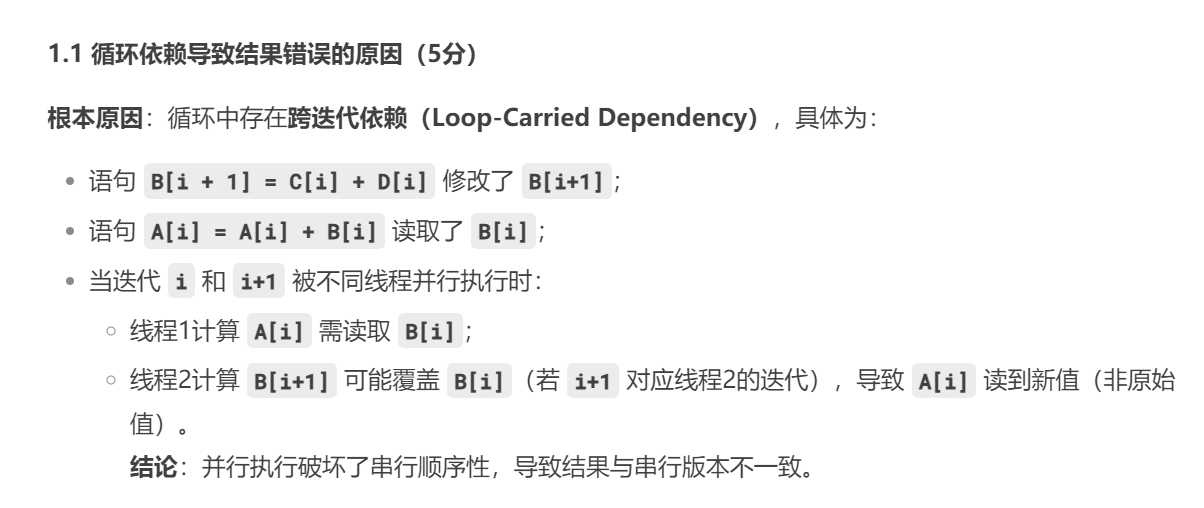
根据以下代码片段，回答以下有关于循环依赖与OpenMP调度的问题：

|  |
| --- |
| #pragma omp parallel for num\_threads(4) schedule(static, 2)  for (i = 0; i < N; i++) {  A[i] = A[i] + B[i];  B[i + 1] = C[i] + D[i];  } |

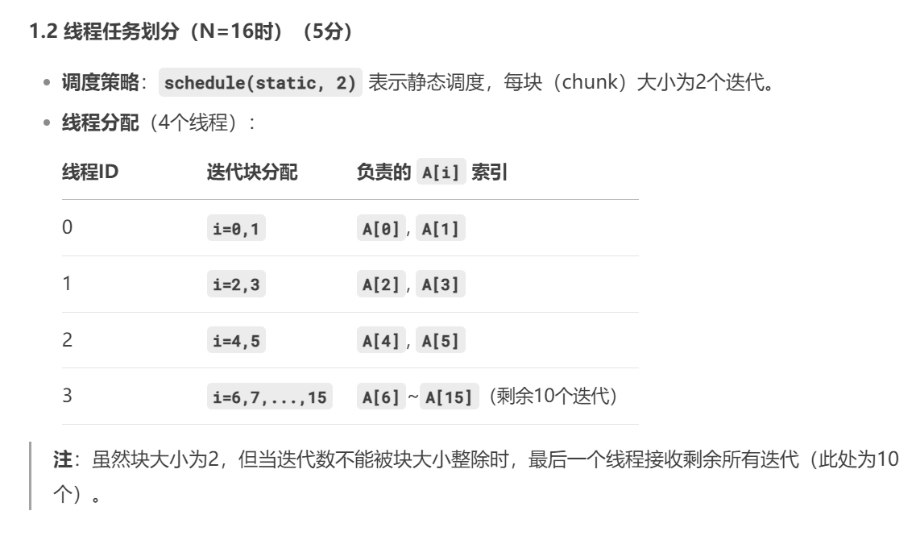
1.1 [ 5分] 由于循环依赖的存在，该并行代码无法产生期望的结果（与去掉OpenMP编译指导语句的串行版本一致的运行结果），请解释原因。

