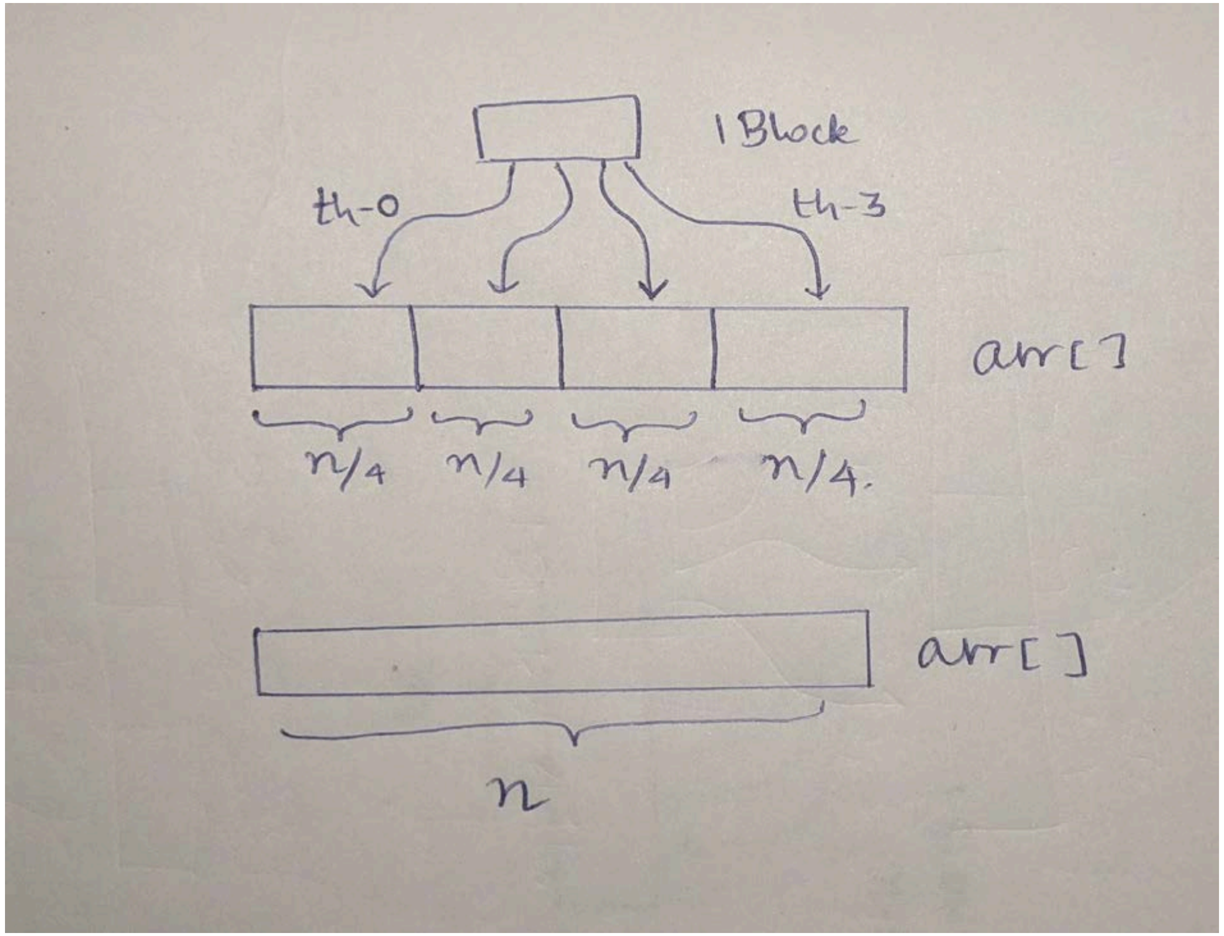
CUDA Merge Sort

M.VENKATAKRISHNA

MERGE SORT PARALLEL ALGORITHM (2-Stages)

