Экзамен

❓ 1. Архитектура, микроархитектура: отличия, примеры

Архитектура компьютера — это концептуальная модель, описывающая функциональность, организацию и поведение компьютерной системы. Она определяет, что может делать процессор и как с ним взаимодействовать.

Ключевые аспекты архитектуры:

1. Набор инструкций (ISA, Instruction Set Architecture) — это набор команд, которые может выполнять процессор. Например, команды сложения, умножения, загрузки данных из памяти и т.д. ISA определяет, как работают программы на уровне машинных кодов.
2. Организация памяти (адресное пространство, выравнивание данных, кэширование).
3. Режимы работы процессора: пользовательский режим, привилегированный режим, виртуализация и т.п.
4. Типы данных и форматы (целые числа, числа с плавающей точкой, их представление).
5. Прерывания и исключения.
6. Поддержка многопоточности.

Примеры архитектуры:

- x86;
- ARM.
- RISC-V.

Микроархитектура — это конкретная реализация архитектуры на уровне аппаратного обеспечения. Она описывает, как именно реализованы те функции, которые описаны архитектурой. Микроархитектура включает в себя схемы, компоненты и алгоритмы, которые обеспечивают выполнение команд, заданных архитектурой.

Ключевые аспекты микроархитектуры:

1. Устройство процессора: конструкция и взаимодействие компонентов, таких как блоки выполнения инструкций, регистры, кэш-память, буферы.
2. Конвейеризация: процессор может разбивать выполнение одной команды на несколько этапов (например, выборка, декодирование, выполнение, запись результата).
3. Кэширование**: р**еализация кэш-памяти (уровни кэша L1, L2, L3) для ускорения доступа к данным.
4. Тактирование и энергопотребление.

Пример микроархитектуры:

- Intel Skylake (x86);
- ARM Cortex-A76 (ARM);
- AMD Zen 4 (x86-64):

❓ 3. Классификация архитектур (CISC, RISC, VLIW, EPIC)

CISC (Complex Instruction Set Computer) — это архитектура, в которой набор инструкций содержит большое количество сложных команд. Каждая инструкция может выполнять сразу несколько операций (например, загрузка данных из памяти, арифметическая операция и запись результата обратно):

- большой набор инструкций;
- разные форматы инструкций;
- меньший объем программного кода;
- большая сложность оборудования;
- ограниченное количество регистров;
- многочисленные способы адресации.

Примеры CISC-архитектур: x86.

RISC (Reduced Instruction Set Computer) — это архитектура, в которой используется минимальный и упрощенный набор команд. Каждая инструкция выполняет одну операцию и имеет фиксированную длину:

- простые инструкции;
- значительно больше регистров, чем в CISC;
- меньшая сложность оборудования ⇒ снижает энергопотребление;
- больший объем кода;
- ограниченные способы адресации.

Примеры RISC-архитектур: ARM, MIPS, SPARC, RISC-V.

VLIW (Very Long Instruction Word) — это архитектура, в которой несколько операций объединяются в одну длинную инструкцию и выполняются параллельно:

- параллельное выполнение команд: каждая длинная инструкция содержит несколько операций, которые выполняются одновременно;
- проверка зависимостей и планирование исполнения возлагается на компилятор;
- увеличенный размер программного кода.

Примеры VLIW-архитектур: Intel Itanium (IA-64).

EPIC (Explicitly Parallel Instruction Computing) — это архитектура, в которой компилятор явно организует выполнение нескольких независимых инструкций одновременно. EPIC можно рассматривать как развитие идей VLIW с добавлением новых возможностей:

- явный параллелизм: на уровне архитектуры определяются независимые инструкции, которые могут выполняться параллельно, и формирует пакеты инструкций;
- аппаратное предсказание ветвлений: в отличие от VLIW, EPIC использует сложные механизмы предсказания ветвлений на уровне процессора.

Примеры EPIC-архитектур: Intel Itanium.

❓ 4. Классификация архитектур (x86, x86-64, Power, ARM, IA64, RISC-V, MIPS, Alpha, ...)

Архитектуры CISC (Complex Instruction Set Computing)

1. x86: используется в настольных ПК, ноутбуках и серверах.

Характеристики:

- 32-разрядная архитектура;
- обратная совместимость с ранними версиями;
- применение: Windows/Unix системы.

2. x86-64 (x64): используется в современных настольных ПК, серверы.

- расширение x86 с поддержкой 64-разрядных вычислений;
- больше регистров;
- поддержка больших объемов памяти (более 4 ГБ).

Архитектуры RISC (Reduced Instruction Set Computing)

1. ARM (Advanced RISC Machine): используется в мобильных устройствах.

Характеристики:

- высокая энергоэффективность;
- варианты: ARMv7 (32-bit), ARMv8 (64-bit).

2. Power (IBM): используется в серверах и суперкомпьютерах.

Характеристики:

- высокая производительность;
- поддержка SIMD-операций;

3. MIPS (Microprocessor without Interlocked Pipeline Stages): использовалась в высокопроизводительных системах (в частности в сетях).

