Лабораторная работа №5

Сортировка на GPU

Еще одной из часто применяющихся операций к массивам является операция сортировки. Даже для последовательного кода реализация эффективной сортировки может представлять трудности, в случае параллельной сортировки ситуация становится заметно сложнее. Поэтому вначале будет рассмотрен классический способ параллельной сортировки – битоническая сортировка, после этого будет рассмотрена работа поразрядной (radix) сортировки и показано использование готовой функции сортировки, входящей в состав библиотеки CUDPP.

Битоническая сортировка

Классическим примером параллельной сортировки (сортировочной сети) является битоническая сортировка. В ее основе лежит операция B_n (называемая полуочистителем) над массивом, параллельно упорядочивающая элементы пар x_i и $x_i+n/2$ (рис. 5.1).

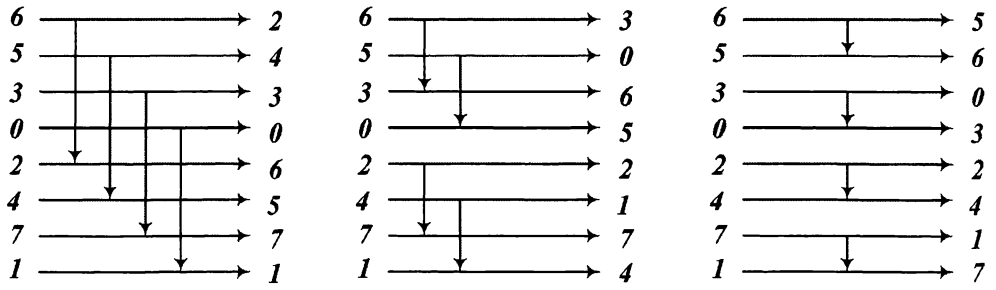


Рис. 5.1. Примеры полуочистителей B_8 , B_4 и B_2

Легко видеть, что полуочиститель может упорядочивать элементы пар как по возрастанию, так и по убыванию.

Битоническая сортировка основана на понятии битонической последовательности и следующем утверждении: если набор полуочистителей правильно сортирует произвольную последовательность нулей и единиц, то он правильно сортирует произвольную последовательность.

Последовательность a_0, a_1, ..., a_{n-1} называется битонической, если она или состоит из двух монотонных частей (то есть либо сначала возрастает, а потом убывает, либо наоборот), или получается из такой последовательности путем циклического сдвига. Например, последовательность 5, 7, 6, 4, 2, 1, 3 является битонической, так как получена из 1, 3, 5, 7, 6, 4, 2 путем циклического сдвига влево на два элемента.

Можно доказать, что если применить полуочиститель B_n к битонической последовательности $a_0, a_1, ..., a_{n-1}$, то получившаяся последовательность будет обладать следующими свойствами:

- обе ее половины также будут битоническими;
- любой элемент первой половины будет не больше любого элемента второй половины;
- хотя бы одна из половин является монотонной.

Применим к битонической последовательности $a_0, a_1, ..., a_{n-1}$ полуочиститель B_n. В результате получим две последовательности длиной $n/2$, каждая из которых будет битонической, и каждый элемент первой будет не больше каждого элемента из второй. Далее применим к каждой из получившихся половин полуочиститель $B_{n/2}$. В результате мы получим уже четыре битонические подпоследовательности длины $n/4$. Применим к каждой из них полуочиститель $B_{n/4}$ и будем продолжать этот процесс до тех пор, пока не придем к $n/2$ последовательностей из двух элементов. Применив к каждой из них полуочиститель B_2, мы отсортируем эти последовательности. Поскольку все эти последовательности уже упорядочены правильно, то, объединив их вместе, получим отсортированную последовательность.

Таким образом, последовательное применение полуочистителей B_n , $B_{n/2}$, ..., B_2 сортирует произвольную битоническую последовательность. Эта операция называется битоническим слиянием и обозначается M_n .

Рассмотрим теперь как можно отсортировать произвольную последовательность при помощи полуочистителей и битонического слияния.

Пусть есть последовательность из 8 элементов $a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7$. Применим к этой последовательности полуочиститель B_2 таким образом, чтобы в соседних парах порядок сортировки был противоположен (рис. 5.2.).

