(2-Way)RADIX-SORT

M.VENKATAKRISHNA

RADIX-SORT : ALGORITHM

EXAMPLE RUN

BIT-0 SPLIT

Number	Bit-Expansion
0	0000
5	0101
2	0010
7	0111
1	0001
3	0011
6	0110
4	0100

COMPACT_0
(COMPACT_0[k] == 1) IF (arr[k] HAS 0 AS BIT-0))

0: 1	1: 0	2: 1	3: 0	4: 0	5: 0	6: 1	7: 1

SCAN-0 : SCAN OF COMPACT_0[]
(Scatter Address for bit-0=0 elements)

0: 0	1: 0	2: 1	3: 1	4: 1	5: 1	6: 2	7: 3

COMPACT_1
(COMPACT_1[k] == 1) IF (arr[k] HAS 1 AS BIT-0))

0: 0	1: 1	2: 0	3: 1	4: 1	5: 1	6: 0	7: 0

SCAN-1 : SCAN OF COMPACT_1[]
(Scatter Address for bit-0=1 elements)

0: 0	1: 0	2: 1	3: 1	4: 1	5: 1	6: 2	7: 3

AFTER REARRANGE BASED ON SCAN ARRAYS

Number	Bit-Expansion
0	0000
5	0101
2	0010
7	0111
1	0001
3	0011
6	0110
4	0100

Number	Bit-Expansion
0	0000
2	0010
6	0110
4	0100
5	0101
7	0111
1	0001
3	0011

BIT-1 SPLIT

Number	Bit-Expansion
0	0000
2	0010
6	0110
4	0100
5	0101
7	0111
1	0001
3	0011

COMPACT_0

(COMPACT_0[k] == 1) IF (arr[k] HAS 0 AS BIT-1))

0: 1	1: 0	2: 0	3: 1	4: 1	5: 0	6: 1	7: 0

SCAN-0 : SCAN OF COMPACT_0[]

(Scatter Address for bit-1=0 elements)

0: 0	1: 1	2: 1	3: 1	4: 2	5: 3	6: 3	7: 4

COMPACT_1

(COMPACT_1[k] == 1) IF (arr[k] HAS 1 AS BIT-1))

0: 0	1: 1	2: 1	3: 0	4: 0	5: 1	6: 0	7: 1

SCAN-1 : SCAN OF COMPACT_1[]

(Scatter Address for bit-1=1 elements,

OFF-SET=4)

0: 0	1: 0	2: 1	3: 2	4: 2	5: 2	6: 3	7: 3

AFTER REARRANGE BASED ON SCAN ARRAYS

Number	Bit-Expansion
0	0000
2	0010
6	0110
4	0100
5	0101
7	0111
1	0001
3	0011

Number	Bit-Expansion
0	0000
4	0100
5	0101
1	0001
2	0010
6	0110
7	0111
3	0011

BIT-2 SPLIT

Number	Bit-Expansion
0	0000
4	0100
5	0101
1	0001
2	0010
6	0110
7	0111
3	0011

COMPACT_0
(COMPACT_0[k] == 1) IF (arr[k] HAS 0 AS BIT-2))

0: 1	1: 0	2: 0	3: 1	4: 1	5: 0	6: 0	7: 1

SCAN-0 : SCAN OF COMPACT_0[]
(Scatter Address for bit-2=0 elements)

0: 0	1: 1	2: 1	3: 1	4: 2	5: 3	6: 3	7: 3

COMPACT_1
(COMPACT_1[k] == 1) IF (arr[k] HAS 1 AS BIT-2))

0: 0	1: 1	2: 1	3: 0	4: 0	5: 1	6: 1	7: 0

SCAN-1 : SCAN OF COMPACT_1[]
(Scatter Address for bit-2=1 elements,
OFFSET = 4)

0: 0	1: 0	2: 1	3: 2	4: 2	5: 2	6: 3	7: 4

AFTER REARRANGE BASED ON SCAN ARRAYS

Number	Bit-Expansion
0	0000
4	0100
5	0101
1	0001
2	0010
6	0110
7	0111
3	0011

Number	Bit-Expansion
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

BIT-3 SPLIT: ARRAY IS NOW SORTED

Number	Bit-Expansion
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

SORTED ARR[] : {0,1,2,3,4,5,6,7}

RADIX-SORT: IMPLEMENTATION USING CUDA

STEP-1: COMPACT EACH ELEMENT OF arr[] BASED ON BIT-i

3-STEPS

KERNEL: compacter()

```
//Takes arr[] as input and produces compact_0[] for bit-0, and compact_1[] for bit-1, in bit-poisiton 'i'
__global__ void compacter(unsigned int* arr, unsigned int* compact_0, unsigned int* compact_1, int i)
{
    int idx = threadIdx.x;

    unsigned int num = arr[idx];

    //Predicate: Check if bit-i is 1 or 0
    compact_0[idx] = !( (num >> i) & 1);
    compact_1[idx] = ( (num >> i) & 1);
}
```

