ние режима производит утилита *nvidia-smi* с флагом *-dm* или *–driver-model=*. При отключенном UVA peer-to-peer копирование возможно только с помощью вызова *cudaMemcpyPeer*, в котором явно указываются индексы устройств.

Наличие общего адресного пространства делает избыточным параметр *kind* вызова *cudaMemcpy*. Поэтому он может быть задан *cudaMemcpyDefault* для любых операций peer-to-peer доступа. При отсутствии поддержки peer-to-peer, выключенном peer access или физическом отсутствии прямой связи между GPU на системах с несколькими PCI-E шинами вызов *cudaMemcpy* (или *cudaMemcpyPeer*) произведет копирование данных через оперативную память управляющего устройства.

3.5. Разделяемая память

Разделяемая память размещена непосредственно в каждом мультипроцессоре и доступна для чтения и записи всем нитям блока. Ее наличие отличает CUDA от традиционных графических API. Разделяемая память может существенно улучшить производительность GPU-приложения в случае, если ее удается использовать как буфер, заменяющий обращения к глобальной памяти. Всего каждому мультипроцессору устройства с compute capability 2.x может быть доступно 16 или 48 Кбайт разделяемой памяти в зависимости от размера L1-кэша, который может быть настроен программно. Объем разделяемой памяти делится поровну между всеми блоками нитей, запущенными на мультипроцессоре. Разделяемая память также используется для передачи значений аргументов ядра.

Размер разделяемой памяти может быть задан в CUDA-ядре при определении массивов с атрибутом *__shared__* или в параметрах запуска ядра. В последнем случае размеры используемых *__shared__*-массивов могут не указываться. Если таких массивов несколько, то все они будут указывать на начало выделенной блоку дополнительной разделяемой памяти, т.е. если они должны следовать друг за другом, то потребуется явно указывать смещение:

```
__global__ void kernel1(float* a)
{
    // Явно задано выделить 256*4 байт на блок.
    __shared__ float buf [256];

    // Запись значения из глобальной памяти в разделяемую.
```

```
    buf [threadIdx.x] = a [blockIdx.x * blockDim.x + threadIdx.x];

    ...
}

__global__ void kernel2(float* a)
{
    // Размер явно не указан.
    __shared__ float buf [];

    // Запись значения из глобальной памяти в разделяемую.
    buf [threadIdx.x] = a [blockIdx.x * blockDim.x + threadIdx.x];

    ...
}

// Запустить ядро и задать выделяемый (под buf)
// объем разделяемой памяти в байтах.
kernel2<<<dim3(n / 256), dim3(256), k * sizeof(float)>>> ( a );

__global__ void kernel3(float* a, int k)
{
    // Размер явно не указан.
    __shared__ float buf1 [];

    // Размер явно не указан, считаем, что он передан как k.
    __shared__ float buf2 [];

    buf1 [threadIdx.x]     = a [blockIdx.x * blockDim.x + threadIdx.x];
    buf2 [k + threadIdx.x] = a [blockIdx.x * blockDim.x + threadIdx.x + k];

    ...
}
```

В CUDA-программе нельзя гарантировать, что операция, только что завершенная в текущей нити, уже выполнена и в нитях других варпов. По этой причине любая коллективная операция с разделяемой памятью, после которой нить будет использовать значения, измененные другими нитями, должна завершаться *барьерной синхронизацией* – *__syncthreads()*. Многие приложения используют следующую последовательность действий:

- загрузить необходимые данные в shared-память из глобальной памяти;
- __syncthreads ();

- выполнить вычисления над загруженными данными;
- __syncthreads ();
- записать результат в глобальную память.

