GPU Project : Batch merge and merge path sort

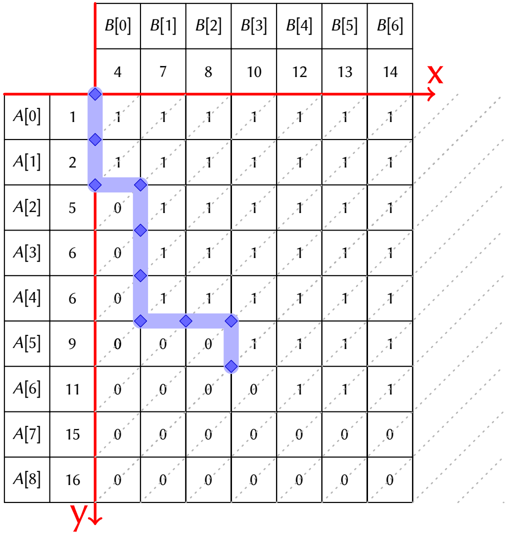
Arthur **Zucker** , Clément **Apavou**

14 décembre 2020

Table of contents

Introduction

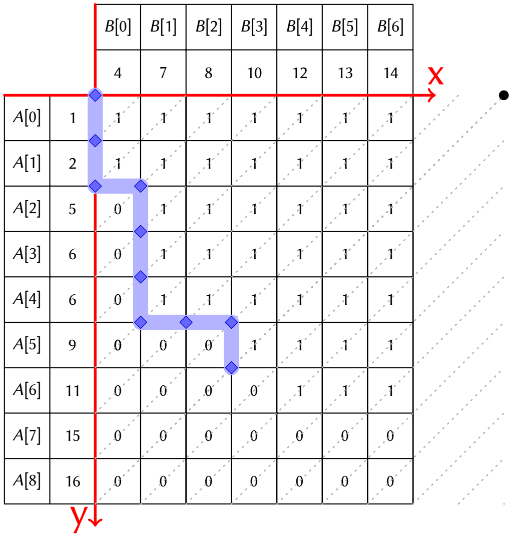
Path merge : How it works



		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

1 Compute P and K :

Path merge : How it works



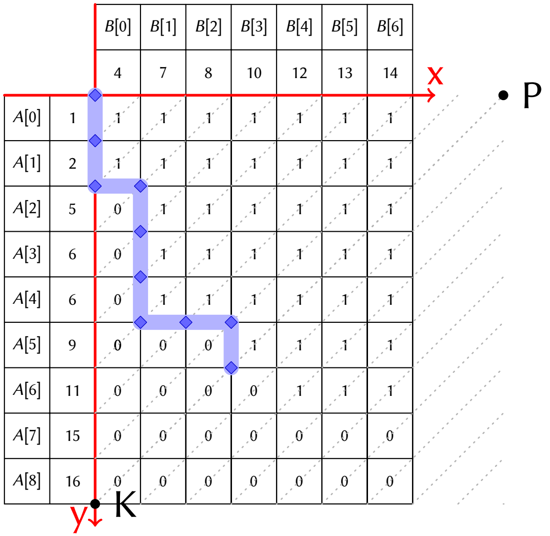
		B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]
		4	7	8	10	12	13	14
A[0]	1	1	1	1	1	1	1	1
A[1]	2	1	1	1	1	1	1	1
A[2]	5	0	1	1	1	1	1	1
A[3]	6	0	1	1	1	1	1	1
A[4]	6	0	1	1	1	1	1	1
A[5]	9	0	0	0	1	1	1	1
A[6]	11	0	0	0	0	1	1	1
A[7]	15	0	0	0	0	0	0	0
A[8]	16	0	0	0	0	0	0	0

- P

1 Compute P and K :

$\longrightarrow$ P=(9,0)

Path merge : How it works



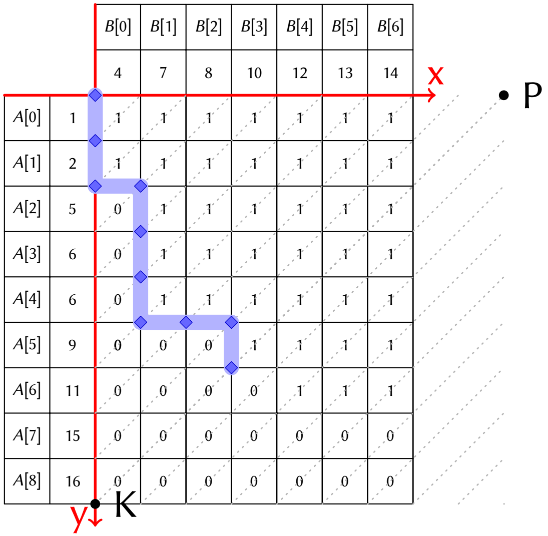
		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

