**Лабораторная работа №3**

**Основы работы с технологией CUDA. Реализация** **базовых операций над массивами на CUDA: редукции, префиксной суммы, построения гистограмм и сортировки**

**Цель:** изучение алгоритмов параллельной реализации базовых операций над массивами (операции редукции, префиксной суммы, построения гистограмм и сортировки) и основы оптимизации программ на CUDA.

При подготовке к лабораторной работе рекомендуется изучить материалы, предоставленные в списке литературы, а также прочие материалы по теме лабораторной работы, представленные в открытых источниках.

Далее следует краткий конспект теоретического материала для лабораторной работы, задания и требования к лабораторной работе, а также контрольные вопросы для самопроверки.

1. **Реализация параллельной редукции на CUDA**

Редукцией массива А= a0, a1,a2,…an-1  относительно некоторой бинарной ассоциативной операции будет называться следующая величина (3.1):

А=(…((a0a1) a2) …an-1), (3.1)

где  - операция сложения, умножения, минимум или максимум из двух чисел.

Последовательный программный код, реализующий операцию редукции с базовой операцией сложения, приведен на рисунке 3.1.

|  |
| --- |
| for (int i = 0; i < n; i++)  sum += a[i]; |

Рисунок 3.1 – Программный код, реализующий операцию редукции

Параллельная реализация операции редукции с базовой операцией сложения приведена в выражении (3.2):

А=(a0+…+ak)+(ak+1 +…+am)+… (3.2)

поток\_1 поток\_2

Реализация операции параллельной редукции на CUDA приведена в выражении (3.3), где каждый блок отвечает за нахождение всех соответствующих ему элементов массива.

А=(a0+…+an-1)+(an +…+a2n-1)+… (3.3)

блок\_1 блок\_2

Задача редукции массива разбивается на подзадачи – нахождение сумм отдельных частей массива, которые параллельно выполняются каждой нитью за log2N шагов (см. рисунок 3.2).

Алгоритм реализации параллельной редукции на CUDA состоит из следующих этапов:

1. каждому блоку сопоставляем часть массива;
2. каждый блок копирует данные в разделяемую память;
3. в рамках одного блока данные иерархически суммируются в разделяемой памяти;
4. сохраняем результат в глобальной памяти.

Ядро, реализующее операцию параллельной редукции, приведено на рисунке 3.3. Как видно из программного кода (рисунок 3.3) и схемы иерархического суммирования условный оператор приводит к ветвлению практически во всех warp'ах.

Ветвления можно избежать, перераспределяя данные и операции по нитям (рисунок 3.4). Ядро программы на CUDA приведено на рисунке 3.5. В результате почти полностью избавились от ветвления, однако получили много конфликтов по банкам (для каждого следующего шага цикла степень конфликта удваивается).

Конфликты по банкам решаются путем изменения порядка суммирования (рисунок 3.6): суммирование начинается с наиболее удаленных на dimBlock.x/2 элементов массива. В процессе суммирования расстояние уменьшается вдвое. Ядро программы на CUDA приведено на рисунке 3.7. В результате избавились от конфликтов по банкам и от ветвления, но на первой итерации половина нитей простаивает. Поэтому первое суммирование сделаем при загрузке. Ядро приведено на рисунке 3.8.

D:\Alex Books\CUDA-course\Images\4-6.emf

Рисунок 3.2 – Параллельное иерархическое суммирование элементов массива

|  |
| --- |
| \_\_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 memory  \_\_syncthreads ();  for ( int s = 1; s < blockDim.x; s \*= 2 ) {  if ( tid % (2\*s) == 0 ) // heavy branching !!!  data [tid] += data [tid + s];  \_\_syncthreads ();  }  if ( tid == 0 ) // write result of block reduction  outData[blockIdx.x] = data [0];  } |

Рисунок 3.3 – Простейший вариант ядра, реализующего операцию параллельной редукции

D:\Alex Books\CUDA-book\Images\5-3.emf

Рисунок 3.4 – Параллельное иерархическое суммирование элементов массива с перераспределением данных и операций по нитям

|  |
| --- |
| \_\_global\_\_ void reduce2 ( 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 memory  \_\_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];  } |

Рисунок 3.5 – Ядро, реализующее операцию параллельной редукции с перераспределением данных и операций по нитям

D:\Alex Books\CUDA-course\Images\4-9.emf

Рисунок 3.6 – Параллельное иерархическое суммирование элементов массива с изменением порядка перебора пар

|  |
| --- |
| \_\_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];  \_\_syncthreads ();  for ( int s = blockDim.x / 2; s > 0; s >>= 1 )  {  if ( tid < s )  data [tid] += data [tid + s];  \_\_syncthreads ();  }  if ( tid == 0 )  outData [blockIdx.x] = data [0];  } |