- простая и быстрая архитектура.

4. RISC-V: используется в IoT, исследованиях, встраиваемыех системах.

- открытая архитектура;
- модульная структура: можно адаптировать под конкретные задачи.

5. Alpha**: и**спользовалась в серверах и рабочих станциях.

- 64-разрядная архитектура, разработанная DEC.

Гибридные архитектуры

IA-64 (Itanium Architecture) (Intel): используется в серверных решениях.

- использует EPIC (Explicitly Parallel Instruction Computing) для параллелизма;
- ограниченная поддержка из-за низкой популярности.

❓ 5. Кодирование инструкций на примере MIPS и x86 (пояснение)

x86/x86-64 (первое изображение)

Компоненты инструкции:

1. **Instruction Prefixes (Префиксы):**

- Префиксы длиной 1 байт (или больше) модифицируют инструкцию:
- Например, переключают адресацию или изменяют разрядность операндов.
- Используются для Legacy, REX, VEX или EVEX.

2. **Opcode (Код операции):**

- Основная часть инструкции. Задает тип операции (например, сложение, перемещение данных).
- Длина: 1, 2 или 3 байта.

3. **ModR/M:**

- Однобайтовый код, который определяет, какие регистры или адреса памяти участвуют в операции.
- Состоит из:
- **Mod** (2 бита): Тип адресации (регистровая или память).

- **Reg/Opcode** (3 бита): Используемый регистр или дополнительные данные о команде.
- **R/M** (3 бита): Адресуемый регистр или память.

4. **SIB (Scale-Index-Base):**

- Используется при сложной адресации (опционально).
- Состоит из:
- **Scale** (2 бита): Масштаб (умножение индекса на 1, 2, 4, 8).
- **Index** (3 бита): Индексный регистр.
- **Base** (3 бита): Базовый регистр.

5. **Displacement (Смещение):**

- Дополнительное смещение для вычисления адреса в памяти (1, 2 или 4 байта).

6. **Immediate (Немедленные данные):**

- Константа, непосредственно встроенная в инструкцию (1, 2 или 4 байта).

Пример:

MOV EAX, [EBX + 4]

- Префиксы: Нет.
- Opcode: MOV (однобайтовый).
- ModR/M: Указывает на регистр EAX и память.
- SIB: Указывает базовый регистр EBX и смещение 4.
- Displacement: 4 байта.
- Immediate: Нет.

MIPS (второе изображение)

Архитектура MIPS использует фиксированную длину инструкций (32 бита), что делает её схему более предсказуемой.

Типы инструкций:

1. **I-type (Immediate type):**

- Используется для операций с константами или адресами.
- Формат:

- **Opcode (6 бит)**: Код операции.
- **rs (5 бит)**: Регистр источника.
- **rt (5 бит)**: Регистр назначения.
- **Immediate (16 бит)**: Константа или смещение.

2. **R-type (Register type):**

- Используется для операций между регистрами.
- Формат:
- **Opcode (6 бит)**: Обычно 0 (указывает на R-type инструкцию).
- **rs (5 бит)**: Первый регистр источника.
- **rt (5 бит)**: Второй регистр источника.
- **rd (5 бит)**: Регистр назначения.
- **shamt (5 бит)**: Сдвиг (для операций сдвига).
- **funct (6 бит)**: Функция (определяет конкретную операцию).

3. **J-type (Jump type):**

- Используется для переходов (jump).
- Формат:
- **Opcode (6 бит)**: Код перехода.
- **Offset (26 бит)**: Смещение, добавляемое к PC.

Пример (R-type):

ADD $t1, $t2, $t3

- Opcode: 0 (R-type).
- rs: $t2 (первый источник).
- rt: $t3 (второй источник).
- rd: $t1 (результат).
- shamt: 0 (нет сдвига).
- funct: 32 (код операции ADD).

❓ 6. Типы инструкций. Примеры

- арифметико-логические: ADD, SUB, AND, OR, XOR, MUL, DIV;
- для работы с плавающей запятой: ADD.D, SUB.S, MUL.D;
- строковые: REP MOVSB (x86);
- управление потоком исполнения: JR;
- передача данных:
  - загрузки и сохранения данных из/в память;
  - загрузки констант в регистры;
  - безусловные инструкции пересылки между регистрами процессора;
  - условные инструкции пересылки.
- мультимедийные (SIMD): ADD.PS, SUB.PS, MUL.PS;
- для работы с графикой: операции для работы с пикселями и вертексами, операции сжатия/восстановления.
- системные: вызовы ОС, для работы с виртуальной памятью;

❓ 8. Регистровый файл. Разновидности регистрового файла (дополнение)