因为对于parallel for调度来说，满足并行的一个要求就是parallel for内的线程循环之间不能有循环依赖。**OpenMP编译器不会检查使用parallel-for指令并行化的循环中迭代之间的依赖关系。OpenMP无法正确并行化一个或多个迭代的结果依赖于其他迭代的循环。**

****



1.2 [5分] 若N = 16，请问各个线程分别计算数组A的哪些元素？



1.3 [ 5分]若N的值很大，且事先未知。实验发现，将schedule(static, 2)中的第二个参数trunk\_size由2改为一个较大的值后（如10000），程序的性能有所提升。此时，还应修改原先的static调度方式以实现负载均衡。请问应将调度方式改为dynamic或guided中的哪一种？为什么？

Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.

However, in a **guided** schedule, as chunks are completed, the size of the new chunks decreases.



1. **编程题（共 1 小题，每小题 15 分，共 15 分）**

1.5CM

现在有一个单精度浮点类型的数组A[N]，已知该数组的数据都分布范围为[0,10)。为了对这些数据的分布有一个更为直观的感受，可以画一个直方图：首先把这些数据的范围划分成10个同等大小的区间，也就是10个桶，然后统计每个桶中元素的数目。现在需要你编写一个MPI并行程序，实现函数histogram\_gen，该函数的功能是完成上述过程，并将每个桶中元素的数目通过printf函数输出到标准输出。请注意，数组A保存在Rank 0中，你需要想办法把它分配到各个进程。

需要实现的函数原型：void histogram\_gen(float\* A, int N, MPI\_Comm comm)

* **MPI的全称是Message Passing Interface，即消息传递接口。**
* **MPI是一种用于编写并行计算机程序的标准库，主要用于多进程间的通信和数据交换。**
* **MPI不是一种编程语言，而是一组函数库，可以被多种编程语言调用，最常见的是C、C++和Fortran。**
* **MPI适用于分布式内存的多处理器系统。**
  + **集合通信是 MPI中的一种通信模式，它涉及一组进程（通常是通信域中的所有进程）之间的协同操作。**

**集合通信的特点**

* + **全局性：集合通信涉及通信域中的所有进程，而不是单独的两个进程。**
  + **同步性：所有参与的进程必须同时调用集合通信函数，否则会导致程序挂起或错误。**
  + **高效性：集合通信通常经过优化，能够利用底层硬件和网络的特性，提高通信效率。**

**集合通信的用途：数据分发与收集、数据规约与聚合、进程同步、高效通信**

**集合通信操作是全局同步的，所有进程必须按照相同的顺序调用集合通信函数MPI 通过调用顺序来匹配集合通信操作，确保每个进程的发送和接收操作能够正确对应。**

****

* + **MPI Safety问题是指在MPI编程中，由于MPI的实现或使用不当，导致程序在多进程环境下出现未定义行为、数据竞争、死锁或其他错误的现象。**
  + **例子：MPI 通信操作（如 MPI\_Send 和 MPI\_Recv）的顺序可能导致死锁**

**MPI\_SendRecv：**

* + **在一个调用中同时完成发送和接收操作**
  + **发送和接收操作是原子的，避免了死锁风险**

**// 模板框架**

**int rank, size;**

**MPI\_Init(&argc, &argv);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);**

**// 数据分发（不均等分割）**

**int counts[size], displs[size];**

**for (int i=0; i<size; i++) {**

**counts[i] = N / size + (i < N%size ? 1 : 0); // ★ 整除处理**

**displs[i] = (i==0) ? 0 : displs[i-1] + counts[i-1];**

**}**

**// 集体通信（选对API！）**

**MPI\_Scatterv**(**sendbuf, counts, displs, MPI\_FLOAT, recvbuf, local\_size, MPI\_FLOAT, 0, MPI\_COMM\_WORLD); // ★ 不等长分发**

