# *Лабораторная работа №3*
# *Работа с глобальной и разделяемой памятью в NVIDIA CUDA при обработке векторов и матриц*

## Работа с константной памятью

***Константная память*** выделяется непосредственно в коде программы при помощи спецификатора constant. Все нити сетки могут читать из нее данные, и чтение из нее кешируется. CPU имеет доступ к ней как на чтение, так и на запись при помощи следующих функций:

cudaError_t cudaMemcpyToSymbol (.const char *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind );

cudaError_t cudaMemcpyFromSymbol ( void *dst, const char *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind );

cudaError_t cudaMemcpyToSymbolAsync ( const char *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStreamt stream );

cudaError_t cudaMemcpyFromSymbolAsync ( void *dst, const char *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream t stream );

В качестве значения параметра kind выступает одна из следующих констант, задающих направление копирования, cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost и cudaMemcpyDeviceToDevice. Параметр stream позволяет организовывать несколько потоков команд, если вы не используете эту возможность, то следует использовать поток по умолчанию – 0.

Пример: построение таблицы значений функции с заданным шагом.

Рассмотрим пример построения таблицы значений функции y=sin(√x) с заданным шагом. Для этого необходимо выделить два массива одинакового размера для хранения результата: один – в памяти CPU, другой – в памяти GPU. После этого запускается ядро, заполняющее массив в глобальной памяти заданными значениями. После завершения ядра необходимо скопировать результаты вычислений из памяти GPU в память CPU и освободить выделенную глобальную память.

```
// ядро, осуществляющее заполнение массива:
__global__ void tableKernal ( float *devPtr, float step )
{                       // получить глобальный адрес нити:
        int index = blockIdx.x * blockDim.x + threadIdx.x;
                        // вычислить значение аргумента:
        float x = step *index;
                        // используем «быстрые» версии функции
        devPtr [index] =  __sinf( __sqrtf ( x ) );
}
void buildTable ( float *res, int n, float step )
{       float *devPtr;
```

```
                    // убедимся, что n кратно 256:
        assert (n % 256 ==0);
                    // выделяем глобальную память под таблицу:
        cudaMalloc &devPtr, n *sizeof (float) );
                    // запускаем ядро для вычисления значений:
        tableKernel<<<dim3(n/256), dim3(256)>>>(devPtr, step );
                    // копируем результат из глобальной памяти в память CPU:
        cudaMemcpy res, devPtr, n *sizeof(float), cudaMemcpyDeviceToHost );
                    // освобождаем выделенную глобальную память
        cudaFree (devPtr);
}
```

Пример: транспонирование матрицы

В качестве следующего примера рассмотрим задачу транспонирования квадратной матрицы A размера NxN, будем считать, что N кратно 16.

Поскольку сама матрица A двумерна, то будет удобно использовать двумерную сетку и двумерные блоки. В качестве размера блока выберем 16x16, это позволит запустить до трех блоков на одном мультипроцессоре. Тогда для транспонирования матрицы можно использовать следующее ядро:

```
__global__ void transposel 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];
}
```

Пример: умножение двух матриц

Несколько более сложным примером (к которому мы еще вернемся при работе с разделяемой памятью) будет задача умножения двух квадратных матриц A и B (будем считать, что они обе имеют размер NxN, где N кратно 16). Матрица-произведение C двух матриц A и B задается при помощи формулы:

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \cdot b_{k,j}$$

Как и в предыдущем примере, будем использовать двумерные блоки 16x16 и двумерную сетку. Ниже приводится простейший пример умножения.

```
__global__ void matMult ( float *a, float *b, int n, float * c )
{       int bx = blockIdx.x             // индексы блока
        int by = blockIdx. y;
        int tx = threadIdx.x;           // индексы нити внутри блока
        int ty = threadIdx.y;
        float sum = 0.0f ;              // здесь накапливается результат
        int ia = n * BLOCK_SIZE * by + n * ty;      // смещение для a [i] [0]
        int ib = BLOCK_SIZE * bx + tx;              // смещение для b [0] [j]
                                        // умножаем и суммируем
```

```
    for ( int k = 0; k < n; k++ )
        sum += a [ia + k] * b [ib + k*n] ;
                                    // сохраняем результат в глобальной памяти
                                    // смещение для записываемого элемента:
    int ic = n * BLOCK_SIZE * by + BLOCK SIZE * bx;
    c [ic + n * ty + tx] = sum;
}
```

Как легко можно убедиться, в данном примере для нахождения одного элемента произведения двух матриц нам нужно прочесть 2xN значений из глобальной памяти и выполнить 2xN арифметических операций. В данном случае основным фактором, лимитирующим быстродействие данной программы, является чтение из глобальной памяти (а не вычисления), подобные случаи называются ***memory bound***.

# Работа с разделяемой памятью

Пример: умножение матриц

Рассмотрим использование разделяемой памяти на примере умножения двух квадратных матриц A и B. Как и ранее, будем использовать двумерные блоки размера 16x16 и будем считать, что размер матриц N кратен 16. Каждый блок будет вычислять одну 16x16 подматрицу C' искомого произведения.