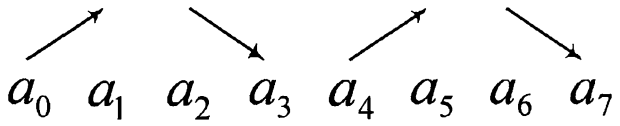


Рис. 5.2. Результат применения полуочистителя B_2 с чередующимся порядком упорядочивания к последовательности из восьми элементов.

Как видно, первые четыре элемента получившейся последовательности образуют битоническую последовательность. Аналогично последние четыре элемента также образуют битоническую последовательность. Поэтому каждая из этих половин может быть отсортирована битоническим слиянием, однако проведем слияние таким образом, чтобы порядок сортировки в половинах был противоположным. В результате обе половины образуют вместе битоническую последовательность длины 8, и ее можно отсортировать.

Собственно, в этом и заключается битоническая сортировка для сортировки последовательности из *n* элементов она разбивается пополам и каждая из половин сортируется в своем направлении. После этого полученная битоническая последовательность сортируется битоническим слиянием.

При этом битоническая сортировка распадается на отдельные шаги, на каждом из которых применяется какойлибо полуочиститель. Всего для битонической сортировки последовательности из *n* элементов требуется $\log_2 n$ x($\log_2 n$+1)/2 шагов.

Ниже приводится ядро, которое осуществляет битоническую сортировку набора элементов в разделяемой памяти, фактически каждый блок берет свои 1024 элемента, загружает их в разделяемую память и там сортирует.

```
#define BLOCK_SIZE 512   // Размер блока.

                                // Отдельный компаратор – сравнивает пару ключей
                                // и производит обмен соответствующих элементов и
                                // ключей для обеспечения заданного порядка.
__device__ void Comparator (uint& keyA, uint& valA, uint& keyB, uint& valB, uint dir)
{
        uint t;
        if ( (keyA > keyB) == dir )    // Поменять местами (keyA, valA) и (keyB, valB)
        {       t = keyA; keyA = keyB; keyB = t;
                t = valA; valA = valB; valB = t;
        }
}

__global__ void bitonicSortShared ( uint * dstKey, uint * dstVal, uint * srcKey,
                                    uint * ssrcVal, uint arrayLength, uint dir )
{       __shared__ uint sk [BLOCK_SIZE * 2];
        __shared__ uint sv [BLOCK_SIZE * 2];
        int index = blockId.x * BLOCK_SIZE * 2 + threadId.x.x;
        sk [threadldx.x ]       =       srcKey [index];
        sv [threadldx.x ]       =       srcVal [index];
        sk [threadldx.x + BLOCK_SIZE]   =       srcKey [index + BLOCK_SIZE];
        sv [threadldx.x + BLOCK_SIZE]   =       srcVal [index + BLOCK_SIZE];
        for ( uint size = 2; size < arrayLength; size <<= 1 )
        {
                                                // Битоническое слияние
                uint ddd = dir ( (threadIdx.x & (size / 2)) != 0 );
                for ( uint stride = size >> 1; stride > 0; stride >>= 1)
                {
                  __syncthreads ();
                  uint pos = 2 * threadIdx.x - (threadIdx.x & (stride - 1));
                  Comparator( sk [pos], sv [pos], sk [pos+stride], sv [pos+stride], ddd );
                }
        }
                                                // Последний шаг - битоническое слияние.
        for ( uint stride = arrayLength >> 1; stride > 0; stride >>= 1 )
        {
```

```
            __syncthreads ();
            uint pos = 2 * threadIdx.x - (threadIdx.x & (stride - 1));
            Comparator ( sk [pos], sv [pos], sk [pos + stride], sv [pos + stride], dir );
        }
        __syncthreads ();
        dstKey [index]          =       sk [threadIdx.x] ;
        dstVal [index ]         =       sv [threadIdx.x] ;
        dstKey [index + BLOCK_SIZE]     =       sk[threadIdx.x + BLOCK_SIZE];
        dstVal [index + BLOCK_SIZE]     =       sv [threadIdx.x + BLOCK=SIZE];
}
```

Данный алгоритм можно перенести и для сортировки очень большой последовательности, однако при этом придется отказаться от использования разделяемой памяти, и на каждом шаге будет очень много операций с глобальной памятью, что плохо сказывается на быстродействии.

Поразрядная сортировка

Еще одним вариантом сортировки, который хорошо ложится на архитектуру CUDA, является поразрядная сортировка (radix sort).