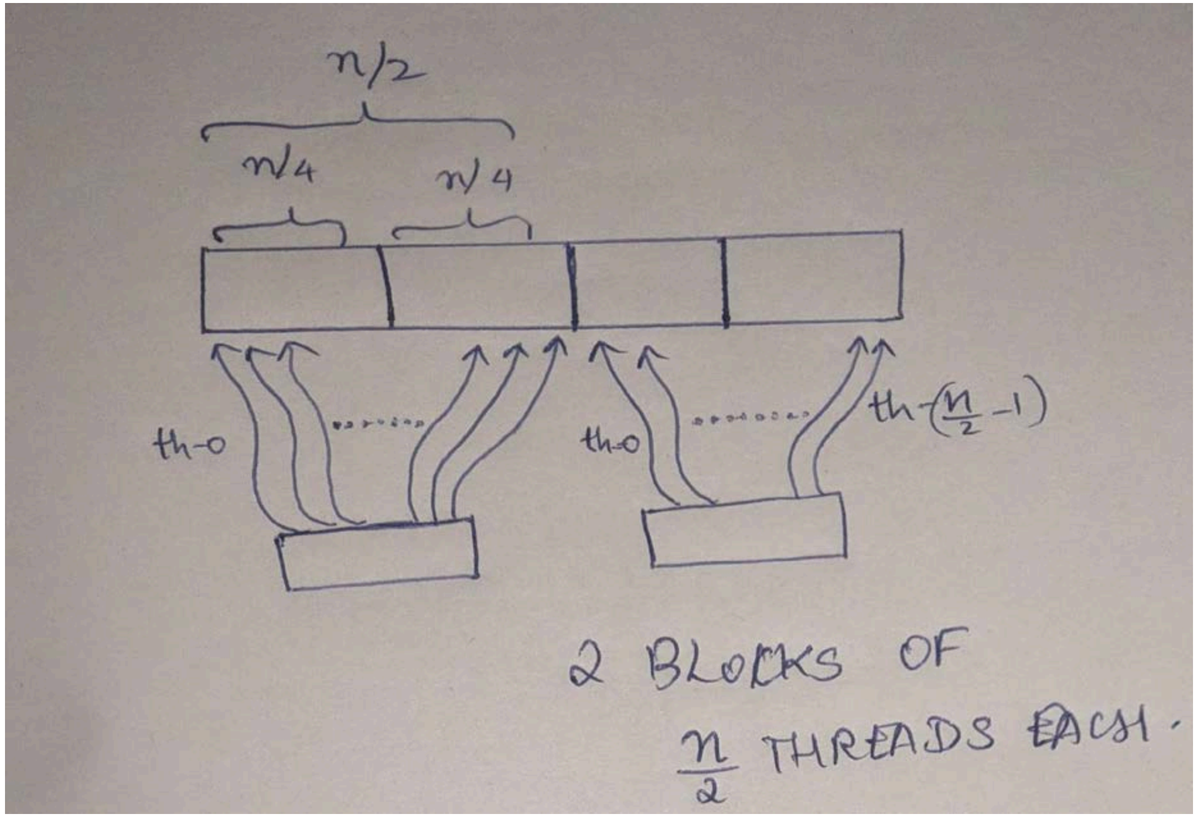
EXPLAINED WITH 4 INITIAL SPLITS

Stage-1

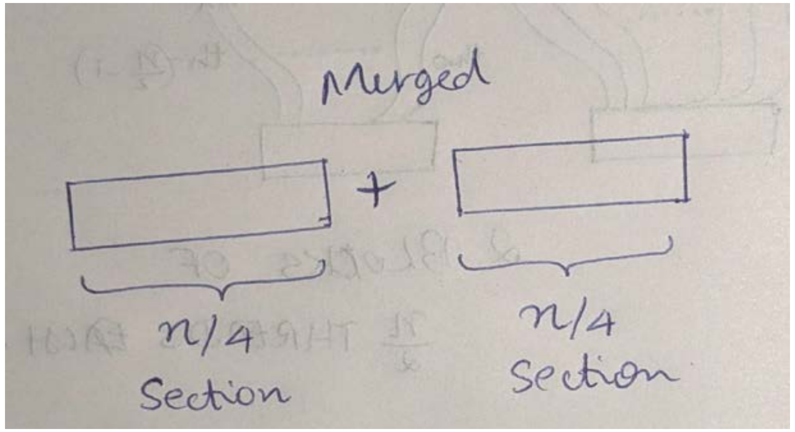


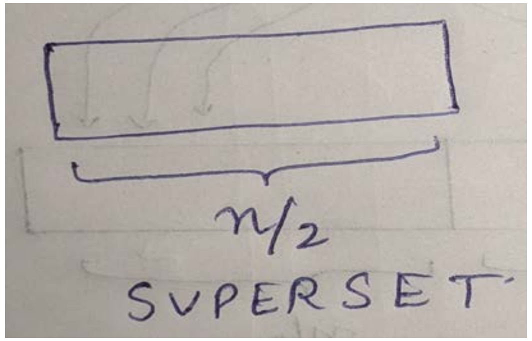
- The original array arr[] is of size n
- The array arr[] is initially split into 4 sections
- A block of 4 threads is launched : **Each thread is responsible for sorting each of the 4 sections**. (BUBBLE-SORT IS EMPLOYED FOR THIS)

STAGE-2 (Iteration-1)