1 Compute P and K :

$\longrightarrow$ P=(9,0)

$\longrightarrow$ K=(0,9)

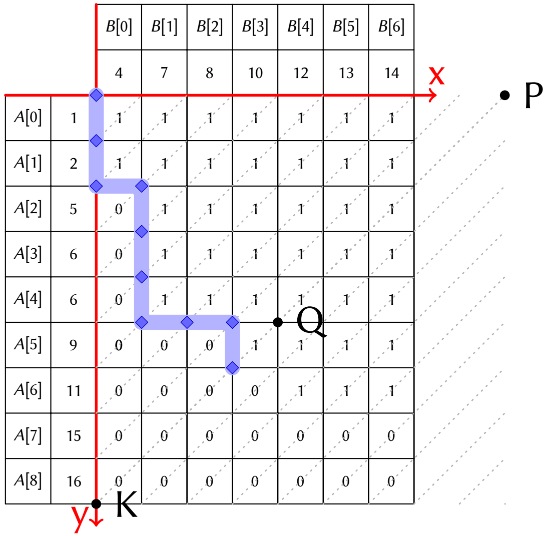
Path merge : How it works



		B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]
		4	7	8	10	12	13	14
A[0]	1	1	1	1	1	1	1	1
A[1]	2	1	1	1	1	1	1	1
A[2]	5	0	1	1	1	1	1	1
A[3]	6	0	1	1	1	1	1	1
A[4]	6	0	1	1	1	1	1	1
A[5]	9	0	0	0	1	1	1	1
A[6]	11	0	0	0	0	1	1	1
A[7]	15	0	0	0	0	0	0	0
A[8]	16	0	0	0	0	0	0	0

1. Compute P and K :
   $\longrightarrow$ P=(9,0)
   $\longrightarrow$ K=(0,9)
2. offset = 3

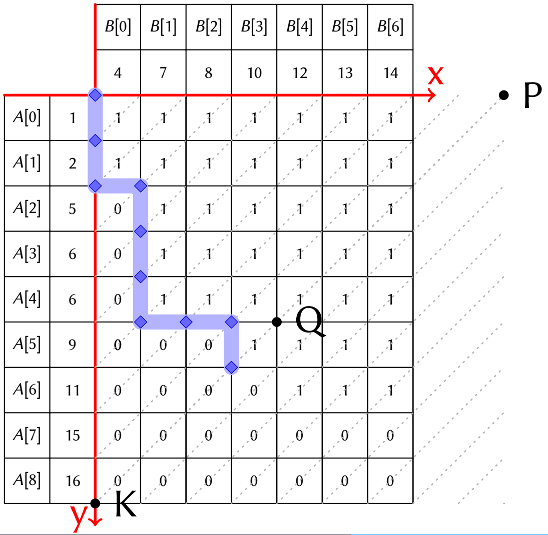
Path merge : How it works



		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

1. Compute P and K :
   $\longrightarrow$ P=(9,0)
   $\longrightarrow$ K=(0,9)
2. offset = 3
3. Q = (5,4)

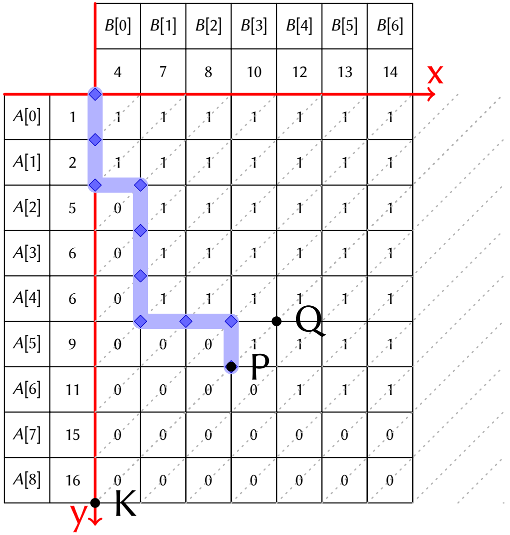
Path merge : How it works



		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

1. Compute P and K :
   $\longrightarrow$ P=(9,0)
   $\longrightarrow$ K=(0,9)
2. offset = 3
3. Q = (5,4)
4. P = $(Q_x\text{-}1, Q_y+1)$=(4,5)

