Ответы на контрольные вопросы

Вопрос 1: Какие основные типы памяти существуют в архитектуре CUDA и чем они отличаются по скорости доступа?

В архитектуре CUDA есть несколько типов памяти:

Регистровая память - самая быстрая память. Находится прямо в процессорных ядрах. Каждый поток имеет свои регистры. Скорость доступа примерно 1 такт.

Разделяемая память (shared memory) - быстрая память, которая доступна всем потокам внутри одного блока. Находится на чипе близко к вычислительным ядрам. Скорость доступа примерно 5-30 тактов. Используется для обмена данными между потоками в блоке.

Локальная память - на самом деле хранится в глобальной памяти, но предназначена для локальных переменных потока, которые не поместились в регистры. Медленная, как глобальная память.

Глобальная память - самая медленная, но самая большая память. Доступна всем потокам на GPU. Скорость доступа примерно 400-800 тактов. Находится в отдельных чипах памяти GDDR.

Константная память - часть глобальной памяти, но с кэшированием. Используется для данных, которые не меняются. Быстрее глобальной при повторных обращениях.

Текстурная память - тоже часть глобальной памяти с особым кэшированием, оптимизирована для 2D и 3D доступа.

Разница в скорости огромная - регистры в сотни раз быстрее глобальной памяти.

Вопрос 2: В каких случаях использование разделяемой памяти позволяет ускорить выполнение CUDA-программы?

Разделяемая память помогает ускорить программу в таких случаях:

Когда данные используются многократно - если один элемент из глобальной памяти нужен нескольким потокам или используется несколько раз, лучше загрузить его в разделяемую память один раз, а потом быстро читать оттуда.

При взаимодействии между потоками - когда потоки в блоке обмениваются данными между собой. Например, при свертке, когда соседние потоки работают с перекрывающимися данными.

Для избежания конфликтов доступа - можно изменить паттерн доступа к данным через разделяемую память, чтобы сделать его более эффективным.

При редукции (сворачивании) данных - когда нужно суммировать или найти максимум среди элементов. Потоки загружают данные в разделяемую память и обрабатывают их локально.

Важно: разделяемая память не всегда ускоряет программу. Если данные читаются только один раз и нет взаимодействия между потоками, накладные расходы на копирование в разделяемую память могут замедлить выполнение.

Вопрос 3: Как шаблон доступа к глобальной памяти влияет на производительность GPU-программы?

Шаблон доступа очень сильно влияет на скорость работы:

Коалесцированный доступ - когда соседние потоки в варпе обращаются к соседним ячейкам памяти. Например, поток 0 читает элемент 0, поток 1 - элемент 1, и так далее. В этом случае GPU объединяет все обращения в одну большую транзакцию памяти, что очень быстро.

Некоалесцированный доступ - когда потоки обращаются к разбросанным ячейкам памяти. Например, с большим шагом или в случайном порядке. Тогда GPU вынужден делать много отдельных транзакций памяти, что медленно.

Конкретный пример: если 32 потока в варпе читают 32 последовательных float элемента, это одна транзакция 128 байт. Если эти же потоки читают данные с шагом 100 элементов, понадобится до 32 отдельных транзакций.

Разница в производительности может быть в 5-10 раз. Поэтому при разработке CUDA-программ нужно следить, чтобы потоки обращались к памяти последовательно.

Вопрос 4: Почему одинаковый алгоритм на GPU может показывать разное время выполнения при разных способах обращения к памяти?

Дело в том, как устроена архитектура GPU и его подсистема памяти:

Пропускная способность памяти ограничена - GPU может передавать определенное количество данных в единицу времени. При неэффективном доступе эта пропускная способность используется плохо.

Размер транзакций памяти фиксирован - GPU читает память блоками по 32, 64 или 128 байт. Если нужен один байт из блока, все равно прочитается весь блок. При плохом паттерне доступа много данных читается впустую.

Кэш-память работает по-разному - последовательный доступ хорошо кэшируется, случайный - плохо. Кэш промахи очень дорогие по времени.

Конфликты банков разделяемой памяти - при одновременном обращении нескольких потоков к одному банку памяти возникают конфликты, и обращения выполняются последовательно, а не параллельно.

Задержки памяти - глобальная память имеет задержку 400-800 тактов. При хорошей организации доступа GPU может скрывать эти задержки, переключаясь между варпами. При плохой - потоки простаивают в ожидании данных.

Поэтому один и тот же алгоритм может работать в разы быстрее или медленнее в зависимости от того, как организован доступ к памяти.

Вопрос 5: Как размер блока потоков влияет на производительность CUDA-ядра?

Размер блока критически важен для производительности:

Слишком маленький блок (например, 32-64 потока) - плохо использует ресурсы GPU. Мультипроцессоры не загружены полностью, мало варпов для скрытия задержек памяти.