Динамический оконный регистровый файл или регистровый стек

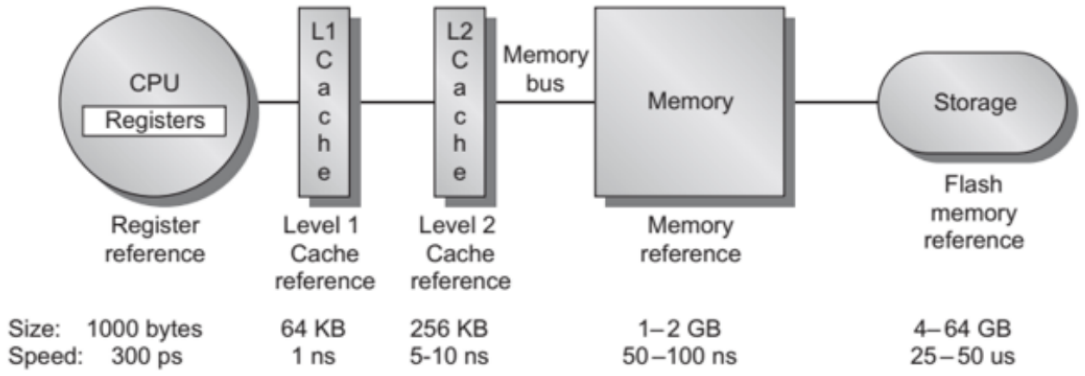
- используется в архитектуре IA-64;
- количество глобальных регистров: 32, стековых регистров — 96;
- регистровое окно произвольного размера;
- автоматическое сохранение/загрузка регистров в случае переполнения регистрового файла;
- количества локальных регистров и регистров входных/выходных параметров произвольно;
- достоинства:
  - ПО не нужно заботься о сохранении/загрузки регистров при вызове процедур;

- эффективное использование регистров из-за изменяемого размера окна.

- недостатки:
  - высокая сложность аппаратной реализации автоматического сохранения/загрузки.

❓ 9. Организация памяти. Устройство управления памятью (MMU)

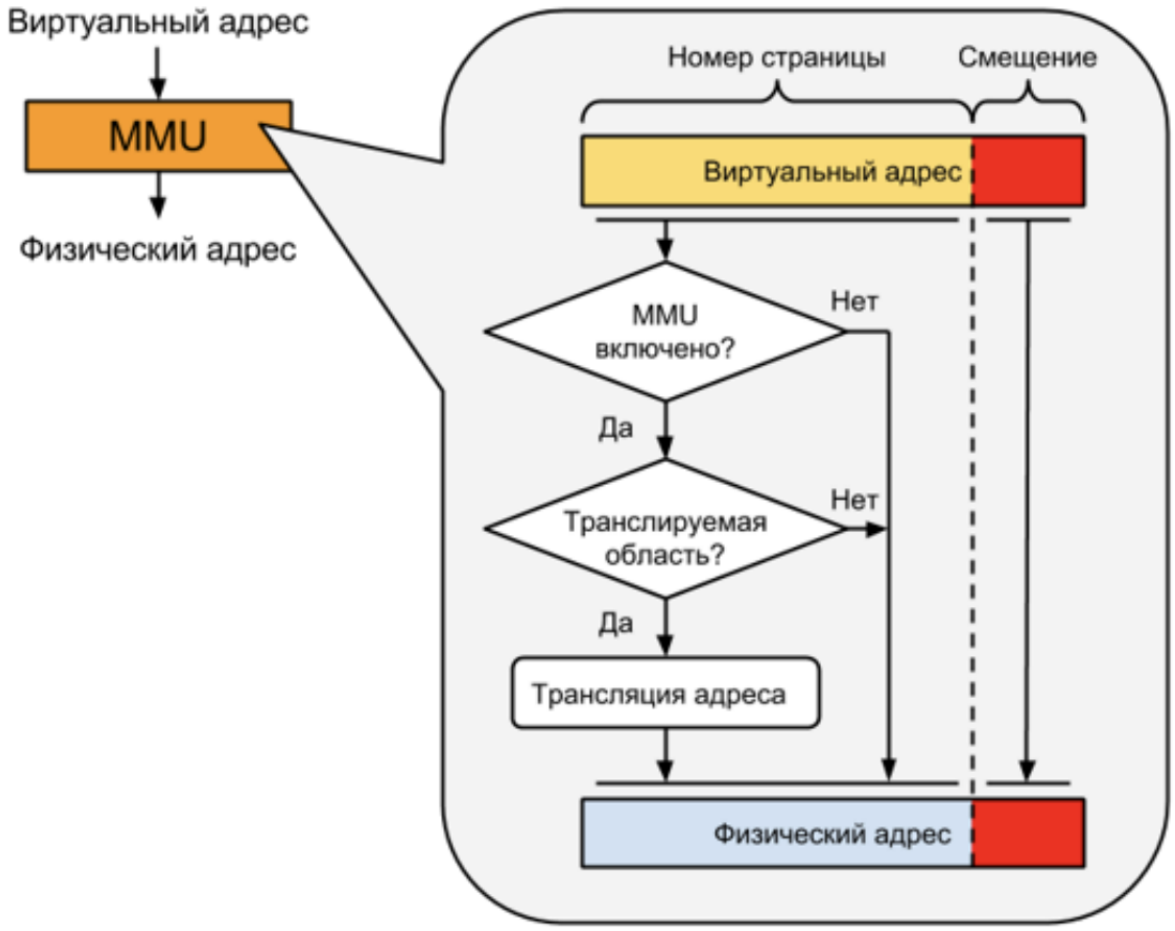
Организация памяти основана на временной (в ближайшее время произойдет еще одно обращение туда же) и пространственной локальности (следующее обращение произойдет по соседнему адресу).



