Лабораторная работа №4

Построение гистограмм

Одной из часто встречающихся операций является построение гистограмм. Пусть есть массив элементов a_o, a_1, ..., a_{n-1} и некоторый критерий, позволяющий разделить все элементы на k групп (фактически просто сопоставить каждому элементу целое число от 0 до k - 1).

Тогда гистограммой для исходного массива будет называться массив C_0, C_1, ..., C_{k-1}, каждый элемент которого C_i равен числу элементов исходного массива при надлежащего i-й группе. Подобная задача часто встречается в обработке изображений и в ряде других задач. Далее для простоты будем считать, что входной массив содержит значения от 0 до 255, значение каждого байта и является номером группы. В этом случае гистограмма будет просто массивом из 256 счетчиков (целых чисел).

Данная задача легко реализуется следующим фрагментом последовательного кода.

```
for (int i = 0; i < k; i++ ) C [ i ] = 0;

for (int i = 0; i < n; i++ ) C [a[i]]++;
```

Рассмотрим теперь, каким образом можно реализовать вычисление гистограммы большого массива на CUDA. Для начала разобьем входной массив на части, каждую часть будет обрабатывать отдельный блок.

Поскольку каждая из нитей будет постоянно изменять значение счетчиков, то размещать их в глобальной памяти будет очень неэффективно, гораздо удобнее создать для каждого блока свой экземпляр гистограммы и разместить его в разделяемой памяти.

Однако тут возникает следующая проблема – любая из нитей любого блока может в любой момент времени увеличить значение любого из 256 счетчиков. Поэтому необходимо предусмотреть некоторый механизм, обеспечивающий атомарность доступа к массиву счетчиков (в противном случае попытка увеличить значение одного и того же счетчика сразу несколькими нитями будет выполнена некорректно).

Простейшим способом для обеспечения атомарности доступа к счетчикам будет использование атомарных операций (например, atomicAdd). Однако атомарные операции для величин в разделяемой памяти доступны лишь для GPU с compute capability 1.2 и выше.

Рассмотрим сначала один частный случай, позволяющий полностью избавиться от необходимости обеспечения атомарности доступа. Это происходит в случае, когда число групп равно 64 (или меньше), то есть мы рассматриваем только старшие 6 бит из каждого байта для его классификации.

Тогда если для каждого счетчика отвести по одному байту, то размер одной гистограммы составит Всего 64 байта. Таким образом, в разделяемой памяти мультипроцессора можно будет разместить до 256 таких гистограмм. Поэтому если сделать размер блока 64 нити, то можно

каждой нити выделить свой экземпляр гистограммы, и при этом можно будет запустить до четырех блоков на одном мультипроцессоре.

Поскольку размер счетчика всего 1 байт, то блок может за раз обработать не более 255 байт данных.

Счетчики в разделяемой памяти можно организовать двумя способами:

- для каждой нити есть свои 64 подряд идущих байта счетчиков;
- для каждого 6-битового значения есть 64 счетчика (по одному под каждую нить блока).

Фактически речь идет о том, каким образом лучше вычислять положение внутри выделенного массива счетчика для нити с номером tid и со значением данных data.

Посмотрим на эффективность данного подхода с точки зрения конфликта банков. Поскольку смещение счетчика задается формулой tid * 64 + data, то видно, что этот счетчик расположен в банке ((tid * 64 + data) / 4) % 16 = (data >> 2) % 16 .

Таким образом, наличие или отсутствие конфликтов банков полностью определяется входными данными; если среди данных очень много близких значений, то мы можем получить конфликт высокого порядка.

Оценим с этой же точки зрения второй вариант: когда гистограмма нити – это не набор подряд идущих байт, то есть смещение определяется формулой tid +64 * data. В этом случае номер банка будет задаваться следующим выражением: ((tid+64*data)/4)%16=(tid>>2)%16.

Тогда конфликты вообще не зависят от входных данных, однако использование в качестве параметра tid величины threadldx.x ведет к конфликту банков 4-го порядка номер банка будет определяться битами 2..5 величины threadIdx.x.

Однако если в качестве tid взять не сам threadIdx.x, а величину, получающуюся из нее перестановкой битов, то можно полностью избавиться от всех конфликтов по банкам памяти. Для этого разобьем биты threadIdx.x (а их всего 6, поскольку число нитей на блоке равно 64) на две группы младшие два бита и старшие четыре бита. После этого построим tid, просто переставив эти две группы битов местами:

tid = (threadIdx.x>>4) | ((threadIdx.x & 0x0F)<<2);

При таком выборе нить будет выполнять запись в банк threadIdx.x&0x0F.