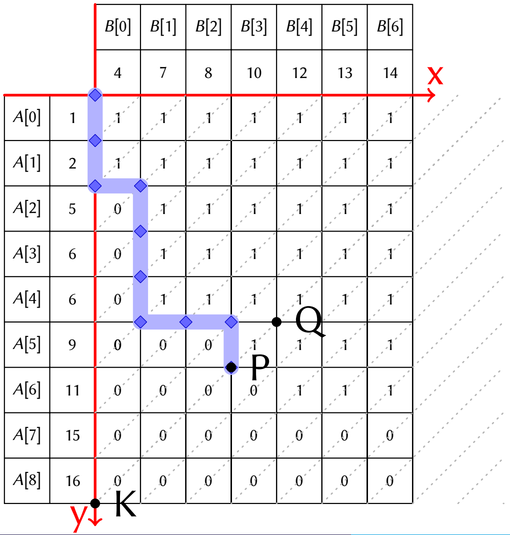
Path merge : how it works



		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

1. Compute P and K :
   $\longrightarrow$ P=(9,0)
   $\longrightarrow$ K=(0,9)
2. offset = 3
3. Q = (5,4)
4. P = $(Q_x\text{-}1, Q_y+1)$=(4,5)

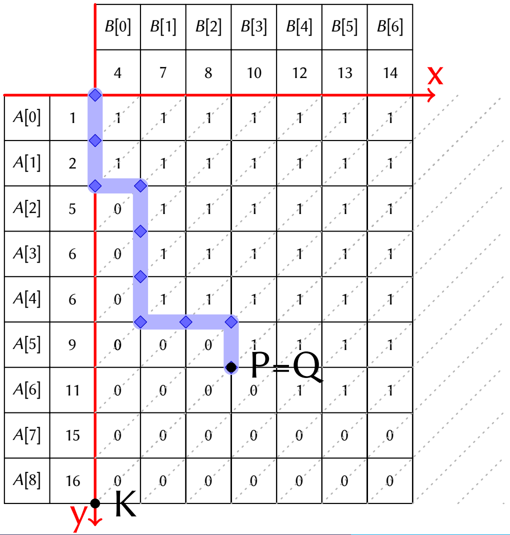
Path merge : how it works



		B[0]	*B*[1]	*B*[2]	*B*[3]	*B*[4]	*B*[5]	*B*[6]
		4	7	8	10	12	13	14
A[0]	1	1	1	1	1	1	1	1
A[1]	2	1	1	1	1	1	1	1
A[2]	5	0	1	1	1	1	1	1
A[3]	6	0	1	1	1	1	1	1
A[4]	6	0	1	1	1	1	1	1
A[5]	9	0	0	0	1	1	1	1
A[6]	11	0	0	0	0	1	1	1
A[7]	15	0	0	0	0	0	0	0
A[8]	16	0	0	0	0	0	0	0

1. Compute P and K :
   $\longrightarrow$ P=(9,0)
   $\longrightarrow$ K=(0,9)
2. offset = 3
3. Q = (5,4)
4. P = $(Q_x\text{-}1, Q_y+1)$=(4,5)
5. Q = P =(4,5)