Рисунок 3.7 – Ядро, реализующее операцию параллельной редукции, с изменением порядка перебора пар

|  |
| --- |
| \_\_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]; // sum  \_\_syncthreads ();  for ( int s = blockDim.x / 2; s > 0; s >>= 1 )  {  if ( tid < s )  data [tid] += data [tid + s];  \_\_syncthreads ();  }  if ( tid == 0 )  outData [blockIdx.x] = data [0];  } |

Рисунок 3.8 – Ядро, реализующее операцию параллельной редукции, с уменьшением числа блоков

Рассмотренные ядра строят суммы для каждого куска массива, соответствующего отдельному блоку, в результате получая массив частичных сумм. Для подсчета суммы элементов большого массива необходимо исходный массив разбить на блоки (например, по 512 элементов), для каждого блока применить ядро для нахождения частичной суммы, затем произвести общее суммирование на CPU.

1. **Реализация нахождения префиксной суммы на CUDA**

Префиксной суммой (**Parallel Prefix Sum, scan**) относительно массива {a0, a1,….., an-1} с некоторой бинарной ассоциативной операцией будет называться массив следующего вида:

{I, a0, a0⊕a1, a0⊕a1⊕a2, a0⊕…..⊕an-2} ,

где I – нейтральный элемент относительно используемой бинарной операции (нулевой элемент как нейтральный элемент относительно сложения, единичный – относительно умножения).

Последовательный программный код, реализующий операцию нахождения префиксной суммы, приведен на рисунке 3.9.

|  |
| --- |
| sum[0]=0;  for (int i = 1; i < n; i++)  sum[i] = sum[i-1] + a[i-1]; |

Рисунок 3.9 – Программный код, реализующий операцию нахождения префиксной суммы

Параллельная реализация Scan выполняется в два этапа:

1) построение дерева сумм (sum tree);

2) построение результирующего массива по sum tree.

Алгоритм параллельной реализации построения дерева сумм похож на операцию редукции массива, основные положения которого представлены ниже:

* используется одна нить на 2 элемента массива (см. рисунок 3.10);
* данные загружаются в разделяемую память;
* после загрузки данных выполняется барьерная синхронизация•\_\_syncthreads ();
* для построения дерева сумм необходимо log2(N) проходов.