Пусть имеется массив из 32-битовых беззнаковых целых чисел a_0, a_1, ..., a_{n-1}. Тогда можно вместо сортировки элементов по всей совокупности бит осуществить несколько сортировок отдельно по каждому из битов.

Если сперва отсортировать этот массив по 0-му биту, затем по 1-му и т. д., заканчивая сортировкой по 31-му биту, то в результате мы получим полностью отсортированный по всей совокупности битов массив.

Рассмотрим поразрядную сортировку для классической реализации, т.е. без использования технологии многопоточности.

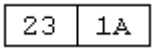
ВНИМАНИЕ! СЛЕДУЮЩИЙ ДАЛЕЕ ТЕКСТ, ВЫДЕЛЕННЫЙ СИНИМ ЦВЕТОМ, ПРИВЕДЕН КАК СПРАВОЧНЫЙ МАТЕРИАЛ, ПРОГРАММЫ, КОТОРЫЕ В НЕМ ПРИСУТСТВУЮТ, СПЕЦИАЛЬНО ОТЛАЖИВАТЬ НЕ НУЖНО, ЕСЛИ В ЭТОМ НЕ ПОЯВИТСЯ НЕОБХОДИМОСТЬ

Поразрядная сортировка беззнаковых целых чисел

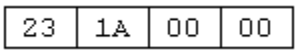
В качестве интересного примера рассмотрим компьютерное представление целых беззнаковых чисел. На хранение каждого из базовых типов выделяется строго определенное количество байт. Как правило, распределение такое:

unsigned char - 1 байт,
unsigned short int - 2 байта,
unsigned long int - 4 байта.

При этом большинство процессоров использует обратный порядок записи: от младшего байта к старшему. Так число типа short int i = 669110 = 0x1A23 непосредственно в памяти хранится так:



Если это число имеет тип long int, то под него выделяется 4 байта: long int i = 669110 = 0x00001A23.



В конце расположены ведущие нули. Таким образом, необходимо провести сортировку от начала числа к концу. Каждый байт может принимать в общем случае 256 значений: m=256.

```
// Создать счетчики.
// data-сортируемый массив, counters-массив для счетчиков, N-число элементов в data
template<class T>
void createCounters(T *data, long *counters, long N) {
   // i-й массив count расположен, начиная с адреса counters+256*i
            memset( counters, 0, 256*sizeof(T)*sizeof(long) );

            uchar *bp = (uchar*)data;
            uchar *dataEnd = (uchar*)(data + N);
            ushort i;

            while ( bp != dataEnd ) {
                        // увеличиваем количество байт со значением *bp
                        // i - текущий массив счетчиков
                        for (i=0; i<sizeof(T); i++)
                                    counters[256*i + *bp++]++;
            }
}

// Функция radixPass принимает в качестве параметров
// номер байта Offset,
// число элементов N,
// исходный массив source,
// массив dest, куда будут записываться числа, отсортированные по байту Offset
// массив счетчиков count, соответствующий текущему проходу.

template<class T>
void radixPass (short Offset, long N, T *source, T *dest, long *count) {
   // временные переменные
            T *sp;
            long s, c, i, *cp;
            uchar *bp;


   // шаг 3
            s = 0;      // временная переменная, хранящая сумму на данный момент
```

```
        cp = count;
        for (i = 256; i > 0; --i, ++cp) {
                c = *cp;
                *cp = s;
                s += c;
        }

        // шаг 4
        bp = (uchar *)source + Offset;
        sp = source;
        for (i = N; i > 0; --i, bp += sizeof(T) , ++sp) {
                cp = count + *bp;
                dest[*cp] = *sp;
                ++(*cp);
        }
}
```

Процедура сортировки заключается в осуществлении sizeof(T) проходов по направлению от младшего байта к старшему.

```
// сортируется массив in из N элементов
// T - любой беззнаковый целый тип
template<class T>
void radixSort (T* &in, long N) {
        T *out = new T[N];
        long *counters = new long[sizeof(T)*256], *count;
        createCounters(in, counters, N);

        for (ushort i=0; i<sizeof(T); i++) {
                count = counters + 256*i;       // count - массив счетчиков для i-го разряда

                if ( count[0] == N ) continue;    // (*** см. ниже)

                radixPass (i, N, in, out, count); // после каждого шага входной и
                swap(in, out);                  // выходной массивы меняются местами
        }
              // по окончании проходов
        delete out;         // вся информация остается во входном массиве.
        delete counters;
}
```

Обратим внимание на то, что если число проходов нечетное (например, T=unsigned char), то из-за перестановки swap() реально уничтожается исходный массив, а указатель *in* приобретает адрес *out*. Об этом следует помнить, используя указатели на сортируемый массив во внешней программе.