Ниже приводится соответствующий листинг программы на CUDA.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#define     NUM_BINS    64
#define     N           (64*255*64)
typedef unsigned int uint;
typedef unsigned char uchar;
```

```
inline __ device__ void addByte ( uchar * base, uint data )
{       base[64*data]++;                }



inline __ device__ void addWord ( uchar * base, uint data )
{                       // Используем только 6 старших бит из каждого байта.
        addByte (base, (data >> 2) & 0x3FU );
        addByte ( base, (data >> 10) & 0x3FU );
        addByte ( base, (data >> 18) & 0x3FU );
        addByte ( base, (data >> 26) & 0x3FU );



__global__ void histogram64Kernel (uint * partialHistograms, uint * data, uint dataCount)
{
        int tid = (threadIdx.x >> 4) | ((threadIldx.x & 0x0F) << 2);
                                        // Массив счетчиков в разделяемой памяти.
        __shared__ uchar hist [64*64];
        uchar * base = hist + tid;
                                        // Обнулить счетчики
        #pragma unroll
        for ( int i = 0; i < 64 / 4; i++ )
                ((uint *)hist) [threadIdx.x + i * 64] = 0;
        __syncthreads ();
        for ( uint pos = blockIdx.x*blockDim.x + threadIdx.x; pos < dataCount;
                                                pos += blockDim.x*gridDim.x)
        {       uint d = data [pos];
                addWord ( base, d );
        }
                                // Объединить гистограммы для отдельных нитей
        __syncthreads ();
        uint sum = 0;
        uint pos = 0;
        base = hist + threadIdx.x * 64;
        for ( int i = 0; i < 64; i++, pos += 64 )           sum += base [i];
        partialHistograms [blockIdx.x * 64 + threadIdx.x] = sum;
}

#define MERGE_THREADBLOCK_SIZE 256
```

```
__global__ void mergeHistogram64Kernel (uint * histogram, uint * partialHistograms,
                                                        uint histogramCount )
{       __shared__ uint data [MERGE_THREADBLOCK_SIZE];
        uint sum = 0;
        for ( uint i = threadIdx.x; i < histogramCount; i += MERGE_THREADBLOCK_SIZE )
                sum += partialHistograms [blockIdx.x + i * 64];
        data [threadIdx.x] = sum;
        for ( uint stride = MERGE_THREADBLOCK_SIZE / 2; stride > 0; stride >>= 1 )
                {
                        __syncthreads ();
                        if ( threadIdx.x < stride )
                                data [threadIdx.x] += data [threadIdx.x + stride];
                }
        if ( threadIdx.x == 0 )         histogram [blockIdx.x] ==data [0];
}


void histogramm ( uint * data, uint * histogram, int byteCount )
{
        // Число partialHistogram –  это число блоков для запуска ядра histogram64Kernel
        uint sz = 64 * 255;
        uint histogramCount = (byteCount/4 + sz - 1) / sz;      //(byteCount + 254) / 255;
        uint * partialHistograms;

        cudaMalloc ( (void **) &partialHistograms, histogramCount * 64 * sizeof (uint ) );
        histogram64Kernel<<<histogramCount, 64>>> ( partialHistograms, data,
                                                        byteCount / sizeof (uint) );
        mergeHistogram64Kernel<<<NUM_BINS, MERGE_THREADBLOCK_SIZE>>> (histogram,
                                                        partialHistograms, histogramCount );

        cudaFree ( partialHistograms );
}

void randomInit ( uint * a, int n, uint * h )
{
        for ( int i = 0; i < n; i++ )
                {       uchar b1 rand ( ) & 0xFF;
                        uchar b2 rand ( ) & 0xFF;
                        uchar b3 rand ( ) & 0xFF;
                        uchar b4 rand ( ) & 0xFF;
                        a [ i] =b1 | (b2 << 8) | (b3 << 16) | (b4 << 24);
                        h [(a[i] >> 2 ) & 0x3F]++;
                        h [(a[i] >> 10) & 0x3F]++;
                        h [(a[i] >> 18) & 0x3F]++;
                        h [(a[i] >> 26) & 0x3F]++;
                }
}
```

