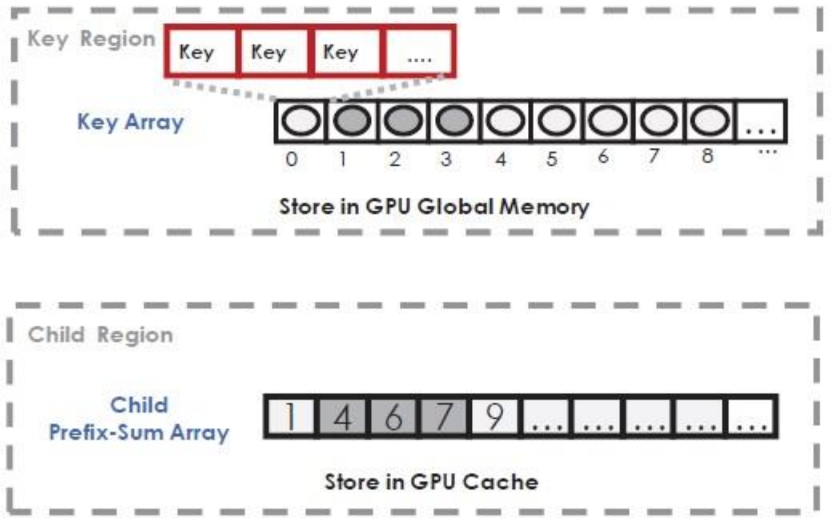
Structure of Harmonia

- Structure of Harmonia consist of precisely two things.

1. vector <struct Node> key_region
2. vector <int> child_prefix_sum

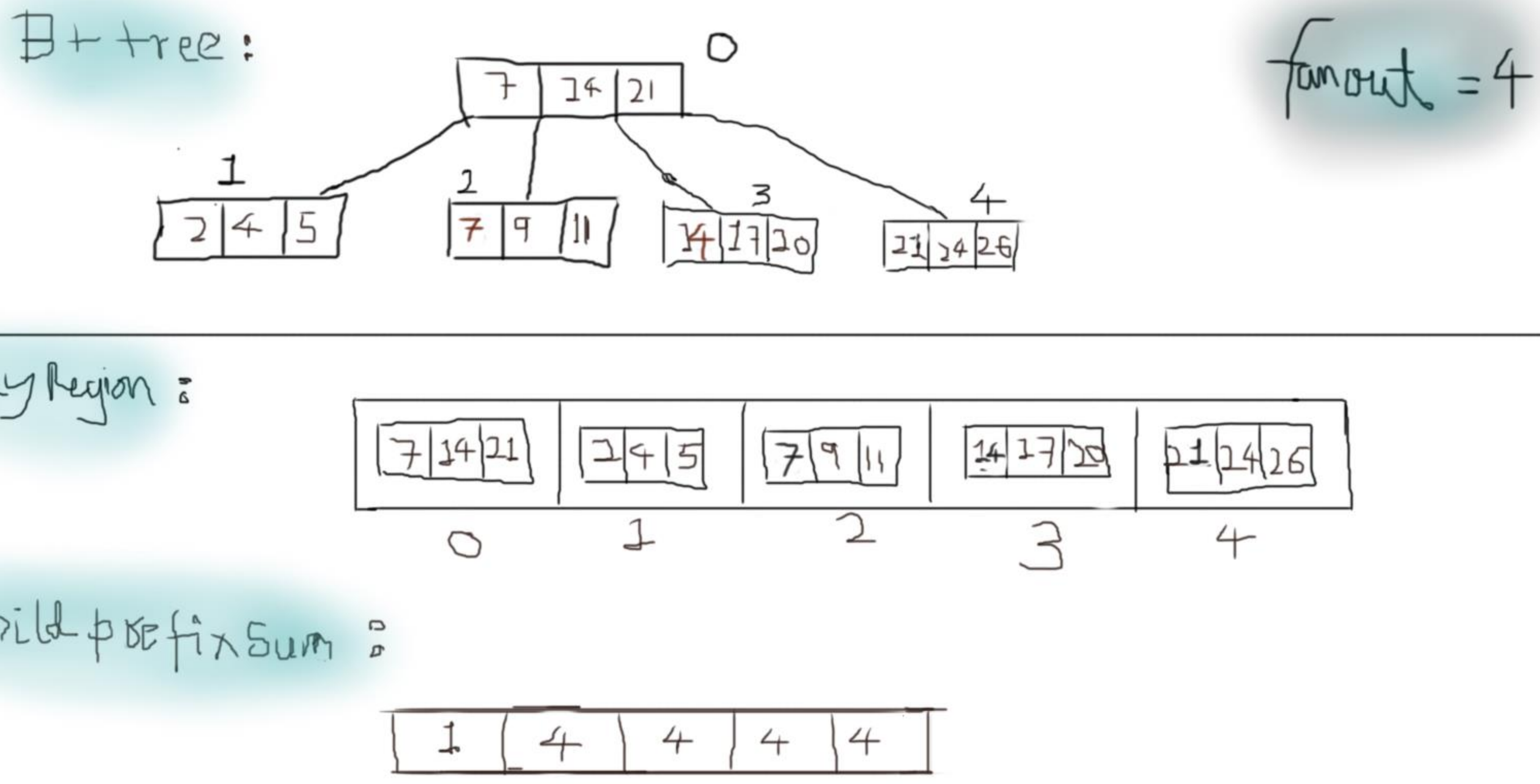
- Apart from this two there one additional global variable used to initialize the child_prefix_sum

