GPU Computing

Lab 10

BFS

Parallelo

Struttura di grafo a liste in C++

array A offset dei gradi sommati

array E delle liste di adiacenza

Restituisce il grado di **i**

Dice se **j** è nel vicinato di **i**

```
struct GraphStruct {
    node nNodes{0}; // num of graph nodes
    node_sz nEdges{0}; // num of graph edges
    node_sz* cumulDegs{ nullptr }; // cumsum of node degrees
    node* neighs{ nullptr }; // list of neighbors for all nodes (edges)

    // return the degree of node i
    node_sz deg(node i) {
        return ( cumulDegs[i + 1] - cumulDegs[i] );
     }
     // check whether node i is a neighbor of node j
     bool areNeighbor(node i, node j) {
         for (unsigned k = 0; k < deg(j); k++)
          if (neighs[cumulDegs[j]+k] == i)
             return true;
           return false;
     }
}
```

CUDA BFS

Frontiera: array F_a

Visitati: array X_a

Costo: array C_a

Algorithm 1. CUDA_BFS (Graph $G(V,E)$, Source Vertex S)

1: Create vertex array V_a from all vertices and edge Array E_a from all edges in $G(V,E)$,
2: Create frontier array F_a, visited array X_a and cost array C_a of size V.
3: Initialize F_a, X_a to false and C_a to ∞
4: $F_a[S] \leftarrow$ true, $C_a[S] \leftarrow 0$
5: **while** F_a not Empty **do**
6: **for** each vertex V in parallel **do**
7: Invoke CUDA_BFS_KERNEL$(V_a, E_a, F_a, X_a, C_a)$ on the grid.
8: **end for**
9: **end while**

BFS Kernel

Frontiera: array F_a

Visitati: array X_a

Costo: array C_a

Algorithm 2. CUDA BFS_KERNEL $(V_a, E_a, F_a, X_a, C_a)$

```
1:  tid ← getThreadID
2:  if F_a[tid] then
3:      F_a[tid] ← false, X_a[tid] ← true
4:      for all neighbors nid of tid do
5:          if NOT X_a[nid] then
6:              C_a[nid] ← C_a[tid]+1
7:              F_a[nid] ← true
8:          end if
9:      end for
10: end if
```

Kernel cudaBFS (hints)

Frontiera: array F_a

Visitati: array X_a

Costo: array C_a

```
__global__ void cudaBFS(GraphStruct *str, bool *Fa, bool *Xa, int *Ca, bool *done, int n) {
   int nodeID = threadIdx.x + blockIdx.x * blockDim.x;   // node ID
   if (nodeID > n)
      return;
   if (Fa[nodeID]) {
         *done = false;
         Fa[nodeID] = false;
         Xa[nodeID] = true;
         int deg = str->cumDegs[nodeID + 1] - str->cumDegs[nodeID];
         int start = str->cumDegs[nodeID];
         for (int i = 0; i < deg; i++) {
              int neighID = str->neighs[start + i];
              if ( !Xa[neighID] ) {
                   Ca[neighID] = Ca[nodeID] + 1;
                   Fa[neighID] = true;
              }
         }
     }
}
```

Merge sort

Parallel sorting in place

Implementazione C

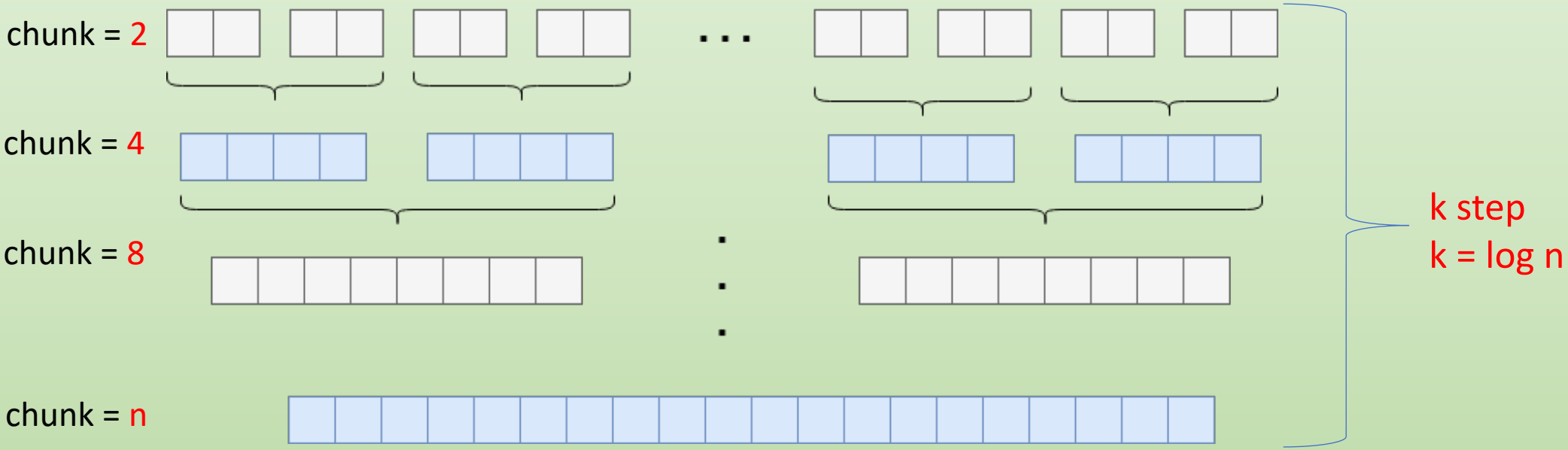
```
/* Main program to sort an array */
int main() {
    . . .

    mergeSort(arr, 0, arr_size - 1);

    . . .
    return 0;
}
```

```
/* l is for left index and r is right index of the
 sub-array of arr to be sorted */

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

