Ответы на контрольные вопросы

1. Что понимается под гетерогенной параллелизацией?

Гетерогенная параллелизация - это когда в одной программе мы используем разные типы вычислительных устройств. Обычно это CPU (процессор) и GPU (видеокарта), которые работают вместе над одной задачей.

Суть в том, что каждое устройство делает то, что умеет лучше всего. CPU хорош в сложной логике, быстром переключении между задачами и работе с небольшими объемами данных. GPU отлично справляется с обработкой огромных массивов данных, когда нужно сделать одну и ту же операцию много раз.

Простой пример: обработка видео. CPU читает файл, решает какие эффекты применять, управляет всем процессом. GPU берет каждый кадр и обрабатывает миллионы пикселей одновременно, применяя фильтры или изменяя цвета.

2. В чем принципиальные различия архитектур CPU и GPU?

CPU и GPU устроены совершенно по-разному под разные задачи.

CPU (процессор):

- Имеет мало ядер (обычно 4-16), но каждое ядро очень мощное
- Может быстро выполнять сложные инструкции и принимать решения
- Хорошо справляется с последовательными задачами где много логики
- Имеет большой кеш (быстрая память рядом с ядрами) для быстрого доступа к данным
- Может быстро переключаться между разными программами
- Частота работы высокая (3-5 ГГц)

GPU (видеокарта):

- Имеет тысячи маленьких простых ядер (от 2000 до 10000+)
- Каждое ядро слабее чем ядро CPU, но их очень много
- Все ядра делают одну и ту же работу одновременно (это называется SIMD - Single Instruction Multiple Data)
- Меньше кеша на ядро, но очень быстрая память для больших массивов
- Плохо работает с задачами где много ветвлений (if-else)
- Частота работы ниже (1-2 ГГц), но компенсируется количеством ядер

Аналогия: CPU - это как один очень умный математик, который может решать разные сложные задачи. GPU - это как тысяча школьников, каждый из которых умеет только складывать числа, но все вместе они могут сложить миллион чисел очень быстро.

3. Какие типы задач лучше подходят для выполнения на GPU, а какие на CPU?

Задачи для GPU:

- Обработка изображений и видео (каждый пиксель обрабатывается одинаково)
- Матричные операции (умножение больших матриц)
- Физические симуляции (расчет движения множества частиц)
- Обучение нейронных сетей (куча одинаковых вычислений)
- Криптография (перебор вариантов)
- Научные расчеты с большими данными

Общее правило: если у тебя есть большой массив данных и ты делаешь с каждым элементом одно и то же - это для GPU.

Задачи для CPU:

- Работа с базами данных (много логики, мало данных)
- Управление файлами и сетью
- Логика игр (что делают персонажи, проверка правил)
- Обработка текста и парсинг
- Задачи с большим количеством условий и ветвлений
- Операционная система

Общее правило: если в коде много if-else, циклы разной длины, обращения к разным местам памяти - это для CPU.

Почему так? GPU плохо работает когда разные потоки делают разные вещи. Если в коде есть if-else, то половина потоков ждет пока другая половина выполняет свою ветку. Это называется divergence и очень замедляет GPU.

CPU наоборот легко справляется с ветвлениями благодаря предсказанию переходов и другим технологиям.

4. Почему не все алгоритмы эффективно распараллеливаются с использованием OpenMP?

Есть несколько причин почему OpenMP не всегда помогает:

1. Зависимость данных Если каждый следующий шаг зависит от результата предыдущего, то распараллелить не получится. Пример: вычисление чисел Фибоначчи где каждое число зависит от двух предыдущих.

2. Маленький объем работы Создание потоков требует времени. Если задача маленькая (например сложить 10 чисел), то накладные расходы на создание потоков будут больше чем выигрыш от параллелизма.

3. Гонки данных (race conditions) Когда несколько потоков пытаются изменить одну и ту же переменную, нужна синхронизация. Это замедляет программу. Если

синхронизации слишком много, параллельная версия может быть даже медленнее последовательной.

4. Неравномерная нагрузка Если разные потоки делают разное количество работы, то быстрые потоки будут ждать медленные. Это называется load imbalance.

5. Доступ к памяти Если все потоки пытаются читать из одного и того же места в памяти, возникает конкуренция за доступ. Это может стать узким местом.

Пример из нашей лабы: сортировка выбором. На каждом шаге мы ищем минимум в оставшейся части массива. Можем распараллелить поиск минимума, но сам следующий шаг начнется только когда закончится текущий. Плюс нужна синхронизация для сравнения локальных минимумов. В итоге выигрыш небольшой или вообще отсутствует.

5. В чем заключается основная идея алгоритма сортировки слиянием?

Сортировка слиянием работает по принципу "разделяй и властвуй". Основная идея очень простая:

Шаг 1: Разделяй Берем массив и делим его пополам. Потом каждую половину делим еще пополам. И так далее, пока не получим кучу массивов по одному элементу. Один элемент - это уже отсортированный массив.

Шаг 2: Властвуй (сливай) Теперь начинаем сливать маленькие отсортированные массивы в большие. Когда сливаем два отсортированных массива, мы просто смотрим на первые элементы каждого и берем меньший. Получается новый отсортированный массив.