Takes as input :

1) **arr[] :-** Input array
2) **i :** The bit whose compact is to be produced for all elements in arr[]

Produces as Output:

1) **compact_0[] :** Compact array :
   (compact_0[k] == 1) if arr[k] has bit-i as 0
2) **compact_1[] :** Compact array:
   (compact_1[k] == 1) if arr[k] has bit-i as 1

compacter () : Function

- Produces compact_0[] and compact_1[] for bit-i of elements of arr[]
  - **compact_0[k] == 1** if arr[k] has bit-i as 0
  - **compact_1[k] == 1** if arr[k] has bit-i as 1

compacter() : Kernel call:

```
compacter <<< 1, n >>> (d_arr, d_compact_0, d_compact_1, bit_no);     //Launch kernel : 1 block of n threads
```

- n = size of arr[]
- **1 block of n threads** are launched
- **Each thread** is responsible for producing compact_0[] and compact_1[] of **1 element in arr[]**

STEP-2: PRODUCE SCAN ARRAYS FOR COMPACT ARRAYS

(THESE WILL BE USED AS SCATTER ADDRESSES TO REARRANGE arr[] AFTER BIT_i SPLIT)

sum_scan_blelloch(d_scan[], d_compact_0, n+1)

- Takes as input compact[] array and number of elements
- Performs **EXCLUSIVE BLELLOCH SCAN** on compact[] array and produces scan output in scan[] array

- **THIS FUNCTION IS USED TO PRODUCE SCAN OF COMPACT_0[] AND COMPACT_1[] ARRAYS**

- **THIS scan_0[] and scan_1[] arrays WILL ACT AS SCATTER ADDRESS ARRAYS to PLACE THE ELEMENTS IN ARR[] AFTER BIT-i SPLIT**

STEP-3: REARRANGE ACCORDING TO BIT-i SPLIT BASED ON SCATTER ADDRESS

KERNEL: scatter_sort()

```
__global__ void scatter_sort(unsigned int* arr, unsigned int* scan_0, unsigned int* scan_1,
                             unsigned int* compact_0, int n, unsigned int* temp_out)
{
    int idx = threadIdx.x;

    int num_of_0 = scan_0[n];    //Total number of elements with 0-bit in respective position
    int pos;

    if (compact_0[idx] == 1)    //If element arr[idx] has bit-0 in current position
    {
        pos = scan_0[idx];    //The index position in output array where the arr[idx] is supposed to go
        temp_out[pos] = arr[idx];
    }

    else if (compact_0[idx] == 0)  //If element arr[idx] has bit-1 in current position
    {
        pos = scan_1[idx] + num_of_0;    //num_of_0 : Offset
        temp_out[pos] = arr[idx];
    }

    __syncthreads;   //Barrier


    //Copy from temp_out[] back to arr[]
    arr[idx] = temp_out[idx];

}
```

- **Takes as input :**
  - **arr[] :** Input array
  - **scan_1[] and scan_2[] :** Scatter address arrays used to rearrange elements of arr[] according to based on bit-i split
  - **compact_0[] :** In order to find out if element in arr[] has bit-i as 0 or 1

- **Produces Output :**
  - The **arr[] rearranged** according to bit-i split.

scatter_sort() Function

- **Rearranges** the arr[] after **split of bit-i** by using **scan_0[] and scan_1[] arrays as scatter address** arrays.

KERNEL CALL scatter_sort()

```
scatter_sort <<< 1, n >>> ( d_arr, d_scan_0, d_scan_1, d_compact_0, n, d_temp_out);
```

- Launch n (arr[].size) threads.
- 1 thread is responsible for 1 element of arr[] .
- Each thread places its respective element in its rearranged position.