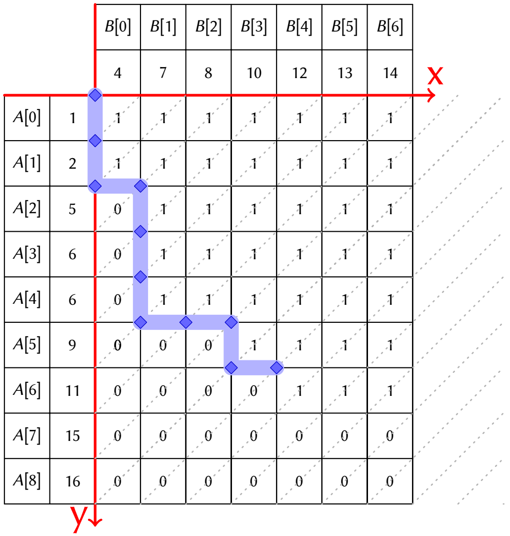
Path merged : thread 9



		B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]
		4	7	8	10	12	13	14
A[0]	1	1	1	1	1	1	1	1
A[1]	2	1	1	1	1	1	1	1
A[2]	5	0	1	1	1	1	1	1
A[3]	6	0	1	1	1	1	1	1
A[4]	6	0	1	1	1	1	1	1
A[5]	9	0	0	0	1	1	1	1
A[6]	11	0	0	0	0	1	1	1
A[7]	15	0	0	0	0	0	0	0
A[8]	16	0	0	0	0	0	0	0

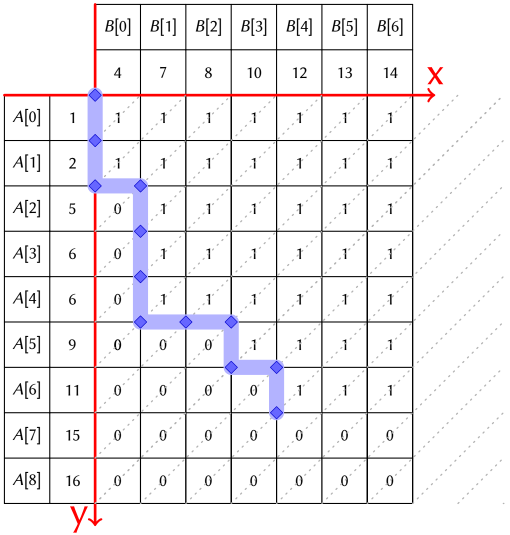
1. Compute P and K :
   $\longrightarrow$ P=(9,0)
   $\longrightarrow$ K=(0,9)
2. offset = 3
3. Q = (5,4)
4. P = $(Q_x\text{-}1, Q_y+1)$=(4,5)
5. Q = P =(4,5)

Path merged : thread 9, path



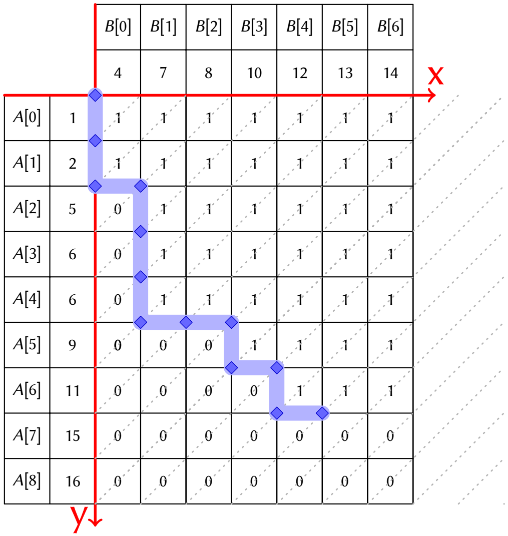
		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

Path merged : thread 9, path



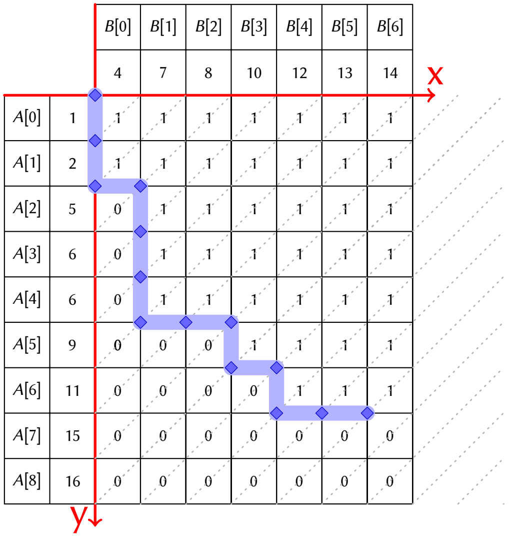
		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

