```
__constant__ float contsData [256]; // константная память GPU
float              hostData  [256]; // данные в памяти CPU
...
// Скопировать данные из памяти CPU в константную память GPU
cudaMemcpyToSymbol(constData, hostData,
    sizeof(data), 0, cudaMemcpyHostToDevice);
```

Поскольку константная память кэшируется, то она подходит для размещения небольшого объема часто используемых неизменяемых данных, которые должны быть доступны всем нитям. Дополнительно в устройствах с compute capability 2.x доступно аналогичное константной памяти кэширование произвольного участка глобальной памяти (инструкция LDU или LoaD Uniform). Данный режим будет автоматически активирован при запросе только для чтения по адресу памяти, не зависящему от индекса нити.

```
__global__ void kernel( float *g_dst, const float *g_src )
{
    g_dst = g_src[0];            // не зависит от индекса нити -> uniform load
    g_dst = g_src[blockIdx.x];   // не зависит от индекса нити -> uniform load
    g_dst = g_src[threadIdx.x];  // зависит от индекса -> non-uniform
}
```

## 3.2. Глобальная память

Основную часть DRAM GPU занимает глобальная память. При корректной работе ядер динамически выделенная глобальная память сохраняет целостность данных на протяжении всего времени жизни приложения, что, в частности, позволяет использовать ее как основное хранилище для передачи данных между несколькими ядрами.

Глобальная память может быть выделена как статически, так и динамически. Динамическое выделение и освобождение глобальной памяти в коде хоста производится при помощи следующих вызовов:

```
cudaError_t cudaMalloc(void ** devPtr, size_t size);

cudaError_t cudaFree(void * devPtr);

cudaError_t cudaMallocPitch(void ** devPtr, size_t * pitch,
    size_t width, size_t height);
```

С появлением device-функций *malloc* и *free* в CUDA 3.2, на GPU с compute capability 2.0 и выше динамическое выделение и освобождение глобальной памяти возможно не только в хост-коде но и в коде CUDA-ядра. Однако при этом динамическое выделение происходит лишь относительно нитей ядра, тогда как общий пул памяти под эти аллокации выделяется заранее. Размер пула может быть установлен вызовом функции *cudaLimitMallocHeapSize* или по умолчанию равен 8 Мб. В следующем примере два CUDA-ядра выделяют и освобождают глобальную память GPU:

```
#include <stdio.h>
#include <stdlib.h>

__global__ void mass_malloc(void** ptrs, size_t size)
{
  ptrs[threadIdx.x] = malloc(size);
}

__global__ void mass_free(void** ptrs)
{
  free(ptrs[threadIdx.x]);
}

int main(int argc, char* argv[])
{
  if (argc != 3)
  {
    printf("Usage: %s <nthreads> <size>\n", argv[0]);
    return 0;
  }

  int nthreads = atoi(argv[1]);
  size_t size = atoi(argv[2]);

  // Установить размер пула. Это действие должно быть
  // выполнено до запуска каких-либо ядер.
  cudaDeviceSetLimit(cudaLimitMallocHeapSize, 16 * 1024 * 1024);

  void** ptrs; cudaMallocHost(&ptrs, sizeof(void*) * nthreads);
  memset(ptrs, 0, sizeof(void*) * nthreads);
  mass_malloc<<<1, nthreads>>>(ptrs, size);
  cudaDeviceSynchronize();
  for (int i = 0; i < nthreads; i++)
    printf("Thread %d got pointer: %p\n", i, ptrs[i]);
```

```
  mass_free<<<1, nthreads>>>(ptrs);
  cudaDeviceSynchronize();
  cudaFreeHost(ptrs);

  return 0;
}
```

```
$ ./device_malloc_test 8 24
Thread 0 got pointer: 0x2013ffb20
Thread 1 got pointer: 0x2013ffb70
Thread 2 got pointer: 0x2013ffbc0
Thread 3 got pointer: 0x2013ffc10
Thread 4 got pointer: 0x2013ffc60
Thread 5 got pointer: 0x2013ffcb0
Thread 6 got pointer: 0x2013ffd00
Thread 7 got pointer: 0x2013ffd50
```