Как видно по рис. 3.1, для вычисления подматрицы C' произведения AxB приходится постоянно обращаться к двум полосам (подматрицам) исходных матриц A и B. Обе эти полосы имеют размер Nx16, и их элементы многократно используются в расчетах.

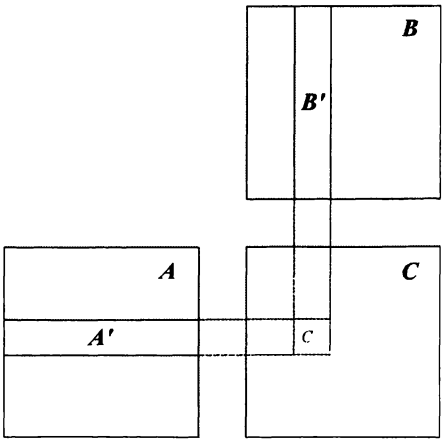


Рис. 3.1. Для вычисления элементов C' нужны только элементы из A' и B'

Идеальным вариантом было разместить копии этих полос в разделяемой памяти, однако для реальных задач это неприемлемо из-за небольшого объема имеющейся разделяемой памяти (так, если N равно 1024, то одна полоса будет занимать в памяти 1024x16x4 == 64 Кб).

Однако если каждую из этих полос разбить на квадратные подматрицы 16x16, то становится видно, что результирующая матрица C просто является суммой попарных произведений подматриц из этих двух полос:

$C == A'_1 \times B'_1 + A'_2 \times B'_2 + ... + A'_{N/16} \times B'_{N/16}$.

За счет этого можно выполнить вычисление подматрицы C' всего за N/16 шагов. На каждом таком шаге в разделяемую память загружаются одна 16x16 подматрица A и одна подматрица B (при этом нам потребуется 16x16x4x2 == 2 Кбайта разделяемой памяти на блок), при этом каждая нить блока загружает ровно по одному элементу из каждой из этих подматриц, то есть каждая нить делает всего два обращения к глобальной памяти на один шаг.

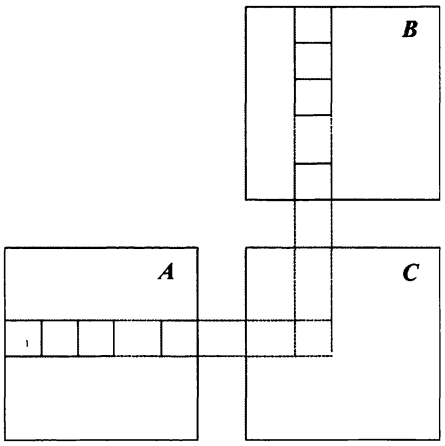


Рис. 3.2. Разложение требуемой подматрицы в сумму произведений матриц 16x16.

После этого считается произведение подматриц, загруженных в разделяемую память, и суммируется нужный элемент произведения, и идет переход к следующей паре подматриц.

Поскольку каждая нить загружает только по одному элементу из каждой из подматриц, а используем все эти элементы, то после загрузки необходимо поставить синхронизацию, чтобы убедиться, что обе подматрицы загружены полностью (а не только 32 элемента, загруженных данным warp'ом). Точно так же синхронизацию необходимо поставить и после вычисления произведения загруженных подматриц до загрузки следующей пары (чтобы убедиться, что текущие подматрицы уже не нужны никакой нити).

Кроме того, в данном варианте все обращения к глобальной памяти будут coalesced. ***coalescing – использование возможности GPU объединять несколько запросов к глобальной памяти в один***.

```
  #define       BLOCK_SIZE          16        // Размер блока.

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

        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;
    int aStep = BLOCK_SIZE;              // Шаг перебора подматриц A.
          // Индекс начала первой подматрицы B, обрабатываемой блоком:
    int bBegin BLOCK_SIZE * bx;
    int bStep BLOCK_SIZE * n;            // Шаг перебора подматриц B.

    float sum = 0.0f;                    // Вычисляемый элемент C'.
                                         // Цикл по 16*16 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() ;
          // Вычисляем нужный 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;
}
```

Таким образом, при вычислении произведения матриц на каждый элемент произведения C нужно выполнить всего 2xN/16 чтений из глобальной памяти, в отличие от предыдущего варианта без использования глобальной памяти, где требовалось на каждый элемент 2xN чтений. Количество арифметических операций не изменилось и осталось равным 2xN  1. В табл. 3.1 приведено сравнение быстродействия с использованием и без использования разделяемой памяти.

*Сравнение быстродействия двух вариантов умножения матриц.  Табл. 3.1.*

| ***Способ умножения матриц*** | ***Затраченное время в миллисекундах*** |
|---|---|
| Без использования разделяемой памяти | 2484.06 |
| С использованием разделяемой памяти | 133.47 |

Как видно из приведенной таблицы, быстродействие выросло больше чем на порядок. В первом случае основное время было затрачено на чтение из глобальной памяти, причем почти все оно было uncoalesced, а на вычисления было затрачено менее одной десятой от времени чтения из глобальной памяти.