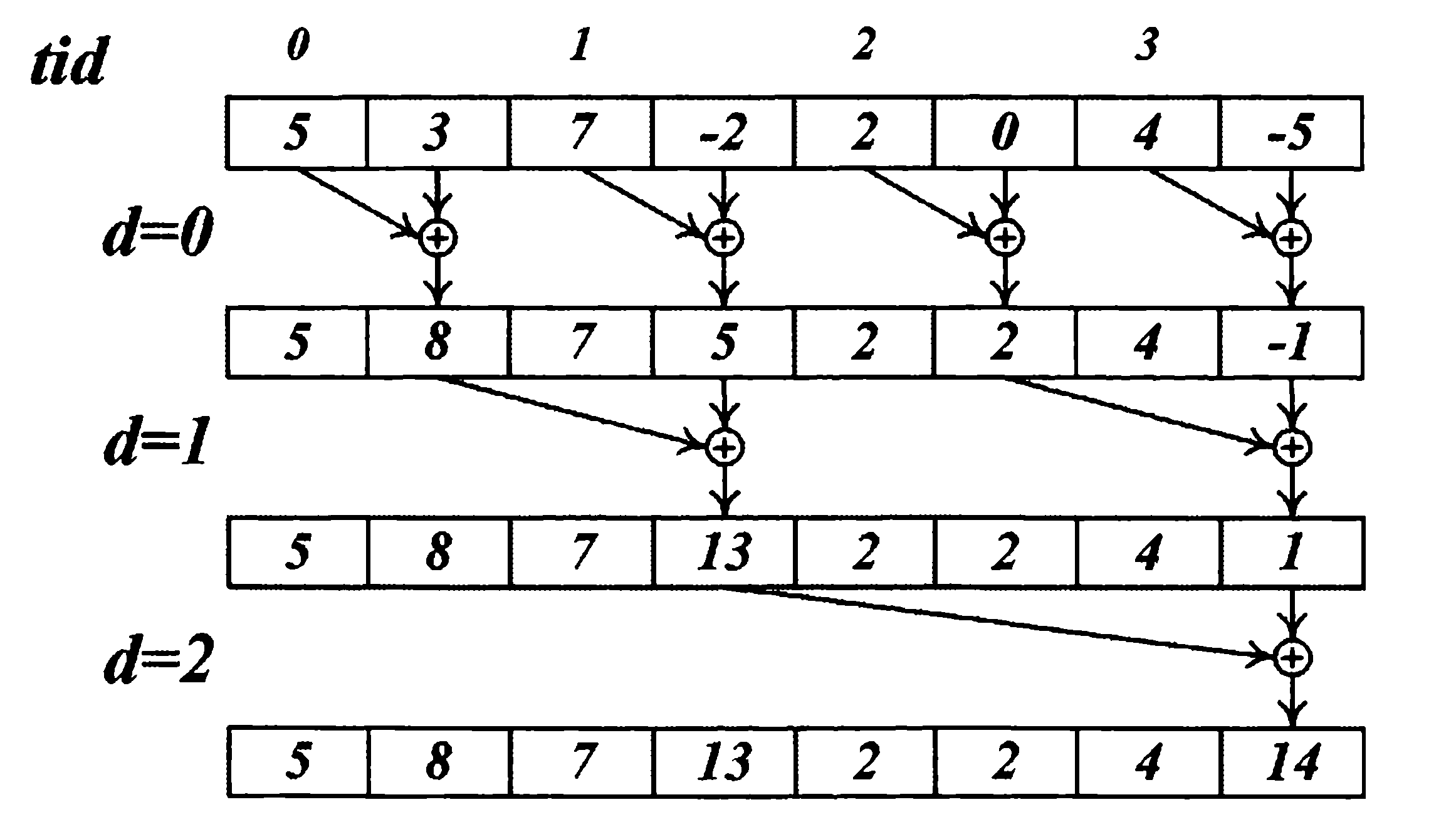
****

Рисунок 3.10 – Схема построения дерева сумм

Ядро, реализующее построение дерева сумм, приведено на рисунке 3.11.

|  |
| --- |
| #define BLOCK\_SIZE 256  \_\_global\_\_ void scan1 ( float \* inData, float \* outData, int n )  {  \_\_shared\_\_ float temp [2\*BLOCK\_SIZE];  int tid = threadIdx.x;  int offset = 1;    temp [tid] = inData [tid]; // load into shared memory  temp [tid+BLOCK\_SIZE] = inData [tid+BLOCK\_SIZE];  for ( int d = n >> 1; d > 0; d >>= 1 ){  \_\_syncthreads ();  if ( tid < d )  {  int ai = offset \* (2 \* tid + 1) - 1;  int bi = offset \* (2 \* tid + 2) - 1;  temp [bi] += temp [ai];  }  offset <<= 1;  } |

Рисунок 3.11 – Ядро, реализующее операцию построения дерева сумм

Параллельный алгоритм, формирующий результирующего массива по дереву сумм, основан на следующих положениях:

* используется одна нить на 2 элемента массива (см. рисунок 3.12);
* обнуляется последний элемент, содержащий сумму всех элементов;
* на каждом шаге один из элементов пары копируется на место второго, а на место первого записывается сумма исходных элементов;
* для построения результирующего массива необходимо log2(N) проходов.