1. psum



(b) Harmonia tree structure

Example

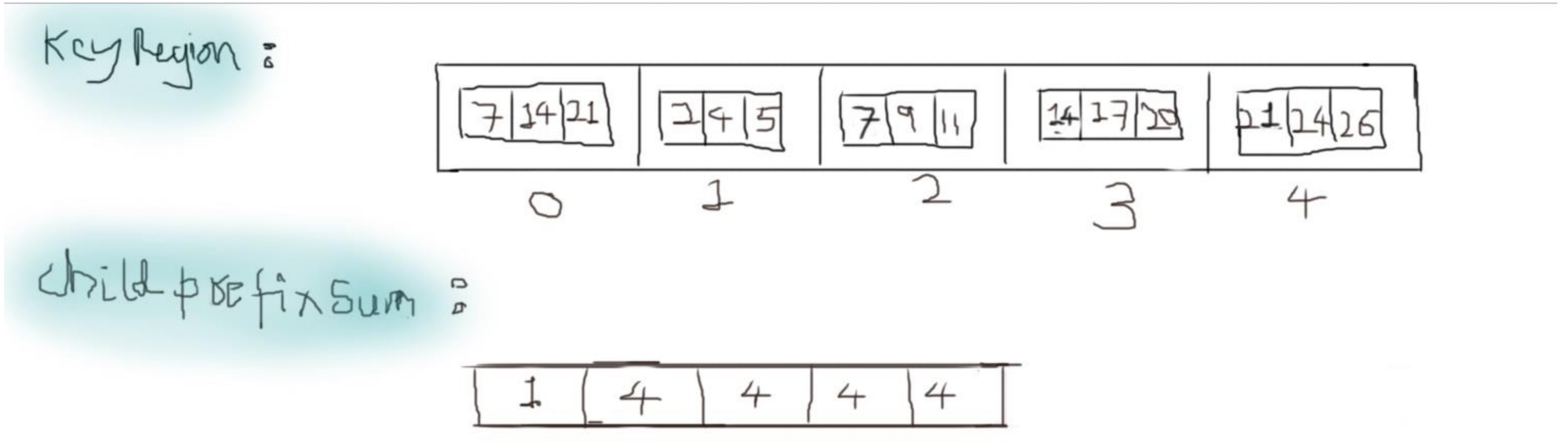

B+ tree:
0
7 | 14 | 21
1
2 | 4 | 5
2
7 | 9 | 11
3
14 | 17 | 20
4
21 | 24 | 26
fanout = 4
KeyRegion:
7 | 14 | 21
2 | 4 | 5
7 | 9 | 11
14 | 17 | 20
21 | 24 | 26
0
1
2
3
4
childprefixSum:
1 | 4 | 4 | 4 | 4

Algorithm createHarmonia(Blocks , column_size)

```
Create a new vector of blocks called newBlocks
for each block inside Blocks do
        store block address of current Block node in curBlock named block pointer
        create temporary Node t and add keycount , all keys and all records in to it from the curBlock
        push  the temporary Node t into the key_region vector
        push all the child of the curBlock into the newBlocks vector and increment psum for each child being pushed
        if it is first child of the curBlock then
                push psum to the child_prefix_sum vector
for end

If size of newBlocks vector is empty then
        clear Blocks vector and we are done.
else
        clear Blocks vector
        recursively call createHarmonia with newBlocks and column_size i.e. createHarmonia( newBlocks , column_size)
```

Searches

- Several other operations like Range queries and Update to the database requires the searching of the particular key firstly and then followed by that respective operation.
- Hence we will only discuss searches here in detail. Rest two operation are just variation of the search operation.
- Let’s search the value 17 in our previously constructed structure of Harmonia.