```
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    /* create temp arrays */
    int *L = new int[n1];
    int *R = new int[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; j = 0; k = l; // Initial index of subarrais
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i]; i++;
        } else {
            arr[k] = R[j]; j++;
        }
        k++;
    }
    /* Copy the remaining elements of of L[], if there are any */
    while (i < n1) {
         arr[k] = L[i];
         i++; k++;
    }
    /* Copy the remaining elements of R[], if there are any */
    while (j < n2) {
         arr[k] = R[j];
         j++; k++;
    }
}
```

Procedura merge in parallelo

- ✓ Array n-dimensionale ($n = 2^k$)
- ✓ Chunk = $2, 4, 8, \ldots, 2^k$



Schema call

Il numero thread si dimezza via via…

Uso di array alternati per passaggi intermedi

```
. . .

for (int chunk = 2; chunk <= N; chunk *= 2) {
    int nThreads = N / chunk;


    dim3 block(min(nThreads, BLOCK_SIZE));
    dim3 grid((nThreads + block.x - 1) / block.x);


    printf("grid: %d, block: %d, chunk: %d\n", grid.x, block.x, chunk);


    if (array2sorted)
        cudaMergeSort<<<grid, block>>>(array, sorted, N, chunk);
    else
        cudaMergeSort<<<grid, block>>>(sorted, array, N, chunk);
    array2sorted = !array2sorted;
}
```

kernel (1 th per chunk)

chunk = 2,4,8,…

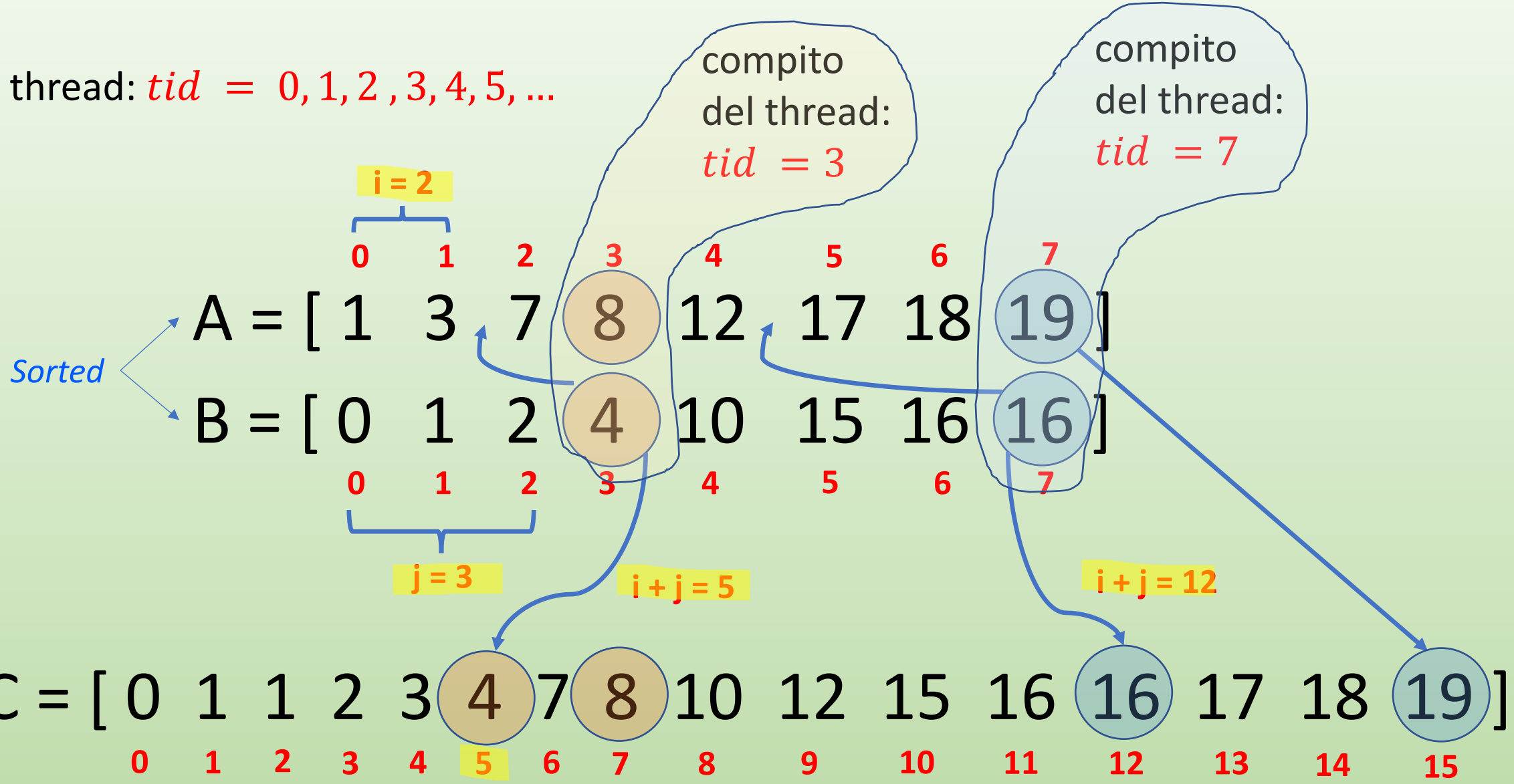