3.5.1. Пример: перемножение матриц

Рассмотрим блочный вариант перемножения матриц с использованием разделяемой памяти. Пусть каждый блок GPU вычисляет одну подматрицу C' размером 16×16 (будем считать, что размер матриц N кратен 16). Как показано на рис. 3.8, для вычисления подматрицы C' произведения $A \cdot B$ приходится постоянно обращаться к двух полосам-подматрицам $N \times 16$ исходных матриц A' и B'. Идеальным вариантом было бы разместить копии этих полос в разделяемой памяти, однако с разделяемой памятью размером 48 Кбайт это невозможно: в случае $N = 1024$ одна полоса будет занимать в памяти $1024 \cdot 16 \cdot 4 = 64$ Кбайта. Однако, если каждую из полос разбить на квадратные подматрицы 16×16, то результирующая матрица C' будет суммой попарных произведений подматриц из этих двух полос (рис. 3.9). С таким разбиением вычисление подматрицы C' можно выполнить за $N/16$ шагов, на каждом из которых в разделяемую память загружается одна 16×16 подматрица A и одна подматрица B, что потребует $16 \cdot 16 \cdot 4 \cdot 2 = 2$ Кбайта разделяемой памяти на блок. Каждая нить блока загружает по одному элементу из каждой подматрицы, т.е. на один шаг требуется лишь два обращения в глобальную память.

Поскольку каждая нить загружает только по одному элементу подматрицы, и затем все элементы используются всеми нитями блока, необходимо добавить синхронизацию, которая бы гарантировала полную загрузку всех элементов подматриц (а не только 32 элементов, загруженных данным варпом). Аналогично, синхронизацию необходимо добавить и после вычислений, до загрузки очередной пары, чтобы элементы текущих подматриц гарантированно не использовались какой-либо нитью. Кроме того, в этой версии обращения к глобальной памяти всех нитей блока будут объединены в одну транзакцию (coalesced).

```
__global__ void matmul2(float* a, float* b, int n, float* c)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
```