**MPI\_Reduce(local\_data, global\_data, count, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD); // ★ 结果归约**

**MPI\_Finalize();**

****

**// 模板框架**

**void\* thread\_func(void\* arg) {**

**ThreadArgs\* args = (ThreadArgs\*)arg; // ★ 必须用结构体传参**

**int start = args->start;**

**for (int i=start; i<end; i+=thread\_count) { // ★ 循环步进分配**

**pthread\_mutex\_lock(&mutex);**

**// 临界区操作**

**pthread\_mutex\_unlock(&mutex);**

**}**

**return NULL;**

**}**

**int main() {**

**pthread\_t threads[N];**

**ThreadArgs args[N]; // ★ 每个线程独立参数**

**for (int i=0; i<N; i++) {**

**args[i].id = i;**

**pthread\_create(&threads[i], NULL, thread\_func, &args[i]);**

**}**

**// ...join...**

**}**

****

* + **Busy-waiting：低延迟但高CPU占用，适合极短操作。**
    - **避免了内核态切换的开销**
    - **内核态切换是指CPU在执行程序时，从用户态切换到内核态（或反之）的过程**
  + **Mutex：高延迟但节省CPU，适合通用场景。**

**// 模板框架**

**#pragma omp parallel num\_threads(4)**

**{**

**#pragma omp for reduction(+:sum) schedule(dynamic, 16) // ★ 归约+动态调度**

**for (int i=0; i<N; i++) {**

**sum += data[i];**

**}**

**#pragma omp single // ★ 单线程执行区域**

**{**

**printf("Thread %d reports\n", omp\_get\_thread\_num());**

**}**

**}**

****

****

****

****

****

**Pthreads 作用域规则**

**共享变量：全局变量/静态变量 → 所有线程可见（需同步）**

**私有变量：栈变量（函数内局部变量） → 默认线程私有**

**Pthreads 临界区实现**

**pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;**

**void\* thread\_func(void\* arg) {**

**pthread\_mutex\_lock(&mutex); // 进入临界区**

**// 修改共享资源（如全局链表插入）**

**pthread\_mutex\_unlock(&mutex); // 退出临界区**

**return NULL;**

**}**

****

**2. OpenMP 临界区实现**

**#pragma omp parallel**

**{**

**#pragma omp critical (my\_critical) // 命名临界区**

**{**

**// 共享资源操作（如文件写入）**

**}**

**}**

**Pthreads 任务调度**

**静态调度：手动划分任务**

**for (int i = rank; i < N; i += thread\_count) {**

**process(i); // 每个线程处理等间隔数据**

**}**

**动态调度：任务队列 + 锁**

**while (1) {**

**pthread\_mutex\_lock(&queue\_lock);**

**task = get\_next\_task(); // 取任务**

**pthread\_mutex\_unlock(&queue\_lock);**

**if (no\_task) break;**

**process(task);**

**}**

****

****

**在OpenMP中，变量的作用域是指可以在并行块中访问变量的线程集。**

MPI参考函数接口



1. **CUDA代码分析（共 2 小题，每小题 15 分，共 30 分）**

1.5CM

1. 线程执行分析：

\_\_global\_\_ void func(float\* input, float\* output, int size) {

int tid = threadIdx.x + blockIdx.x \* blockDim.x;

if (tid \* 2 > size) return;

if (tid %2){

output[tid] = input[tid\*2]\*input[tid\*2]+input[tid\*2+1]\*input[tid\*2+1];