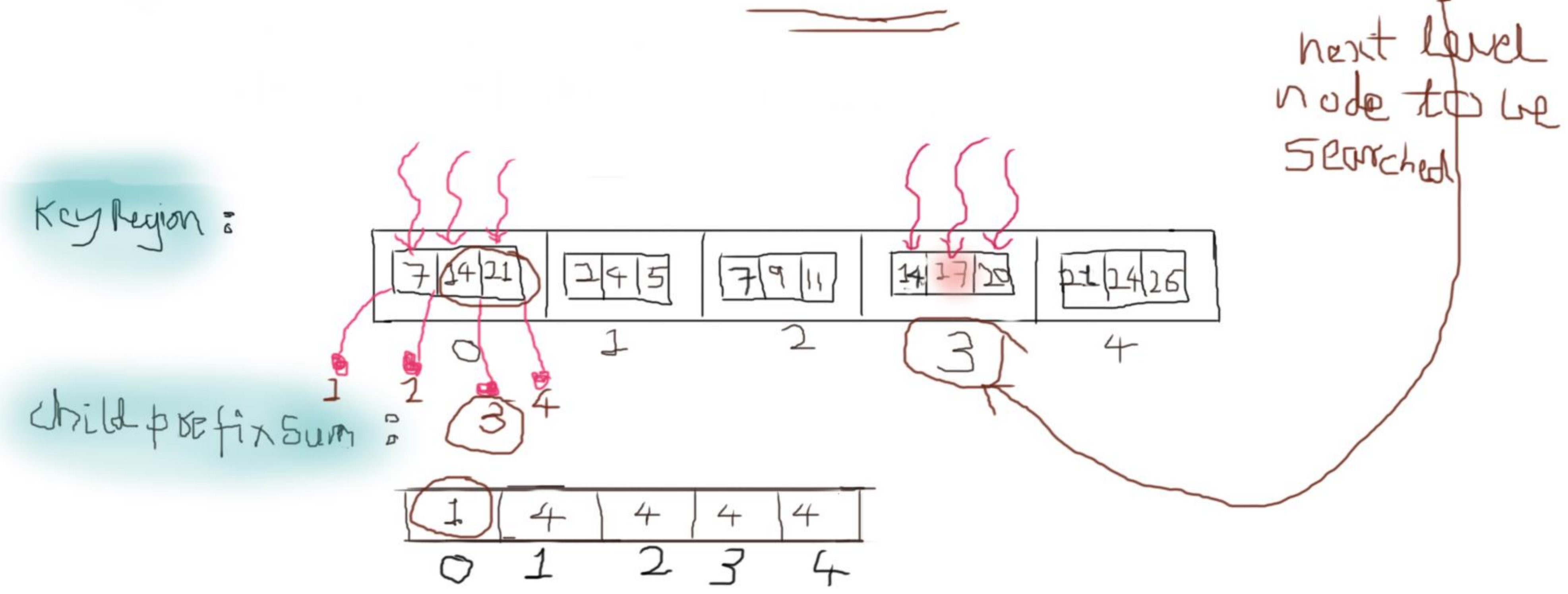
So, How we will Search using given structure?

- Following equation will help in searching in current structure of Harmonia.