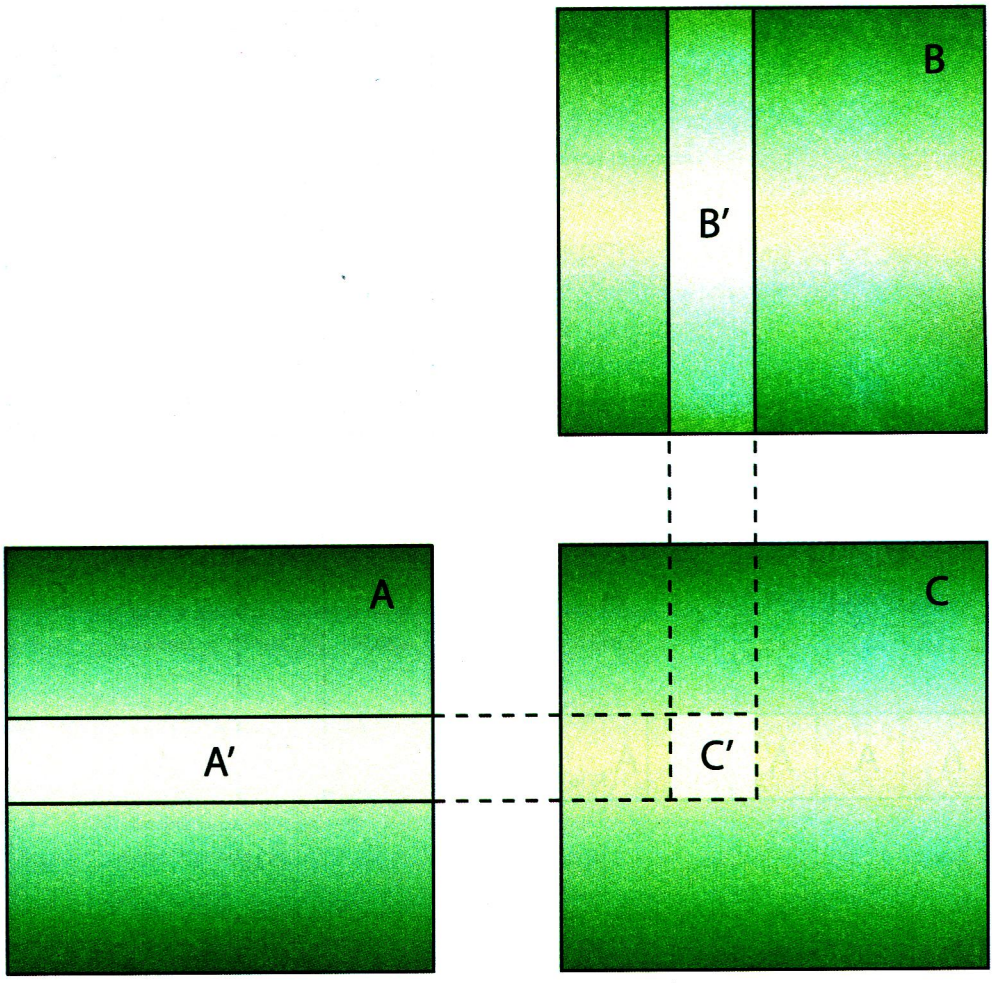

Рис. 3.8. Для вычисления элементов C' нужны только элементы из A' и B'