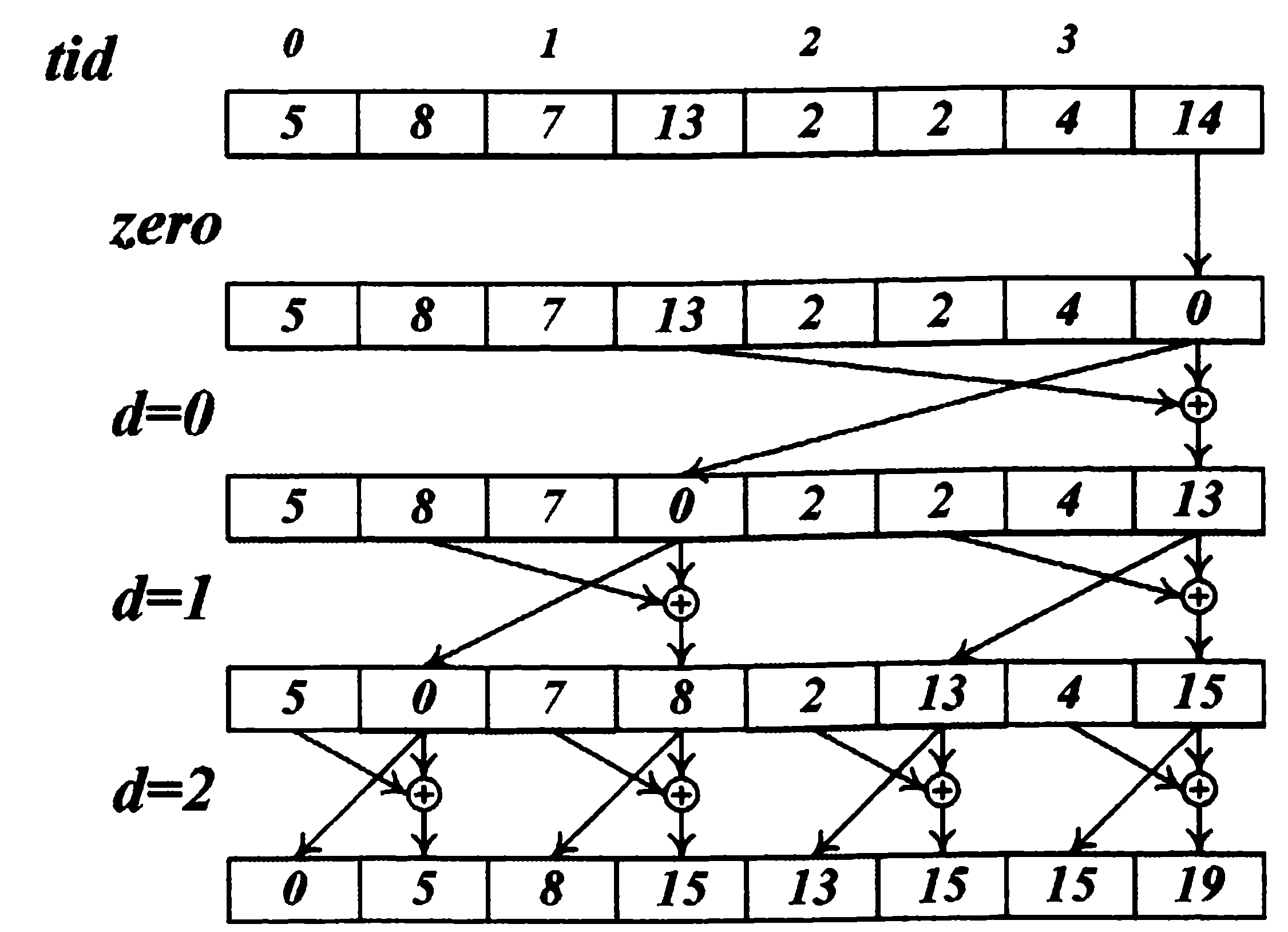


Рисунок 3.12 – Схема построения результирующего массива, содержащего префиксные суммы

Ядро, реализующее построение результирующего массива, приведено на рисунке 3.13.

|  |
| --- |
| if ( tid == 0 ) temp [n-1] = 0; // clear the last element  for ( int d = 1; d < n; d <<= 1 )  {  offset >>= 1;  \_\_syncthreads ();  if ( tid < d )  {  int ai = offset \* (2 \* tid + 1) - 1;  int bi = offset \* (2 \* tid + 2) - 1;  float t = temp [ai];  temp [ai] = temp [bi];  temp [bi] += t;  }  }  \_\_syncthreads ();  outData [2\*tid] = temp [2\*tid]; // write results  outData [2\*tid+1] = temp [2\*tid+1];  } |

Рисунок 3.13 – Ядро, реализующее операцию построения префиксной суммы

Данная реализация приводит к конфликтам банков до 16 порядка, поэтому требуется следующая оптимизация:

* добавим по одному «выравнивающему» элементу на каждые 16 элементов в разделяемой памяти;
* к каждому индексу добавим соответствующее смещение.

Ядро, эффективно осуществляющее нахождение префиксной суммы, приведено на рисунке 3.14.