Глобальная переменная может быть объявлена статически при помощи атрибута *__device__*. В этом случае память будет выделена при инициализации модуля с CUDA-ядрами и данными (в зависимости от способа загрузки, в момент старта приложения или при вызове *cuModuleLoad*). В следующем примере утилита *nm* показывает, что в объектном представлении *cubin* (двоичный файл CUDA-модуля), глобальные данные размещаются таким же образом как на CPU: "B" – неинициализированная переменная, "D" – инициализированная переменная, "R" – только для чтения. Отличие же состоит в том, что в cubin статические переменные не становятся приватными (соответствующие литеры в нижнем регистре) и не декорируются. Правила видимости не имеют смысла, поскольку в CUDA отсутствует линковка, и все CUDA-объекты должны быть самодостаточными, без внешних зависимостей.

```
__device__ float val1;
__device__ float val2 = 1.0f;
__device__ const float val3 = 2.0f;
__constant__ float val_pi = 3.14;
__device__ static float val4;

__global__ void kernel(float* val5)
{
    if (!threadIdx.x && !blockIdx.x)
    {
        val4 = sin(val1 + val2 * val3 + val_pi);
        *val5 = val4;
```

```
    }
}
```

```
$ nvcc -keep -c test.cu
$ nm test.sm_10.cubin | grep val
0000000000000004 B val1
0000000000000004 D val2
0000000000000000 D val3
0000000000000000 B val4
0000000000000018 R val_pi
```

```
float val1;
float val2 = 1.0f;
const float val3 = 2.0f;
float val_pi = 3.14;
static float val4;

#include <cmath>

void kernel(float* val5)
{
    val4 = sin(val1 + val2 * val3 + val_pi);
    *val5 = val4;
}
```

```
$ g++ -c test.c
$ nm test.o | grep val
0000000000000000 r _ZL4val3
0000000000000004 b _ZL4val4
0000000000000000 B val1
0000000000000000 D val2
0000000000000004 D val_pi
```

Доступ к многомерным массивам в глобальной памяти может быть более эффективен при наличии выравнивания строк. Выравнивание обеспечивается добавлением фиктивных элементов в конец каждой строки и соответствующих сдвигов при индексации. Функция *cudaMallocPitch* выделяет память с выравниванием строк и возвращает сдвиг через параметр *pitch*. Например, если выделена память для матрицы с элементами типа *T*, то для получения адреса элемента, расположенного в строке *row* и столбце *col*, используется следующая формула:

```
T * item = (T *) ((char *) baseAddress + row * pitch) + col;
```

Хотя функция *cudaMalloc* возвращает обычный указатель, его значение имеет смысл только для адресного пространства GPU. Для заполнения памяти GPU данными хоста и наоборот, необходимо использовать специальные функции копирования:

```
cudaError_t cudaMemcpy (
    void * dst, const void * src, size_t size,
    enum cudaMemcpyKind kind );

cudaError_t cudaMemcpyAsync (
    void * dst, const void * src, size_t size,
    enum cudaMemcpyKind kind, cudaStream_t stream );

cudaError_t cudaMemcpy2D (
    void * dst, size_t dpitch, const void * src, size_t spitch,
    size_t width, size_t height, enum cudaMemcpyKind kind );

cudaError_t cudaMemcpy2DAsync (
    void * dst, size_t dpitch, const void * src, size_t spitch,
    size_t width, size_t height, enum cudaMemcpyKind kind,
    cudaStream_t stream );
```

Копирование данных в глобальной памяти внутри одного GPU обычно производится на порядок быстрее, чем внутри памяти хоста, например, для GPU Tesla C2050 характерная скорость – 144 Гбайт/сек. Копирование данных между памятью хоста и памятью GPU значительно медленнее. Наиболее распространенный в данный момент стандарт интерфейса PCI Express 2.0 способен обеспечить пропускную способность до 8 Гбайт/сек. С учетом потерь на кодирование, латентности и задержки, на практике, как правило, удается добиться не более 4 Гбайт/сек при копировании между хостом и одним GPU и не более 6 Гбайт/сек – при копировании между хостом и несколькими GPU одновременно.