Такая организация процедуры также является прототипом для сортировки строк: достаточно сделать проходы radixPass от последней буквы к первой.

Строка, помеченная (***) представляет собой небольшую оптимизацию. Бывает так, что сортируемые типы используются не полностью. Например, в переменных ulong

хранятся числа до 65536 или до 16777216. При этом остается еще один или два старших разряда, которые всегда равны нулю. В переменной count[0] хранится общее количество нулевых байтов на текущей сортируемой позиции, и если оно равно общему количеству чисел, то сортировка по этому разряду не нужна. Так radixSort сама адаптируется к реальному промежутку, в котором лежат числа.

Поразрядная сортировка целых чисел со знаком

Числа со знаком хранятся абсолютно также, за исключением одной важной детали. А именно, если выписать число в двоичном представлении, то первый бит старшего байта является *знаковым*.

Рассмотрим short int i = 669110 = 0x1A23 = 00011010'001000112. Здесь старший байт 0x1A, и его знаковый бит выделен. Он равен нулю, так как число положительное. У отрицательного числа знаковый бит равен 1. При этом все остальные биты числа инвертированы, т.е вместо 0 хранится 1, вместо 1 - ноль. Таким образом, -669110 имеет вид 11100101'110111012.

Внутреннее представление чисел в двоичном виде (байты хранятся в обратном порядке):

Каким образом такое представление влияет на сортировку? Очень просто - все отрицательные числа воспринимаются как очень большие (еще бы, первый бит равен 1) положительные числа.

Если для отрицательных чисел верно |a| > |b|, то, благодаря инвертированным битам, (unsigned)a < (unsigned)b. Поэтому порядок, в котором сортируются такие числа, остается естественным: большее по модулю отрицательное число стоит впереди.

Например, последовательность
-302 -249 1258 2330 -2948 2398 -543 3263
после сортировки станет такой:
1258 2330 2398 3263 -2948 -543 -302 -249.

Как видно, и положительные и отрицательные числа отсортировались правильно. Однако их взаимное расположение необходимо подкорректировать.
Для этого модифицируем последний проход сортировки, работающий со старшими байтами и производящий окончательную расстановку. Все, что нам необходимо - узнать номер первого отрицательного числа n*umNeg,* и заполнять массив dest сначала числами после *numNeg* (отрицательными), а потом – остальными (положительными).

Определить по старшему байту знак числа можно без битовых операций, по результатам сравнения. После шага 1 в *count*[*i*] содержится количество байт, равных *i*. Отрицательных чисел столько же, сколько байт с единичным старшим битом, т.е равных 128...255:

```
long numNeg=0;   // количество отрицательных чисел
for(i=128;i<256;i++) numNeg += count[i];
```

На шаге 3 увеличиваем начальную позицию положительных чисел на numNeg, а отрицательные записываем с начала массива. Это приводит к следующей процедуре:

```
// проход поразрядной сортировки по старшим байтам,
// для целых чисел со знаком Offset = sizeof(T)-1.
template<class T>
void signedRadixLastPass (short Offset, long N, T *source, T *dest, long *count) {
        T *sp;
        long s, c, i, *cp;
        uchar *bp;

        // подсчет количества отрицательных чисел
        long numNeg=0;
        for(i=128;i<256;i++) numNeg += count[i];

        // первые 128 элементов count относятся к положительным числам.
        // отсчитываем номер первого числа, начиная от numNeg
        s = numNeg;
        cp = count;
        for (i = 0; i < 128; ++i, ++cp) {
                c = *cp;
                *cp = s;
                s += c;
        }

        // номера для отрицательных чисел отсчитываем от начала массива
        s = 0;
        cp = count+128;
        for (i = 0; i < 128; ++i, ++cp) {
                c = *cp;
                *cp = s;
                s += c;
        }


        bp = (uchar *)source + Offset;
        sp = source;
        for (i = N; i > 0; --i, bp += sizeof(T) , ++sp) {
                cp = count + *bp;
                dest[*cp] = *sp;
                ++(*cp);
        }
}
```