|  |
| --- |
| #define LOG\_NUM\_BANKS 4  #define CONLICT\_FREE\_OFFS(i) ((i) >> LOG\_NUM\_BANKS)  // sums – для scan больших массивов  \_\_global\_\_ void scan2 ( float \* inData, float \* outData, /\*float \* sums, \*/ int n ) {  \_\_shared\_\_ float temp [2\*BLOCK\_SIZE+CONFLICT\_FREE\_OFFS(2\*BLOCK\_SIZE)];  int tid = threadIdx.x;  int offset = 1;  int ai = tid  int bi = tid + (n / 2);  int offsA = CONFLICT\_FREE\_OFFS(ai);  int offsB = CONFLICT\_FREE\_OFFS(bi);  temp [ai + offsA] = inData [ai + 2\*BLOCK\_SIZE\*blockIdx.x];  temp [bi + offsB] = inData [bi + 2\*BLOCK\_SIZE\*blockIdx.x];  for ( int d = n>>1; d > 0; d >>= 1, offset <<= 1 )  {  \_\_syncthreads ();  if ( tid < d )  {  int ai = offset \* (2 \* tid + 1) - 1;  int bi = offset \* (2 \* tid + 2) - 1;  ai += CONFLICT\_FREE\_OFFS(ai);  bi += CONFLICT\_FREE\_OFFS(bi);  temp [bi] += temp [ai];  }  }  if ( tid == 0 )  {  //int i = n - 1 + CONFLICT\_FREE\_OFFS(n-1); // для scan больших массивов  //sums [blockIdx.x] = temp [i]; // для scan больших массивов  temp [i] = 0; // clear the last element  }  for ( int d = 1; d < n; d <<= 1 )  {  offset >>= 1;  \_\_syncthreads ();  if ( tid < d )  {  int ai = offset \* (2 \* tid + 1) - 1;  int bi = offset \* (2 \* tid + 2) - 1;  float t;  ai += CONFLICT\_FREE\_OFFS(ai);  bi += CONFLICT\_FREE\_OFFS(bi);  t = temp [ai];  temp [ai] = temp [bi];  temp [bi] += t;  }  }  \_\_syncthreads ();  outData [ai + 2\*BLOCK\_SIZE\*blockIdx.x] = temp [ai + offsA];  outData [bi + 2\*BLOCK\_SIZE\*blockIdx.x] = temp [bi + offsB];  } |

Рисунок 3.14 – Программный код ядра, эффективно реализующее нахождение префиксной суммы

Рассмотренный код хорошо работает для небольших массивов, целиком, помещающихся в разделяемую память.

В общем случае для нахождения префиксной суммы необходимо выполнить следующие этапы:

* выполняем отдельный scan для каждого блока (см. рисунок 3.14, включая комментарии с пометкой «для больших массивов»);
* для каждого блока запоминаем сумму элементов (перед обнулением);
* применяем scan к массиву сумм (см. рисунок 3.15);
* к каждому элементу, кроме элементов 1-го блока добавляем значение, соответствующее данному блоку (см. ядро scanDistribute на рисунке 3.15).

|  |
| --- |
| \_\_global\_\_ void scanDistribute(float \* data, float \* sums)  {  data[threadIdx.x + blockIdx.x\*2\*BLOCK\_SIZE] += sums[blockIdx.x];  }  void scan ( float \* inData, float \* outData, int n )  {  int numBlocks = n / (2\*BLOCK\_SIZE);  float \* sums; // суммы элементов для каждого блока  float \* sums2; // результаты scan этих сумм  if ( numBlocks < 1 ) numBlocks = 1;  // выделяем память под массивы  cudaMalloc ( (void\*\*)&sums, numBlocks \* sizeof ( float ) );  cudaMalloc ( (void\*\*)&sums2, numBlocks \* sizeof ( float ) );  // поблочный scan  dim3 threads ( BLOCK\_SIZE, 1, 1 ), blocks ( numBlocks, 1, 1 );  scan2<<<blocks, threads>>> ( inData, outData, sums, 2\*BLOCK\_SIZE );  // выполняем scan для сумм  if ( n >= 2\*BLOCK\_SIZE )  scan ( sums, sums2, numBlocks );  else  cudaMemcpy ( sums2, sums, numBlocks\*sizeof(float), cudaMemcpyDeviceToDevice );  // корректируем результат  threads = dim3 ( 2\*BLOCK\_SIZE, 1, 1 );  blocks = dim3 ( numBlocks - 1, 1, 1 );  scanDistribute<<<blocks,threads>>> ( outData + 2\*BLOCK\_SIZE, sums2 + 1 );  cudaFree ( sums );  cudaFree ( sums2 );  } |

Рисунок 3.15 – Фрагмент программного кода, реализующий scan больших массивов

1. **Реализация построения гистограммы на CUDA**

Гистограммой исходного массива элементов a0,….., an-1 будет называться массив с0,….., сk-1, каждый элемент которого равен числу элементов исходного массива, принадлежащий каждой группе, разделенный по некоторому критерию.

Последовательный программный код, реализующий операцию построения гистограммы, приведен на рисунке 3.16.

|  |
| --- |
| for (int i = 0; i < k; i++)  c[i] = 0;  for (int i = 0; i < n; i++)  c[a[i]]++; |

Рисунок 3.16 – Программный код, реализующий операцию построения гистограммы

Далее рассмотрено решение построения гистограммы, которое работает на графических картах, поддерживающих CUDA с compute capability 1.0 и выше (вариант с atomicAdd доступен для GPU c compute capability выше 1.2).