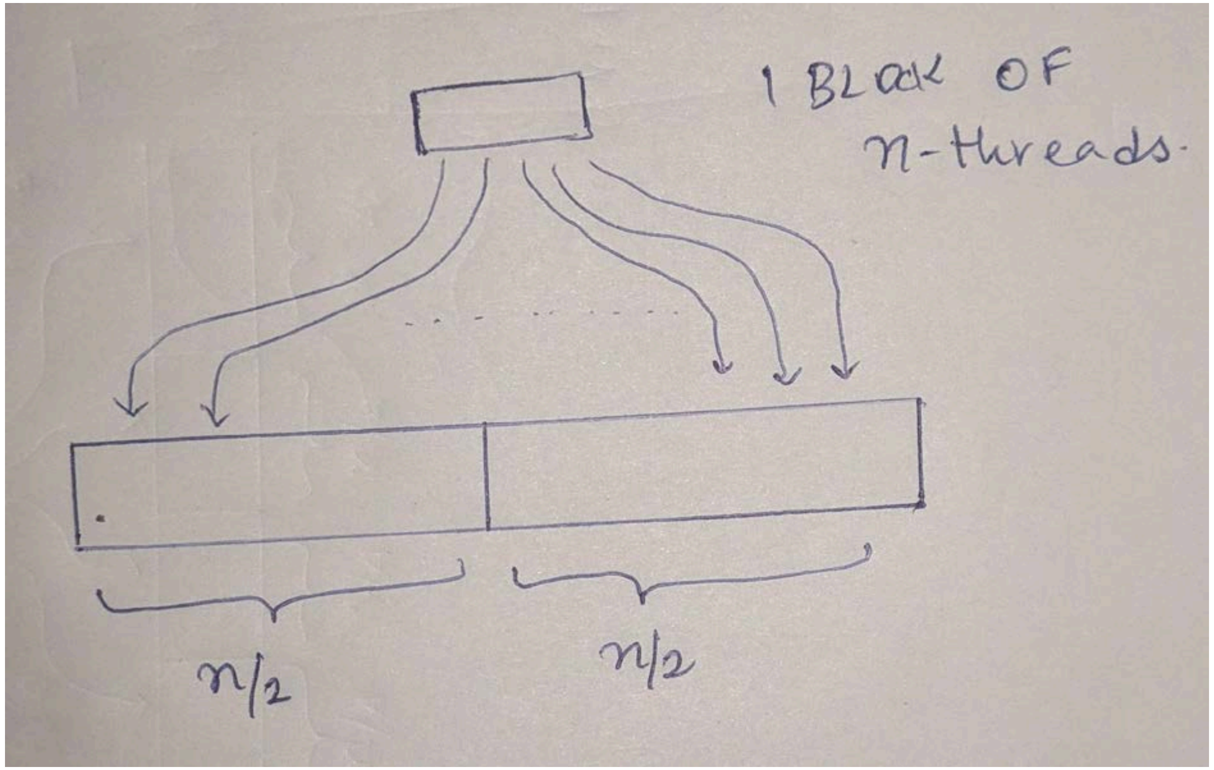


- Now, each section of n/4 elements has been sorted (STAGE-1)
- **2 blocks of n/2 threads** are launched.
- Each block parallelly **merges a pair of 2 sections of size: n/4 to produce a sorted SUPERSET of size: n/2**
- Each Block of n/2 threads is responsible for producing **1 SUPERSET of size: n/2**

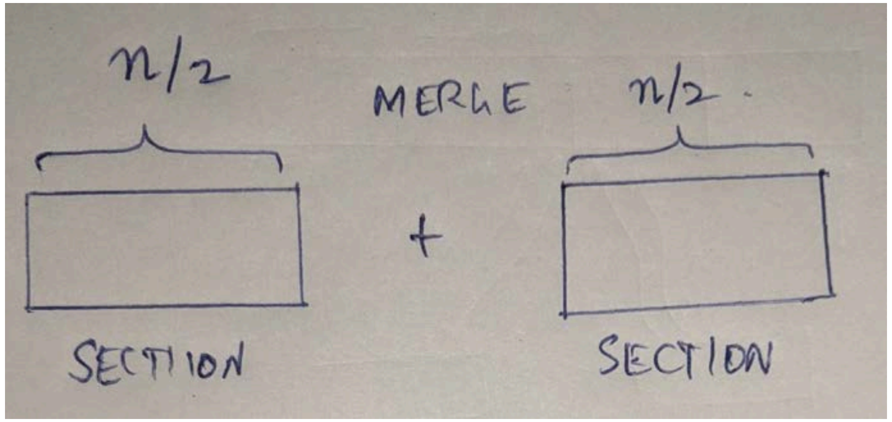


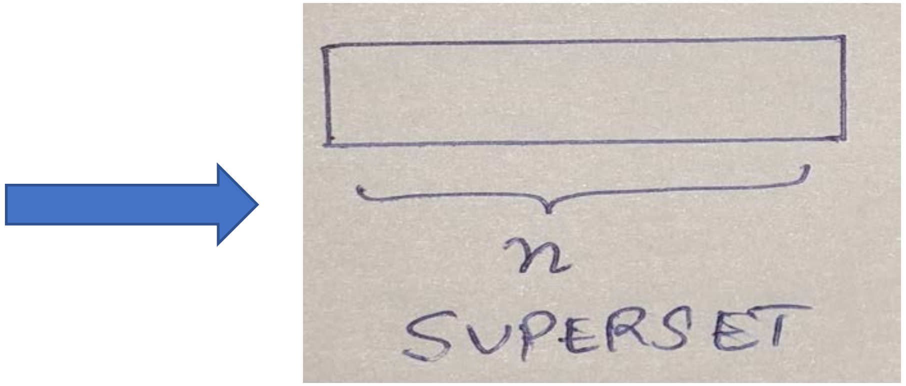


STAGE-2 (Iteration-2)