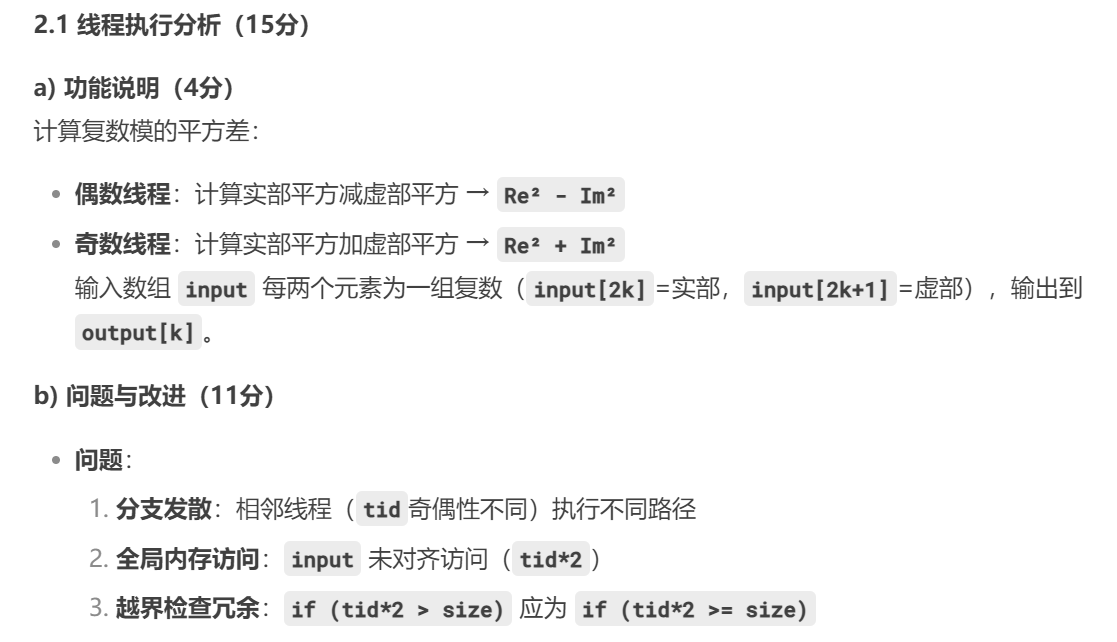
} else {

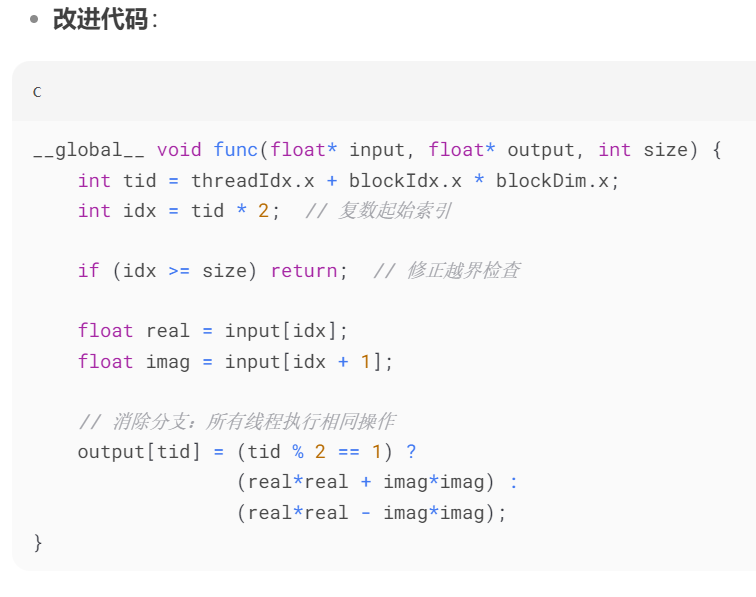
output[tid] = input[tid\*2]\*input[tid\*2]-input[tid\*2+1]\*input[tid\*2+1];