Во втором случае свыше 80% времени было затрачено на вычисления, доступ к глобальной памяти занял менее 13%, и весь доступ был coalesced.

### Пример: умножение матрицы на транспонированную

Рассмотрим в качестве следующего примера частный случай умножения матриц, когда матрица A умножается на свою транспонированную матрицу $AxA^T$. В данном случае хотя у нас всего одна входная матрица, но в разделяемой памяти нужно по-прежнему держать две подматрицы 16x16: одна из них соответствует исходной матрице A, а вторая транспонированной матрице A. Для данного случая можно переписать ядро, выполняющее данное умножение следующим образом:

```
__global__ void matMult ( 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;
    float sum = 0.0f;               // Вычисляемый элемент C
                                    // 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];
        __syncthreads();            // Синхронизация, чтобы убедиться,
                                    // что обе подматрицы загружены.
                                    // 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 ;
}
```

Однако в данном случае, в отличие от умножения произвольных матриц, можно заметить, что доступ ко второй матрице в разделяемую память (ats) будет осуществляться по столбцам, то есть нити одного полу-warp'а будут обращаться к элементам столбца этой

матрицы (в отличие от обращения к первой матрице as, где обращение будет идти по строкам).

Поскольку матрица имеет размер 16x16, то, как легко видеть, каждый ее столбец полностью находится в одном банке, таким образом мы получаем конфликт 16-го порядка. Для избавления от этого конфликта можно воспользоваться очень простым приемом – добавим в матрицу ats один фиктивный (то есть не используемый нами) столбец (рис.3.3).

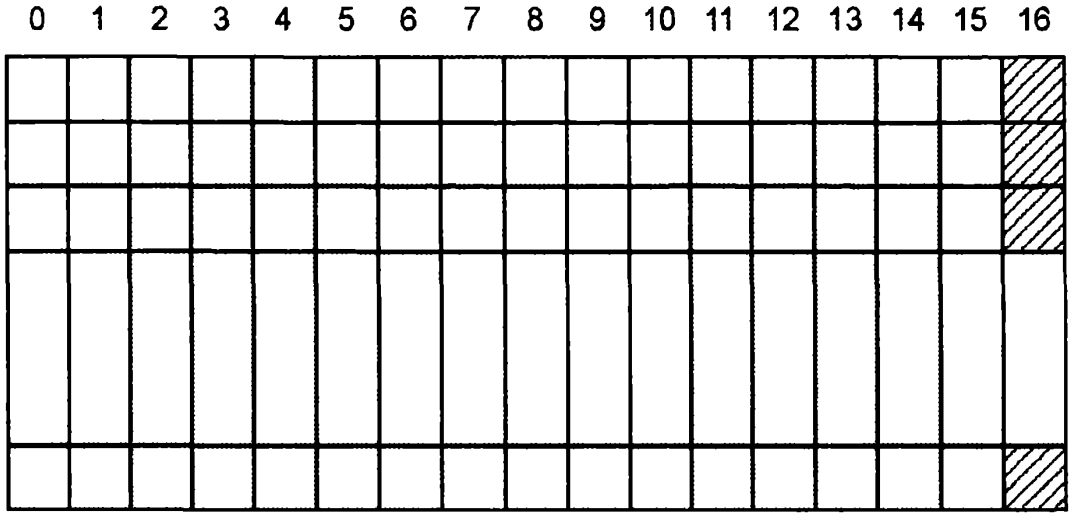


Рис. 3.3. Добавление к матрице 16x16 одного столбца для избавления от конфликта банков.

Как легко можно убедиться, теперь как первые 16 элементов каждой строки, так и каждый столбец получившейся матрицы будет занимать все 16 банков памяти, таким образом, мы полностью избавились от конфликтов по банкам. Добавление некоторого количества пустых элементов к исходным данным таким образом, чтобы избавиться от конфликтов по банкам памяти - довольно распространенный прием. Подобный прием также позволяет выполнить выравнивание для доступа к глобальной памяти. Далее приводится соответствующее ядро.

```
__global__ void matMult ( 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;
    int atBegin = n * BLOCK_SIZE * bx;
    float sum = 0.0;        // Вычисляемый элемент C.
                        // 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];
                                    // Загрузить подматрицы в разделяемую память:
        as [ty] [tx] = a [ia + n * ty + tx];
        ats [ty] [tx] = a [iat + n * ty + tx];
        __syncthreads() ;   // Синхронизация, чтобы убедиться,
                            // что обе подматрицы загружены.
                            // Находим нужный элемент произведения подматрицы:
        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;
}
```

*Сравнение быстродействия двух вариантов умножения матриц.  Табл. 3.2.*

| ***Способ умножения матриц*** | ***Затраченное время в миллисекундах*** |
|---|---|
| Без выравнивания в разделяемой памяти | 551.73 |
| С выравниванием в разделяемой памяти | 120.59 |

Как видно из табл. 3.2, избавление от конфликтов по банкам памяти поднимает быстродействие более чем в четыре раза. Однако конкретные данные, как для табл. 3.1, так и для табл. 3.2 зависят от конкретной среды программирования.