Иерархия памяти

- Обмен данными происходит поблочно и только с соседними уровнями иерархии.
- Если на уровень пришел запрос, данных для которого нет, то тогда запрос передается на нижний уровень.

MMU — это аппаратный компонент процессора, отвечающий за управление доступом к памяти. Оно выполняет преобразование виртуальных адресов в физические, обеспечивает защиту памяти (проверка прав доступа) и поддержку виртуальной памяти



❓ 12. Кэш: классификация. Пространственная и временная локальность

По уровню:

- L1 (Level 1):
  - самый быстрый и маленький (обычно 32–128 КБ);
  - расположен внутри ядра процессора;
  - разделяется на кэш данных и кэш инструкций (раздельные кэши).
- L2 (Level 2):
  - больше по объёму, но медленнее (обычно 256 КБ – 8 МБ);

    - может быть общим для всех ядер или индивидуальным для каждого ядра.
- L3 (Level 3):
    - еще больше и медленнее (до нескольких десятков МБ);
    - обычно общий для всех ядер процессора.
- L4 (Level 4):
    - редко встречается;
    - может быть организован как часть оперативной памяти или отдельный модуль.

По методу размещения данных: (см. вопрос №13)

- с прямым отображением (direct-mapped);
- полностью ассоциативный (fully associative);
- с множественно-ассоциативным отображением (k-way set associative).

По способу записи данных: (см. вопрос №11)

- write-through;
- write-back.

По взаимодействию уровней:

- инклюзивный: все данные, которые находятся в кэше более высокого уровня, обязательно содержатся и в кэше более низкого уровня;
- эксклюзивный: данные хранятся только на одном уровне кэша.

Временная локальность: часто используемые данные хранятся в кэше до тех пор, пока не потребуется замена.

Пространственная локальность: вместо загрузки только одного элемента данных кэш загружает целую строку данных, чтобы захватить соседние элементы.

❓ 15. Кэш: способы оптимизации ПО, prefetching

1. Предвыборка данных (Prefetching) — это техника загрузки данных в кэш заранее, до того как процессору понадобятся эти данные. Это позволяет уменьшить задержки, связанные с кэш-промахами (cache misses)

- аппаратная предвыборка (Hardware Prefetching): процессор автоматически анализирует паттерны доступа к памяти (например, линейный обход массива);
- программная предвыборка (Software Prefetching): программист вручную добавляет инструкции для предвыборки данных.

2. Оптимизация циклов:

- перестановка циклов (Loop Interchange);
- разбиение на блоки (Loop Blocking): делит цикл на небольшие блоки, чтобы данные одного блока помещались в кэш.

3. Конвейеризация доступа — это разделение операций работы с кэшем на несколько этапов, выполняющихся параллельно. Она увеличивает пропускную способность и снижает время ожидания.

4. Использование write-back кэша.

5. Многоуровневый кэш минимизирует задержки при кэш-промахах.

6. Не дожидаться загрузки всей кэш-линии, передавать данные по мере попадания в кэш.

❓ 16. Векторные архитектуры. Примеры

Векторные архитектуры — это архитектуры вычислительных систем, оптимизированные для выполнения операций над массивами данных (векторами).

Ключевые особенности векторных архитектур:

- векторные регистры: хранят сразу несколько элементов данных (например, массив чисел);
- векторные инструкции: операции выполняются над всеми элементами вектора одновременно;

- конвейеризация: ускоряет выполнение векторных операций за счёт параллельной обработки нескольких этапов вычислений.

Примеры векторных архитектур:

- NEC SX Series — японская серия суперкомпьютеров с мощной векторной архитектурой:
  - длина векторных регистров доходила до 256 элементов;
  - использовались в научных и инженерных вычислениях.
- Vector Facility в IBM System/370 — добавление векторных инструкций к архитектуре IBM System/370:
  - использовались в финансовых и научных вычислениях.

❓ 24. Механизмы ускорения выборки инструкций (внеочередное исполнение инструкций, переименование регистров, технологии микро- и макро-fusion)

1. Внеочередное исполнение инструкций (Out-of-Order Execution) — техника, при которой процессор может выполнять инструкции в любом порядке, а не строго в порядке их поступления (по коду программы), при условии, что не нарушаются зависимости данных.

Принцип работы:

- анализ зависимостей: перед выполнением каждой инструкции процессор проверяет, от каких данных она зависит. Если данные уже доступны, инструкция может быть выполнена сразу.
- станции резервирования: инструкции, которые не могут быть выполнены сразу, помещаются в буфер ожидания (резервирования) до тех пор, пока необходимые данные не станут доступны.

2. Переименование регистров (Register Renaming) — устранение архитектурного конфликта — использования одного регистра несколькими инструкциями.

