ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ

Assignment 4: Гибридные и распределённые параллельные вычисления

ВОПРОС 1: В чём заключается отличие гибридных вычислений от вычислений только на CPU или только на GPU?

Гибридные вычисления это когда мы используем и процессор и видеокарту одновременно для решения одной задачи.

Когда мы работаем только на CPU, у нас есть несколько ядер (обычно 4-16), которые работают на высокой частоте. Процессор хорош для последовательных задач где нужна логика и сложные вычисления.

Когда мы работаем только на GPU, у нас есть тысячи маленьких ядер, которые работают медленнее но могут обрабатывать много данных параллельно. GPU идеален для простых однотипных операций над большими объёмами данных.

В гибридных вычислениях мы делим задачу на части: то что требует сложной логики отправляем на CPU, а массовые параллельные операции на GPU. Или можем просто разделить данные пополам и обрабатывать их одновременно на обоих устройствах. Это позволяет использовать сильные стороны обоих типов процессоров и получить максимальную производительность.

Основная идея: не ждать пока одно устройство закончит работу, а задействовать все доступные вычислительные ресурсы сразу.

ВОПРОС 2: Для каких типов задач целесообразно распределять вычисления между CPU и GPU?

Гибридный подход имеет смысл для задач которые можно разделить на две части с разными характеристиками:

1. Задачи с неоднородной нагрузкой
Например обработка видео: GPU декодирует и обрабатывает кадры (массовые операции с пикселями), а CPU занимается анализом содержимого и принятием решений.

2. Задачи с большими объёмами данных
Когда данных настолько много что GPU не может всё уместить в памяти, CPU может обрабатывать часть данных пока GPU работает с другой частью.

3. Конвейерные вычисления

Когда результат одной стадии нужен для следующей. CPU может готовить данные для GPU, GPU обрабатывает, CPU собирает результаты. Работает как конвейер.

4. Задачи с разной сложностью операций
Часть данных требует простых операций (умножение, сложение) - идёт на GPU. Другая часть требует сложной логики, условий, рекурсии - идёт на CPU.

5. Научные симуляции
Физические расчёты (движение частиц) на GPU, сложная математика и контроль симуляции на CPU.

НЕ имеет смысла использовать гибридный подход когда:
- Задача маленькая (накладные расходы на синхронизацию будут больше выигрыша)
- Задача полностью последовательная
- Нет чёткого способа разделить данные или вычисления

ВОПРОС 3: В чём разница между синхронной и асинхронной передачей данных между CPU и GPU?

Синхронная передача (по умолчанию):
Когда мы копируем данные на GPU функцией cudaMemcpy, программа на CPU останавливается и ждёт пока копирование полностью завершится. CPU просто стоит и ничего не делает пока данные едут. Это простой и надёжный способ но мы теряем время.

Пример: посылаешь файл другу и сидишь смотришь на прогресс-бар пока файл не загрузится, ничего другого не делаешь.

Асинхронная передача:
Используем cudaMemcpyAsync. Программа запускает копирование данных но сразу продолжает выполнение следующих инструкций, не дожидаясь завершения. CPU может делать другую полезную работу пока данные копируются в фоновом режиме. Потом когда нам нужны эти данные, мы вызываем cudaDeviceSynchronize чтобы дождаться завершения.

Пример: отправляешь файл и пока он загружается, делаешь другие дела, периодически проверяешь загрузился ли.

Ключевая разница:
- Синхронная = блокирующая операция, CPU ждёт
- Асинхронная = неблокирующая операция, CPU продолжает работать

Для асинхронной передачи нужно использовать специальные потоки CUDA (CUDA streams) и закреплённую память (pinned memory).

ВОПРОС 4: Почему асинхронная передача данных может повысить производительность программы?

Асинхронная передача повышает производительность за счёт перекрытия операций - пока одно происходит, делаем другое:

1. Перекрытие вычислений и передачи данных
CPU может готовить следующую порцию данных пока GPU обрабатывает предыдущую порцию. Или GPU может вычислять пока данные копируются. Вместо "копируй → вычисляй → копируй → вычисляй" получаем "копируй И вычисляй одновременно".

2. Конвейерная обработка
Разбиваем большую задачу на куски. Пока GPU обрабатывает кусок 1, CPU готовит кусок 2. Когда GPU закончит с куском 1 и начнёт кусок 2, CPU готовит кусок 3. Получается непрерывный поток работы.

3. Скрытие задержек
Передача данных между CPU и GPU медленная (узкое место). Если делать её асинхронно, можем скрыть эту задержку полезной работой. CPU не простаивает в ожидании.

4. Использование нескольких GPU
Можем запускать операции на разных GPU параллельно через асинхронные вызовы.

Практический пример:
Синхронно: копируй 100мс → вычисляй 200мс → копируй обратно 100мс = 400мс
Асинхронно: копируй И вычисляй одновременно, копируй обратно И следующую порцию вычисляй = может быть 250мс

Важно понимать: асинхронность полезна только если есть что делать параллельно. Если просто запустить асинхронное копирование и сразу ждать, никакого выигрыша не будет.