## Параллельная редукция

Одной из часто встречающихся задач являются так называемая редукция массив (reduction). Пусть у нас есть массива0,a1,...,aN-1 и некоторая бинарная операция или функция (например сложение).

Тогда следующее выражение будет называться редукцией массива a0,a1,...,aN-1 относительно заданной операции (в нашем случае сложения):

A=((a0+a1)+...+aN-1

В качестве бинарной операции также может выступать умножение, минимум или максимум, давая в результате произведение всех элементов массива или минимальный/максимальный элемент массива. Точно также скалярное произведение двух больших массивов является редукцией поэлементного произведения этих массивов.

На примере реализации редукции в CUDA мы рассмотрим основные элементы оптимизации программы.

Обратите внимание на то, что в CUDA есть ограничение - количество блоков по каждому измерению (т.е. размер grid'a по каждому измерению не может превышать 65535). В нашем случае, поскольку мы работаем с одномерным массивом и одномерный grid является наиболее подходящим способом организации блоков, это накладывает ограничения на размер входного массива. Однако данное ограничение не принципиально - можно легко переписать рассматриваемые ниже варианты для двух- или трехмерных grid'ов.

Первое с чем нужно определиться - это что является лимитирующим быстродействие фактором - арифметические операции или доступ к памяти. В нашем случае очевидно, что лимитировать нас будет именно доступ к памяти и именно этот доступ и нужно оптимизировать в первую очередь.

Первый шаг довольно прост - каждому блоку соответствует часть массива (делим массив поровну между всеми блоками). Задача блока найти сумму всех элементов своей части и записать результирующее значение в выходной массив. При этом каждой нити соответствует по одному элементу массива, сначала каждая нить загружает свой элемент в shared-память, а затем иерархически суммирует, как показано на рис 3.7.

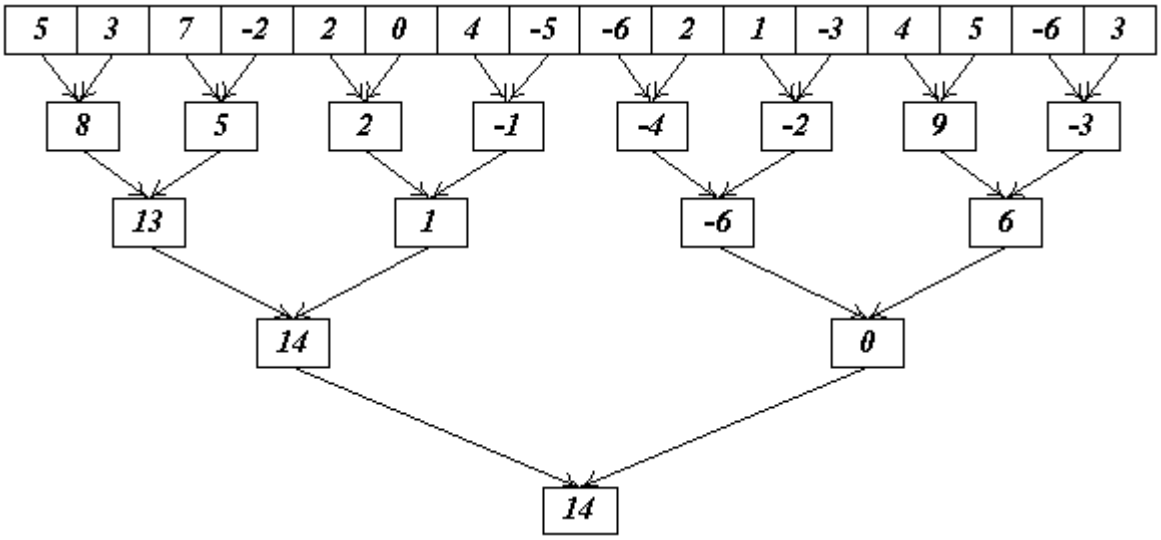


Рис 3.7. Иерархическое суммирование элементов внутри блока.

Далее мы рассмотрим именно оптимизацию этого шага - нахождение каждым блоком своей суммы наиболее эффективным образом. Будем использовать одномерный массив блоков, где каждый блок является одномерным массивом нитей. Ниже приводится ядро, осуществляющего параллельную редукцию описанным методом:

```
__global__ void reduce1 ( int * inData, int * outData )
{
   __shared__ int data [BLOCK_SIZE];

   int  tid = threadIdx.x;
   int  i   = blockIdx.x * blockDim.x + threadIdx.x;
```

```
   data [tid] = inData [i];       // load into shared memeory

   __syncthreads ();

   for ( int s = 1; s < blockDim.x; s *= 2 )
   {
      if ( tid % (2*s) == 0 )
         data [tid] += data [tid + s];

      __syncthreads ();
   }

   if ( tid == 0 )               // write result of block reduction
      outData [blockIdx.x] = data [0];
}
```

На рисунке 3.8 проиллюстрирована схема работы данного подхода.