Входной массив разбивается на части, каждую часть будет обрабатывать отдельный блок. Наиболее эффективный вариант создать для каждого warp'а свой вариант гистограммы и разместить в разделяемой памяти. 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 записи будут выполнены последовательно одна за другой, и в памяти останется последнее записанное значение. При этом порядок, в котором могут произойти записи, заранее неизвестен. Для такого решения зарезервируем пять старших бит счетчика под уникальный идентификатор (рисунок 3.17) нити внутри warp'а (threadIdx.x & 0x1f). Тогда 192 нитей в блоке дают 6 warp’ов, то есть получается. 6\*256\*4=6Кбайт разделяемой памяти на блок.

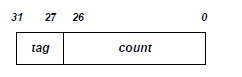
****

Рисунок 3.17 – Формат счетчика в каждой гистограмме

На рисунке 3.18 приводится функция, которая реализует гарантированное атомарное увеличение счетчиков.

|  |
| --- |
| #define N (6\*1024\*1024)  #define NUM\_BINS 256 // число счетчиков в гистограмме  \_\_device\_\_ inline void addByte( volatile unsigned \* warp\_hist, unsigned data, unsigned ttag)  {  unsigned count;  do { // прочесть текущее значение счетчика и снять идентификатор нити  count = warp\_hist[data] & 0x07FFFFFFU;  // увеличить его на единицу и поставить свой идентификатор  count = ttag | (count + 1);  warp\_hist[data] = count; //осуществить запись  } while (warp\_hist[data] != count); // проверить, прошла ли запись  } |

Рисунок 3.18 – Функция, реализующая гарантированное атомарное увеличение счетчиков в гистограмме

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. Каждый блок создает в глобальной памяти отдельную гистограмму. Для сведения результатов в единую гистограмму используется отдельное ядро mergeHistogramKernel. Листинг программы, демонстрирующий нахождение гистограммы для массива байт приведен на рисунке 3.18.

|  |
| --- |
| #define LOG2\_WARP\_SIZE 5 // логарифм размера warp's по основанию 2  #define WARP\_SIZE 32 // Размер warp'а  #define WARP\_N 6 // Число warp'ов в блоке  \_\_global\_\_ void histogramKernel ( unsigned \* result, unsigned \* data, int n )  {  \_\_shared\_\_ unsigned hist[NUM\_BINS\*WARP\_N]; //1536 элементов  // очистить счетчики гистограмм  for (int i = 0; i < NUM\_BINS / WARP\_SIZE; i++)  hist[threadIdx.x + i\*WARP\_N\*WARP\_SIZE/\*число нитей в блоке=192\*/] = 0;    int warp\_base = (threadIdx.x >> LOG2\_WARP\_SIZE) \* NUM\_BINS;  unsigned ttag = threadIdx.x << (32 - LOG2\_WARP\_SIZE); // получить id для данной нити    \_\_syncthreads ();  int global\_tid = blockIdx.x \* blockDim.x + threadIdx.x;  int numThreads = blockDim.x \* gridDim.x;  for (int i = global\_tid; i < n; i += numThreads)  {  unsigned data4 = data [i];  addByte(hist + warp\_base, (data4 >> 0) & 0xFFU, ttag);  addByte(hist + warp\_base, (data4 >> 8) & 0xFFU, ttag);  addByte(hist + warp\_base, (data4 >> 16) & 0xFFU, ttag);  addByte(hist + warp\_base, (data4 >> 24) & 0xFFU, ttag);  }  \_\_syncthreads();  // объединить гистограммы данного блока и записать результат в глобальную память  // 192 нити суммируют данные до 256 элементов гистограмм  for (int bin = threadIdx.x; bin < NUM\_BINS; bin += (WARP\_N\*WARP\_SIZE))  { unsigned sum = 0;  for (int i = 0; i < WARP\_N; i++)  sum += hist [bin + i\*NUM\_BINS] & 0x07FFFFFFU;  result[blockIdx.x \* NUM\_BINS + bin] = sum;  }  }  // объединить гистограммы, один блок на каждый NUM\_BINS элементов  \_\_global\_\_ void mergeHistogramKernel (unsigned \* out\_histogram, unsigned \* partial\_histograms, int histogram\_count)  {  unsigned sum = 0;  for (int i = threadIdx.x; i < histogram\_count; i += 256)  sum += partial\_histograms[blockIdx.x + i \* NUM\_BINS];    \_\_shared\_\_ unsigned data[NUM\_BINS];  data[threadIdx.x] = sum;    for (unsigned stride = NUM\_BINS / 2; stride > 0; stride >>= 1)  {  \_\_syncthreads ();  if (threadIdx.x < stride) data[threadIdx.x] += data[threadIdx.x + stride];  }  if (threadIdx.x == 0 ) out\_histogram[blockIdx.x] = data[0];  }  void histogram( uint \* histogram, void \* data, unsigned byte\_сount)  {  int n = byte\_сount / 4;  int num\_blocks = n / (WARP\_N\*WARP\_SIZE);  int num\_partials = 240;  unsigned \*partial\_histograms = nullptr;  unsigned int h[NUM\_BINS] = {0};  int \*pdata = (int\*)data;    //выделить память под гистограммы блока  cudaMalloc((void\*\*)&partial\_histograms, num\_partials\*NUM\_BINS\* sizeof(unsigned));  // построить гистограмму для каждого блока  histogramKernel <<<dim3(num\_partials), dim3(WARP\_N\*WARP\_SIZE) >>> (partial\_histograms, (unsigned\*)data, n);    //объдинить гистограммы отдельных блоков вместе  mergeHistogramKernel<<<dim3(NUM\_BINS), dim3(256)>>> (histogram, partial\_histograms, num\_partials);  // освободить выделенную память  cudaFree(partial\_histograms);  } |