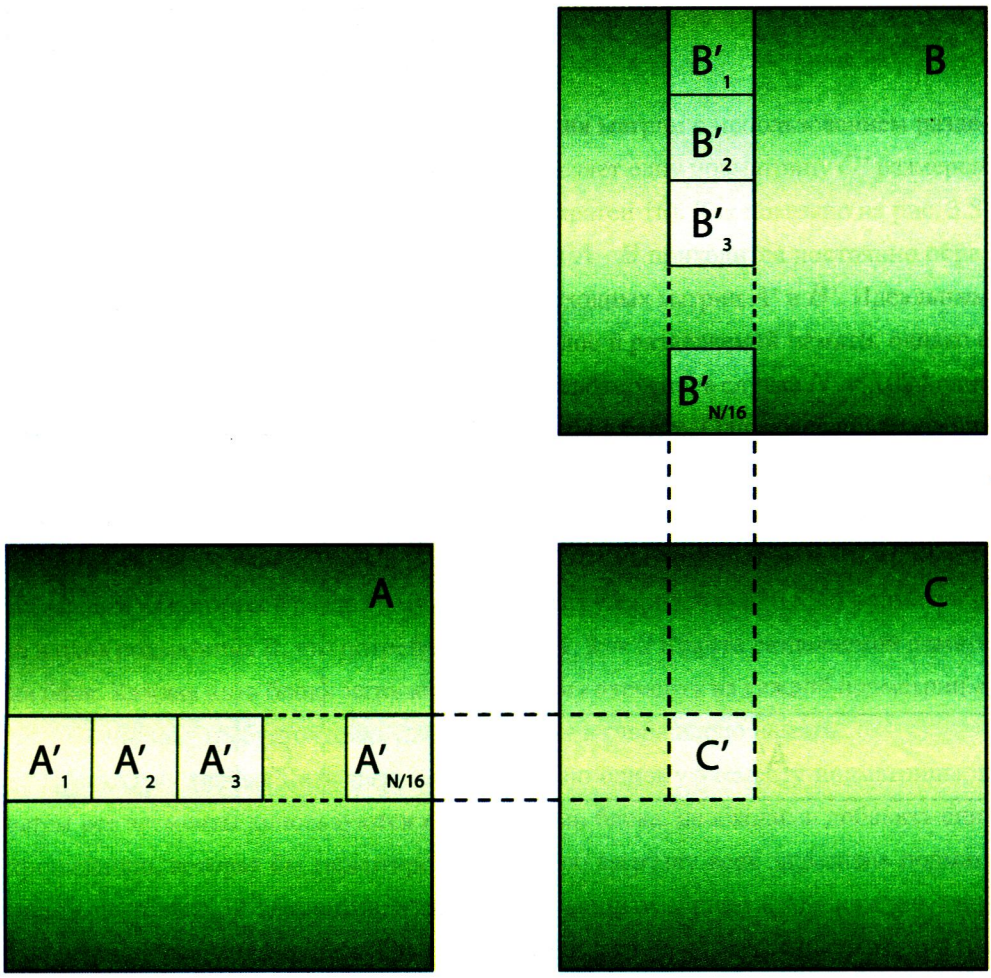

Рис. 3.9. Разложение требуемой подматрицы в сумму произведений матриц 16 × 16

```
int tx = threadIdx.x;
int ty = threadIdx.y;

// Индекс начала первой подматрицы A обрабатываемой блоком.
int aBegin = n * BLOCK_SIZE * by;
int aEnd = aBegin + n - 1;

// Шаг перебора подматриц A.
int aStep = BLOCK_SIZE;

// Индекс первой подматрицы B обрабатываемой блоком.
int bBegin = BLOCK_SIZE * bx;

// Шаг перебора подматриц B
int bStep = BLOCK_SIZE * n;

// Вычисляемый элемент C'.
float sum = 0.0f;

// Цикл по всем подматрицам
for (int ia = aBegin, ib = bBegin; ia <= aEnd; ia += aStep, ib += bStep)
{
    // Очередная подматрица A в разделяемой памяти.
    __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];

    // Очередная подматрица B в разделяемой памяти.
    __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];

    // Загрузить по одному элементу из A и B в разделяемую память.
    as [ty][tx] = a [ia + n * ty + tx];
    bs [ty][tx] = b [ib + n * ty + tx];

    // Дождаться когда обе подматрицы будут полностью загружены.
    __syncthreads();

    // Вычислить элемент произведения загруженных подматриц.
    for (int k = 0; k < BLOCK_SIZE; k++)
        sum += as [ty][k] * bs [k][tx];

    // Дождаться пока все остальные нити блока закончат вычислять
    // свои элементы.
    __syncthreads();
}

// Записать результат.
int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
```

```
    c [ic + n * ty + tx] = sum;
}
#define kernel matmul2
#include "main.h"
```

В отличие от версии, использующей только глобальную память, в данном варианте вместо $2N$ чтений из глобальной памяти достаточно сделать $2N/16$. Количество арифметических операций не изменилось и осталось равным $2N - 1$. Это привело к соответствующему изменению числа обращений в глобальную память (таблица 3.2) и увеличению производительности в 3,5 раза (таблица 3.3). Для сравнения, перемножение матриц также было выполнено с помощью функции библиотеки CUBLAS, дополнительно использующей аналогичный алгоритм блочного перемножения на уровне регистров [4].

Таблица 3.2. Результаты профилирования вариантов перемножения матриц 2048×2048 на Tesla C2070

Метод	Количество чтений из глобальной памяти на варп в одном SM	Количество записей в глобальную память на варп в одном SM
«в лоб», без использования разделяемой памяти	38535168	9408
С использованием разделяемой памяти	1196032	9344

3.5.2. Эффективный доступ к разделяемой памяти

Для повышения пропускной способности разделяемая память разбита на 16 банков в устройствах с compute capability 1.x и на 32 банка – в более современных архитектурах[1]. В каждый момент времени банк может выполнять одно чтение или запись 32-битного слова. Подряд идущие 32-битные слова попадают в различные

[1] Без ограничения общности все иллюстрации в данном разделе даны для 16 банков.