Наилучшая скорость копирования данных между хостом и GPU может быть достигнута при использовании *pinned-памяти*. Pinned-память – это память *хоста*, которая может быть либо выделена функциями *cudaHostAlloc* или *cudaMallocHost*, заменяющими стандартный вызовы *malloc* или *new*, либо получена переводом обычной памяти в категорию pinned функцией *cudaHostRegister* (см. Unified Virtual Address Space).

```
cudaError_t cudaHostAlloc (void** pHost, size_t size, unsigned int flags);
cudaError_t cudaMallocHost(void** devPtr, size_t size);
```

```
cudaError_t cudaFreeHost(void* devPtr);

cudaError_t cudaHostRegister(void* ptr, size_t size, unsigned int flags);
cudaError_t cudaHostUnregister(void* ptr);
```

При асинхронном копировании памяти между CPU и GPU с помощью функции *cudaMemcpyAsync* в CUDA версии 4.0 должна использоваться только pinned-память. Начиная с CUDA версии 4.0 может использоваться обычная (не pinned) память, но в этом случае вызов *cudaMemcpyAsync* будет синхронным. Выделение большого количества pinned-памяти может отрицательно сказаться на быстродействии всей системы.

В CUDA используется *прямая адресация* глобальной памяти GPU, т.е., в отличие от OpenCL, для хранения адресов подходят обычные указатели, и для них корректна адресная арифметика. Если память выделяется с помощью *cudaMalloc*, то выделенный диапазон имеет смысл только в контексте GPU-ядра. Память, выделенная на одном GPU некорректна по отношению к другому GPU (см. контексты GPU). Если память выделяется *cudaHostAlloc(..., , cudaHostAllocMapped)*, то выделенный на CPU диапазон памяти становится доступен как для CPU, так и для всех GPU (см. Unified Virtual Address Space).

### 3.2.1. Кэширование

Для более эффективного использования глобальной памяти в GPU архитектуры Fermi (compute capability 2.0 и 2.1) реализованы кэши первого и второго уровня (Рис. 3.1).

Кэш первого уровня (L1) находится на каждом мультипроцессоре, тогда как кэш второго уровня (L2) – общий и имеет размер 768 Кбайт. Кэш L1 и разделяемая память расположены на одном физическом носителе, объем которого может быть разделен между ними одним из двух способов: 48 Кбайт разделяемой памяти и 16 Кбайт L1-кэша или 16 Кбайт разделяемой памяти и 48 Кбайт L1-кэша. Переключение производится функцией *cudaFuncSetCacheConfig*:

```
// Device code
__global__ void MyKernel()
{
    ...
}
```