Рисунок 3.18 – Листинг фрагмента программы, демонстрирующий нахождение гистограммы для массива байт

1. **Реализация битонической сортировки на CUDA**

На GPU хорошо реализуются алгоритмы сортировки, которые имеют многопроходный и параллельный характер. Такими характеристиками обладаютсети компараторов (comparator networks), которые хорошо ложатся на архитектуру GPU. Компаратор - это устройство с n входами и n выходами, которое выполняет некоторую перестановку элементов в зависимости от результатов их сравнения.

Одной из простейших сетей компараторов является полуочиститель (half-cleaner) Bn, упорядочивающий пары элементов xi и xi+n/2 на заданном расстоянии i от 0 до n/2-1.

Компаратор *Bn* можно реализовать как для упорядочивания элементов внутри пары по возрастанию, так и по убыванию.

Применение определенной последовательности полуочистителей позволяет отсортировать произвольный массив заданного размера, т.е. из отдельных полуочистителей можно построить сортирующую сеть.

Утверждение. Если сеть компараторов упорядочивает (сортирует) произвольную последовательность из нулей и единиц, то она является сортирующей, т.е. она упорядочивает произвольную последовательность чисел.

Определение. Последовательность *a0,...,an-1* называется битонической, если она сперва убывает, а потом возрастает, либо получается из такой последовательности циклическим сдвигом.

Примеры:

1. 1,3,4,7,6,5,2;
2. 5,7,6,4,2,1,3 (получена сдвигом 1,3,5,7,6,4,2).

Утверждение. Если применить сеть *Bn* к битонической последовательности *a0,...,an-1* (длины *n*) из нулей и единиц, то получившееся в результате последовательность *b0,...,bn-1* будет обладать следующими свойствами:

1. ее верхняя и нижняя половины также будут битоническими;
2. любой элемент первой (верхней) половины всегда будет меньше или равен любого элемента второй (нижней) половины;
3. хотя бы одна из половин - чистая (т.е. монотонная).

Пусть у нас есть битоническая последовательность a0,...,an-1. Если мы применим к ней сеть Bn, то мы в результате получим две половины, каждая из которых является битонической и все элементы первой половины меньше или равны элементов второй.

Если к битонической последовательности длины n применить получистители Bn,Bn/2,…,B8,B4,B2, то в результате мы получим отсортированную последовательность. Таким образом построили сортирующую сеть для битонических последовательностей. Подобная сеть также называется битоническим слиянием (bitonic merge), структура которой представлена на рисунке 3.19.

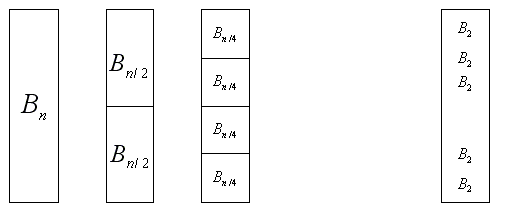


Рисунок 3.19 – Битоническое слияние

Операция битонического слияния *Mn* может быть представлена рекурсивно в следующем виде (см. рисунок 3.20).

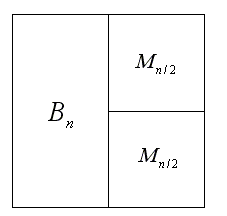


Рисунок 3.20 - Рекурсивное представление битонического слияния

Рассмотрим, как при помощи сетей *Bn* и *Mn* построить сеть, сортирующую произвольную последовательность длины *n*. Пусть у нас есть последовательность *a0,a1,a2,a3,a4,a5,a6,a7*. Тогда битоническая сортировка выполняется в несколько этапов:

1) Применим к каждой паре произвольной последовательности элементов полуочиститель *B2* с чередующимся порядком сортировки. Тогда каждая четверка элементов будет образовывать битоническую последовательность.