Пример:

Исходный: [5, 2, 8, 1, 9, 3]

Разделяем:
[5, 2, 8] и [1, 9, 3]
[5, 2] [8] и [1, 9] [3]
[5] [2] [8] и [1] [9] [3]

Сливаем:
[2, 5] [8] и [1, 9] [3]
[2, 5, 8] и [1, 3, 9]
[1, 2, 3, 5, 8, 9]

Почему это хорошо:

- Сложность O(n log n) - это быстро для больших данных
- Стабильная сортировка (элементы с одинаковыми значениями сохраняют порядок)
- Хорошо параллелится - можно сортировать разные части независимо
- Предсказуемое время работы (не зависит от входных данных как quicksort)

Минус:

- Требует дополнительную память для слияния (O(n))

6. Какие сложности возникают при реализации сортировки слиянием на GPU?

Реализация на GPU - это не просто взять обычный код и запустить. Есть несколько проблем:

1. Управление памятью Нужно копировать данные с CPU на GPU и обратно. Это занимает время. Если массив маленький, время копирования может быть больше чем время сортировки.

2. Рекурсия Классическая сортировка слиянием использует рекурсию. На GPU рекурсия работает плохо или вообще не поддерживается в старых версиях CUDA. Приходится переписывать алгоритм итеративно.

3. Синхронизация Потоки в разных блоках не могут синхронизироваться напрямую. Приходится делать несколько запусков kernel'ов и использовать cudaDeviceSynchronize(), что медленно.

4. Разделение работы Нужно правильно разделить массив между блоками и потоками. Если сделать неправильно, часть GPU будет простаивать.

5. Коалесцентный доступ к памяти GPU работает быстро когда соседние потоки читают соседние ячейки памяти. При слиянии мы читаем из разных мест, что замедляет работу.

6. Дополнительная память Нужен временный массив для слияния. На GPU память ограничена, особенно быстрая shared memory.

Как решаем:

- Сначала сортируем маленькие куски простым алгоритмом (insertion sort)
- Потом сливаем куски итеративно, удваивая размер на каждом шаге
- Используем много блоков чтобы загрузить весь GPU
- Оптимизируем доступ к памяти

7. Как выбор размера блока и сетки влияет на производительность вычислений на GPU?

Размер блока (block size) и размер сетки (grid size) - это критические параметры для производительности.

Размер блока (сколько потоков в блоке):

Обычно выбирают 128, 256 или 512 потоков. Почему?

- GPU выполняет потоки группами по 32 (это называется warp)
- Размер блока должен быть кратен 32 для эффективности
- Слишком маленький блок (32-64) - не загружаем GPU полностью
- Слишком большой блок (1024) - может не хватить ресурсов (регистров, shared memory)
- 256 - это обычно золотая середина

Размер сетки (сколько блоков):

Зависит от задачи и размера блока:

gridSize = (totalWork + blockSize - 1) / blockSize

Нюансы:

- Нужно достаточно блоков чтобы загрузить все SM (streaming multiprocessors) на GPU
- Современные GPU имеют 20-100+ SM
- Каждый SM может выполнять несколько блоков одновременно
- Обычно делают блоков в 2-4 раза больше чем SM

Что влияет на выбор:

1. **Occupancy (загруженность)** Сколько процентов ресурсов GPU мы используем. Хотим близко к 100%.

2. **Shared memory** Если используем много shared memory на блок, меньше блоков поместится на SM.

3. **Регистры** Каждому потоку нужны регистры. Если kernel использует много регистров, меньше потоков поместится.

4. **Характер задачи** Если много вычислений - можно меньше потоков. Если много обращений к памяти - нужно больше потоков чтобы скрыть latency.

Практический совет: Начинай с 256 потоков на блок. Потом экспериментируй с 128, 512. Смотри на время выполнения и выбирай лучшее.

8. Почему гетерогенный подход может быть эффективнее использования только CPU или только GPU?

Гетерогенный подход дает лучшее от обоих миров. Вот почему:

1. Каждому свое Разные части программы имеют разные требования. Логика и управление лучше на CPU. Массовые вычисления - на GPU. Если использовать только GPU, он будет простаивать на простых операциях. Если только CPU - будет медленно считать большие массивы.

2. Параллельная работа Пока GPU считает, CPU может делать другую работу - читать следующую порцию данных, готовить результаты, логировать. Оба устройства работают одновременно.

3. Оптимизация под размер задачи Маленькие задачи быстрее на CPU (не тратим время на копирование данных на GPU). Большие задачи - на GPU (окупаются накладные расходы). Можем динамически выбирать где запускать.

4. Меньше узких мест Если весь pipeline на CPU, он становится узким местом. Если только на GPU - узкое место в копировании данных. Гетерогенный подход распределяет нагрузку.

Реальный пример: Обучение нейросети:

- CPU: загружает данные с диска, делает augmentation (случайные преобразования изображений), подготавливает батчи
- GPU: считает forward pass и backward pass (основные вычисления)
- Пока GPU считает один батч, CPU уже готовит следующий
- Результат: нет простоя, максимальная скорость

Если бы использовали только GPU: GPU ждал бы пока загрузятся данные и подготовятся батчи. Простой дорогого оборудования.

Если бы использовали только CPU: Обучение заняло бы в 100 раз больше времени. Некоторые модели вообще невозможно обучить на CPU за разумное время.

Вывод: Гетерогенный подход - это как хорошая команда. Каждый делает то, что умеет лучше всего, и все работают одновременно. Это быстрее, эффективнее и дешевле чем использовать только одно устройство.