- Now, each section of n/2 elements has been sorted (STAGE-2 Iteration 1)
- **1 blocks of n threads** are launched.
- This block **merges a pair of 2 sections of size: n/2 to produce a sorted SUPERSET of size: n**
- This Block of n threads is responsible for producing **1 SUPERSET of size-n**



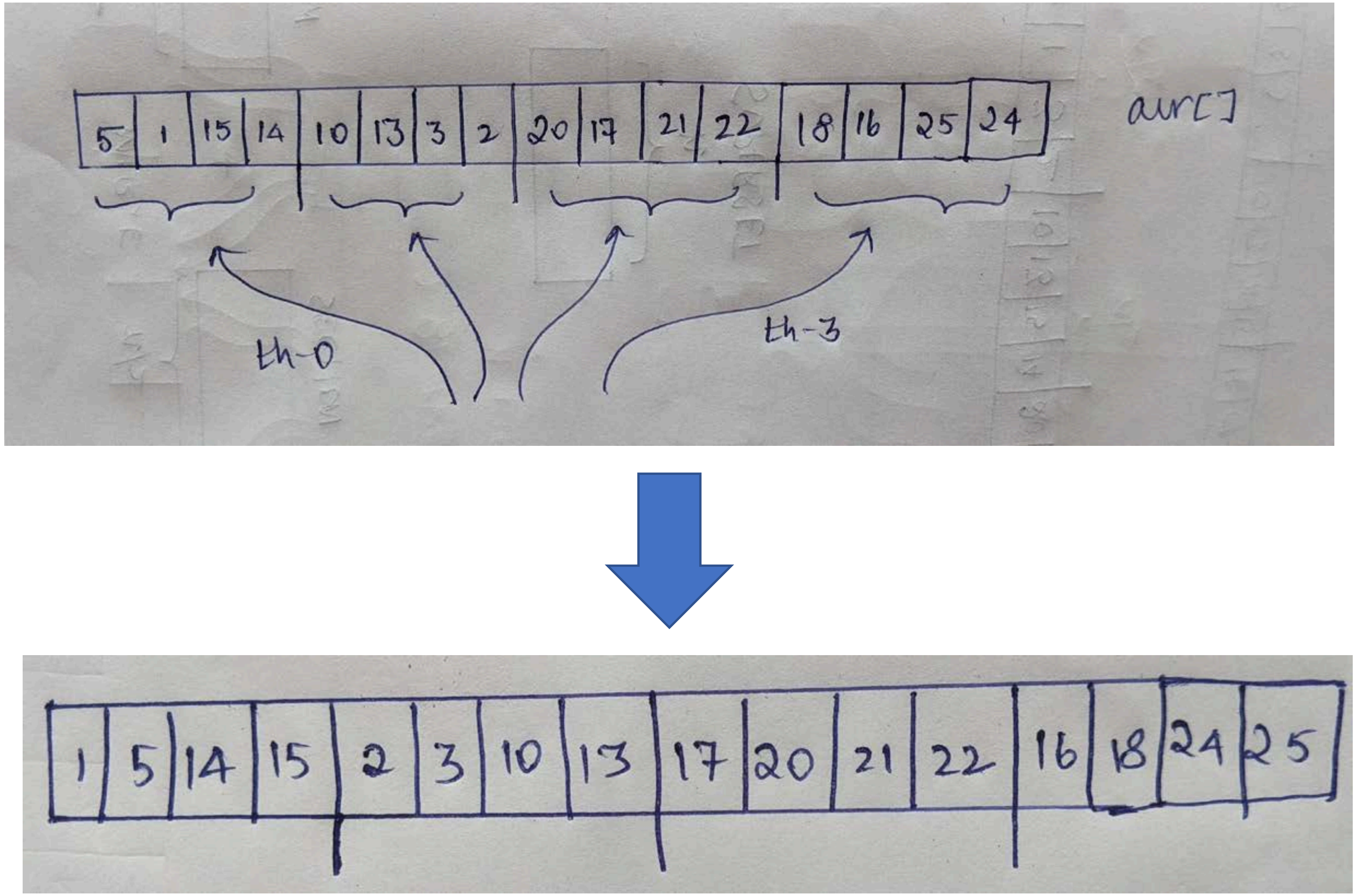


MERGE SORT PARALLEL ALGORITHM