```
int main ( int argc, char * argv [] )
{
        uint * a          =       new uint [N];
        uint * hDev       =       NULL;
        uint * aDev       =        NULL;
        uint      h       [NUM BINS];
        uint      hHost   [NUMBINS];
        cudaEvent t       start, stop;
        float             gpuTime = 0.0f;

        memset            (hHost, 0, sizeof ( hHost ) );
        randomInit        (a, N, hHost );

        cudaEventCreate         ( &start );
        cudaEventCreate         ( &stop );

        cudaEventRecord (start, 0);

        cudaMalloc        ( (void **) &aDev, N * sizeof ( uint ) );
        cudaMalloc        ( (void **) &hDev, NUM_BINS * sizeof ( uint ) );
        cudaMemcpy        (aDev, a, N * sizeof T uint ), cudaMemcpyHostToDevice );

        histogram         (aDev, hDev, N*4 );

        cudaMemcpy        ( h, hDev, NUM BINS * sizeof ( uint ), cudaMemcpyDeviceToHost );
        cudaFree          ( aDev );
        cudaFree          ( hDev );

        cudaEventRecord ( stop, 0);
        cudaEventSynchronize ( stop );

        cudaEventElapsedTime ( &gpuTime, start, stop );

        printf ( "Elapsed time: %,2f\n", gpuTime );

        for ( int i = 0; i < NUM_BINS; i++ )
                if ( h [i] != hHost [i] )
                        printf ( "Diff at %d  %d, %d\n", i, h [i], hHost [i] );
        delete a;
        return 0;
}
```

Для общего случая (классификация по 256 группам) такой подход уже не сработает, однако есть очень красивый прием, позволяющий и в этом случае полностью обойтись без использования атомарных операций. Самым простым вариантом было бы выделить каждой нити свою таблицу счетчиков (гистограмму), однако этот вариант, очевидно, не подходит из-за слишком большого объема требуемой разделяемой памяти (фактически нужно каждой нити выделить 256 x 4 = 1 Кбайт разделяемой памяти).

Поскольку выделить каждой нити свою гистограмму невозможно, попробуем разделить все нити блока на группы и выделить свою гистограмму в разделяемой памяти для такой группы нитей. Однако на самом деле подобное деление нитей на группы уже есть – все нити блока уже

поделены на warp'ы. При этом подобное деление имеет очень большой плюс – все нити warp'а физически выполняются одновременно, что заметно упрощает обеспечение атомарности доступа.

Для обеспечения атомарности в пределах одного warp'а можно воспользоваться следующим свойством: если несколько нитей warp'а одновременно производят запись по одному и тому же адресу, то все эти записи будут выполнены последовательно одна за другой, и в памяти останется последнее записанное значение. При этом порядок, в котором могут произойти эти записи, заранее не известен.

Поскольку нам нужно точно знать, что каждая нить выполнила свое увеличение счетчика на единицу, то можно зарезервировать старший байт счетчика под уникальный идентификатор нити внутри warp'а (threadIdx.x & 0x1F). Тогда при записи каждая нить записывает значение со своим идентификатором, а после записи каждая нить читает записанное значение и, сравнивая идентификатор значения со своим идентификатором, проверяет, прошла для нее операция или нет. Ниже приводится функция, которая и реализует подобное гарантированное атомарное увеличение счетчиков.

```
inline __device__ void addByte ( volatile uint * warpHist, uint data, uint threadTag )
{
        uint count;
        do
          {// Прочесть текущее значение счетчика и снять идентификатор нити.
              count   =       warpHist [data] & TAG_MASK;