Таблица 3.3. Быстродействие вариантов перемножения матриц $C = AB$, 2048×2048 на Tesla C2070

Метод	Время, мс
Без использования разделяемой памяти	324,63
С использованием разделяемой памяти	93,26
С использованием CUBLAS	30,84

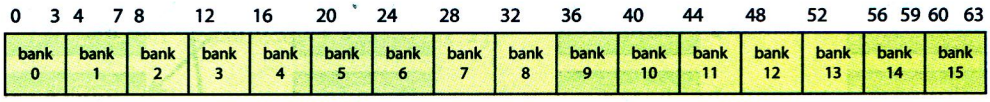

Рис. 3.10. Разбиение 64 байт разделяемой памяти на банки

подряд идущие банки. Если все 32 нити варпа обращаются к 32 32-битным словам, находящимся в разных банках, то данные будут получены без дополнительных задержек. Подобная организация разделяемой памяти имеет цель оптимизировать типичный для приложений характер запросов: чтение различными нитями варпа подряд идущих 32-битных значений (рис. 3.10). Обращения к одному банку могут быть выполнены только последовательно. Одновременный запрос данных из одного банка несколькими нитями называется *конфликтом банков* и характеризуется *порядком конфликта* – максимальным числом обращений в один банк. Если имеет место конфликт второго порядка хотя бы для одного банка, то скорость доступа к разделяемой памяти снижается вдвое. Особым случаем является обращение всех 32 нитей варпа к одному и тому же элементу одного банка: тогда включается режим broadcast-запроса, и конфликта не возникает.

На рис. 3.11 приведены примеры бесконфликтного доступа к разделяемой памяти. В частности, конфликта не возникает при непоследовательном соответствии нитей и банков.

На рис. 3.12 приведены два примера доступа к разделяемой памяти с конфликтами по банкам памяти. В случае, показанном слева, возникает 8 конфликтов вто-