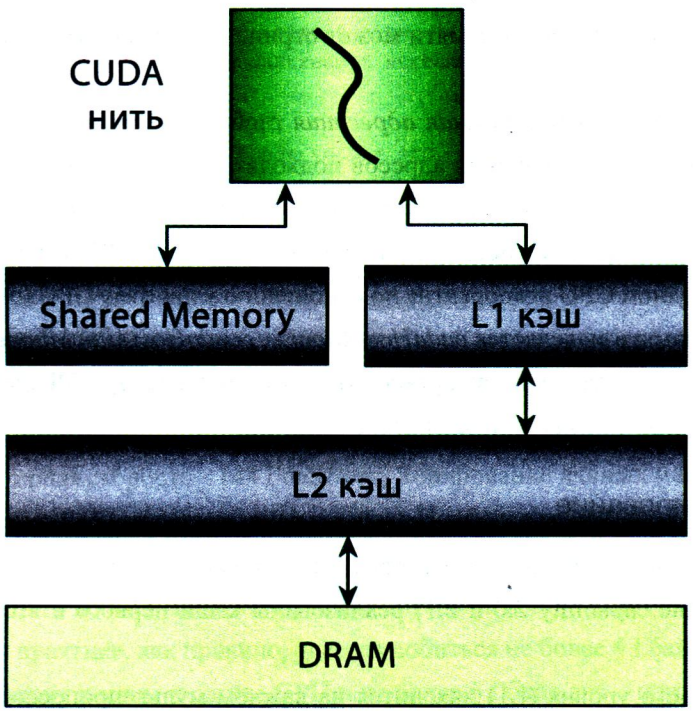


*Рис. 3.1.* Fermi – подсистема памяти

```
// Host code
// cudaFuncCachePreferShared: 48 KB разделяемой памяти
// cudaFuncCachePreferL1: 16 KB разделяемой памяти
// cudaFuncCachePreferNone: без предпочтения, использовать текущий контекст
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared);
```

По умолчанию используется конфигурация «без предпочтения». В этом режиме будет использоваться конфигурация текущего контекста, который может быть задан с помощью *cudaDeviceSetCacheConfig*. Если текущий контекст также не имеет предпочтения, то будет использована конфигурация предыдущего запуска любого ядра, если нет конфликта по требованиям к кэшу и разделяемой памяти. По умолчанию изначально используется конфигурация в пользу большего объема разделяемой памяти.

Длина кэш-линии составляет 128 байт и соответствует выровненному по 128 байтам сегменту глобальной памяти. Все транзакции с памятью, проходящие через L1- и L2-кэш, имеют размер 128 байт. Использование L1-кэша можно отключить на этапе компиляции с помощью опций *-Xptxas -dlcm=cg*. При отключенном L1-кэше размер транзакции уменьшается до 32 байт. Этот режим может быть более эффективен в случае, если необходимые данные расположены в коротких разрывных участках памяти.

Если размер слова для каждой нити равен 4 байтам, то запросы в память всех 32 нитей варпа объединяются в один. Если размер слова для каждой нити превышает 4 байта, обращение варпа к памяти делится на независимые запросы по 128 байт: два запроса при размере слова 8 байт и четыре при размере слова 16 байт. Каждому запросу, в свою очередь, соответствуют собственные линии в L1- и L2-кэше.

### 3.2.2. Пример: транспонирование матрицы

Пусть необходимо транспонировать квадратную матрицу $A : N \times N$ (здесь и далее мы будем считать, что $N$ кратно 16). Поскольку матрица – это двумерный массив, то будет удобно использовать двумерную сетку и двухмерные блоки. Размер блока выберем равным $16 \times 16$, что позволит запустить до 3 блоков на одном мультипроцессоре архитектуры 1.x и до 6 – на архитектуре Fermi. Тогда для транспонирования матрицы можно использовать следующее ядро:

```
__global__ void transpose ( float * inData, float * outData, int n )
{
    unsigned int xIndex   = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIndex   = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int inIndex  = xIndex + n * yIndex;
    unsigned int outIndex = yIndex + n * xIndex;

    outData [outIndex] = inData [inIndex];
}
```

### 3.2.3. Пример: перемножение двух матриц

Пусть необходимо перемножить две квадратные матрицы $A, B : N \times N$. Как и в предыдущем примере, будем использовать двумерные блоки $16 \times 16$ и двумерную сетку. Ниже приведено ядро, в точности реализующее общую формулу перемножения матриц:

```
__global__ void matmul1 (float* a, float* b, int n, float* c)
{
    // Индексы блока
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Индексы нити внутри блока
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Переменная для накопления результата
    float sum = 0.0f;

    // Смещение для a [i][0]
    int ia = n * BLOCK_SIZE * by + n * ty;

    // Смещение для b [0][j]
    int ib  = BLOCK_SIZE * bx + tx;

    // Перемножить строку и столбец
    for (int k = 0; k < n; k++)
        sum += a [ia + k] * b [ib + k * n];

    // Смещение для записываемого элемента
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

    // Сохранить результат в глобальной памяти
```

```
    c[ic + n * ty + tx] = sum;
}
#define kernel matmul1
#include "main.h"
```