D:\Alex Books\CUDA-book\Images\5-8.emf

2) Применим к каждой такой четверке элементов операцию битонического слияния *М4* с чередующимся порядком сортировки (сначала по возрастанию, потом по убыванию). Получим битоническую последовательность.

3) Применим операцию битонического слияния *М8*. Получим отсортированную последовательность.

Битоническая сортировка требует *log2n\*(log2n+1)/2* проходов. Фрагмент программного кода, выполняющего битоническую сортировку приведен на рисунке 3.21.

|  |
| --- |
| #define N 1024  #define BLOCK\_SIZE (N/2) //размер блока  \_\_device\_\_ void Comparator(unsigned int& keyA, unsigned int& valA, unsigned int& keyB, unsigned int& valB, unsigned int dir)  {  unsigned int t;  if ((valA > valB) == dir) //поменять местами (keyA, valA) и (keyB, valB)  {  t = keyA; keyA = keyB; keyB = t;  t = valA; valA = valB; valB = t;  }  }  \_\_global\_\_ void bitonicSortShared(unsigned int\* dstKey, unsigned int \* dstVal, unsigned int\* srcKey, unsigned int\* srcVal, unsigned int arrayLength, unsigned int dir)  {  \_\_shared\_\_ unsigned int sk[BLOCK\_SIZE\*2];  \_\_shared\_\_ unsigned int sv[BLOCK\_SIZE\*2];  int index = blockIdx.x \* BLOCK\_SIZE\*2 + threadIdx.x;    sk[threadIdx.x] = srcKey[index]; sv[threadIdx.x] = srcVal[index];  sk[threadIdx.x + BLOCK\_SIZE] = srcKey[index + BLOCK\_SIZE]; sv[threadIdx.x + BLOCK\_SIZE] = srcVal[index + BLOCK\_SIZE];    for (unsigned int size = 2; size < arrayLength; size <<=1 )  {//битоническое слияние  unsigned int ddd = dir ^ ((threadIdx.x&(size / 2)) != 0);  for(unsigned int stride = size >> 1; stride > 0; stride >>=1)  {  \_\_syncthreads ();  unsigned int 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 (unsigned int stride = arrayLength >> 1; stride > 0; stride >>= 1)  {  \_\_syncthreads ();  unsigned int 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];  } |

Рисунок 3.21 – Фрагмент программного кода, выполняющего битоническую сортировку

**Лабораторные задания** (№ варианта = (№ студента в списке)%2)

**Общее задание:** изучить параллельную реализацию базовых операций над массивами на CUDA, нарисовать схему алгоритмов, запустить программу с реализацией базовых операций на GPU и CPU (по вариантам), измерить время выполнения**.** Написать программу для верификации результатов. **Результаты занести в отчёт.**

|  |  |
| --- | --- |
| **Вариант** | **Задание** |
| **0** | 1. Дан массив А из N элементов (задаются случайно). Выполнить редукцию массива А с базовой операцией min. 2. Дан массив А из N натуральных элементов от 0 до 255 с нормальным распределением. Построить гистограмму, содержащую число каждого элемента массива. |
| **1** | 1. Дан массив А из N элементов (задаются случайно). Найти префиксную сумму элементов массива с базовой операцией сложения. 2. Дан массив А из N натуральных элементов (задаются случайно). Выполнить сортировку массива. |

**Контрольные вопросы**

1. Чем отличается параллельная и последовательная реализация операции редукции?
2. Какова сложность алгоритма параллельной редукции?
3. Какие подходы используются при реализации операции редукции для избавления от ветвления и конфликта по банкам в программе?
4. Что такой операция префиксного суммирования?
5. В чем особенность параллельной реализации алгоритма префиксного суммирования?
6. Алгоритм выполнения scan над большими массивами (массив не помещается в разделяемую память).
7. На каких принципах основано построение гистограммы?
8. Какими способами возможно обеспечить атомарность операций при реализации алгоритма построения гистограммы на CUDA?
9. Что такое сеть компараторов и для чего она нужна?
10. Что такое битоническое слияние?
11. Этапы битонической сортировки.

**Требования к сдаче работы**

1. При домашней подготовке изучить теоретический материал по тематике лабораторной работы, представленный в списке литературы ниже, выполнить представленные примеры, занести в отчёт результаты выполнения.
2. Продемонстрировать выполнение лабораторных заданий.
3. Ответить на контрольные вопросы.
4. Показать преподавателю отчет.

**Литература**

1. <http://www.nvidia.ru/object/cuda-parallel-computing-ru.html>
2. А.В. Боресков, А.А. Харламов. Основы работы с технологией Cuda. – М: ДМК Пресс, 2010. – 232 с.