EXAMPLE RUN WITH 4 INITIAL SPLITS

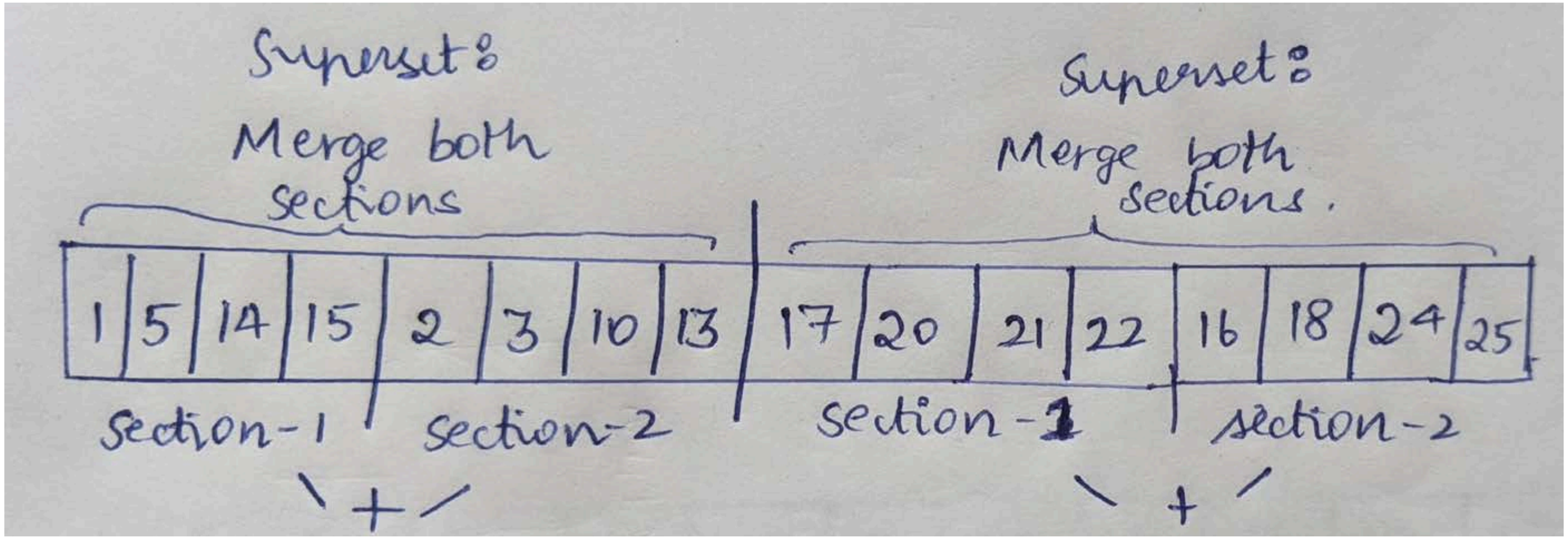
Example **arr[] = {5,1,15,14,10,13,3,2,20,17,21,22,18,16,25,24**}

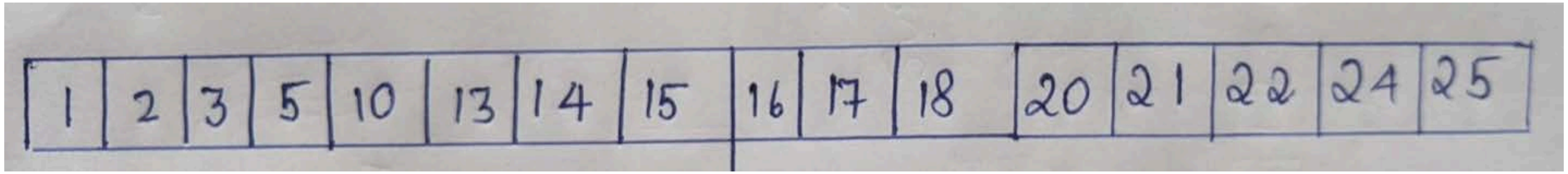
STAGE-1:
ARRAY IS DIVIDED INTO SECTIONS AND EACH SECTION IS SORTED



- **4 threads** are launched
- Each thread **individually sorts a section of 4 elements**