                      // Увеличить его на единицу и поставить свой идентификатор.
              count   =       threadTag | (count + 1);
                      // Осуществить запись.
              warpHist [data] = count;
          }               // Проверить, прошла ли запись этой нитью.
            while ( warpHist [data] ! = count );
}
```

Каждая нить читает текущее значение счетчика, снимает идентификатор записавшей его нити и увеличивает на единицу. После этого нить добавляет к полученному значению свой идентификатор и записывает полученное значение.

Дальше нить читает только что записанное значение и сравнивает идентификатор из него со своим идентификатором. Если они совпадают, значит, данная нить успешно произвела операцию, в противном случае ей потребуется еще как минимум одна попытка.

Если каждая из 32 нитей увеличивает значение своего счетчика (то есть, нет конфликтов), то все проверки пройдут успешно, и управление вернется из функции после первой же итерации, то есть в этом случае накладные расходы минимальны.

Пусть есть всего две нити (из 32), которые пытаются увеличить значение одного и того же счетчика. Тогда на первой итерации цикла каждая из них попытается записать в счетчик одно и то же значение (на единицу больше текущего), но со своим идентификатором. В результате в памяти останется результат записи одной из этих нитей.

Далее каждая нить читает записанное значение (обратите внимание на использование ключевого слова volatile, сообщающего компилятору, что элементы массива могут быть изменены совершенно неожиданно, и если сразу после записи следует чтение по тому же адресу, то не нужно оптимизировать это, а следует выполнить и запись, и чтение).

У одной из нитей идентификатор обязательно совпадет, и она выйдет из цикла. В результате в цикле останется только одна нить, которая опять прочтет значение, увеличит его на единицу и запишет со своим идентификатором. Поскольку другая нить уже вышла из цикла, то это будет единственная запись по данному адресу, и она успешно завершится.

Тем самым если у нас имеет место конфликт 2-го порядка, то нужно две итерации цикла. Можно показать, что число итераций данного цикла увеличит значение одного и того же счетчика.

Таким образом, ядро находит гистограммы для каждого warp'а. Потом ядро суммирует гистограммы отдельных warp'ов и записывает полученную гистограмму в глобальную память.

В результате каждый блок создает в глобальной памяти отдельную гистограмму для своей части массива. Чтобы свести все эти гистограммы вместе в одну, нужно отдельно ядро, использующее редукцию.

Ниже приводится полный листинг программы, демонстрирующий нахождение гистограммы для массива байт. При этом байты передаются как массив 32-битовых целых, каждое целое задает четыре байта.

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

typedef unsigned int uint;
typedef unsigned char uchar;

#define     N       (6*1024*1024)       // Размер массива.

#define     LOG2_WARP_SIZE      5       // Логарифм размера warp'а по основанию 2
#define     WARP_SIZE           32      // Размер warp'а
#define     TAG_MASK    0x07FFFFFU      // Маска для снятия идентификатора нити.
#define     NUM_BINS            256     // Число счетчиков в гистограмме.
#define     NUM_WARPS           6       // Число warp'ов в блоке.
#define     MERGE_THREADBLOCK_SIZE  256

inline __device__ void addByte ( volatile uint * warpHist, uint data, uint threadTag)
{
    uint count;
    do
      {
        // Прочесть текущее значение счетчика и снять иденификатор нити.
        count = warpHist [data] & TAG_MASK;
        // Увеличить его на единицу и поставить свой идентификатор.
        count = threadTag | (count + 1);
        // Осуществить запись.
        warpHist [data] = count;
    }
    while ( warpHist [data] ! = count ); / / Проверить, прошла ли запись этой нитью.
}

inline __device__ void addWord ( volatile uint * warpHist, uint data, uint tag )
{
    addByte warpHist, (data >> 0) & 0xFFU, tag ) ;
    addByte warpHist, (data >> 8 ) & 0xFFU, tag ) ;
    addByte warpHist, (data >> 16) & 0xFFU, tag ) ;
    addByte warpHist, (data >> 24) & 0xFFU, tag ) ;
}

__global__ void histogramKernel ( uint * partialHistograms, uint * data, uint dataCount )
{
                                        // Своя гистограмма на каждый warp.
    __shared__ uint hist [NUM_BINS * NUM_WARPS];
```

```
        uint *warpHist hist+ (threadIdx.x >> LOG2_WARP_SIZE) * NUM_BINS;
                                                // Очистить счетчики гистограмм.
        #pragma unroll
        for ( uint i = 0; i < NUM_BINS / WARP_SIZE; i++ )
                hist [threadIdx.x + i *NUM_WARPS *WARP_SIZE] = 0;
                                                // Получить id для данной нити.
        uint tag = threadIdx.x << (32 - LOG2_WARP_SIZE);
        __syncthreads();
                        // Построить гистограммы по заданному набору элементов.
        for ( uint pos = blockIdx.x * blockDim.x + threadIldx.x;  pos < dataCount;
                                                        pos += blockDim.x * gridDim.x )