ВОПРОС 5: Какие основные функции MPI используются для распределения и сбора данных между процессами?

MPI это библиотека для запуска программы на нескольких компьютерах или процессорах одновременно. Вот основные функции:

1. MPI_Init и MPI_Finalize
MPI_Init - запускает MPI окружение в начале программы
MPI_Finalize - закрывает MPI окружение в конце
Обязательно вызывать в начале и конце main().

2. MPI_Comm_size
Узнаём сколько всего процессов запущено. Например если mpirun -np 4, то вернёт 4.

3. MPI_Comm_rank
Узнаём номер текущего процесса (от 0 до N-1). Каждый процесс получает свой уникальный номер. Это как ID процесса.

4. MPI_Send и MPI_Recv
Базовые функции для отправки и получения данных между двумя конкретными процессами. Процесс 0 может отправить данные процессу 1, процесс 1 получает их.

5. MPI_Scatter
Главный процесс (обычно ранг 0) разделяет массив на равные части и рассылает каждому процессу его часть. Одна операция раздаёт всем.

6. MPI_Scatterv
То же что Scatter но части могут быть разного размера. Нужен когда массив не делится нацело на количество процессов.

7. MPI_Gather
Обратная операция к Scatter. Каждый процесс отправляет свою часть данных главному процессу, который собирает всё в один массив.

8. MPI_Gatherv
Gather для частей разного размера.

9. MPI_Reduce
Каждый процесс имеет число, функция собирает эти числа и применяет операцию (сумма, максимум, минимум). Результат получает главный процесс.

10. MPI_Barrier
Точка синхронизации. Все процессы доходят до барьера и ждут пока все остальные дойдут. Потом все продолжают вместе. Нужно для точных замеров времени.

11. MPI_Bcast

Главный процесс рассылает одни и те же данные всем остальным процессам. Например настройки программы.

Типичная схема работы:
1. MPI_Init - инициализация
2. MPI_Comm_rank, MPI_Comm_size - узнаём кто мы и сколько нас
3. MPI_Scatter - раздаём данные
4. Локальная обработка каждым процессом своей части
5. MPI_Gather - собираем результаты
6. MPI_Finalize - завершение

ВОПРОС 6: Как количество процессов MPI влияет на время выполнения программы и почему?

Влияние количества процессов на производительность нелинейное и имеет ограничения:

Теоретически:
Если задача хорошо параллелится, удвоение процессов должно ускорить программу в 2 раза. 4 процесса - в 4 раза быстрее чем 1 процесс.

На практике:
Реальное ускорение всегда меньше идеального из-за накладных расходов.

Что происходит при увеличении процессов:

1. Положительные эффекты:
- Каждый процесс обрабатывает меньше данных
- Больше вычислительной мощности задействовано
- Можем использовать память нескольких машин

2. Отрицательные эффекты (накладные расходы):
- Коммуникация между процессами занимает время
- Нужно разделить данные (Scatter) - это время
- Нужно собрать результаты (Gather) - это время
- Синхронизация процессов (Barrier) - ждём самого медленного
- Каждый запуск процесса имеет начальные затраты

График зависимости обычно выглядит так:
- 1 процесс: 1000мс (базовая линия)
- 2 процесса: 550мс (ускорение 1.8x, не 2x из-за коммуникации)
- 4 процесса: 300мс (ускорение 3.3x)

- 8 процессов: 180мс (ускорение 5.5x)
- 16 процессов: 150мс (ускорение 6.7x) - уже мало улучшений

Почему рост замедляется:
- Время на коммуникацию растёт с количеством процессов
- Каждый процесс получает всё меньше работы
- Наступает момент когда время на коммуникацию больше времени вычислений

Закон Амдала:
Даже если часть программы идеально параллелится, есть последовательные участки (инициализация, сбор результатов, вывод). Эти участки ограничивают максимальное ускорение.

Оптимальное количество процессов зависит от:
- Размера задачи (большие задачи лучше масштабируются)
- Соотношения вычислений к коммуникации
- Доступных ресурсов
- Скорости сети между узлами

Правило: имеет смысл добавлять процессы пока время вычислений значительно больше времени коммуникации.

ВОПРОС 7: Какие факторы ограничивают масштабируемость распределённых параллельных программ?

Масштабируемость это насколько хорошо программа ускоряется при добавлении ресурсов. Вот что её ограничивает:

1. Последовательные участки кода (закон Амдала)
Части программы которые нельзя распараллелить. Инициализация, финализация, сбор результатов, вывод - всё это делает один процесс. Если 5% программы последовательны, максимальное ускорение не больше 20x даже с тысячей процессов.

2. Коммуникация между процессами
Процессам нужно обмениваться данными. Чем больше процессов, тем больше сообщений. Сеть имеет ограниченную пропускную способность. При большом количестве процессов можем получить перегрузку сети.