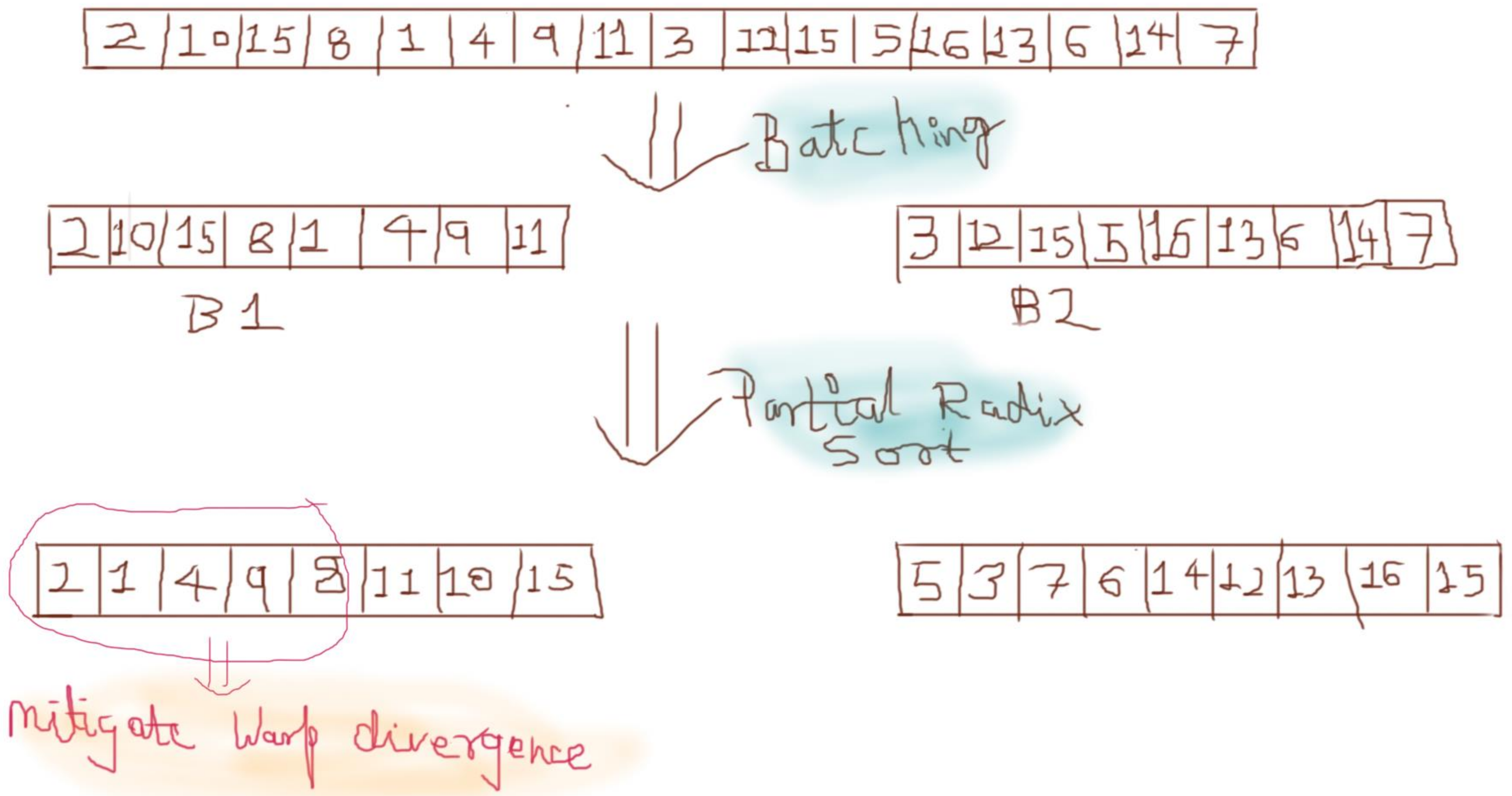
$$child\_idx = PrefixSum[node\_idx] + i - 1 \qquad (1)$$