Принцип работы: архитектурные регистры, упомянутые в коде, переименовываются в физические регистры процессора.Каждая

инструкция получает свой уникальный физический регистр для хранения результата.

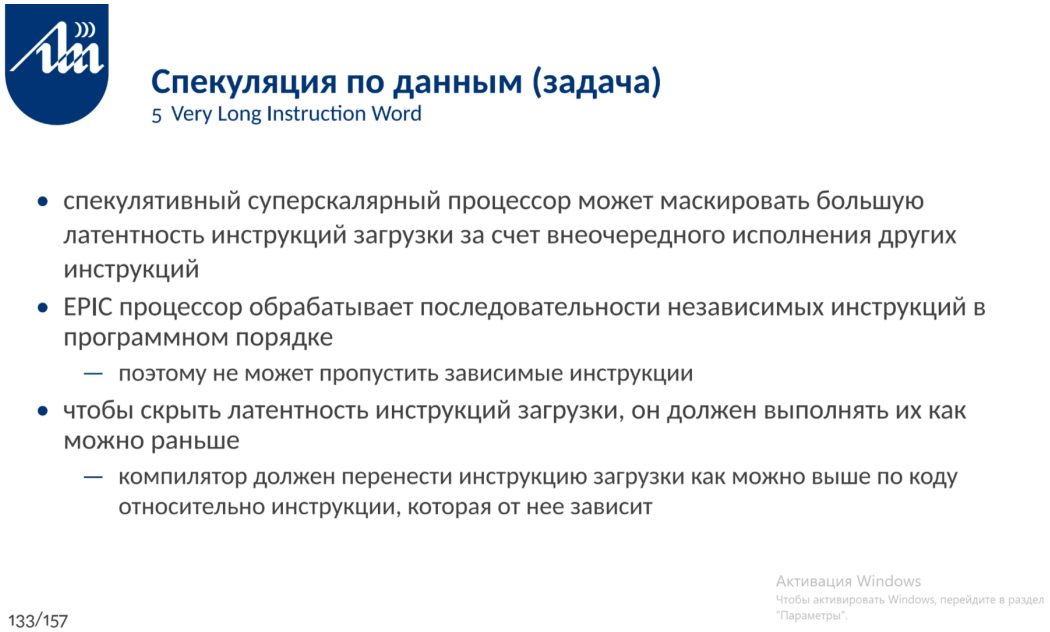
3. Технология Микро-Fusion (Micro-Fusion) — техника, используемая на этапе декодирования инструкций. Некоторые сложные инструкции (например, MOV и ADD) могут быть представлены как одна микрокоманда внутри процессора.

Пример: инструкция ADD [mem], imm (сложение с немедленным значением в памяти) декодируется как одна команда на этапе микро-Fusion.

4. Макро-Fusion (Macro-Fusion) — техника объединяет две соседние инструкции высокого уровня (макроинструкции) в одну. Обычно это применяется к инструкциям ветвления и сравнения.

Пример: CMP reg, imm (сравнение) + JE label (переход) объединяются в одну макроинструкцию.

? 29. EPIC. Механизмы поддержки спекуляции (дополнение)



❓ 32. Задание ядра, потоки и блоки потоков на примере перемножения двух матриц в CUDA

- Thread — наименьшая единица исполнения на GPU, выполняет одну инструкцию или вычислительную операцию.
- Block — группа потоков (threads), которые обрабатываются GPU в пределах одного блока. Потоки внутри одного блока могут обмениваться данными через локальную память.
- Grid — сетка, содержащая блоки, каждый из которых, в свою очередь, содержит потоки. Grid позволяет задать масштаб задачи для GPU.

Потоки одной сетки исполняют одну и ту же функцию (вычисление элемента результирующей матрицы), называемую kernel (аналог функции в C/C++).

Каждый поток в CUDA идентифицируется уникальным индексом, который определяется с использованием встроенных переменных:

- threadIdx — индекс потока в блоке;
- blockIdx — индекс блока в сетке;
- blockDim — количество потоков в одном блоке;
- gridDim — количество блоков в сетке.

```
// C[i][j] = Sum(k=0, k<K-1) (A[i][k] * B[k][j]).
// C[row * N + col] = Sum(k=0, k<K-1) A[row * K + k] * B[k * 
__global__ matrixMulKernel(const float *A, const float *B, fl
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if(row < M && col < N) {
        float sum = 0.0f;
        for(int k = 0; k < K; k++) {
            sum += A[row * K + k] * B[k * N + col];  // скаля
        }
        C[row * N + col] = sum;
```

```
    }
}

void matrixMul(const float *A, const float *B, float *C, int
    size_t size_a = M * K * sizeof(float);
    size_t size_b = M * K * sizeof(float);
    size_t size_c = M * N * sizeof(float);

    float *dev_a, *dev_b, *dev_c;

    cudaMalloc(&dev_a, size_a);
    cudaMalloc(&dev_b, size_b);
    cudaMalloc(&dev_c, size_c);

    cudaMemcpy(dev_a, A, size_a, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, B, size_b, cudaMemcpyHostToDevice);

    dim3 blockDim(16, 16);
    dim3 gridDim((N + blockDim.x - 1) / blockDim.x, (M + bloc

    matrixMulKernel<<<gridDim, blockDim>>>(dev_a, dev_b, dev_

    cudaMemcpy(C, dev_c, size_c, cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b)l
    cudaFree(dev_c);
}
```

