Билет 40

Методы эффективной организации параллельных вычислений на графических процессорах

1) Утилизация латентности памяти

- *Цель*: эффективно загружать Ядра
- *Проблема*: латентность памяти
- *Решение*: CPU: Сложная иерархия кешей

GPU: Много нитей, покрывать обращения одних нитей в память вычислениями в других за счёт быстрого переключения контекста. За счёт наличия сотен ядер и поддержки миллионов нитей (потребителей) на GPU легче утилизировать всю полосу пропускания

2) SIMT (Single instruction, multiple thread) и масштабирование

Виртуальная

- GPU может поддерживать миллионы виртуальных нитей
- Виртуальные блоки независимы (программу можно запустить на любом количестве SM)

Аппаратная

- Мультипроцессоры независимы
- Можно «нарезать» GPU с различным количеством SM

3) Асинхронность в CUDA

Чтобы GPU больше времени работало в фоновом режиме, параллельно с CPU, некоторые вызовы являются асинхронными. Отправляют команду на устройство и сразу возвращают управление хосту. К таким вызовам относятся:

- Запуски ядер (если CUDA_LAUNCH_BLOCKING не установлена на 1)
- Копирование между двумя областями памяти на устройстве
- Копирование с хоста на устройство менее 64KB
- Копирования, выполняемые функциями с окончанием *Async
- cudaMemSet – присваивает всем байтам области памяти на устройстве одинаковое значение (чаще всего используется для обнуления)

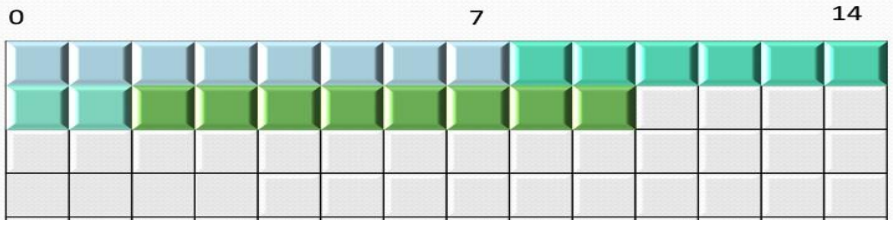
4) Работа с глобальной памятью

- Расположена в DRAM GPU
- Объем до 4Gb
    - Параметр устройства totalGlobalMem
- Кешируется в кешах L1 и L2
    - L1 – на каждом мультипроцессоре (максимальный размер – 48KB, минимальный – 16KB)
    - L2 – на устройстве (максимальный размер - 768 KB, параметр устройства l2CacheSize

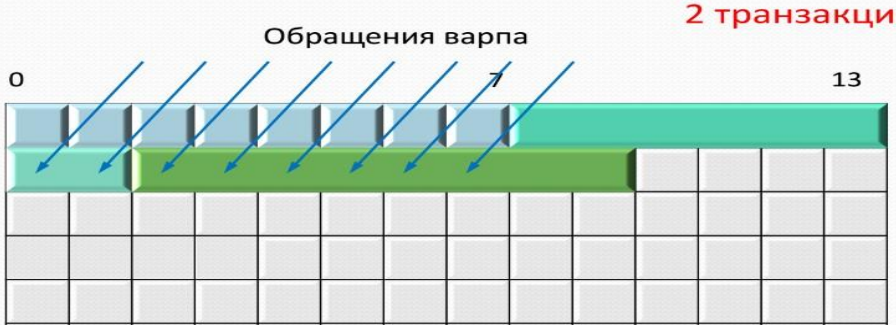
Общие принципы эффективной работы:

- Обращения нитей варпа в память должны быть пространственно-локальными
- Начала строк матрицы должны быть выровнены

Пусть транзакция – 8*4=32 байта, адрес транзакции выровнен по 32 байта
Если ширина матрицы не кратна 32 байтам – большая часть строк не выровнена

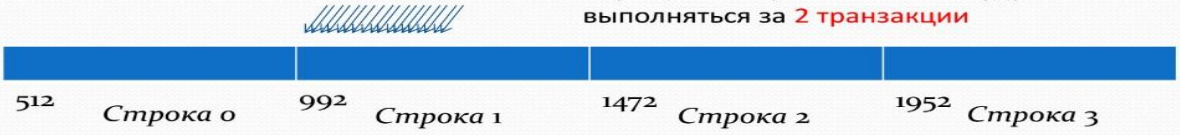


2 транзакции!