Searching : 17

(i) $\underline{\text{child\_id}} = 1 + 3 - 1 = 3$

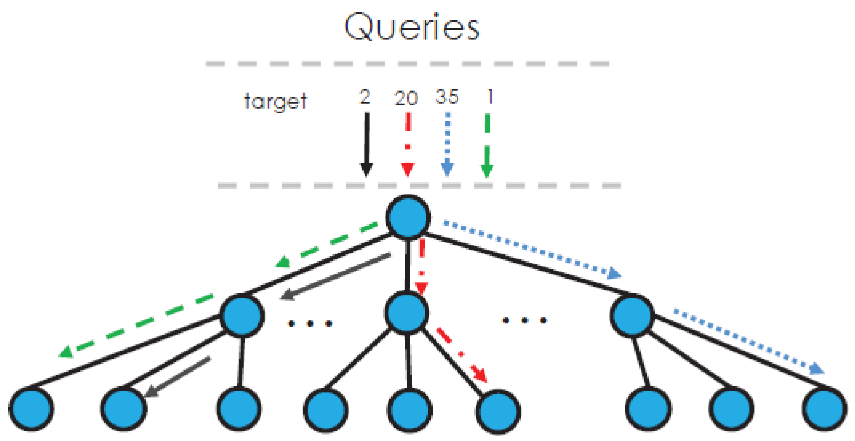


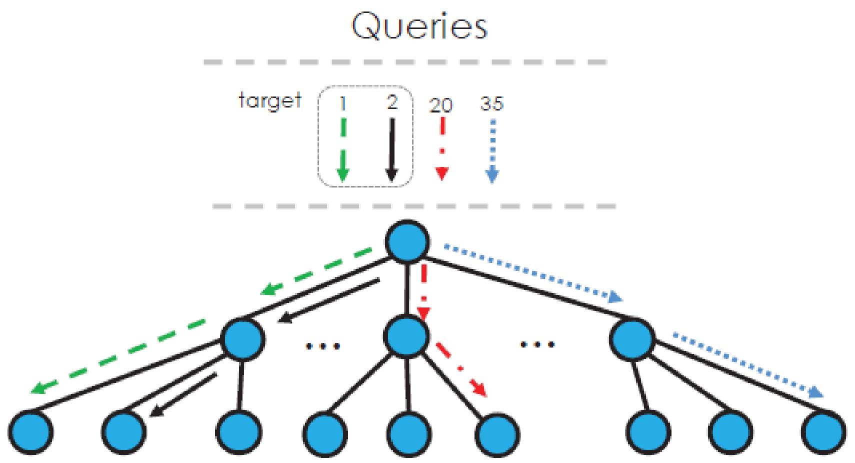
Optimization 1: Batch-wise Partial Radix Sort on GPU (on search keys)



But was is the use of that?

- It make memory access coalesced and also mitigate warp divergence





- Tuning of the number of MSB bits used for sorting is need to get optimal performance.