В данном случае при вычислении одного элемента произведения двух матриц на $2N-1$ арифметических операций приходится $2N$ чтений из глобальной памяти. При таком соотношении производительность GPU-ядра ограничена скоростью работы глобальной памяти (в англоязычной литературе – *memory bound*). Значительно большую эффективность может иметь блочный алгоритм перемножения матриц с применением разделяемой памяти (см. раздел о разделяемой памяти).

### 3.2.4. Оптимизация работы с глобальной памятью

Глобальная память обладает высокой латентностью, поэтому для достижения высокой эффективности приложений доступ к ней необходимо оптимизировать.

**Выравнивание.** При чтении и записи значений глобальной памяти на низком уровне используются выровненные 32-, 64- и 128-битные слова. По этой причине все функции выделения глобальной памяти CUDA API всегда возвращают адреса, выровненные по 256-байтам. Если при кратном выравниванию размере элементов, базовый адрес массива по какой-либо причине оказался невыровненным, то чтение каждого элемента вместо одного [0..3] (рис. 3.2, верхняя строка) потребует двух обращений – [0..3] и [4..7], полностью покрывающих невыровненный запрос (рис. 3.2, нижняя строка).

Аналогичная ситуация возникает при использовании элементов массива, размер которых не кратен выравниванию:

```
struct vec3
{
    float x, y, z;
};
```

В этом случае при выровненном базовом адресе и длине элемента в 12 байт, выровненым будет только каждый четвертый элемент, а все остальные потребуют по два обращения. Проблема может быть решена добавлением фиктивного элемента или директивы выравнивания по 16 байтам:

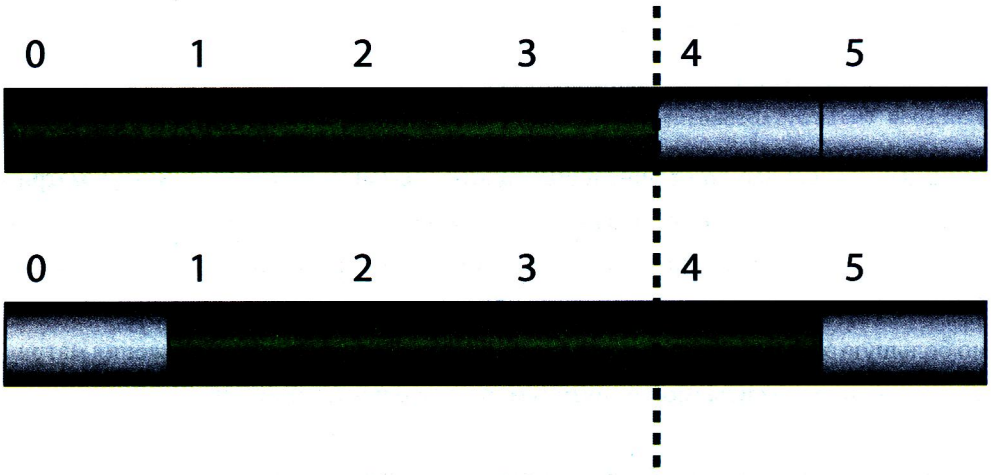


*Рис. 3.2.* Пример выровненного (сверху) и невыровненного (внизу) 4-байтового блока

```
struct __attribute__((aligned(16))) vec3
{
    float x, y, z;
};
```

Теперь все элементы массива будут располагаться по адресам, кратным 16, что обеспечит чтение одного элемента за раз. Таким образом, эффективность доступа может быть повышена ценой увеличения расхода памяти.