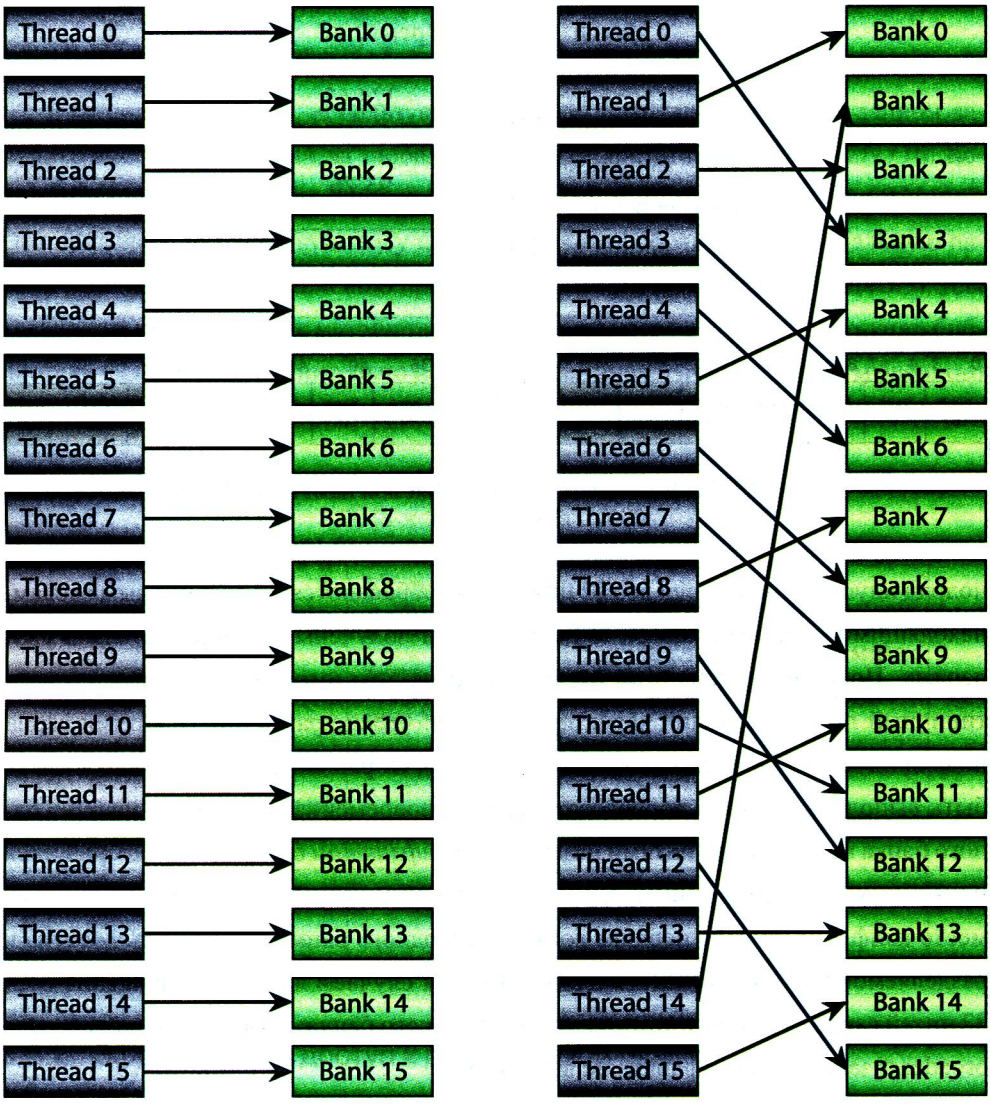

Рис. 3.11. Примеры доступа к разделяемой памяти, в которых не возникает конфликт банков

рого порядка, и скорость работы с разделяемой памятью снижается вдвое, а справа – конфликты 4, 5 и 6-го порядков, приводящие к 6-кратному замедлению.

Рассмотрим также некоторые примеры кода CUDA-ядер. Обычно адреса, по которым производится доступ в разделяемую память, линейно зависят от номера нити. В первом случае конфликтов не будет, однако ситуация меняется при использовании элементов размером менее 32 бит. В следующем фрагменте кода элементы *buf[0]*, *buf[1]*, *buf[2]* и *buf[3]* лежат в одном и том же банке памяти, что приведет к конфликту четвертого порядка. Аналогично в третьем случае будет получен конфликт 2-го порядка.

```
// Нет конфликтов.
__shared__ float buf [128];
float v = buf [baseIndex + threadIdx.x];

// Конфликт 4-го порядка.
__shared__ char buf [128];
char v = buf [baseIndex + threadIdx.x];

// Конфликт 2-го порядка.
__shared__ short buf [128];
short v = buf [baseIndex + threadIdx.x];
```

3.5.3. Пример: умножение матрицы на транспонированную

Рассмотрим перемножение матрицы на транспонированную: $C = AA^T$. В данном случае имеется только одна входная матрица, однако в разделяемой памяти по-прежнему необходимо иметь две подматрицы 16×16, соответствующие A и A^T:

```
__global__ void matmult1(float* a, int n, float* c)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Индекс первой подматрицы A, обрабатываемой блоком.
    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;
```

