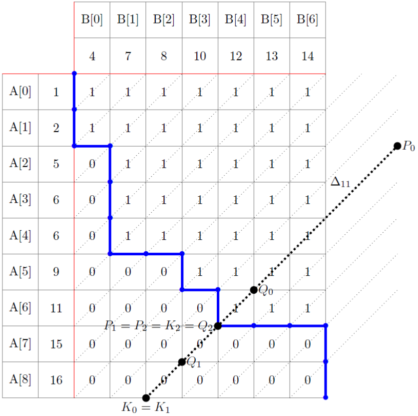
Batch merge path sort

2 avril 2021

Introduction

▶ **Objective** : Sorting several arrays $(M_i)_{i\in\{1,\dots,N\}}$ using the parallel **merge path algorithm**.




```
__device__ void trifusion(int * a, int * b, int * sol, int modA, int modB, int idx)
```

The sorting algorithm

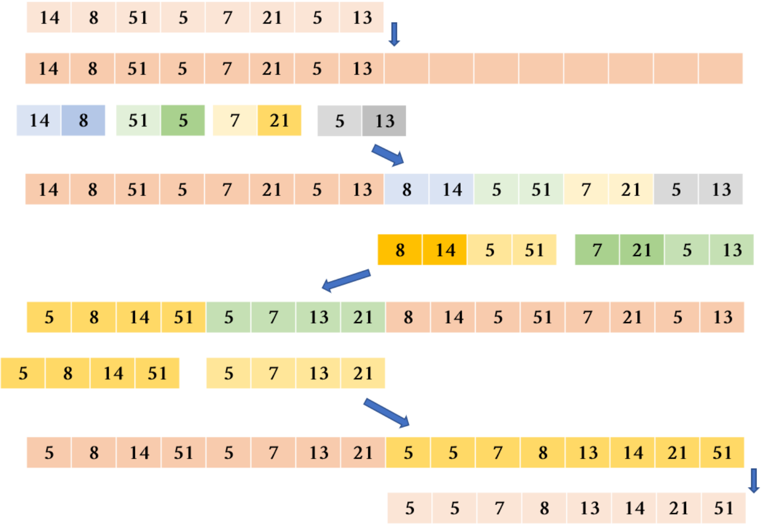
- For simplicity, let us consider the case of one array (N=1) with size 2^n.

- We use the same principle as in the classic **merge sort** algorithm, replacing the merge step with the merge path algorithm.

Input : An array M of size 2^n (in CPU).
Goal : Sort it.

1. Allocate in GPU an array $[M_0, M_1]$ of size $2 \cdot 2^n$ and copy $M_1 \leftarrow M$.
2. For each $i \in 1, 2, ..., n-1$:
   - Let $k_i = (i \mod 2)$. Split M_{k_i} into 2^{n-i} arrays $[A_0, A_1, ..., A_{2^{n-i}-1}]$ of size 2^i.
   - $\forall s \in \{1, ..., 2^{n-1-i}\}$: Apply the merge path algorithm on $[A_{2s-2}, A_{2s-1}]$ by putting the sorted result on $M_{k_{i+1}}$.
3. **Return** $M_{k_{n-1}}$

Sorting algorithm scheme



Implementation

- For each i on step 2 :
  - We should apply the parallel merge path algorithm 2^{n-i} times.
  - Each merge path needs $2^{i-1} + 2^{i-1} = 2^i$ threads.

So, we should call a kernel function that uses $2^{n-i} \cdot 2^i = 2^n$ threads.

```
kernel_batch_sort<<< 1, d >>> (mGPU, i, d);
```

- The thread $k = 2^{i+1} * q + r$ will be responsible for the r^{th} diagonal of the q^{th} path merge algorithm.

```
__global__ void kernel_batch_sort(int * M, int i, int d){
    // which sort array?

    int size = ((int) pow(2,i));

    // which merge? find offset of M corresponding to A and B
    int offset =   (threadIdx.x /(2*size) * 2*size;
    int idx_start_a = offset + (i%2)*d;
    int idx_start_b = idx_start_a + size;
    int m =  intermediate_threadIdx % (2*size);

    // device function
    trifusion(M+ idx_start_a, M+idx_start_b, M+offset + (!(i%2))*d, size, size, m);
}
```

Using shared memory

- The merge path algorithms need to go the GPU several times. We can use the shared memory instead.
- For each i, we should copy the arrays to be sorted into the shared memory before call the (merge path algorithm).