**Объединение запросов.** Оптимизации, позволяющие объединять запросы всех нитей полуварпа (compute capability 1.x) или всего варпа в одно обращение к непрерывному блоку глобальной памяти (coalescing), могут на порядок ускорить работу GPU-приложения.

Для того, чтобы GPU с compute capability 1.0 или 1.1 произвел объединение запросов нитей половины варпа, необходимо выполнение следующих условий:

- Все нити должны обращаться к 32-битным словам, давая в результате один 64-байтовый блок, или к 64-битным словам, давая один 128-байтовый блок;
- Полученный блок должен быть выровнен по своему размеру, т.е. адрес 64-байтового блока кратен 64, а адрес 128-байтовго блока кратен 128;

- Все 16 слов, к которым обращаются нити, должны находиться внутри этого блока;

- Нити должны обращаться к словам последовательно: k-ая нить должна обращаться к k-му слову (при этом допускается, что отдельные нити пропустят обращение к соответствующим словам).

Если нити полуварпа не удовлетворяют какому-либо из данных условий, то каждое обращение к памяти происходит как отдельная транзакция. На рис. 3.3 приведены типичные шаблоны обращения к памяти, приводящие к объединению запросов в одну транзакцию: слева выполнены все условия, а справа для части нитей пропущено обращение к соответствующим словам, что позволяет добавить фиктивные обращения и свести к случаю слева. На рис. 3.4 слева для нитей 4 и 5 нарушен порядок обращения к словам, а справа нарушено условие выравнивания: несмотря на то, что слова образуют непрерывный блок из 64 байт, начало этого блока (по адресу 132) не кратно его размеру.

На GPU с compute capability 1.2 и 1.3 объединение запросов в один будет происходить, если слова, к которым обращаются нити, лежат в одном сегменте размером 32 байта (если все нити обращаются к 8-битным словам), 64 байта (если все нити обращаются к 16-битным словам) и 128 байт (если все нити обращаются к 32-битным или 64-битным словам). Объединенный сегмент (блок) должен быть выровнен по 32, 64 или 128 байтам соответственно. В случае compute capability 1.2 и 1.3 порядок, в котором нити обращаются к словам, не играет никакой роли и ситуация на рис. 3.4 слева приведет к объединению всех запросов в одну транзакцию, а для случая справа произойдет объединение запросов в две транзакции.

В устройствах архитектуры Fermi объединение запросов в память происходит для всех 32 нитей (рис. 3.5). В верхней части рисунка приведен пример использования L1-кэша (размер транзакции – 128 байт). В данном случае нарушение выравнивания может и не привести к дополнительной транзакции при успешном попадании остатка в L1-кэш. На нижней части рисунка приведен пример с отключенным L1-кэшем (размер транзакции – 32 байта). Такой режим доступа выгоднее использовать в случае обращения нитей варпа в разрозненные участки глобальной памяти или, наоборот, при обращении всех нитей варпа к одному и тому же элементу.