❓ 34. Синхронизация потоков, дивергенция потоков, функции голосования в CUDA. Примеры

Синхронизация потоков используется для обеспечения того, чтобы все потоки в блоке достигли определенной точки перед продолжением

выполнения. Это полезно, например, при совместном использовании памяти между потоками.

```
__global__ void reduceSum(float *input, float *result, int n)
    __shared__ float sharedData[256];

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

    sharedData[tid] = (idx < n) ? input[idx] : 0.0f;
    __syncthreads();  // синхронизация потоков

    // суммирование с использованием дерева редукции
    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            sharedData[tid] += sharedData[tid + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        result[blockIdx.x] = sharedData[0];
    }
}
```

Дивергенция потоков возникает, когда потоки в одном векторе выполнения (warp, обычно 32 потока) исполняют разные ветви программы из-за условных операторов.

```
__global__ void computeConditionally(int *data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n) {
        if (data[idx] % 2 == 0) {
            data[idx] *= 2;
        } else {
            data[idx] += 1;
        }
```

```
    }
}
```

Функции голосования позволяют потокам в векторе выполнения координироваться, проверяя общие условия. Примеры таких функций:

- `__all(condition)` — возвращает true, если все потоки в векторе выполнения удовлетворяют condition;
- `__any(condition)` — возвращает true, если хотя бы один поток удовлетворяет condition;
- `__ballot_sync(mask, condition)` — возвращает битовую маску потоков, удовлетворяющих condition.

```
__global__ void voteExample(int *data, int *result, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    bool condition = (idx < n && data[idx] > 0);

    // все ли потоки имеют положительные значения
    bool allPositive = __all_sync(0xFFFFFFFF, condition);

    // есть ли хотя бы одно положительное значение
    bool anyPositive = __any_sync(0xFFFFFFFF, condition);

    if (threadIdx.x == 0) {
        result[0] = allPositive;
        result[1] = anyPositive;
    }
}
```

❓ 36. Понятие occupancy в CUDA. Пример расчета

Occupancy (заполняемость) — это отношение количества активных потоков к максимально возможному числу потоков на одном многоядерном процессоре GPU.

Входные данные:

- максимальное количество потоков на один мультипроцессор (SM): 2048;
- максимальное количество блоков на SM: 32;
- максимальное количество регистров на SM: 65536;
- максимальный объем shared memory на SM: 48 KB;

- размер блока: 128 потоков;
- кол-во регистров на поток: 32;
- объем shared memory на блок: 1 KB.

Расчет:

1. Максимальное число блоков по потокам: 2048 / 128 = 16.
2. Максимальное число блоков по регистрам: 65536 / (128 * 32) = 16.
3. Максимальное число блоков по shared memory: 48 / 1 = 48.
4. Максимальное число блоков на SM: min(16, 16, 48, 32) = 16.
5. Число активных потоков: максимальное число блоков на SM * размер блока = 16 * 128 = 4096.
6. Occupancy = число активных потоков / максимальное количество потоков = 1 (100%).

❓ 37. Типы памяти в CUDA. Примеры создания и организации доступа

Тип памяти	Область видимости	Доступ	Скорость	Объем
глобальная	все потоки	RW	низкая	очень большая
разделяемая	блок	RW	высокая	ограничен
константная	все потоки	R	высокая (кэш)	64 Кб
регистровая	поток	RW	самая быстрая	ограничен

локальная	поток	RW	низкая	ограничен

Глобальная память

Доступ к памяти требует коалесцирования (оптимального размещения данных).

```
__global__ void globalMemoryExample(float *input, float *outp
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        output[idx] = input[idx] * 2.0f;
    }
}
```

Разделяемая память

Требует синхронизации потоков при совместном доступе.

```
__global__ void reduceSum(float *input, float *result, int n)
    __shared__ float sharedData[256];

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

    sharedData[tid] = (idx < n) ? input[idx] : 0.0f;
    __syncthreads();  // синхронизация потоков

    // суммирование с использованием дерева редукции
    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            sharedData[tid] += sharedData[tid + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        result[blockIdx.x] = sharedData[0];
    }
}
```

Константная память

```
__constant__ float constData[256];

__global__ void constantMemoryExample(float *output, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        output[idx] = constData[threadIdx.x];
    }
}

float h_constData[256];
cudaMemcpyToSymbol(constData, h_constData, sizeof(h_constData
```

Регистровая память

Локальные переменные ядра, не объявленные как массивы или указатели, хранятся в регистрах.

```
__global__ void addKernel(float *a, float *b, float *c, int n
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        float temp = a[idx] + b[idx];  // temp хранится в рег
        c[idx] = temp;
    }
}
```

Локальная память

```
__global__ void localMemoryExample() {
    int localArray[100];  // массив хранится в локальной памя
    localArray[threadIdx.x] = threadIdx.x * 2;
}
```

❓ 38. Механизм транзакций в CUDA. Пример

В CUDA транзакции относятся к операциям чтения или записи данных между потоками и памятью устройства. Когда потоки обращаются к памяти, GPU группирует эти обращения в транзакции для повышения производительности.