$$N = B - \log_2\left(\frac{2^B}{T} * K\right)$$

B = each key is represented by B bits

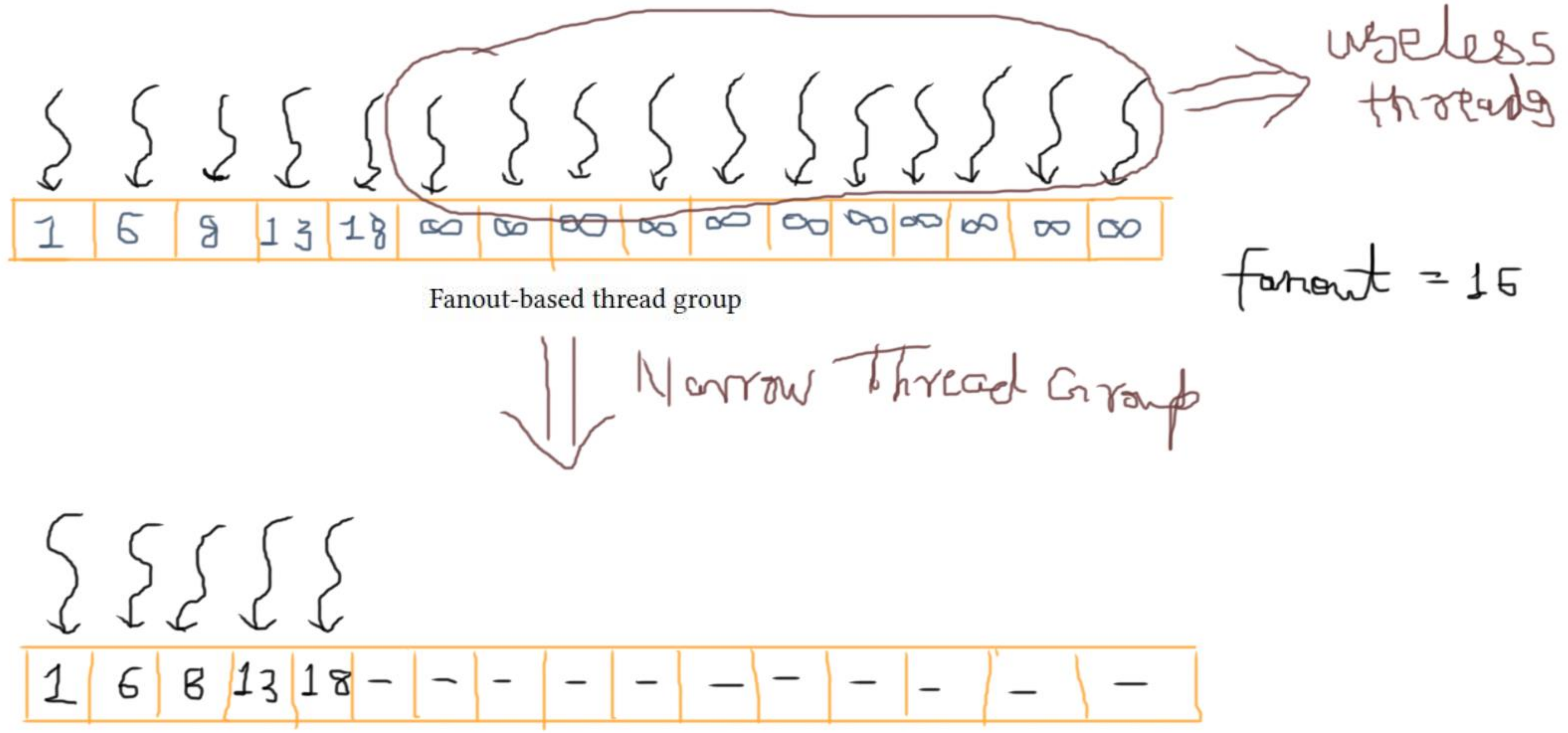
T = the size of traversed B+ tree is T

K = cache line can save K keys

N = MSB N bits that has to be used for sorting by radix sort.

Optimization 2: Removing Unnecessary Computations (inside search procedure) or Narrow Thread Grouping

- Narrow the thread group based on the current number of keys present in the Node.



Fanout-based thread group

Narrowed thread group

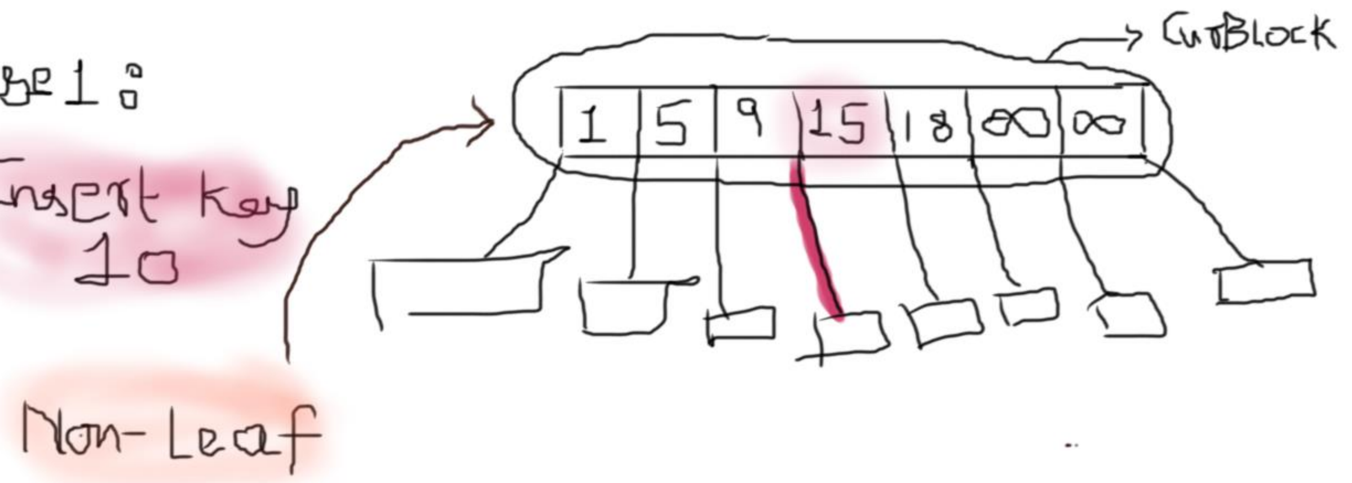
Procedure Search()

```
__global__ void search( struct Node *a , int *b , int asize , int bsize , int *search_keys, int *mutex , int n , char mode){

    // task 1 is to assign individual searches to each threads
    int key = search_keys[blockIdx.x];
    __shared__ int index ;
    __shared__ int prev_index;

    index = 0;
    prev_index = 0;

    __syncthreads();


    // task 2 is to perform search using harmonia
    while(true){

        if( index > asize-1 ){
            break;
        }
        prev_index = index;
        // need to divide this for loop among several threads to implement NTG.

        if( threadIdx.x < a[index].count ){
            if( a[index].keys[ threadIdx.x][0] == key ){
                index = b[index] + threadIdx.x + 1;        eq1
                goto bottom;
            }
            if(threadIdx.x != a[index].count - 1)
            if( a[index].keys[ threadIdx.x][0] < key  && key < a[index].keys[threadIdx.x + 1][0]  ){
                index = b[index] + threadIdx.x +1;
                goto bottom;
            }
            if(threadIdx.x == 0){
                 if( a[index].keys[0][0] > key ){
                index = b[index] + 0;
                goto bottom;
                }
                if( a[index].keys[ a[index].count-1 ][0] < key ){
                index = b[index] + a[index].count;
                goto bottom;
                }

            }

        }
      bottom:   __syncthreads();
  }
```

How InsertKey works?