                {       uint d = data [pos];
                        addWord ( warpHist, d, tag );
                }
                __syncthreads();
                                                // Объединить гистограммы данного блока
                                                // и записать результат в глобальную память.
                                                // 192 нити суммируют данные
                                                // по 256 элементам гистограммы.
        for ( uint bin = threadIdx.x; bin < NUM_BINS; bin += NUM_WARPS * WARP_SIZE )
        {
                uint sum = 0;
                for ( uint i = 0; i < NUM_WARPS; i++ )
                                sum += hist [bin + i * NUM_BINS] & TAG_MASK;
                partialHistograms [blockIdx.x * NUM_BINS + bin] sum;
        }
}

//
// Объединить гистограммы, один блок на каждый из NUM_BINS элементов.
//
__global__ void mergeHistogramKernel ( uint * outHistogram, uint * partialHistograms,
                                                                uint histogramCount )
{
        uint sum = 0;
        for ( uint i = threadIdx.x; i < histogramCount; i += 256 )
                sum += partialHistograms [blockIdx.x + i * NUM_BINS];
        __shared__  uint data [NUM_BINS];
        data [threadIdx.x] = sum;
        for ( uint stride = NUM_BINS / 2; stride > 0; stride >>= 1)
        {
                __syncthreads();
                if ( threadIdx.x < stride ) data [threadIdx.x] += data [threadIdx.x + stride];
        }
        if ( threadIdx.x == 0 ) outHistogram [blockldx.x] = data [0];
}

void histogram ( uint * histogram, void * dataDev, uint byteCount )
{
        assert( byteCount % 4 == 0 );
        int n = byteCount / 4;
        int numBlocks = n / (NUM_WARPS * WARP_SIZE) ;
```

```
        int numPartials = 240;
        uint * partialHistograms = NULL;
                                // Выделить память под гистограммы блоков.
        cudaMalloc ( (void **) &partialHistograms, numpartials * NUMBINS * sizeof ( uint ) );
                                // Построить гистограммы для каждого блока.
        histogramKernel<<<dim3 ( numPartials ), dim3 (NUM_WARPS * WARP_SIZE) >>>
                                                (partialHistograms, (uint *) dataDev, n);
                                // Объединить гистограммы отдельных блоков вместе.
        mergeHistogramKernel<<<dim3(NUM_BINS), dim3(256)>>> ( histogram,
                                                partialHistograms, numPartials );
                                // Освободить выделенную память.
        cudaFree ( partialHistograrns );
                                // Заполнить массив случайными байтами

void randornInit ( uint * a, int n, uint * h )
{
        for ( int i = 0; i < n; i++ )
        {
                uchar b1 = rand ( ) & 0xFF;
                uchar b2 = rand ( ) & 0xFF;
                uchar b3 = rand ( ) & 0xFF;
                uchar b4 = rand ( ) & 0xFF;

                a [i] = b1 | (b2 << 8) | (b3 << 16) | (b4 << 24);
                h [b1]++;
                h [b2]++;
                h [b3]++;
                h [b4]++;
        }
}


int main ( int argc, char * argv [] )
{
        uint    * a     =       new uint [N];
        uint    * hDev  =       NULL;
        uint    * aDev  =       NULL;
        uint    h [NUM_BINS];
        uint    hHost [NUM_BINS];
        cudaEvent_t     start, stop;
        float gpuTime   =       0.0f;

        memset (hHost, 0, sizeof ( hHost ) );
        randomInit (a, N, hHost );

        cudaEventCreate ( &start );
        cudaEventCreate ( &stop );
        cudaEventRecord ( start, 0 );

        cudaMalloc  (void **) &aDev, N * sizeof ( uint ) );
        cudaMalloc (void **) &hDev, NUM_BINS * sizeof ( uint ) );
        cudaMemcpy (aDev, a, N * sizeof T uint ), cudaMemcpyHostToDevice );
```

```
        histogram ( hDev, aDev, 4 * N );

        cudaMerncpy (h, hDev, NUM BINS * sizeof ( uint ), cudaMemcpyDeviceToHost );
        cudaFree ( aDev );
        cudaFree ( hDev );

        cudaEventRecord (stop, 0);
        cudaEventSynchronize (stop );
        cudaEventElapsedTime ( &gpuTime, start, stop );

        printf ( "Elapsed time: %.2f\n", gpuTime );

        for ( int i = 0; i < NUM_BINS; i++ )
                if ( h [i] != hHost [i] )
                        printf ( "Diff at %d  %d, %d\n", i, h [i], hHost [i] );
        delete a;
        return 0;
}
```

Лабораторное задание.

1. Для своего варианта распределения построить гистограмму.
2. Определить, есть ли конфликты банков для обрабатываемого набора данных.
3. *Изобразить графическое представление гистограммы. Это задание предполагается выполнить в качестве 8 лабораторной работы, но если получится сделать сразу, будет просто прекрасно.

№ варианта	Закон распределения
1, 6	Нормальное (кривая Гаусса)
2, 7	Биномиальное (формула Бернулли)
3, 8	Пуассона (формула Пуассона)
4, 9	Равномерное
5, 10	Показательное
	Вспоминайте, какие еще бывают!!!! ☺