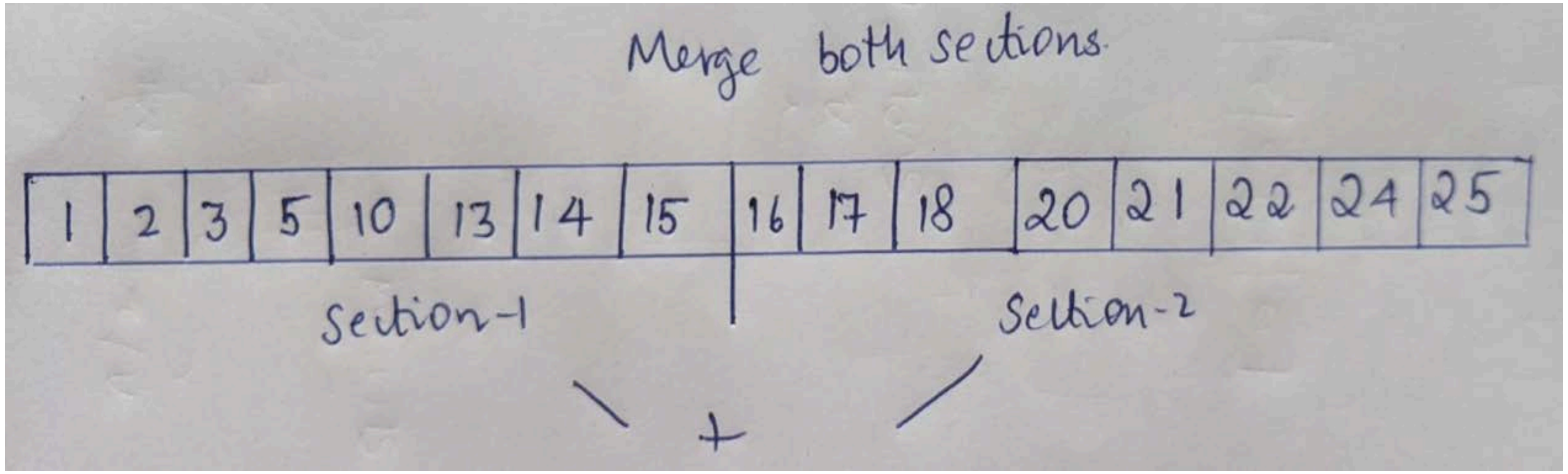
STAGE 2: ITERATION-1
EACH SORTED SECTION IS MERGED IN PAIRS TO PRODUCE SORTED SUPERSETS

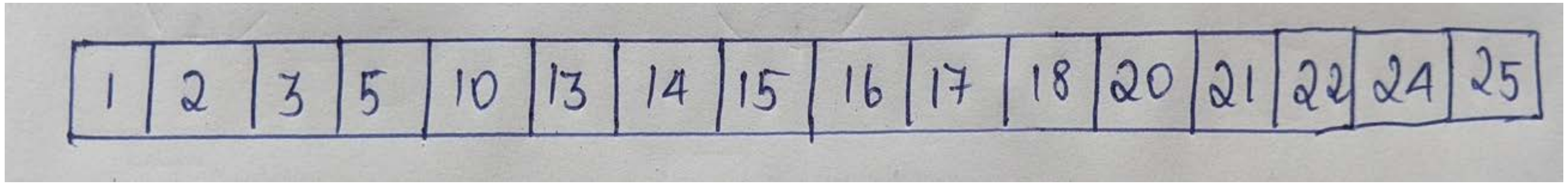




- **2 Sections on left** are **merged** to form a sorted **superset** in left
- **2 sections in right** are **merged** to form a sorted **superset** in right

STAGE 2: ITERATION-2
EACH SORTED SECTION IS MERGED IN PAIRS TO PRODUCE SORTED SUPERSETS

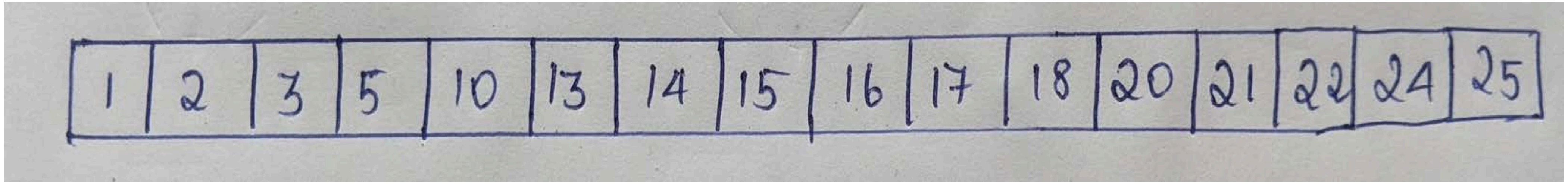




2 Sorted Sections of size-n/2 each are **merged** to form a **sorted superset of size n**

SORTED ARRAY IS FINALLY OBTAINED

arr[] = {1,2,3,5,10,13,14,15,16,17,18,20,21,22,24,25}



MERGE SORT CUDA IMPLEMENTATION

KERNEL : section_sort()

```
//Entire block for the entire array. Each thread takes care of bubble-sorting an individual section of size : *section_length
__global__ void section_sort(int* nums, int section_size)   //(, int n)
{
    //The thread with thread index = idx will take care of nums[] from : [( section_size * idx ) to ( section_size * (idx + 1) - 1 )]
    //For example: idx = 1 and section_size = 20, then, thread with idx = 1 will take care of nums[ 20: 39 ]

    int idx = threadIdx.x;



    //Bubble sort nums[] from index  [ ( section_size * idx ) : ( section_size * (idx + 1) - 1 ) ]
    bubblesort(nums, section_size * idx, (section_size * (idx + 1) - 1));

}
```