3. Синхронизация и ожидание
Барьеры синхронизации заставляют быстрые процессы ждать медленные. Если один процесс тормозит (загрузка на узле, сеть), все ждут.

4. Балансировка нагрузки
Если данные распределены неравномерно, одни процессы закончат быстро и будут ждать другие. Простое деление массива может быть неоптимальным если обработка разных элементов занимает разное время.

5. Размер задачи
Маленькие задачи плохо масштабируются. Накладные расходы на коммуникацию становятся сопоставимы с временем вычислений. Например сортировка 100 элементов на 8 процессах будет медленнее чем на одном.

6. Ограничения памяти
У каждого узла ограниченная память. Нужно распределять данные так чтобы они помещались. Иногда приходится добавлять процессы не для скорости, а чтобы разместить данные.

7. Зависимости по данным
Если результат одной части нужен для вычисления другой части, появляется последовательность. Нельзя начать следующий шаг пока не закончится предыдущий.

8. Накладные расходы MPI
Сама библиотека MPI имеет свои затраты. Запуск процессов, управление коммуникаторами, сериализация данных.

9. Аппаратные ограничения
- Количество физических ядер и узлов
- Скорость межпроцессорных соединений
- Топология сети (некоторые узлы дальше друг от друга)
- Конкуренция за ресурсы (кеш, память, шина)

10. Алгоритмическая эффективность
Некоторые алгоритмы плохо параллелятся по своей природе. Например обход графа в глубину сложно распараллелить.

Практический пример:
Задача занимает 100 секунд, из них 95 секунд параллельные вычисления, 5 секунд последовательная часть.
- 2 процесса: 95/2 + 5 = 52.5 сек (ускорение 1.9x)
- 10 процессов: 95/10 + 5 = 14.5 сек (ускорение 6.9x)
- 100 процессов: 95/100 + 5 = 5.95 сек (ускорение 16.8x, а не 100x!)

Видно что даже 5% последовательного кода сильно ограничивают максимальное ускорение.

ВОПРОС 8: В каких случаях использование распределённых вычислений оправдано, а в каких - неэффективно?

ОПРАВДАНО использовать распределённые вычисления когда:

1. Очень большие объёмы данных
Данные не помещаются в память одной машины. Нужно распределить по нескольким узлам. Например анализ терабайтов данных, обработка больших графов.

2. Долгие вычисления
Задача выполняется часами или днями на одной машине. Распараллеливание может сократить время до минут или часов. Научные симуляции, рендеринг видео, обучение нейросетей.

3. Хорошо параллелящиеся задачи
Задачи где части независимы друг от друга. Обработка изображений, симуляция частиц, перебор вариантов, анализ логов.

4. Много простых операций над данными
Матричные операции, статистические расчёты, применение функции к каждому элементу массива.

5. Доступны вычислительные ресурсы
Есть кластер или можно арендовать облачные ресурсы. Простаивающие мощности лучше использовать.

6. Соотношение вычислений к коммуникации высокое
Каждая часть данных обрабатывается долго, а передаётся быстро. Процессы редко обмениваются данными.

7. Нужна отказоустойчивость
Распределённые системы могут продолжать работу даже если один узел упадёт (при правильной архитектуре).

НЕЭФФЕКТИВНО использовать распределённые вычисления когда:

1. Маленькие задачи
Сортировка 1000 элементов, вычисление суммы маленького массива. Накладные расходы MPI будут больше выигрыша. Одна машина справится быстрее.

2. Сильная связность данных
Каждый шаг требует результатов предыдущего шага. Постоянный обмен данными между процессами. Больше времени на коммуникацию чем на вычисления.

3. Последовательные алгоритмы
Алгоритмы которые невозможно или очень сложно распараллелить. Рекурсивные обходы, некоторые динамические программы.

4. Высокие требования к синхронизации
Процессы должны постоянно согласовывать свои действия. Частые барьеры убивают производительность.

5. Ограниченная пропускная способность сети
Медленная сеть между узлами. Время передачи данных превышает время вычислений.

6. Нет доступа к ресурсам
Если есть только один компьютер, MPI может использовать несколько ядер но эффективнее будет OpenMP или потоки.

7. Простые однопоточные решения быстрые
Если задача решается за секунды на одном ядре, распараллеливание избыточно.

8. Дорого по стоимости
Аренда кластера стоит денег. Если выигрыш в производительности не критичен, лучше подождать подольше но сэкономить.

Правило большого пальца:
Если задача занимает меньше минуты на одной машине - распределённые вычисления избыточны.
Если задача занимает часы-дни и хорошо делится на независимые части - распределённые вычисления оправданы.

Пример хорошей задачи для MPI:
Расчёт погоды на неделю вперёд. Огромные объёмы данных (температура, давление, ветер по всей планете), много вычислений, можно разделить планету на регионы.

Пример плохой задачи для MPI:
Вычисление чисел Фибоначчи. Каждое следующее число зависит от предыдущих, параллелить нечего.