Case 1:

Insert key 10

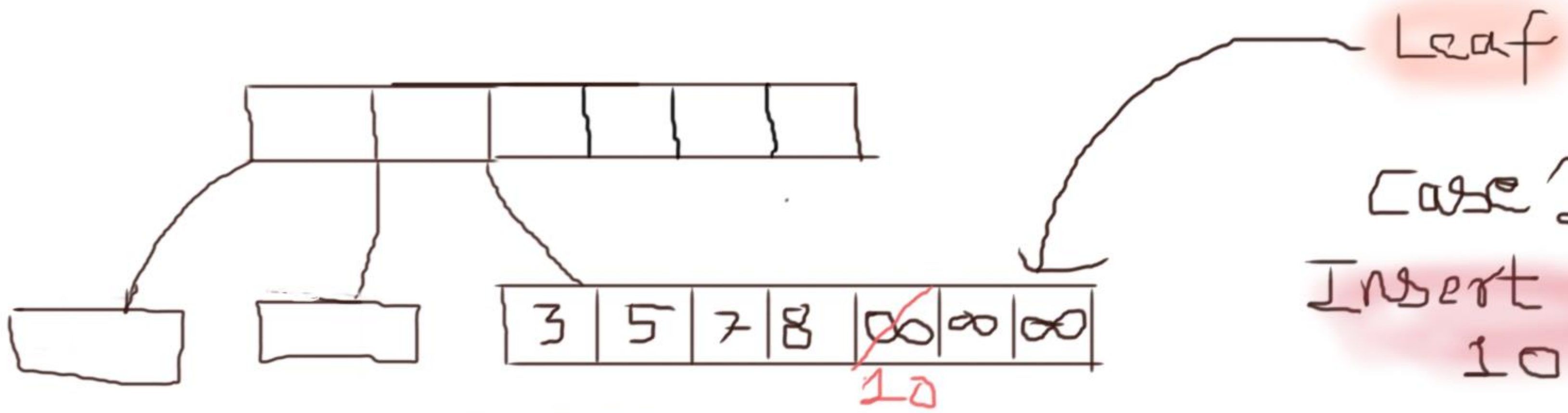


$10 < 15$

value < currBlock → value[i] and not null

Case 2:

Insert key 10



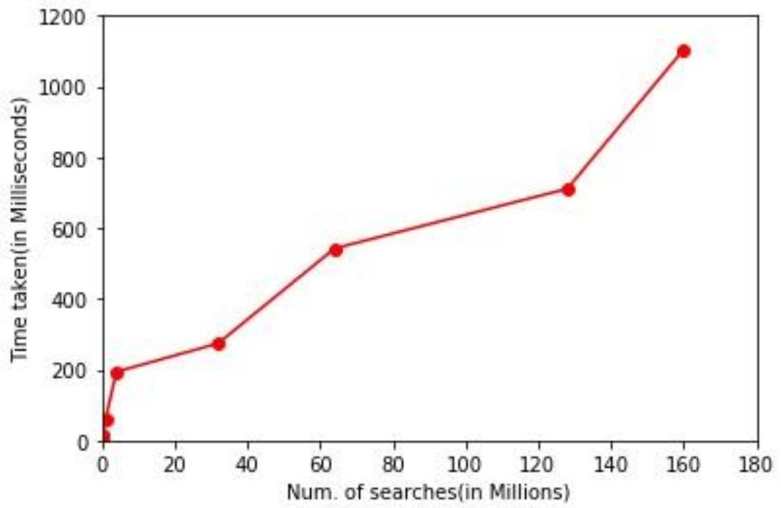
$10 < \infty$

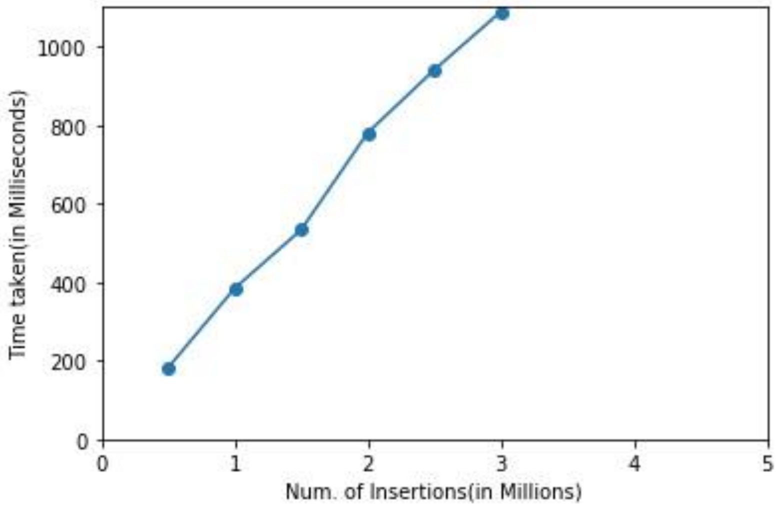
value < currBlock → value[i] and null

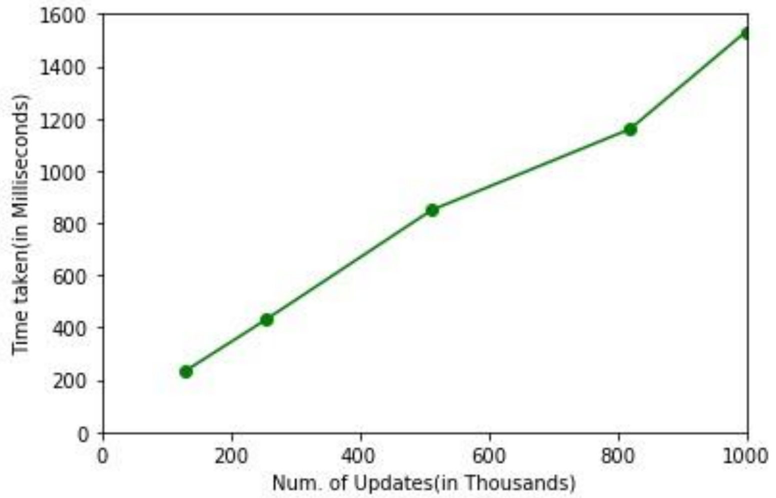
Algorithm InsertKey(curBlock, value , recordptr)

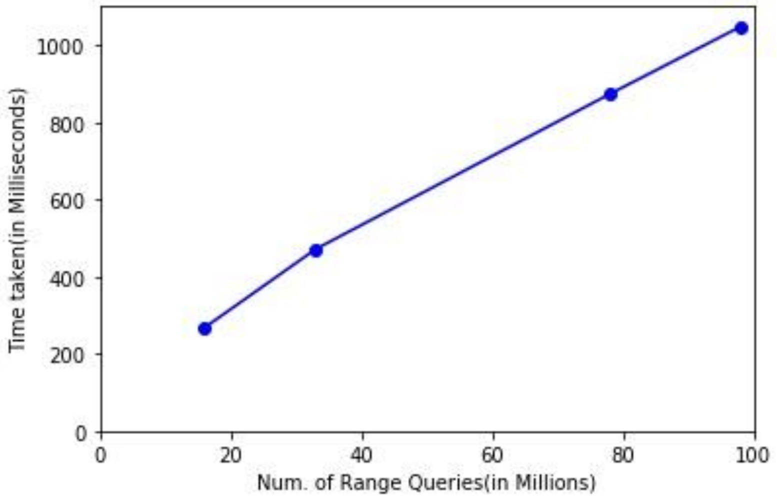
```
for each keys in the curBlock
        if  value < curBlock->value[i]   and curBlock->childBlock[i] is not empty then     // case 1
                recursively call InsertKey on next level of tree pointed by that childBlock[i] i.e.  InsertKey(curBlock->childBlock[i] , value , recordptr)
                if curBlock is at its max capacity
                        split curBlock as Non Leaf node
                return as insertion is completed
        else if value < curBlock->value[i] and curBlock->childBlock[i] is empty then     // case 2
                swap curBlock->value[i], and value
                update the number of keys present in curBlock
for end.

if curBlock is at its max capacity
                split curBlock as Leaf Node.

end.
```

All right but what about performance of all four operations?


Time taken(in Milliseconds)
Num. of searches(in Millions)


Time taken(in Milliseconds)
Num. of Insertions(in Millions)


Time taken(in Milliseconds)
Num. of Updates(in Thousands)


Time taken(in Milliseconds)
Num. of Range Queries(in Millions)