Takes as INPUT:

- nums*   : The input array , which is to be divided into sections and sorted sectionally
- section_size : The size (number of elements) of each section

Produces OUTPUT:

- **nums[]** Input array but with **each section individually sorted**

section_sort() : KERNEL-CALL AND WORKING

```
section_sort <<<1, div_num>>> (d_arr, section_size);
```

- *div_num* = Number of sections into which arr[] is split into.
- **A single block of *div_num* number of threads are launched**.
- **Each thread will be responsible for sorting one specific section of the arr[].**
- **Thread number idx** will be responsible for the section:

  [idx * section_size : (idx+1) * section_size - 1]
- Bubble-Sort() is employed by each thread to sort the section its responsible for.

Stage-2

Merge Sorted Sections

Merge Kernel Call

```
//Number of Threads = Size of Superset
//Number of blocks = Number of supersets
merge <<< superset_num, n/superset_num, n/superset_num >>> (d_arr, section_size, d_out_temp);
```

- **Number of Blocks** = Number of Supersets
- **Number of threads** = Length of each superset

- **Each Block** : Is responsible for a superset (merging the 2 sections of its superset)
- **Each thread** : Is responsible for 1 element of the superset

KERNEL: merge()

```
__global__ void merge(int* arr, int section_length, int* d_out_temp)
{
    //int section_length = *section_size;
    int superset_length = section_length * 2;   //Block will be 2 * (size of 1 section). Because 2 sections are merged
    int idx = threadIdx.x;
    int b_idx = blockIdx.x;

    //Length of arr1[] and arr2[] are section size
    int len1 = section_length;
    int len2 = section_length;

    //-----Select *arr1 and *arr2 and *d_out_curr------------------------

    int* arr1 = arr + (b_idx * superset_length);
    int* arr2 = arr1 + (section_length);
    int* d_out_curr = d_out_temp + (b_idx * superset_length);   //Determine d_out_curr[], the output array for current merge

    //Dynamically allocated shared memory array.
    // scat_ad[] from index [0 to n1-1] is for arr1[].
    //scat_ad[] from index [n1 to n2-1] is for arr2[]

     //Create a shared memory of size n1+n2 to accomodate the scatter-addresses corresonding to each element in arr1[] and arr2[]
    extern __shared__ int scat_ad[];
```

Takes as INPUT:

- **arr*** : Input array
- **section_length** : length of each section, which are to be merged in pairs
- **d_out_temp*** : temporary array used to store intermediate output

Produces As Output:

- **arr[]** but **with pairs of sections of section_length merged** to form **bigger sections of size section_length*2**

Merge Kernel :Details

- **section** : Divisions in arr[] . They will be merged in pairs.
- **superset** : The result formed from merging to 2 sorted sections
- **superset_length** : Length of each superset.
- **arr1*** : Points to first section of superset.
- **arr2*** : Points to second section of superset
- **len1** = **len2** = Length of arr1[] = Length of arr2[]
- **scat_ad[]** : **A shared memory array of length = superset-length**.
    - <u>**It will hold the scatter addresses for each element of sorted superset**</u>

Merge : Scatter Address Calculation

```
//--------------------------------These threads are responsible for arr1[]------------------------------------------------------
if (idx <= len1 - 1)
{
    int idx1 = idx;     //Number of elements in arr1[] that are lesser than arr1[idx]. idx1 = index of current element in arr1[]

    int target = arr1[idx1];    //Target is current element in arr1[]

    //-------------Find idx2----------------------------------------Binary Search Part------------------------------
    int idx2 = bin_search(arr2, target, len2) + 1;    //Number of elements in arr2[] that are lesser than arr1[idx].....

    //Calculate and store the scatter address in array
    //scat_arr1[idx] = idx1 + idx2;     //If there are 2 elements before a number in output array, its index will be 2

    scat_ad[idx] = idx1 + idx2;    //Scatter address correspinding to arr1[idx] = idx1 + idx2
}
```

Each thread is responsible for producing scatter address of one element in the superset