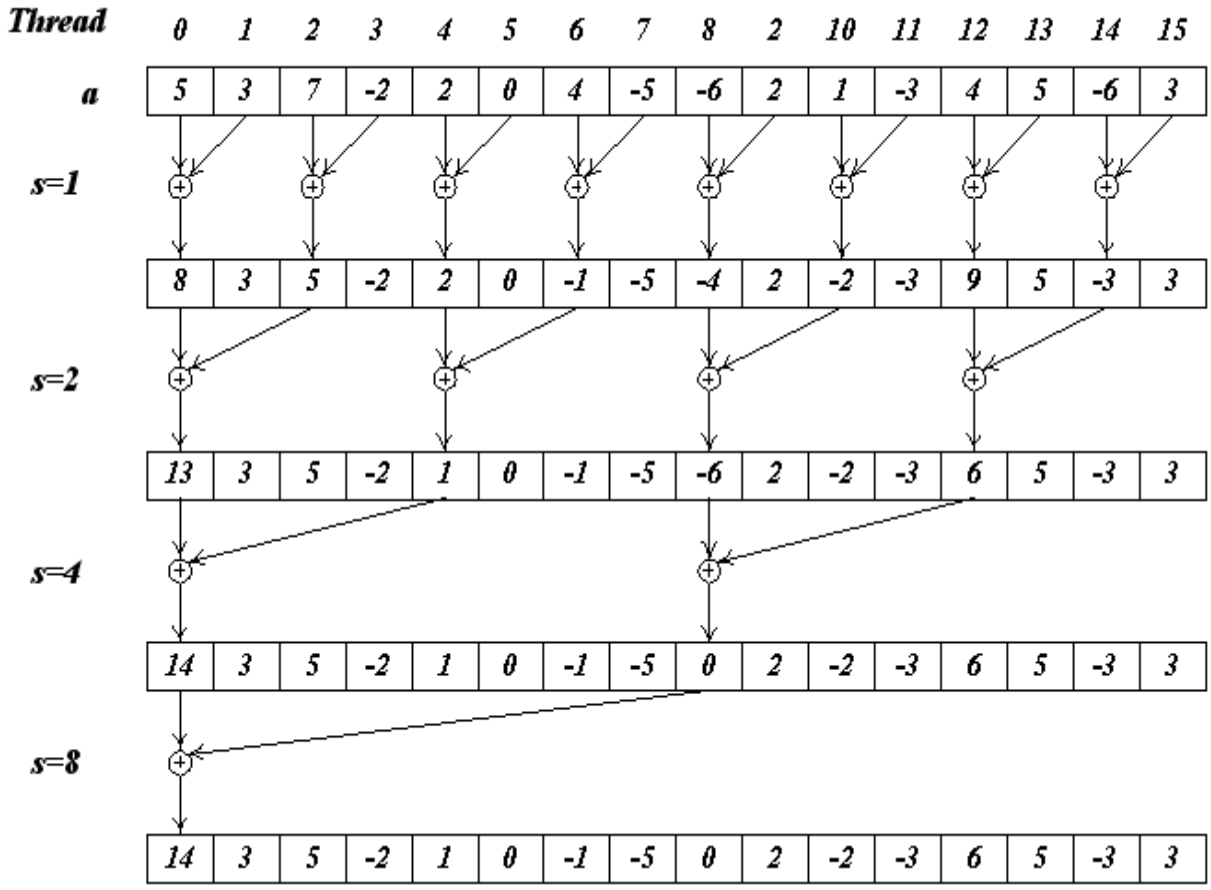


Рис 3.8.

Сразу виден большой недостаток этого подхода - условный оператор приводит к сильному ветвлению кода внутри каждого warp'a. Его можно избежать, переписав ядро следующим образом:

```
__global__ void reduce1 ( int * inData, int * outData )
{
   __shared__ int data [BLOCK_SIZE];
```

```
   int      tid = threadIdx.x;
   int      i   = blockIdx.x * blockDim.x + threadIdx.x;

   data [tid] = inData [i];        // load into shared memeory

   __syncthreads ();

   for ( int s = 1; s < blockDim.x; s <<= 1 )
   {
      int  index = 2 * s * tid;

      if ( index < blockDim.x )
                        data [index] += data [index + s];

      __syncthreads ();
   }

   if ( tid == 0 )                 // write result of block reduction
      outData [blockIdx.x] = data [0];
}
```