MAIN CPU FUNCTION

radix_sort()

```
void radix_sort()
{

    // ---------------------------Iterate 32 times (1 time for each bit of integer) and Make Kernel Call-------------------------
    int bit_no;
    for (bit_no = 0; bit_no < 32; bit_no++)
    {
        compacter <<< 1, n >>> (d_arr, d_compact_0, d_compact_1, bit_no);     //Launch kernel : 1 block of n threads


        //Produce scan arrays
        sum_scan_blelloch(d_scan_0, d_compact_0, n+1);     //Calculate scan_0[] using Blelloch Scan Technique
        scan_1_calculator <<< 1, (n+1) >>> (d_scan_1, d_scan_0, n);    //calculate scan_1[] using scan_0[]


        //Produce bit-0 sort
        scatter_sort <<< 1, n >>> ( d_arr, d_scan_0, d_scan_1, d_compact_0, n, d_temp_out);

    }
```

STEP-1,2,3 ARE REPEATED FOR EACH BIT (BIT-0 TO BIT-31) OF ELEMENTS OF arr[]

- AFTER THAT WE WILL ARRIVE AT THE SORTED OUTPUT

Step Complexity and Work Complexity

- **Step Complexity: O(1)**
  - **Totally 32 steps (One Step for 1 Bit)**
  - **Each step, n work is done parallelly**
- **Work Complexity : O(n)**

4-WAY RADIX SORT

4-Way Radix vs 2-Way Radix Sort

- 4-way radix sort is a **more efficient implementation** of radix sort.
- In normal 2-way radix sort, after each pass of array, **we split the array into 2 parts based on bit-i**
- Whereas, in 4-way radix sort, **after each pass of array, we split the array into 4 parts** based on **2 bits : bit-i , bit: i+1**

4-WAY RADIX-SORT : ALGORITHM

EXAMPLE RUN

BIT-0,1 SPLIT

Number	Bit-Expansion
0	0000
5	0101
2	0010
7	0111
1	0001
3	0011
6	0110
4	0100

Number	Bit-Expansion
0	0000
4	0100
5	0101
1	0001
2	0010
6	0110
7	0111
3	0011

BIT-2,3 SPLIT

Number	Bit-Expansion
0	0000
4	0100
5	0101
1	0001
2	0010
6	0110
7	0111
3	0011

Number	Bit-Expansion
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

ARRAY IS NOW SORTED

Number	Bit-Expansion
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

SORTED ARR[] : {0,1,2,3,4,5,6,7}

4-WAY RADIX SORTING IMPLEMENTATION IN CUDA

KEY-DIFFERENCE 1: COMPACT ARRAYS

- The implementation of 4-Way radix sort is similar to 2-way radix sort. **There are only these key differences**:

```
//Takes arr[] as input and produces compact_0[] for bit-00, compact_1[] for bit-01, compact_2[] for bit-10 and compact_3[] for bit-11 , in
__global__ void compacter(unsigned int* arr, unsigned int* compact_0, unsigned int* compact_1,
                          unsigned int* compact_2, unsigned int* compact_3, int i){ ... }
```

❑ The compacter() kernel function is now **responsible for producing 4 compact[] arrays** :
compact_0[] , compact_1[], compact_2[] and compact_3[].

- **compacter_0[k] == 1** if arr[k] has **(bit-i , bit-i+1) as (“00”)**
- **compacter_1[k] == 1** if arr[k] has **(bit-i , bit-i+1) as (“01”)**
- **compacter_2[k] == 1** if arr[k] has **(bit-i , bit-i+1) as (“10”)**
- **compacter_3[k] == 1** if arr[k] has **(bit-i , bit-i+1) as (“11”)**

KEY DIFFERENCE 2 : SCAN ARRAYS

```
//Produce scan arrays
sum_scan_blelloch(d_scan_0, d_compact_0, n + 1);
sum_scan_blelloch(d_scan_1, d_compact_1, n + 1);
sum_scan_blelloch(d_scan_2, d_compact_2, n + 1);
sum_scan_blelloch(d_scan_3, d_compact_3, n + 1);
```

- Here, 4 scan arrays are produced using Blelloch Scan, for each 2-bit iteration
- 1 scan[] array for each compact[] array

EFFICIENCY OF 4-WAY RADIX SORT

- Finally, by using the 4 scan arrays as scatter address arrays, we can produce arr[] array rearranged according to bit-(i, i+1) split.
- Note that in 4-way radix sort, we **can arrive at sorted output (for integers) only after 16 iterations** (1 iteration for 1 pair of bits).
  - As compared to 2-way radix sort, where we need to do 32 iterations to arrive at sorted output
- Hence**, 4-way radix sort is more efficient** than normal 2-way radix sort

Step Complexity and Work Complexity

- **Step Complexity: O(1)**
  - **Totally 16 steps (One Step for 1 Bit)**
  - **Each step, n work is done parallelly**
- **Work Complexity : O(n)**

END