Соответственным образом изменится и внешняя процедура сортировки. На последнем шаге теперь происходит вызов signedRadixLastPass.

```
template<class T>
void signedRadixSort (T* &in, long N) {
```

```
        T *out = new T[N];
        ushort i;

        long *counters = new long[sizeof(T)*256], *count;
        createCounters(in, counters, N);

        for (i=0; i<sizeof(T)-1; i++) {
                count = counters + 256*i;
                if ( count[0] == N ) continue;
                radixPass (i, N, in, out, count);
                swap(in, out);
        }
        count = counters + 256*i;
        signedRadixLastPass (i, N, in, out, count);

        delete in;
        in = out;
        delete counters;
}
```

Формат IEEE-754

Особенно подходящим местом применения radixSort является компьютерная графика. Есть много координат, которые необходимо отсортировать, причем с наибольшей скоростью. Поэтому было бы очень удобно распространить возможности процедуры на числа с плавающей точкой.

С другой стороны, существует несколько стандартов для представления таких чисел. Наиболее известным следует признать IEEE-754, который используется в IBM PC, Macintosh, Dreamcast и ряде других систем. Зафиксировав формат представления числа, уже можно модифицировать соответствующим образом сортировку.

Любое число можно записать в системе с основанием BASE как $M*BASE^n$, где $|M|<BASE$. Например,

$47110 = 4.7110 * 10^4$, $6.510 = 6.510*10^0$, $-0.003210 = -3.210 * 10^{-3}$. В двоичной системе можно дополнительно сделать первую цифру M равной 1: $101.01_2 = 1.0101_2*2^2$, $-0.000011_2 = -1.1_2*2^{-5}$

Такая запись называется *нормализованной*. M называется *мантиссой*, n - *порядком* числа. Стандарт IEEE-754 предполагает запись числа в виде [знак][порядок][мантисса]. Различным форматам соответствует свой размер каждого из полей. Для чисел одинарной точности (float) это 1 бит под знак, 8 под порядок и 23 на мантиссу. Числа двойной точности (double) имеют порядок из 11 бит, мантиссу из 52 бит. Некоторые процессоры (например, 0x86) поддерживают расширенную точность (long double).

тип/бит	[Знак]	[Порядок]	[Мантисса]	[Смещение]
float	1	8	23	127

double	1	11	52	1023

Рассмотрим числа одинарной точности (float). Как и в целых числах, значение знакового бита 0 означает, что число положительное, значение 1 – число отрицательное. Так как первая цифра мантиссы всегда равна 1, то хранится только ее дробная часть. То есть, если $M=1.01011_2$, то [Мантисса]= 01011_2. Если $M=1.1011_2$, то [Мантисса]=1011_2. Таким образом, в 23 битах хранится мантисса до 24 бит длиной. Чтобы получить из поля [Мантисса] правильное значение *M* необходимо мысленно прибавить к его значению единицу с точкой.

Для удобства представления отрицательного порядка в соответствующем поле реально хранится порядок+смещение. Например, если n=0, то [Порядок]=127. Если n=-7, то [Порядок]=120. Таким образом, его значение всегда неотрицательно.

Таким образом, общая формула по получению обычной записи числа из формата IEEE-754:

N = (-1)[Знак] * 1.[Мантисса] * 2[Порядок]-127 (нормализованная запись, 1)

Для $10.2510 = 1010.01_2 = +1.01001 * 2^3$ соответствующие значения будут [Знак]=0 (+) , [Порядок] = 130(=3+127), [Мантисса] = 010012.

При значениях [Порядок]=1...254 эта система позволяет представлять числа приблизительно от $+-3{,}39*10^{-38}$ до $+-3{,}39*10^{+38}$.

Особым образом обрабатываются [Порядок]=0 и [Порядок]=255.

Если [Порядок]=0, но мантисса отлична от нуля, то перед ее началом необходимо ставить вместо единицы ноль с точкой. Так что формула меняется на

N = (-1)[Знак] * 0.[Мантисса] * 2^{-126} (денормализованная запись, 2)

Это расширяет интервал представления чисел до $+-1{,}18*10^{-38}$. ... $+-3{,}39*10^{+38}$

[Порядок]=0 [Мантисса]=0 обозначают ноль. Возможно существование двух нулей: +0 и -0, которые отличаются битом знака, но при операциях ведут себя одинаково. В этом смысле можно интерпретировать ноль как особое денормализованное число со специальным порядком.