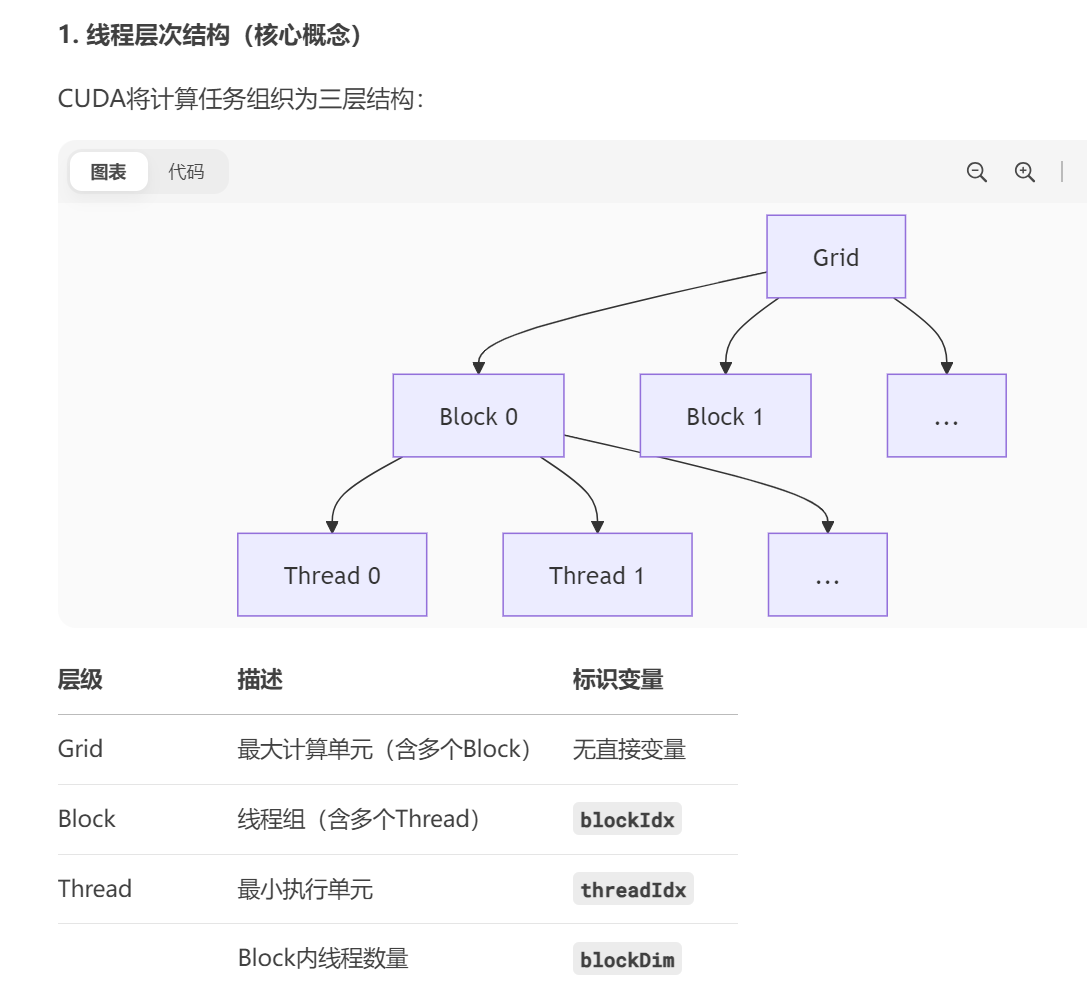
}

}

* 1. 说明以上代码功能。（4分）
  2. 说明其效率存在的问题，如何改进，写出改进后的源码。（11分）









// 一维计算

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

// 三维计算（矩阵场景）

int x = blockIdx.x \* blockDim.x + threadIdx.x;

int y = blockIdx.y \* blockDim.y + threadIdx.y;

int global\_idx = y \* width + x; // 行优先索引





2. 代码分析：

1. 说明以下代码的功能；（5分）实现矩阵转置
2. 说明其效率存在的问题，如何改进，写出改进后的源码；（10分）

\_\_global\_\_ void func(int \* out, int\* in, int n, int m){

int x = blockIdx.x\*blockDim.x+threadIdx.x;

int y = blockIdx.y\*blockDim.y+threadIdx.y;

out[x\*n+y] = in[y\*m+x];