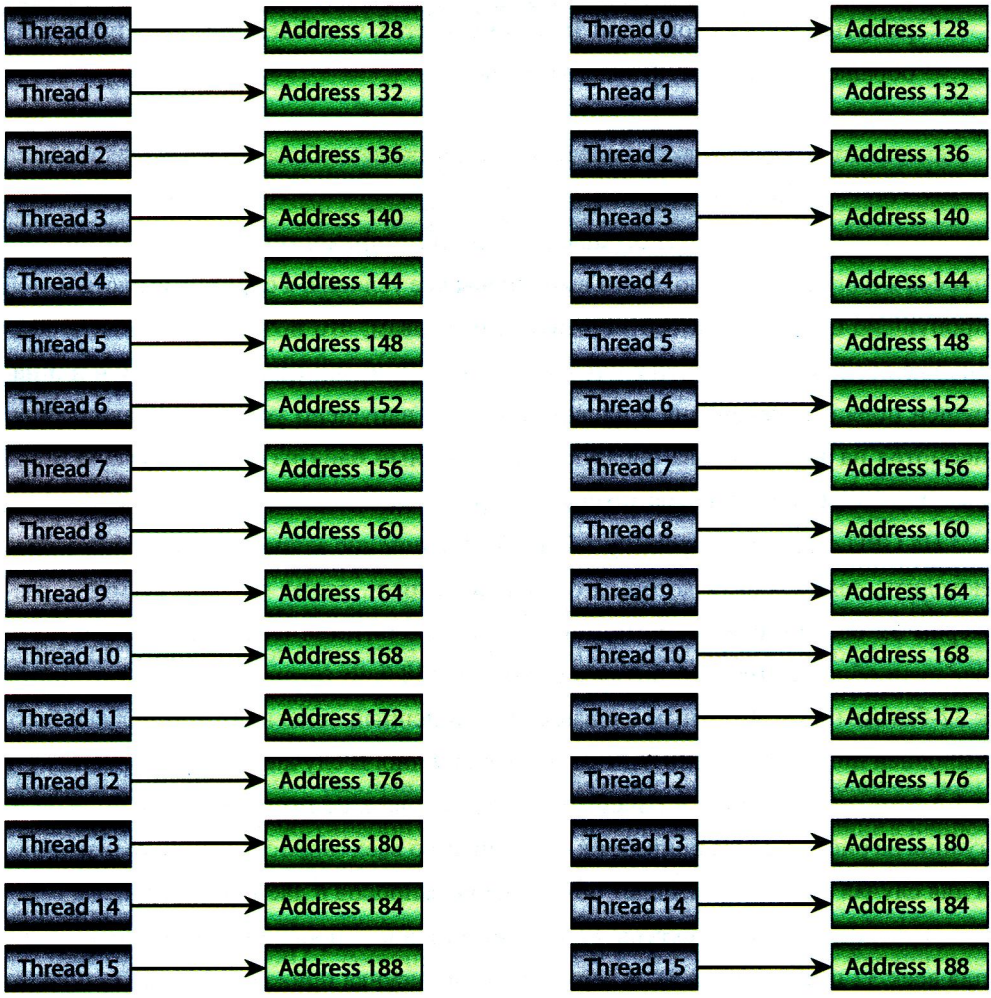


*Рис. 3.3.* Паттерны обращения к памяти, дающие объединение для GPU с compute capability 1.0 и 1.1