Path merged : thread 9, path



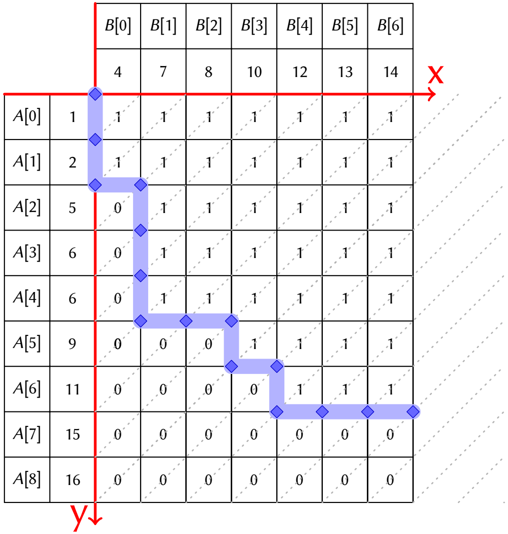
		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

Path merged : thread 9, path



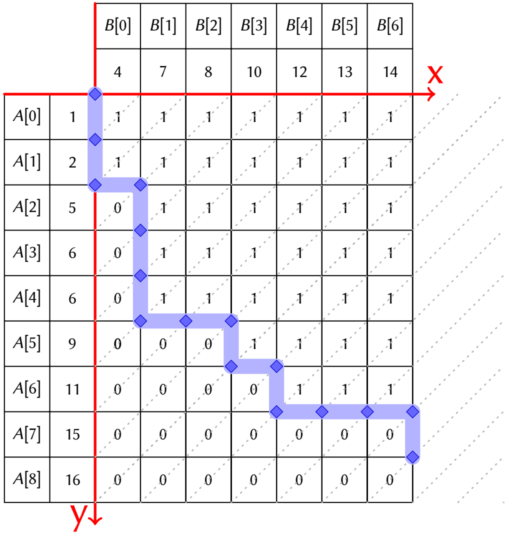
		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

Path merged : thread 9, path



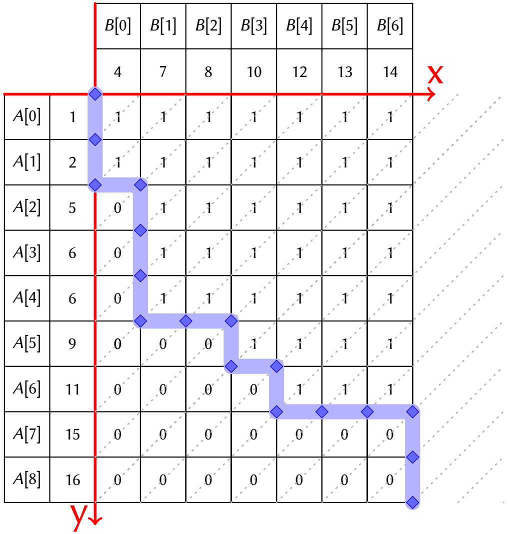
		B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]
		4	7	8	10	12	13	14
A[0]	1	1	1	1	1	1	1	1
A[1]	2	1	1	1	1	1	1	1
A[2]	5	0	1	1	1	1	1	1
A[3]	6	0	1	1	1	1	1	1
A[4]	6	0	1	1	1	1	1	1
A[5]	9	0	0	0	1	1	1	1
A[6]	11	0	0	0	0	1	1	1
A[7]	15	0	0	0	0	0	0	0
A[8]	16	0	0	0	0	0	0	0

Path merged : thread 9, path



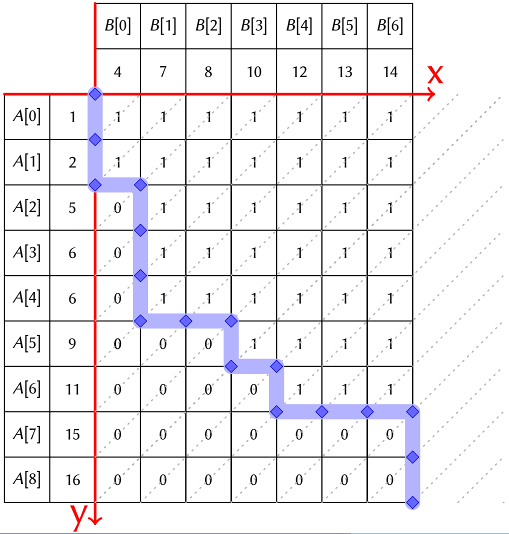
		B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]
		4	7	8	10	12	13	14
A[0]	1	1	1	1	1	1	1	1
A[1]	2	1	1	1	1	1	1	1
A[2]	5	0	1	1	1	1	1	1
A[3]	6	0	1	1	1	1	1	1
A[4]	6	0	1	1	1	1	1	1
A[5]	9	0	0	0	1	1	1	1
A[6]	11	0	0	0	0	1	1	1
A[7]	15	0	0	0	0	0	0	0
A[8]	16	0	0	0	0	0	0	0