}

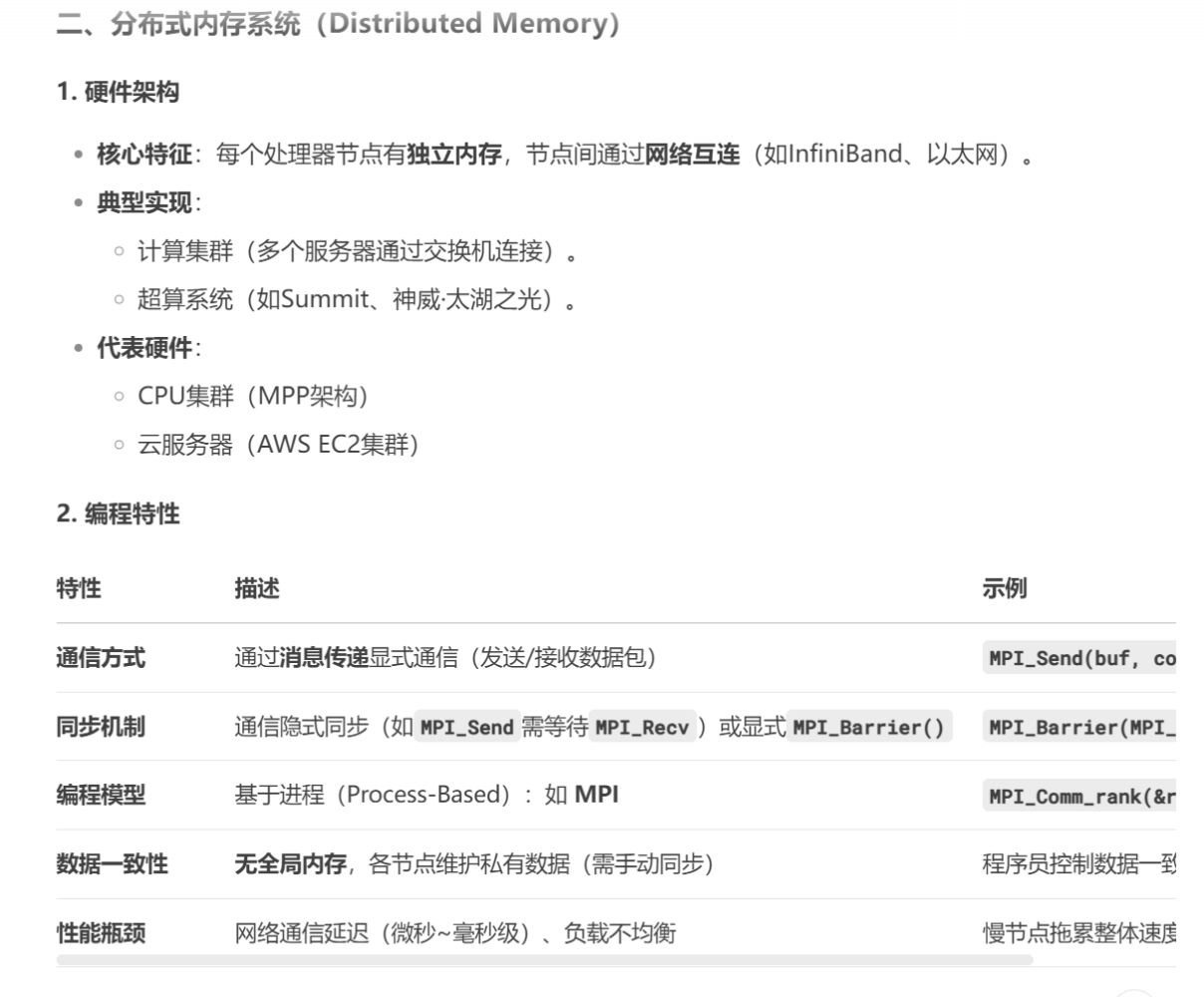
合并访问失败：in[y\*m+x] 导致线程访问非连续内存（跨步为m）

存储体冲突：out[x\*n+y] 写入时相邻线程访问同一存储体（若n是32倍数）

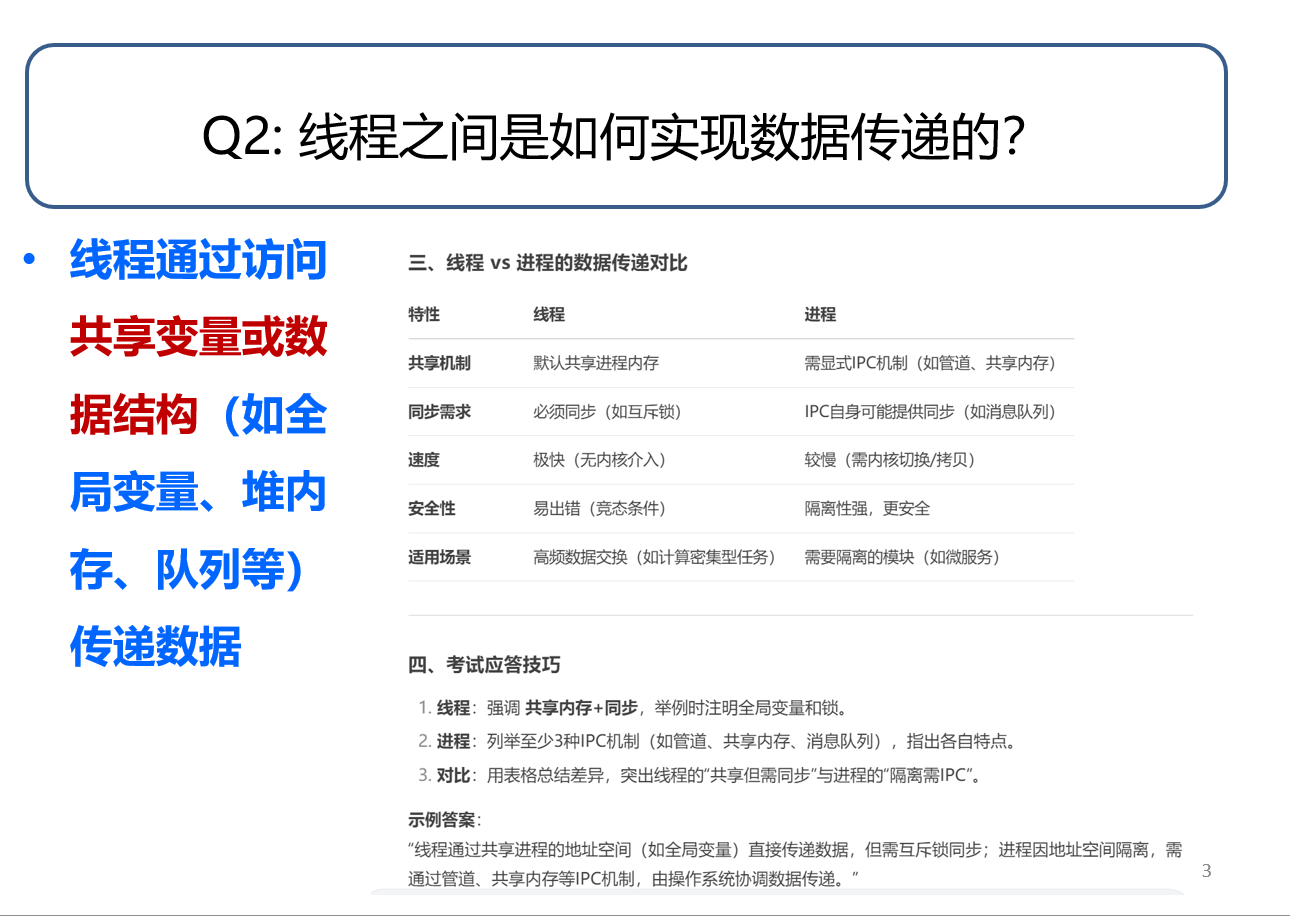
**排版要求**：试卷一级标题须标注分值，至少占两行，字体加粗。行距适当留疏，格式整齐一致，保持卷面美观。各题之间不留答题区域，考生一律将答案写在专用答题纸（本）上。**试卷头版式不可改变,内容可根据考试科目实际需要增减**。

**说明：**此份仅为试卷模板，教师需根据实际内容将红色字迹替换。





* Foster方法论是一种用于设计和分析并行算法的系统方法。包括以下主要步骤：
  + **划分（Partitioning）**：将问题划分为可以独立解决的较小任务。
  + **通信（Communication）**：确定哪些数据需要在不同的处理器之间进行交换。
  + **聚合（Agglomeration）**：将较小的任务组合成较大的任务，以减少通信开销和提高计算效率。
  + **映射（Mapping）**：将任务分配到具体的处理器上。



为何pthreads中的条件变量（condition variable）总是与一个mutex相关联？

* 释放关联的 Mutex（让其他线程可以修改共享数据）。
* 阻塞线程，等待条件变量的信号。
* 被唤醒后，重新获取 Mutex（继续操作共享数据）。
* **线程在等待某个条件成立时，需要先检查条件是否满足，但检查条件本身必须是原子操作**
* **通过互斥锁包裹条件的检查和修改，确保“检查条件”和“进入等待”是原子的，避免竞争条件**
* **如果没有Mutex：线程无法安全地释放和重新获取锁，可能导致共享数据被破坏。**