Calcolo estremi del chunk

Procedura merge

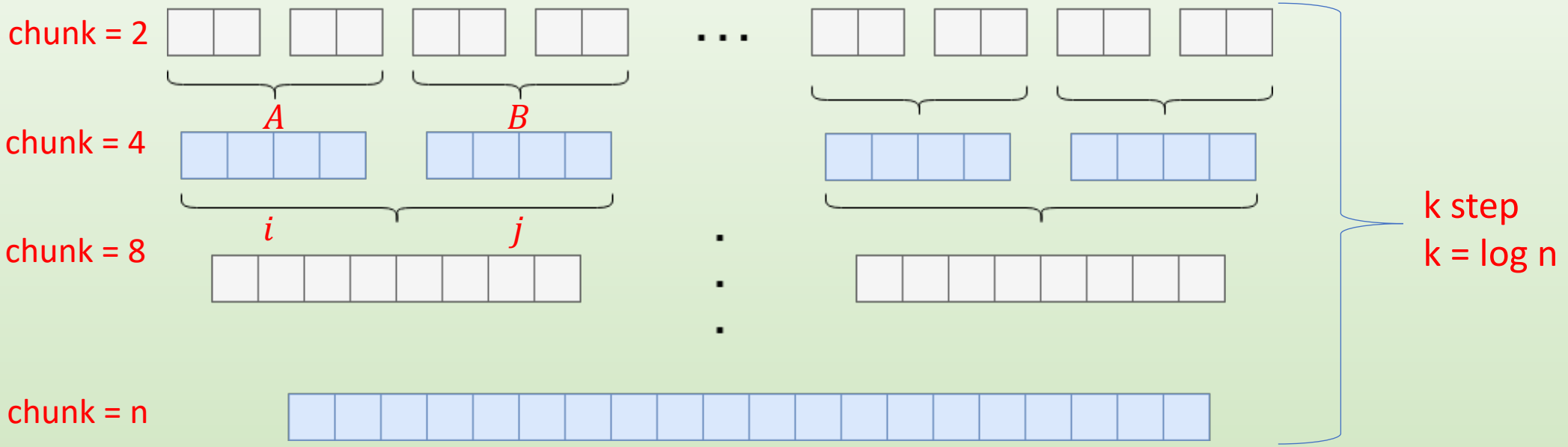
Sistema le ‘code’ eventuali

```
. . .
__global__ void cudaMergeSort(int *array, int *sorted, int n, int chunk) {
    int start = chunk * threadIdx.x + blockIdx.x * blockDim.x;
    if (start > n)
        return;
    int mid = min(start + chunk / 2, n);
    int end = min(start + chunk, n);
    int i = start, j = end, k = mid;

    //cudaMerge(array, sorted, start, mid, end);
    while (i < mid && j < end) {
        if (array[i] <= array[j])
            sorted[k++] = array[i++];
        else
            sorted[k++] = array[j++];
    }
    // Copy the remaining elements array[i] if there are any
    while (i < mid)
        sorted[k++] = array[i++];
    // Copy the remaining elements of array[j] if there are any
    while (j < end)
        sorted[k++] = array[j++];

}
```

Parallel binary merge



Procedura merge a più thread



- ✓ Indice di thread $tid = 0,1,2,3, \ldots$
- ✓ indice elementi $i = tid\%k + (tid - tid\%k) * 2$ (chunk inf A)
  $j = i + k$ (chunk sup B)

Test

```
*** Sorting array size N = 4194304
*** CPU processing... CPU
    elapsed time: 2.69321 (sec)
*** GPU ONE THREAD x chunk processing...
    elapsed time: 0.49219 (sec)
    speedup vs CPU: 5.47
*** GPU MULTI THREAD x chunk processing...
     elapsed time: 0.01057 (sec)
     speedup vs CPU: 254.82
     speedup vs GPU mono: 46.57
```