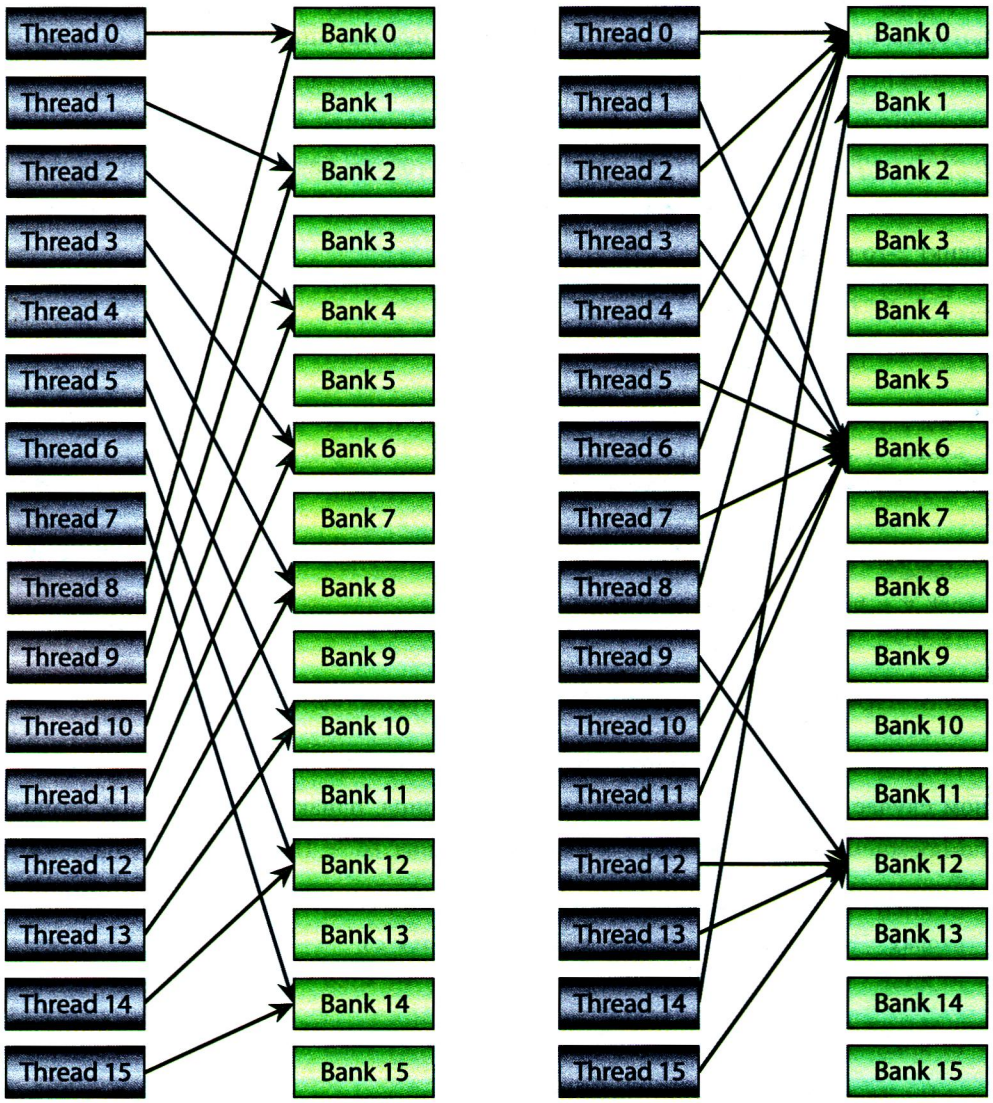

Рис. 3.12. Примеры доступа к разделяемой памяти, в которых возникает конфликт банков

```
    // Индекс первой подматрицы B, обрабатываемой блоком.
    int atBegin = n * BLOCK_SIZE * bx;

    // Вычисляемый элемент C.
    float sum = 0.0f;

    // Цикл по 16*16 подматрицам
    for (int ia = aBegin, iat = atBegin; ia <= aEnd;
        ia += BLOCK_SIZE, iat += BLOCK_SIZE)
    {
        __shared__ float as  [BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float ats [BLOCK_SIZE][BLOCK_SIZE];

        // Загрузить подматрицы в разделяемую память.
        as  [ty][tx] = a [ia  + n * ty + tx];
        ats [ty][tx] = a [iat + n * ty + tx];

        // Синхронизация, чтобы убедиться, что обе подматрицы загружены.
        __syncthreads();

        // Находим нужный элемент произведения подматриц
        for (int k = 0; k < BLOCK_SIZE; k++)
            sum += as [ty][k] * ats [tx][k];

        // Синхронизация, чтобы убедиться, что
        // текущие подматрицы не нужны ни одной нити блока.
        __syncthreads();
    }

    // Записать найденный элемент произведение матриц
    // в глобальную память.
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    c [ic + n * ty + tx] = sum;
}
#define kernel matmult1
#include "main.h"
```