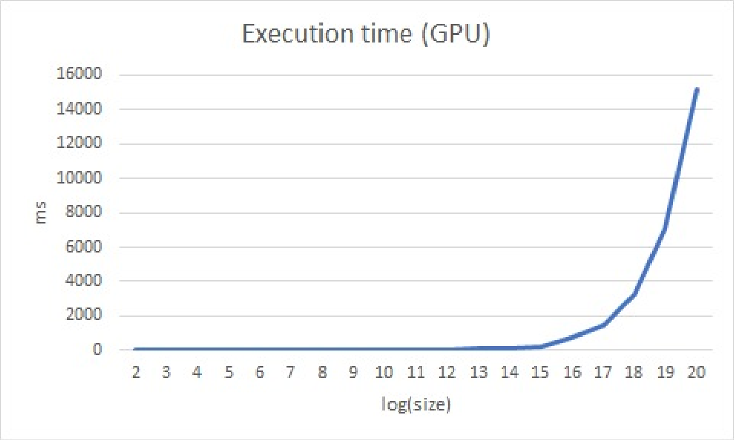
```
__global__ void kernel_batch_sort_shared(int *M, int i, int d){

    int size = ((int) pow(2,i));

    extern __shared__ int A[];

    int offset =  (threadIdx.x /(2*size)) * 2*size;


    // device function

    A[threadIdx.x+(i%2)*d]= M[threadIdx.x+(i%2)*d];

    __syncthreads();
    trifusion(A+offset+(i%2)*d,A+offset+(i%2)*d+size, A+offset+(!(i%2))*d, size, size, threadIdx.x%(2*size));
    M[(!(i%2))*d + threadIdx.x]=A[threadIdx.x+(!(i%2))*d];

}
```

Generalization

- If $d \leq 1024$ we can easily generalize for $N > 1$:
  - We can pass to the kernel a concatenated array $M = [M_1, ..., M_N]$
  - For a given i, the thread $k_{i,j} = 2^{i+1} * q_j + r_j$ from the j^{th} block will be responsible for the r_j^{th} diagonal of the q_j^{th} path merge algorithm on M_j.
- If $d > 1024$: We can consider a "super-block" composed by several blocks. If $d = 2048$ for instance, the 2 first blocks handles the array M_1, the next 2 handles M_2 and so on..

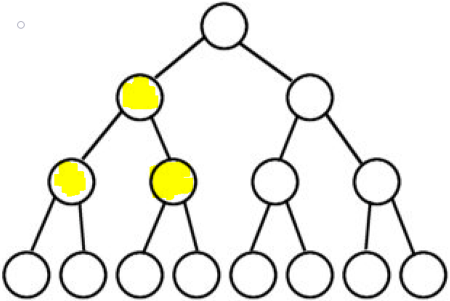
```
__global__ void kernel_batch_sort(int * M, int i, int mul, int d){
    // which sort array?
    int k = (int) blockIdx.x/mul;

    // which sizes of A e B ?
    int size = ((int) pow(2,i));

    // thread 2 from second block must represents thread 1025 of a virtual "superblock", where superblock is mul blocks together)
    int intermediate_threadIdx =  (blockIdx.x % mul) * blockDim.x + threadIdx.x ;

    // which merge? find offset of M corresponding to A and B
    int offset =   k*2*d +  (intermediate_threadIdx /(2*size) * 2*size;
    int idx_start_a = offset + (i%2)*d;
    int idx_start_b = idx_start_a + size;
    int m =  intermediate_threadIdx % (2*size);

    // device function
    trifusion(M+ idx_start_a, M+idx_start_b, M+offset + (!(i%2))*d, size, size, m);
}
```

- We consider one array of size $d = 2^i$, and we vary i and we compute the time execution cost.
- We do the same for a standard sort algorithm in CPU.



	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
Exec. Time (CPU)	0.006	0.011	0.021	0.039	0.083	0.163	0.334	0.687	1.439	3.148	6.612
Exec. Time (GPU)	2.37568	3.43376	5.0039	9.16035	17.1259	27.59	43.5856	71.1766	128.165	215.423	708.18

- For the considered cases, the sort algorithm using the parallel merge path showed a reasonable asymptotic behaviour. However, the execution cost was not better than for a standard sort algorithm in CPU.
- Even the using of shared memory, in this case, doesn't improve the performance of the algorithm.
- The dependency on subproblems :



Batch merge path sort

2 avril 2021