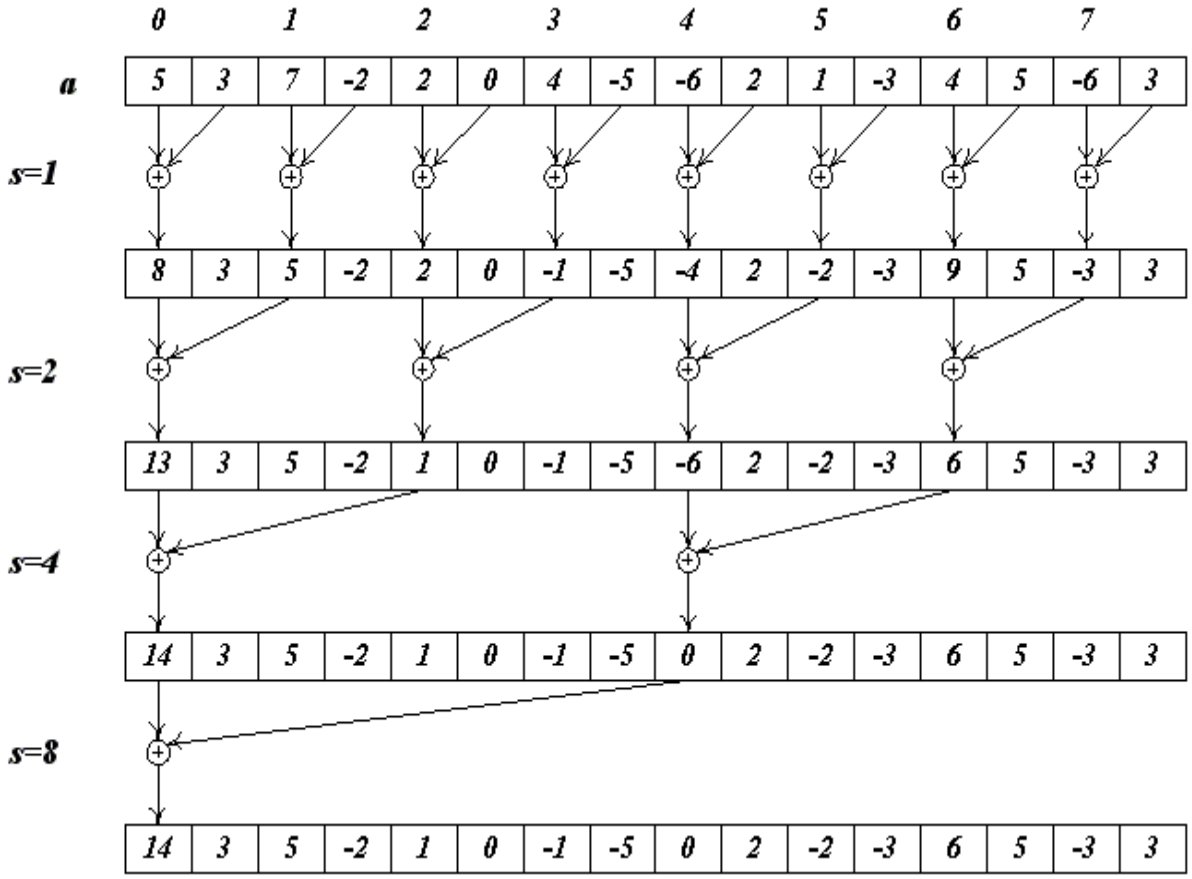


Рис 3.9.

Как видно из рисунка 3.9, получается почти такая же схема, но распределение операций и элементов по нитям изменилось, за счет чего и удалось получить минимальное ветвление.

Однако данный подход имеет серьезный недостаток - он приводит в большому числу конфликтов банков при обращении к shared-памяти - так при s>1 у нас не будет ни одного обращения к банкам с нечетными номерами, а на банки с четными номерами придется двойная нагрузка. По мере увеличения s количество неиспользуемых банков (а, значит, и нагрузка на остальные) будет только расти.

Для того, чтобы избежать подобной проблемы реорганизуем саму схему суммирования - начнем суммировать не с соседних элементов, а наоборот - с элементов, удаленных друг от друга на dimBlock.x/2. На следующем шаге будем суммировать элементы, удаленные друг от друга на dimBlock.x/2 и т.д. (см. рис 3.10).