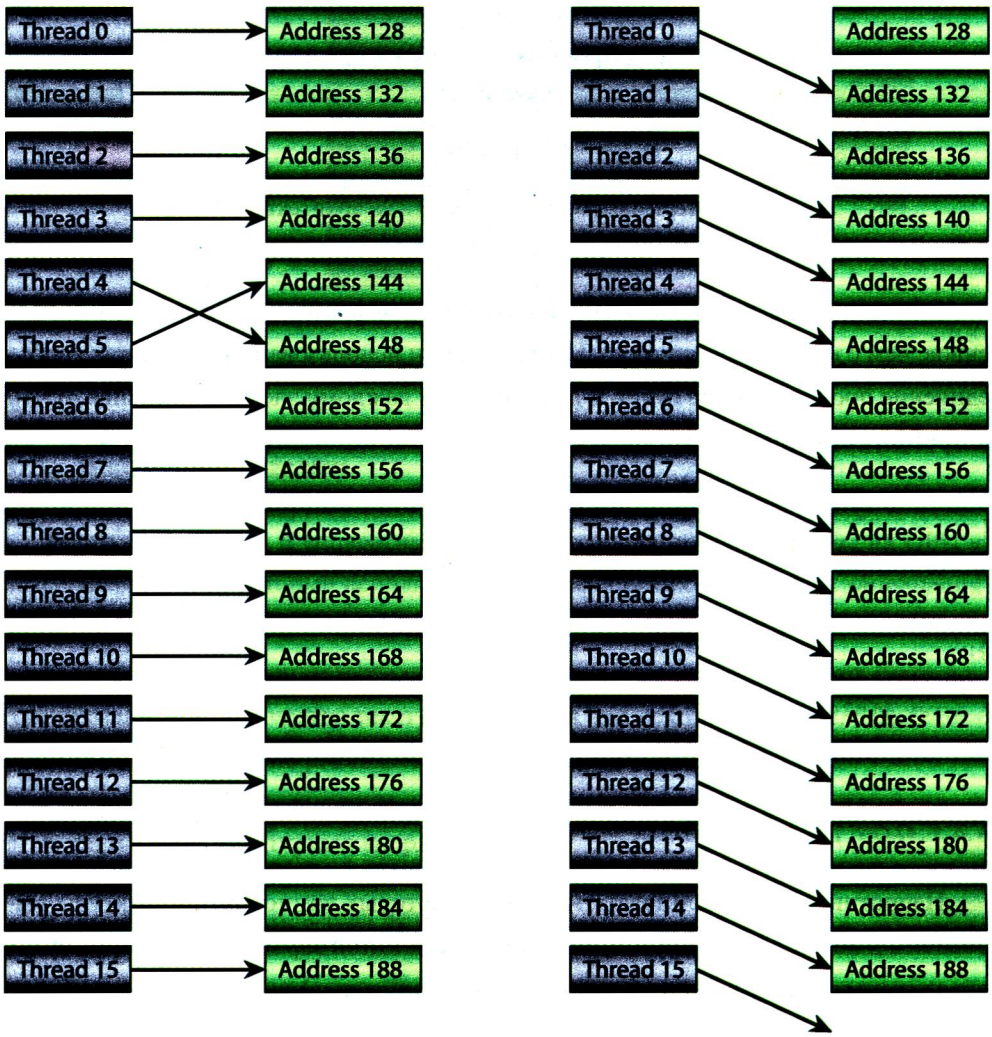


*Рис. 3.4.* Паттерны обращения к памяти, не дающие объединение для GPU с compute capability 1.0 и 1.1

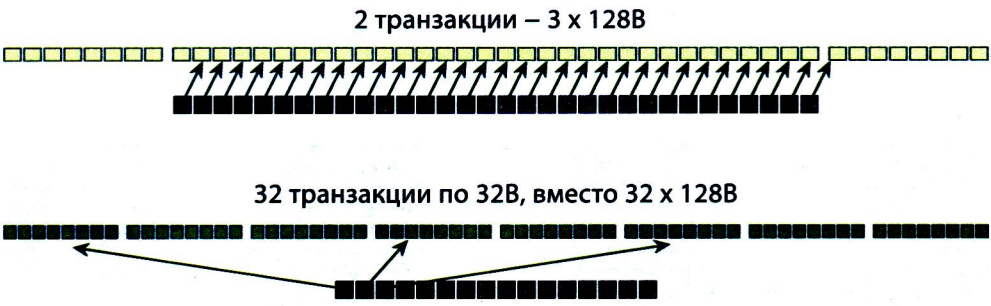


*Рис. 3.5.* Паттерны обращения к памяти на архитектуре Fermi

Объединения запросов может работать более эффективно со структурами массивов, чем с массивами структур. Например, при использовании структуры $A$ объединение происходить не будет, и для доступа к каждому элементу массива $array$ потребуется отдельная транзакция:

```
struct A __attribute__((aligned(16)))
{
    float a;
    float b;
    uint  c;
};

A array [1024];

. . .

A a = array [threadIdx.x];
```

Напротив, если использовать массивы, в которых компоненты структуры выровнены и лежат друг за другом, то запросы всех нитей варпа или полуварпа будут объединены в 3 транзакции (вместо 32 транзакций ранее):

```
float a [1024];
float b [1024];
uint  c [1024];
. . .
float fa = a [threadIdx.x];
float fb = b [threadIdx.x];
uint  uc = c [threadIdx.x];
```