- Матрицы хранятся в линейном виде, по строкам
- Пусть длина строки матрицы – 480 байт (120 float)
  - обращение – matrix[idy*120 + idx]

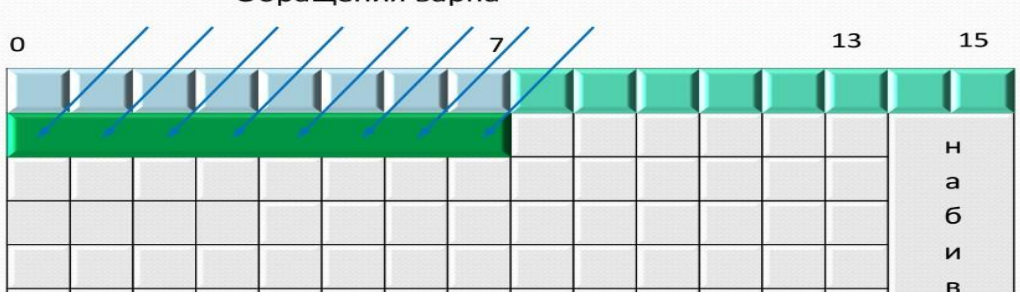
Адрес начала каждой строки, кроме первой, не выровнен по 128 – обращения варпов в память будут выполняться за 2 транзакции



- Дополним каждую строку до размера, кратного 128 байтам – в нашем случае, 480 + 32 = 512, это наш pitch – фактическая ширина в байтах
- Эти байты никак не будут использоваться, т.е. 32/512=6% лишней памяти будет выделено (Но для больших матриц эта доля будет существенно меньше)
- Зато каждая строка будет выровнена по 128 байт
  - Обращение matrix[idy*128+ idx]

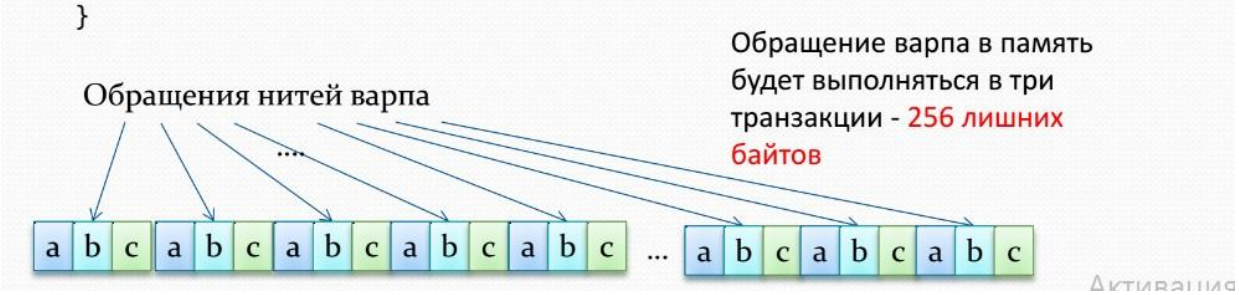
Адрес начала каждой строки выровнен по 128! Обращения варпов в память выполняются за одну транзакцию

1 транзакция!

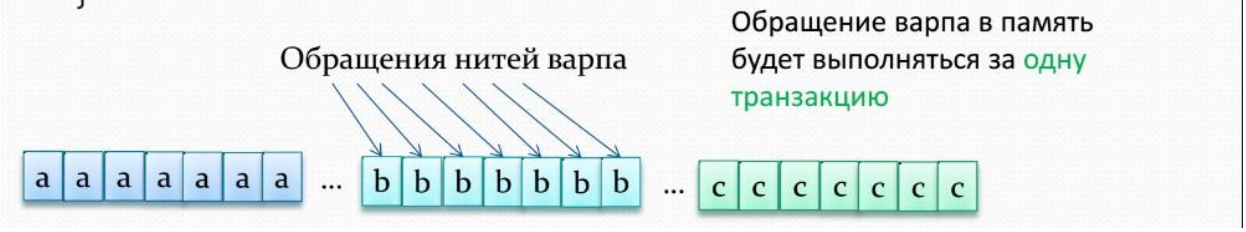


- Использование массивов структур

```
struct example {
   int a;
   int b;
   int c;
}
__global__ void kernel(example * arrayOfExamples) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    arrayOfExamples[idx].c =
         arrayOfExamples[idx].b + arrayOfExamples[idx].a;
}
```




```
struct example {
   int *a;
   int *b;
   int *c;
}
__global__ void kernel(example arrayOfExamples) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    arrayOfExamples.c[idx] =
         arrayOfExamples.b[idx] + arrayOfExamples.a[idx];
}
```



- Выбор режима Кеша L1

Кеш может работать в двух режимах: 16 KB и 48KB. Возможные режимы

  - cudaFuncCachePreferNone – без предпочтений (по умолчанию). Выбирается последняя использованная конфигурация. Начальная конфигурация – 16KB L1
  - cudaFuncCachePreferShared - 16KB L1
  - cudaFuncCachePreferL1 - 48KB L1

Если в ядре не используется общая память, то заведомо стоит включить cudaFuncCachePreferL1.