Типы транзакций:

1. Транзакции с глобальной памятью: память устройства организована в 32-, 64- или 128-байтные строки (memory lines). Потоки в векторе выполнения группируют свои обращения, если они обращаются к последовательным адресам.

   Коалесцированный доступ возникает, когда потоки читают или записывают данные, выровненные по размерам строки памяти. Это снижает число транзакций.

2. Транзакции с общей памятью: общая память разбита на банки (обычно 32 банка). Если потоки обращаются к разным банкам или к одному адресу (broadcast), конфликта нет.

```
__global__ void bankConflictExample(float *input, float *outp
    __shared__ float sharedData[32]; // 32 банка

    int tid = threadIdx.x;

    // Конфликт банков: все потоки читают из разных адресов о
    sharedData[tid] = input[tid];
    __syncthreads();

    output[tid] = sharedData[(tid + 1) % 32];
}

__global__ void noBankConflictExample(float *input, float *ou
    __shared__ float sharedData[32];

    int tid = threadIdx.x;

    // Устранение конфликта: добавляем смещение
    sharedData[tid + tid / 32] = input[tid];
    __syncthreads();
```

```
    output[tid] = sharedData[(tid + 1) % 32 + tid / 32];
}
```

❓ 40. Алгоритм редукции в CUDA. Пример

Редукция — это операция свертки, которая преобразует множество элементов в одно значение с использованием коммутативной и ассоциативной операции, например, суммы, минимума, максимума и т.д.

В наивной редукции каждый поток идет с заданным шагом.

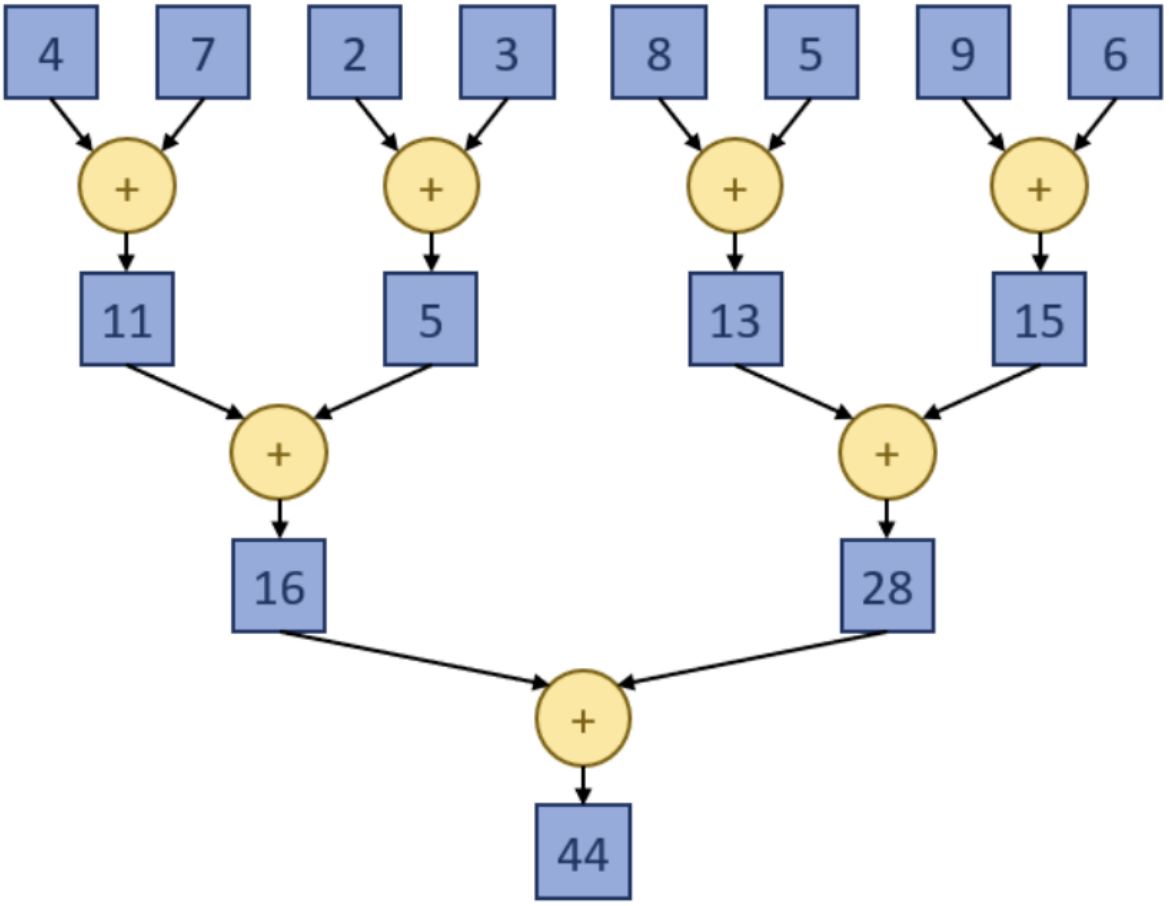
```
__global__ void reduceSum(float *input, float *result, int n)
    __shared__ float sharedData[256];

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

    sharedData[tid] = (idx < n) ? input[idx] : 0.0f;
    __syncthreads();  // синхронизация потоков

    // суммирование с использованием дерева редукции
    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            sharedData[tid] += sharedData[tid + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        result[blockIdx.x] = sharedData[0];
    }
}
```