Значение [Порядок]=255(все единицы) зарезервировано для специальных целей, более подробно описанных в стандарте. Такое поле в зависимости от мантиссы обозначает одно из трех специальных "нечисел", которые обозначают как Inf, NaN, Ind, и может появиться в результате ошибок в вычислениях и при переполнении.

Числа двойной точности устроены полностью аналогичным образом, к порядку прибавляется не 127, а 1023. Увеличение длин соответствующих полей позволяет хранить числа от 10-323.3 до 10308.3, значение [Порядок]=2047(все единицы) зарезервировано под "нечисла".

Поразрядная сортировка целых чисел с плавающей точкой

У рассмотренного представления есть одна интересная особенность. А именно, если значение поля [Порядок] у одного числа больше соответствующего значения у другого, то первое число обязательно больше второго. Это следует непосредственно из формул перевода (1) и (2) и верно для неотрицательных чисел.
Если порядки равны, то сравниваются мантиссы. Это полностью соответствует обычной системе для целых чисел, когда сравниваются сначала старшие двоичные цифры, затем младшие.

Таким образом, можно интерпретировать числа стандарта IEEE-754 как целые соответствующей длины:

a > b равносильно (ulong)a > (ulong)b, если a, b > 0.

Неприятности начинаются с отрицательными числами. Первый бит равен единице, так что это число будет, как и в случае беззнаковых целых, очень большим положительным. Больше любого по-настоящему положительного.

Если a>0, b<0, то всегда (ulong)a < (ulong)b

С другой стороны, никакого инвертирования битов, как в случае целых чисел со знаком, не производится. Поэтому большее по модулю отрицательное число дает большее в беззнаковой целочисленной интерпретации.

a > b равносильно (ulong)a < (ulong)b, если a, b < 0.

Если запустить RadixSort на массиве:

-302 -249 1258 2330 -2948 -543 2398 3263,
после сортировки получим: 1258 2330 2398 3263 -249 -302 -543 -2948.

Сначала идут положительные числа, которые отсортировались абсолютно правильно. Затем идут отрицательные, но в обратном порядке!

Все, что необходимо сделать - это передвинуть их на правильное место тем же способом, что и при сортировке целых со знаком, при этом обратив порядок их расположения. Эти изменения можно провести на шаге 3 последнего прохода, работающего со старшими байтами.

Отрицательные числа расположены в обратном порядке, поэтому при вычислении их расположения необходимо идти от *count*[255]=0, соответствующего наименьшему отрицательному числу до *count*[128]. При этом в массив нужно записывать сумму всех чисел после текущего, ***включая*** само текущее.

```
s = count[255] = 0;                    // отрицательные числа располагаются от
нуля
cp = count+254;
for (i = 254; i >= 128; --i, --cp) {
        *cp += s;
        s = *cp;
}
```

Теперь можно написать окончательную реализацию.

```
// Функция для последнего прохода при поразрядной сортировке чисел с плавающей точкой
template <class T>
void floatRadixLastPass (short Offset, long N, T *source, T *dest, long *count) {
	T *sp;
	long s, c, i, *cp;
	uchar *bp;

	long numNeg=0;
	for(i=128;i<256;i++) numNeg += count[i];

	s=numNeg;
	cp = count;
	for (i = 0; i < 128; ++i, ++cp) {
		c = *cp;
		*cp = s;
		s += c;
	}

	// изменения, касающиеся обратного расположения отрицательных чисел.
	s = count[255] = 0;		//
	cp = count+254;			//
	for (i = 254; i >= 128; --i, --cp) {//
		*cp += s;		// остальное - то же, что и в
		s = *cp;		// signedRadixLastPass
	}

	bp = (uchar *)source + Offset;
	sp = source;
	for (i = N; i > 0; --i, bp += sizeof(T) , ++sp) {
		cp = count + *bp;
		if (*bp<128) dest[ (*cp)++ ] = *sp;
		else dest[ --(*cp) ] = *sp;
	}
}

// поразрядная сортировка чисел с плавающей точкой
template<class T>
void floatRadixSort (T* &in, long N) {
	T *out = new T[N];
	ushort i;

	long *counters = new long[sizeof(T)*256], *count;
	createCounters(in, counters, N);

	for (i=0; i<sizeof(T)-1; i++) {
		count = counters + 256*i;
		if ( count[0] == N ) continue;
		radixPass (i, N, in, out,count);
		swap(in, out);
	}
	count = counters + 256*i;
```

```
            floatRadixLastPass (i, N, in, out,count);

            delete in;
            in = out;
            delete counters;
}
```

Для большего удобства можно объединить все полученные сортировки в одну функцию с дополнительным параметром type = {FLOAT, SIGNED, UNSIGNED}, в зависимости от входного типа данных.