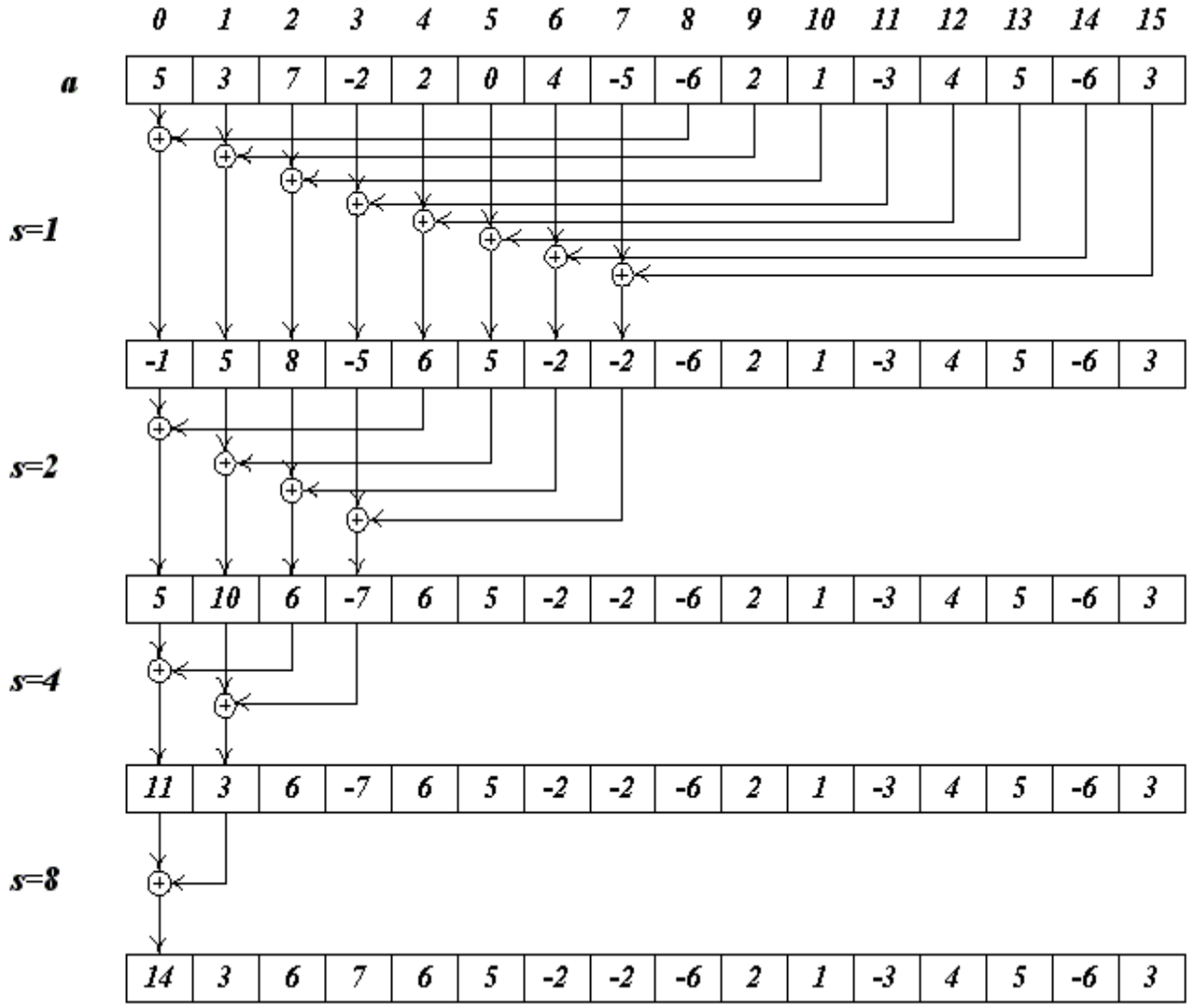


Рис 3.10.

Соответствующая схема реализуется следующим ядром:

```
__global__ void reduce3 ( int * inData, int * outData )
{
   __shared__ int data [BLOCK_SIZE];
   int   tid = threadIdx.x;
   int   i   = blockIdx.x * blockDim.x + threadIdx.x;

   data [tid] = inData [i];        // load into shared memeory
```

```
   __syncthreads ();

   for ( int s = blockDim.x / 2; s > 0; s >>= 1 )
   {
      if ( tid < s )
        data [index] += data [index + s];

      __syncthreads ();
   }

   if ( tid == 0 )                // write result of block reduction
     outData [blockIdx.x] = data [0];
}
```

Хотя мы заметно сократили число конфликтов банков в shared-памяти, но в результате получили, что на первой итерации цикла по s половина всех нитей простаивает.

Хотя с точки зрения ветвления в этом никаких проблем нет (при большом размере блока, кратном 64, у нас все простаивающие нити будут собраны в warp'ы), но все равно имеется неэффективность, которую хочется удалить.