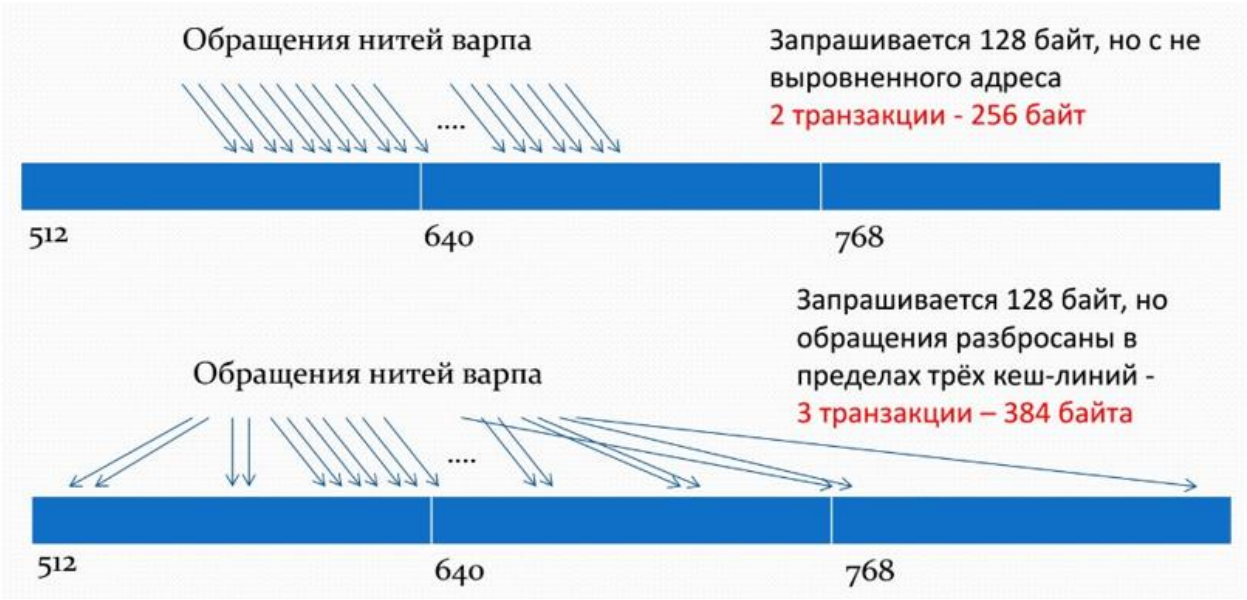
- Избегать косвенной адресации
  - Использование косвенной адресации нежелательно, поскольку требует двух чтений из памяти (сначала A[i], потом A[i][j])
  - A[i] для разных нитей варпа скорее всего будут в одной кеш- линии → одно обращение к кешу
  - Но A[i][j] в общем случае могут быть разбросаны

    - Принципиального решения нет! Скорее всего придется переработать алгоритм.
- Избегать обращений нитей варпа к столбцу матрицы

Матрица расположена по строкам, а обращение происходит по столбцам

Решение – хранить матрицу в транспонированном виде! (В этом случае обращения по столбцам превратятся в обращения к последовательным адресам, а выделять память под транспонированную матрицу также через cudaMallocPitch)

- В случае сильно разряженного доступа проверять работу с отключенным кэшем
    - Транзакция – выполнение загрузки из глобальной памяти сплошного отрезка в 128 байт, с началом кратным 128
    - Инструкция обращения в память выполняется одновременно для всех нитей варпа
        - Выполняется столько транзакций, сколько нужно для покрытия обращений всех нитей варпа
        - Если нужен 1 байт – все равно загрузится 128.



    - Ядра взаимодействуют не с памятью напрямую, а с кешами
    - Транзакция – выполнение загрузки кеш – линии
        - У кеша L1 кеш – линии 128 байт, у L2 – 32 байта
        - Кеш грузит из памяти всю кеш линию, даже если нужен один байт
    - Можно обращаться в память минуя L1
        - Транзакции будут по 32 байта