В отличие от общего случая, доступ к транспонированной матрице в разделяемой памяти (A^T) будет идти не по строкам, а по столбцам, т.е. нити одного варпа будут обращаться к элементам столбца этой матрицы. Поскольку матрица имеет размер 16 × 16, каждый ее столбец полностью располагается в одном банке, что приводит к банк-конфликту 16-го порядка. Избавиться от конфликтов можно, добавив в матрицу A^T один фиктивный столбец (рис. 3.13). Тогда 16 элементов столбца

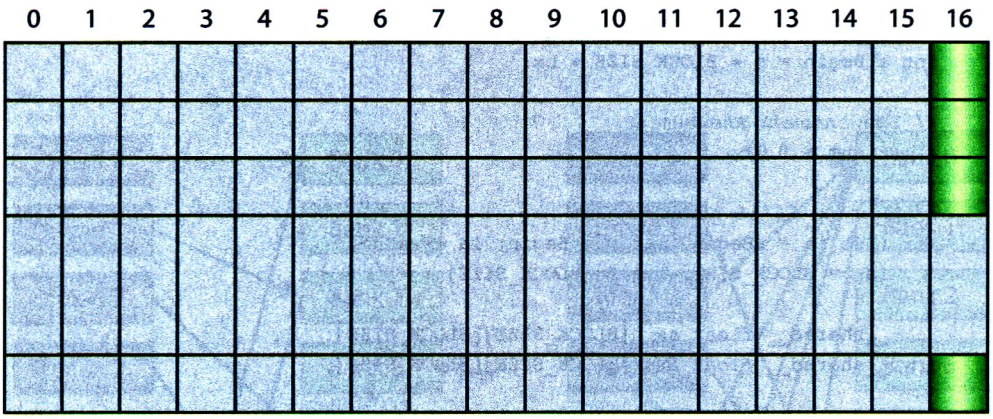

Рис. 3.13. Добавление к матрице 16 × 16 одного столбца для избавления от конфликта банков

будут распределены по всем 16 банкам памяти.

Вставка фиктивных элементов является распространенным приемом выравнивания доступа к памяти и устранения банк-конфликтов:

```
__global__ void matmult2(float* a, int n, float* c)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Индекс первой подматрицы A, обрабатываемой блоком.
    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;

    // Индекс первой подматрицы B, обрабатываемой блоком.
    int atBegin = n * BLOCK_SIZE * bx;

    // Вычисляемый элемент C.
    float sum = 0.0f;

    // Цикл по 16*16 подматрицам
    for (int ia = aBegin, iat = atBegin; ia <= aEnd;
        ia += BLOCK_SIZE, iat += BLOCK_SIZE)
    {
        __shared__ float as  [BLOCK_SIZE][BLOCK_SIZE];
```

```
        __shared__ float ats [BLOCK_SIZE][BLOCK_SIZE + 1]; // +1!

        // Загрузить подматрицы в разделяемую память.
        as  [ty][tx] = a [ia  + n * ty + tx];
        ats [ty][tx] = a [iat + n * ty + tx];

        // Синхронизация, чтобы убедиться, что обе подматрицы загружены.
        __syncthreads();

        // Находим нужный элемент произведения подматриц
        for (int k = 0; k < BLOCK_SIZE; k++)
            sum += as [ty][k] * ats [tx][k];

        // Синхронизация, чтобы убедиться, что
        // текущие подматрицы не нужны ни одной нити блока.
        __syncthreads();
    }

    // Записать найденный элемент произведение матриц
    // в глобальную память.
    int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    c [ic + n * ty + tx] = sum;
}
#define kernel matmult2
#include "main.h"
```

Таблица 3.4. **Быстродействие вариантов перемножения матриц $C = AA^T$, 2048×2048 на Tesla C2070**

Метод	Время, мс
Без выравнивания в разделяемой памяти	596,60
С выравниванием в разделяемой памяти	92,50
С использованием CUBLAS	30,77