Хочется, чтобы уже на первой итерации цикла все нити были загружены. Для этого необходимо уменьшить вдвое количество блоков, но при этом каждому блоку выделить вдвое больше слов. При этом самое первое суммирование может быть выполнено сразу же при загрузке данных в shared-память (т.е. мы не увеличиваем требуемый объем shared-памяти).

```
__global__ void reduce4 ( int * inData, int * outData )
{
   __shared__ int data [BLOCK_SIZE];
   int   tid = threadIdx.x;
   int      i   = 2 * blockIdx.x * blockDim.x + threadIdx.x;

   data [tid] = inData [i] + inData [i+blockDim.x];      // load into shared memeory

   __syncthreads ();

   for ( int s = blockDim.x / 2; s > 0; s >>= 1 )
   {
      if ( tid < s )
        data [tid] += data [tid + s];

      __syncthreads ();
   }

   if ( tid == 0 )                              // write result of block reduction
      outData [blockIdx.x] = data [0];
}
```

```
int main ( int argc, char *  argv [] )
{
   int numBytes = N * sizeof ( int );
   int n        = N;
   int i        = 0;
   int sum      = 0;

                        // allocate host memory
   int * a = new int [N];
   int * b = new int [N];

                        // init with random values
   for ( i = 0; i < N; i++ )
   {
      a [i] = 1; //(rand () & 0xFF) - 127;
      sum  += a [i];
   }

                        // allocate device memory
   int       * adev [2] = { NULL, NULL };
   cudaEvent_t start, stop;
   float     gpuTime = 0.0f;

   cudaMalloc ( (void**)&adev [0], numBytes );
   cudaMalloc ( (void**)&adev [1], numBytes );

                        // create cuda event handles
   cudaEventCreate ( &start );
   cudaEventCreate ( &stop );

                        // asynchronously issue work to the GPU (all to stream 0)
   cudaEventRecord ( start, 0 );
   cudaMemcpy      ( adev [0], a, numBytes, cudaMemcpyHostToDevice );

   for ( i = 0; n >= BLOCK_SIZE; n /= (2*BLOCK_SIZE), i ^= 1 )
         {
                        // set kernel launch configuration
      dim3 dimBlock ( BLOCK_SIZE, 1, 1 );
      dim3 dimGrid  ( n / (2*dimBlock.x), 1, 1 );

      reduce4<<<dimGrid, dimBlock>>> ( adev [i], adev [i^1] );
   }

   cudaMemcpy      ( b, adev [i], 4*N, cudaMemcpyDeviceToHost );
   cudaEventRecord ( stop, 0 );

   cudaEventSynchronize ( stop );
   cudaEventElapsedTime ( &gpuTime, start, stop );

   for ( i = 1; i < n; i++ )
      b [0] += b [i];
```

```
                        // print the cpu and gpu times
    printf ( "time spent executing by the GPU: %.2f milliseconds\n", gpuTime );
    printf ( "CPU sum %d, CUDA sum %d, N = %d\n", sum, b [0], N );

                        // release resources
    cudaEventDestroy ( start );
    cudaEventDestroy ( stop  );
    cudaFree         ( adev [0] );
    cudaFree         ( adev [1] );

    delete a;
    delete b;

    return 0;
}
```

В качестве заключительной оптимизации заметим, что при s<=32 у нас в каждом блоке останется всего по одному warp'у, поэтому синхронизация уже не нужна и проверка tid<s также не нужна (она все равно ничего в этом случае не делает). Поэтому развернем цикл для s<=32:

```
__global__ void reduce5 ( int * inData, int * outData )
{
    __shared__ int data [BLOCK_SIZE];
    int   tid = threadIdx.x;
    int   i   = 2 * blockIdx.x * blockDim.x + threadIdx.x;

    data [tid] = inData [i] + inData [i+blockDim.x];     // load into shared memeory

    __syncthreads ();

    for ( int s = blockDim.x / 2; s > 32; s >>= 1 )
    {
        if ( tid < s )
            data [tid] += data [tid + s];

        __syncthreads ();
    }

    if ( tid < 32 )                             // unroll last iterations
    {
        data [tid] += data [tid + 32];
        data [tid] += data [tid + 16];
        data [tid] += data [tid +  8];
        data [tid] += data [tid +  4];
        data [tid] += data [tid +  2];
        data [tid] += data [tid +  1];
    }

    if ( tid == 0 )                             // write result of block reduction
        outData [blockIdx.x] = data [0];
}
```

В результате выполнения такого ядра над входным массивом, в выходном массиве для каждого блока появится сумма всех элементов данного блока. Тем самым мы получили новый массив, который также следует суммировать аналогичным способом, до тем пор, пока мы не получим окончательный массив размера, меньшего чем dimBlock.x. Элементы этого массива можно просуммировать уже на CPU.

В таблице 3.3 приведены быстродействия для всех рассмотренных подходов.

*Быстродействие для различных типов редукции* *Табл. 3.3*

| Вариант алгоритма | Время выполнения (в миллисекундах) |
|---|---|
| reduce1 | 19.09 |
| reduce2 | 11.91 |
| reduce3 | 10.62 |
| reduce4 | 9.10 |
| reduce5 | 8.67 |

## *Лабораторное задание*

Необходимо написать программу согласно варианту, при этом реализовать 2 функции: одну для выполнения с использованием глобальной памяти, вторую для выполнения с использованием разделяемой памяти. Затем сравнить результаты (возвращаемые значения) и скорость работы, задав большой размер матрицы (Size=1000). Массивы должны быть типа float.

Выполнить разработанную задачу для случая выравнивания и без выравнивания. Показать реально полученную разницу во времени.

**Варианты заданий**

1. Нахождение скалярного произведения векторов (протестировать для Size=1000).
2. Нахождение суммы для каждой строки матрицы.
3. Вычисление среднего арифметического строк (столбцов) матрицы.
4. Выбор максимального элемента для каждой строки матрицы.
5. Нахождение суммы квадратов элементов строк матрицы для каждой строки матрицы.
6. Выбор минимального элемента для каждой строки матрицы.