Сортировка массива по каждому из битов фактически означает некоторую перестановку элементов местами. В случае если сортировка производится по одному биту, то при помощи операции нахождения префиксных сумм можно сразу получить новые позиции для перестановки элементов массива.

Пусть необходимо отсортировать последовательность по k-му биту. Тогда определим массив $b_0, b_1, ..., b_{n-1}$ следующим образом: $b_i = (a_i >> k)$ & 1.

b:	0	1	1	0	1	0	0	1	1	0	1
s:	0	0	1	2	2	3	3	3	4	5	5	,6

Рис. 5.3. Результат применения операции префиксного суммирования к массиву b.

Далее применим к этому массиву операцию *scan*, в результате которой получим массив частичных сумм $s_0, s_1, ..., s_{n-1}$ и полную сумму s_n всех элементов массива. Тогда новое положение элемента a_i будет равно $i - s_i$, если $b_i = 0$ и s_i+Nz в противном случае, где Nz это число нулей в массиве $b_0, b_1, ..., b_{n-1}$ ($Nz = n - s_n$).

Чтобы не возникало проблем с перестановкой элементов массива $a_0, a_1, ..., a_{n-1}$, проще всего ввести дополнительный массив, куда будут записываться элементы, отсортированные по заданному биту.

В результате сортировка массива целых чисел свелась к 32 довольно простым шагам.

Существуют различные оптимизации данного подхода: так, можно сперва осуществлять сортировку в разделяемой памяти в пределах каждого блока, а только потом уже выполнять глобальную сортировку для каждого бита. Эта сильно повысит coalsescing при записи элементов на каждом проходе.

Еще одним вариантом оптимизации является сортировка не по одному биту за раз, а по четыре. При этом процедура расчета нового положения элемента усложняется, однако сокращается общее число проходов метода. В состав CUDA SDK входит ряд примеров на сортировку, в том числе несколько примеров на поразрядную сортировку.

Использование библиотеки CUDPP

Поскольку написание эффективной сортировки на CUDA является достаточно сложным, то можно воспользоваться уже готовой функцией cudppSort, входящей в библиотеку CUDPP.

```
CUDPPResult cudppSort ( CUDPPHandle plan, void * outData, const void * inData,
                        size_t numElements );
```

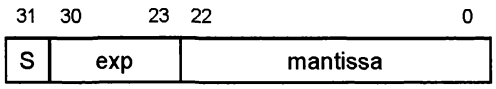
При создании плана в качестве алгоритма используется одно из следующих значений – CUDPP_SORT_RADIX или CUDPP_SORT_RADIX_GLOBAL.

Ниже приводится простой пример, демонстрирующий сортировку массива целых чисел при помощи библиотеки CUDPP.

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

#define             N       (5*1024*1024)       // Количество элементов в массиве.
void randomInit ( int * a, int n )
{
    for ( int i = 0; i < n; i++ )
    a [i] = rand ();
}

int main ( int argc, char * argv [] )
{
    CUDPPconfiguration config;
    CUDPPAlgorithm      algos       []      =       {CUDPP_SORT_RADIX,
CUDPP_SORT_RADIX_GLOBAL };
    CUDPPHandle             plan;
    CUDPPResult             result  =       CUDPP_SUCCESS;
    int                 * array newint [N];
    int                 * arrayDevln    =       NULL;
    int                 * arrayDevOut       =       NULL;
    cudaEvent_t         start, stop;
    float               gpuTime = 0.0f;

    randomlnit ( array, N );
    cudaEventCreate ( &start );
    cudaEventCreate ( &stop );
    cudaEventRecord ( start, 0 );

    cudaMalloc (void **) &arrayDevln, N * sizeof ( int ) );
    cudaMalloc (void **) &arrayDevOut, N * sizeof ( int ) );
    cudaMemcpy (arrayDevIn, array, N * sizeof ( int ), cudaMemcpyHostToDevice );

                            // Создать план для сортировки данных.
    config.datatype         =       CUDPP_INT;
    config.op           =       CUDPP_ADD;
    config.options          =       (CUDPPOption) 0;
```