- **If (thread index < len1):** Current thread is responsible for an element in arr1[]
- **Target = arr1[idx]** (**Current Number whose scatter address is to be calculated**)
- **Idx1** = Number of elements smaller than target in arr1[] = idx
- **Idx2** = Number of elements smaller than target in arr2[]
    - **To calculate idx2, we perform a binary_search for target in arr2[]**
- **<u>Scatter address of arr[idx] in sorted array will be (idx1 + idx2</u>), i.e, the number of elements in superset smaller than target. Hence, store scatt_ad[idx]= idx1+idx2**

```
//-------------------------------------These threads are responsible for arr2[]-------------------------------------------
else if (idx >= len1)
{
    //Number of elements in arr2[] that are lesser than arr2[idx].
    //idx1 = index of current element in arr2[]
    //(idx-len1) because threads with index n1 to n2-1 are responsible for arr2[] index [0: n2-1]
    int idx1 = idx - len1;

    int target = arr2[idx1];    //Target is current element in arr1[]

    //-------------Find idx2---------------------------Binary Search Part-------------------------
    int idx2 = bin_search(arr1, target, len1) + 1;    //Number of elements in arr1[] that are lesser than arr2[idx].  +1 bcos we want appro

    //Calculate and store the scatter address in array
    //scat_arr1[idx] = idx1 + idx2;     //If there are 2 elements before a number in output array, its index will be 2
    scat_ad[idx] = idx1 + idx2;    //Scatter address corresponding to arr2[idx - len1] = idx1 + idx2

}

__syncthreads();   //Barrier to ensure that all threads have finished writing scat_ad[].-----------------Not necessary
```

If idx >= len1: Current thread is responsible for an element in arr2[]

Calculating scatter address is similar to previous case

__syncthread() : Barrier to ensure all threads have finished calculating scatter addresses.

Place the elements in Superset in appropriate sorted positions

```
//------------Store the output in respective position in d_out_temp[] using scatter address so that they are in sorted order----------
/* ... */
d_out_curr[scat_ad[idx]] = arr1[idx];

__syncthreads();
//--------------------------------------Copy sorted elements back to array-----------------------------------------------------


arr1[idx] = d_out_curr[idx];

//printf( "%d ", arr1[idx] );
}
```

1)Store the elements in **current superset from arr[] to d_out[]** in **appropriate sorted position using scatter address** from **scat_ad[] array** (d_out[] is temporary array)
2)Barrier : Ensure all threads have finished copying to d_out[]
3) Copy back to arr[] from d_out[]

merge_sort()

CPU CODE

merge_sort() Function

```
// ...
void merge_sort()
{
    GpuTimer timer;

    // ...


    int h_arr[64] = { 100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,81,80,79,78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,62,61


    int n = sizeof(h_arr) / sizeof(int);      //n = Total size of host array

    int div_num = 8;    //How many parts the array is initially split.
    int section_size = n/div_num;              // section_size = Size of each section after splitting arr[] into div_num parts (Stored in Hos


    //-----------------------------------Create input and output arrays in GPU--------------------------------------------
    int* d_arr, * d_out_temp;                                                                  // *d_out2;
    cudaMalloc((void**)&d_arr, n * sizeof(int));
    cudaMemcpy((void*)d_arr, (void*)h_arr, n * sizeof(int), cudaMemcpyHostToDevice);   //d_arr[] is input array in device

    cudaMalloc((void**)&d_out_temp, n * sizeof(int));           //d_out_temp[] is temporarily used to store sorted block elements
```

- h_arr[] : Input array to be sorted (Stored in HOST)
- d_arr[] : Input array (Stored in Device)
- d_out_temp[] : temporary array (Device)

contd.

- **div_num**: **Number of sections** into which the arr[] is initially SPLIT
- **section_size:** Length of each section = Total array length / number of sections = n /div_num

STAGE – 1: SECTION-SORT KERNEL CALL

```
//------------------------------Stage-1: KERNEL CALL: Bubble Sort Each Section of section_size elements------------------------------

section_sort <<<1, div_num>>> (d_arr, section_size);    //Call div_num threads: Each thread bubble-sorts a sub-section of n/div_num el
```

- section-sort kernel call will bubble-sort all sections in arr[]. (The array is divided into div_num number of sections each of section_size)
- KERNEL calls 1 block of div_num threads, 1 thread for each section in arr[].
- Each thread will sort its section

STAGE-2: ITERATIVELY MERGE SECTIONS UNTIL ENTIRE ARRAY IS SORTED

```
//Initially, section_size = n / div_num
int superset_num = div_num / 2; //Is the total number of supersets in the array, each of which are merged by a separate block. Initially,
while (superset_num >= 1)
{

    /* ... */
    // ...

    //Number of Threads = Size of Superset
    //Number of blocks = Number of supersets
    merge <<< superset_num, n/superset_num, n/superset_num >>> (d_arr, section_size, d_out_temp);

    //UPDATE : superset_num ( halved ) and section_size (doubled)
    superset_num = superset_num/2;
    section_size = section_size*2;

}
```

- Iteratively call merge kernel
- In each iteration, merge sections into supersets.
- After each iteration, **section size is doubled and number of superset is halved** (Because after each iteration, each pair of section is merged, so that current superset is a section for the next iteration)
- **In Final iteration, only a single pair of sections is to be merged and then, sorted array will be obtained**

END