Слишком большой блок (близко к максимуму 1024) - может ограничить количество блоков на мультипроцессоре из-за ограничений по регистрам и разделяемой памяти. Это тоже снижает occupancy.

Оптимальный размер (обычно 128-512) - баланс между загрузкой ресурсов и гибкостью планировщика. Обычно 256 потоков - хороший выбор для большинства задач.

Важные правила:

- Размер блока должен быть кратен размеру варпа (32)
- Нужно учитывать использование регистров и разделяемой памяти
- Для простых ядер можно использовать большие блоки (512-1024)
- Для сложных ядер с большим потреблением ресурсов - меньшие блоки (128-256)

Occupancy - процент загрузки мультипроцессора. Высокий occupancy обычно хорошо, но не всегда. Главное - баланс между occupancy и эффективностью использования памяти.

Вопрос 6: Что такое варп и почему важно учитывать его при разработке CUDA-программ?

Варп (warp) - это группа из 32 потоков, которые выполняются одновременно на GPU в режиме SIMD (Single Instruction Multiple Data). Все потоки в варпе выполняют одну и ту же инструкцию, но над разными данными.

Почему это важно:

Дивергенция ветвлений - если потоки в варпе попадают в разные ветки if-else, они выполняются последовательно. Сначала одна ветка (для части потоков), потом другая (для остальных). Это замедляет выполнение.

Транзакции памяти - GPU объединяет обращения к памяти от потоков одного варпа. Для эффективности нужно, чтобы потоки в варпе обращались к соседним адресам.

Размер блока - должен быть кратен 32, иначе последний неполный варп будет работать неэффективно.

Синхронизация - потоки внутри варпа всегда синхронизированы автоматически, не нужно использовать __syncthreads() для них.

Пример плохого кода:

```
if (threadIdx.x % 2 == 0) {
   // четные потоки
} else {
   // нечетные потоки
}
```

Половина потоков в каждом варпе ждет, пока выполнится другая половина.

Пример хорошего кода: Все потоки в варпе идут по одной ветке кода, разделение происходит на уровне варпов, а не внутри них.

Вопрос 7: Какие факторы необходимо учитывать при выборе конфигурации сетки и блоков потоков?

При выборе конфигурации нужно учитывать много факторов:

Размер данных - количество блоков должно быть достаточным, чтобы обработать все данные. Формула: numBlocks = (N + blockSize - 1) / blockSize.

Ограничения оборудования:

- Максимум 1024 потока на блок (обычно)
- Максимум потоков на мультипроцессоре (например, 2048)
- Максимум блоков на мультипроцессоре (обычно 16-32)

Использование ресурсов:

- Регистры на поток (больше регистров = меньше блоков на SM)
- Разделяемая память на блок (больше памяти = меньше блоков на SM)

Occupancy - нужно стремиться к высокой загрузке мультипроцессоров. Можно использовать CUDA Occupancy Calculator или функцию cudaOccupancyMaxPotentialBlockSize.

Характер задачи:

- Для задач с большим числом вычислений и малым обращением к памяти - большие блоки
- Для задач с частым обращением к памяти - средние блоки для лучшего скрытия задержек

Размер варпа - блок должен быть кратен 32.

Практический подход: начать с 256 потоков на блок, измерить производительность, попробовать 128 и 512, выбрать лучший вариант.

Вопрос 8: Почему оптимизация CUDA-программы часто начинается с анализа работы с памятью, а не с изменения алгоритма?

Память - это главное узкое место в большинстве GPU-программ:

GPU очень быстрый на вычислениях - современные GPU делают тысячи операций за такт. Арифметика почти всегда быстрее, чем доступ к памяти.

Память очень медленная - глобальная память в сотни раз медленнее регистров. Одно обращение к глобальной памяти стоит столько же, сколько сотни арифметических операций.

Пропускная способность памяти ограничена - даже если GPU может делать триллионы операций в секунду, он может прочитать только определенное количество гигабайт данных в секунду.

Большинство программ memory-bound - они ограничены скоростью памяти, а не скоростью вычислений. Процессор простаивает в ожидании данных.

Оптимизация памяти дает больший эффект - улучшение паттерна доступа к памяти может дать ускорение в 5-10 раз, в то время как изменение алгоритма обычно дает 20-30% ускорения.

Примеры оптимизаций памяти:

- Коалесцированный доступ вместо случайного - ускорение в 5-10 раз
- Использование разделяемой памяти для переиспользования данных - ускорение в 2-5 раз
- Правильная организация данных в памяти - ускорение в 2-3 раза

Только после того, как оптимизирован доступ к памяти, имеет смысл думать об алгоритмах. Иначе даже самый лучший алгоритм будет работать медленно из-за плохой работы с памятью.