```
        config.algorithm        =       algos [0];
        result = cudppPlan ( &plan, config, N, 1, 0 );
        if ( result != CUDPP SUCQESS )
        {
                fprintf ( stderr, "Ошибка  создания плана для сортировки\n" );
                return 1;
        }
        cudppSort  ( plan, arrayDevOut, arrayDevIn, N );
        cudaMemcpy ( array, arrayDevOut, N * sizeof ( int ), cudaMemcpyDeviceToHost
);

                                        // Освободить ресурсы.
        cudppDestroyPlan ( plan );
        cudaFree (arrayDevIn );
        cudaFree (arrayDevOut) ;

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

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

                                        // Проверить сортировку.
        for ( int i = 1; i < N; i++ )
                if ( array [i-1l] > array [i] )
                        printf( " Error at  pos , % d \ n ", i );

        return 0;
}
```

Приведенный выше пример работает только для целочисленных величин. Обычно считается, что поразрядная сортировка не подходит для сортировки чисел с плавающей точкой. Однако на самом деле этот не так. Число типа float имеет следующую побитовую структуру (рис. 5.4).



$$f = (-1)^S \times 2^{\exp-127} \times 1.mantissa$$

Рис. 5.4. Побитовая структура 43-битовоuо float-числа

Если мы имеем дело только с положительными 32-битовыми float-величинами, то старший (знаковый) бит у них совпадает и равен нулю, следующими по значимости являются 8 бит экспоненты: если у одноrо числа соответствующее 8-битовое значение больше, то и данное число больше (так как мантисса умножается на $2^{exp-127}$). При совпадении экспонент идет сравнение мантисс. Таким образом, положительные 32-битовые float-величины можно просто сортировать как 32-битовые беззнаковые целые.

С отрицательными величинами уже потребуется преобразование. Можно легко показать, что если для каждого 32-битового float-числа его 32 бита (как беззнаковое целое) подвергнуть следующему довольно простому преобразованию, то полученные величины можно сортировать поразрядно.

Данное преобразование работает следующим образом: если число положительно, то у него выставляется старший бит, в противном случае все его биты инвертируются. Тем самым, когда надо отсортировать массив величин типа float, можно воспользоваться специальным приемом и перевести эти величины в такие беззнаковые целые числа, используя описанное выше преобразование, что их сортировка даст правильный результат. Ниже приводятся ядра для перевода массива в целые числа и из целых чисел (обратите внимание, что массив в обоих случаях объявлен как массив типа uint, для облегчения манипуляции с отдельными битами).

```
template <bool doFlip>
__device__ uint floatFlip(uint f)
{
        if ( doFlip )
        {
                uint mask = ((f >> 31) – 1 | 0x80000000;
                return f ^ mask;
        }
             else return f;
}

//
// Ядро для преобразования всех float'ов в целые числа.
// Каждая нить обрабатывает сразу четыре значения.
//
__global__ void flipFloats ( uint * values, uint numValues )
{
        uint index = __umul24(blockDim.x*4, blockIdx.x) + threadIdx.x;
        if (index < numValues)  values[index] = floatFlip<true>(values[index]);
        index += blockDim.x;

        if (index < numValues) values[lndex] = floatFlip<true> (values[index] );
        index += blockDim.x;

        if (index < numValues) values[index] = floatFlip<true>(values[index]);
        index += blockDim.x;

        if (index < numValues) values[index] = floatFlip<true>(values[index]);
}

//
// Ядро для преобразования целых чисел обратно в float'ы,
// Каждая нить обрабатывает сразу четыре значения
//
__global__ void unflipFloats ( uint * values, uint numValues )
{
        uint index = __umul24(blockDim.x*4, blockIdx.x) + threadIdx.x;
```

```
        if (index < numValues) values [index] = floatUnflip<true> (values [index]);
        index += blockDim.x;

         if (index < numValues) values [index] = floatUnflip<true> (values [index]);
        index += blockDim.x;

         if (index < numValues) values [index] = floatUnflip<true> (values [index]);
        index += blockDim.x;

         if (index < numValues) values [index] = floatUnflip<true> (values [index]);
        index += blockDim.x;
}
```

Задание.

1. Отладить программы битонической и поразрядной сортировки.
2. Запустить каждую из них и засечь время выполнения.
3. Сравнить время выполнения программ.