❓ 41. Алгоритм свертки в CUDA. Пример

Свертка — это операция обработки сигналов или изображений, часто используемая в фильтрации изображений, обработке сигналов и сверточных нейронных сетях (CNN).

Каждый выходной элемент — это взвешенная сумма соседних входных элементов.

```
#define FILTER_DIM 3
#define FILTER_RADIUS ((FILTER_DIM - 1) / 2)

__global__ void convolution_kernel(float *input, float *outpu
                                   int width, int height, flo
    int outRow = blockIdx.y * blockDim.y + threadIdx.y;
```

```
    int outCol = blockIdx.x * blockDim.x + threadIdx.x;

    if (outRow < height && outCol < width) {
        float sum = 0.0f;

        // Применяем фильтр
        for (int filterRow = 0; filterRow < FILTER_DIM; ++fil
            for (int filterCol = 0; filterCol < FILTER_DIM; +
                int inRow = outRow - FILTER_RADIUS + filterRo
                int inCol = outCol - FILTER_RADIUS + filterCo

                // Проверка границ
                if (inRow >= 0 && inRow < height && inCol >=
                    sum += filter[filterRow][filterCol] * inp
                }
            }
        }

        output[outRow * width + outCol] = sum;
    }
}
```

❓ 42. Алгоритм операции инклюзивного scan в CUDA. Пример

Операция инклюзивного scan (или префиксная сумма) принимает массив и выполняет операцию (например, сложение) на всех предыдущих элементах массива для каждого элемента.

```
__global__ void inclusive_scan(int *data, int N) {
    int idx = threadIdx.x;

    if (idx > 0) {
        data[idx] += data[idx - 1];  // каждый элемент добавл
    }
}
```

❓ 43. Алгоритм операции эксклюзивного scan в CUDA. Пример

Эксклюзивный scan (или эксклюзивная префиксная сумма) — это операция, которая вычисляет сумму всех предыдущих элементов массива для каждого элемента, но не включает сам текущий элемент в расчет.

```
__global__ void exclusive_scan(int *data, int N) {
    int idx = threadIdx.x;

    if (idx > 0) {
        data[idx] += data[idx - 1];
    }
    __syncthreads();

    if (idx == 0) {
        data[idx] = 0;  // первый элемент делаем нулевым
    } else if (idx < N) {
        data[idx] -= data[idx - 1];  // убираем текущее значе
    }
}
```

❓ 44. Асинхронное и синхронное копирование в CUDA. Pinned память. Способы выделения

Pinned memory — это специальная память, закрепленная в физической оперативной памяти. Она позволяет устройству (GPU) напрямую (DMA) обращаться к данным хоста, что ускоряет передачу данных.

```
float* hostData;
cudaHostAlloc((void**)&hostData, size * sizeof(float), cudaHo
```

Способы выделения pinned memory:

- `int *h_pinned_data;`
  `cudaHostAlloc(&h_pinned_data, N * sizeof(int), cudaHostAllocDefault);`
- `int *h_pinned_data;`
  `cudaMallocHost(&h_pinned_data, N * sizeof(int));`

❓ 45. CUDA Stream. Создание, инициализация и синхронизация

CUDA Stream — последовательность команд для GPU (запуски ядер, копирования памяти и т.д.), исполняемая строго последовательно.

Только команды из разных потоков, отличных от потока по-умолчанию, могут выполняться параллельно.

```
#include <iostream>
#include <cuda_runtime.h>

__global__ void kernel(int *data, int value) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    data[idx] += value;
}

int main() {
    const int N = 256;
    int *h_data, *d_data;

    cudaMallocHost(&h_data, N * sizeof(int));
    cudaMalloc(&d_data, N * sizeof(int));

    for (int i = 0; i < N; ++i) h_data[i] = i;

    // создание и инициализация потока
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    kernel<<<N / 64, 64, 0, stream>>>(d_data, 10);
```

```
    // асинхронное копирование данных обратно на хост
    cudaMemcpyAsync(h_data, d_data, N * sizeof(int), cudaMemc

    // синхронизация потока
    cudaStreamSynchronize(stream);

    for (int i = 0; i < 10; ++i) {
        std::cout << h_data[i] << " ";
    }
    std::cout << std::endl;

    cudaFree(d_data);
    cudaFreeHost(h_data);
    cudaStreamDestroy(stream);

    return 0;
}
```