Path merged : thread 9, path



		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

Path merged : thread 9, path



		$B[0]$	$B[1]$	$B[2]$	$B[3]$	$B[4]$	$B[5]$	$B[6]$
		4	7	8	10	12	13	14
$A[0]$	1	1	1	1	1	1	1	1
$A[1]$	2	1	1	1	1	1	1	1
$A[2]$	5	0	1	1	1	1	1	1
$A[3]$	6	0	1	1	1	1	1	1
$A[4]$	6	0	1	1	1	1	1	1
$A[5]$	9	0	0	0	1	1	1	1
$A[6]$	11	0	0	0	0	1	1	1
$A[7]$	15	0	0	0	0	0	0	0
$A[8]$	16	0	0	0	0	0	0	0

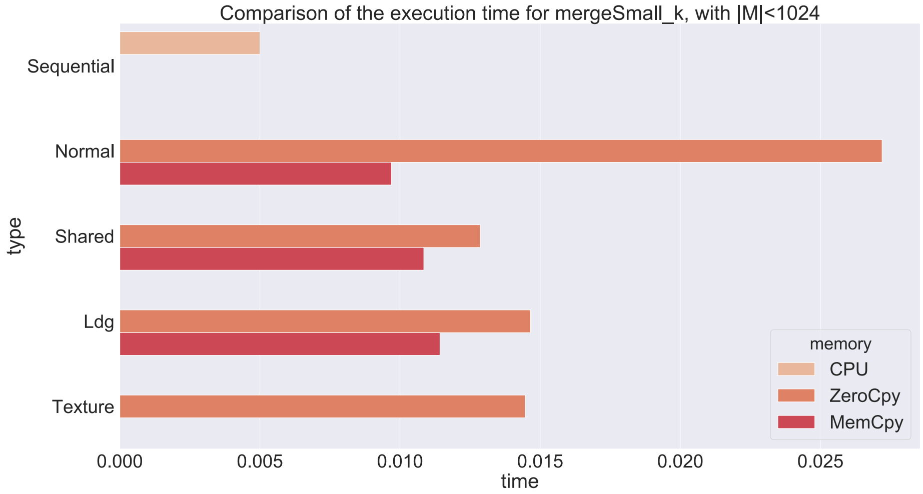
Merge path and sort

Question 1 : `mergeSmall_k`

Concept

- Two sorted arrays A and B such as : $|A| + |B| \leq 1024$
- `mergeSmall_k` merges A and B
- Using one block of threads

Question 1 : Different type of memory

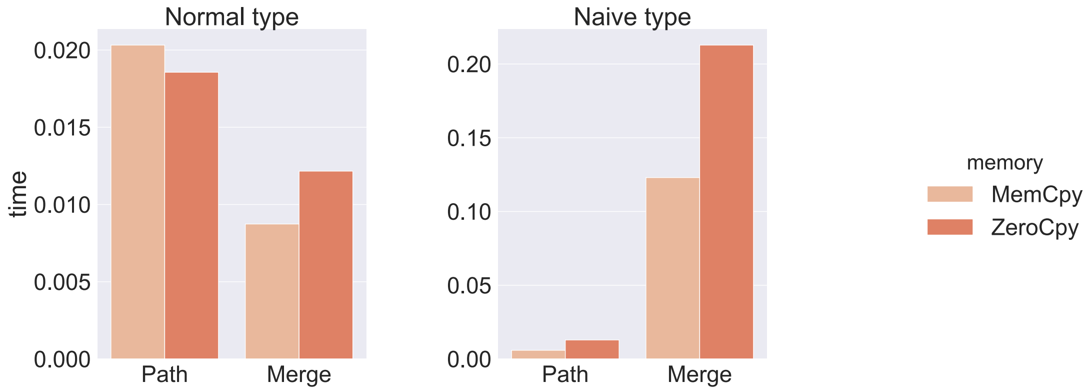


Question 2 : `pathBig_k` and `mergeBig_k`

Concept

- Two sorted arrays A and B such as : $|A| + |B| = d$
- `pathBig_k` finds the merge path
- `mergeBig_k` merges A and B

Question 2 : Different type of memory and methods

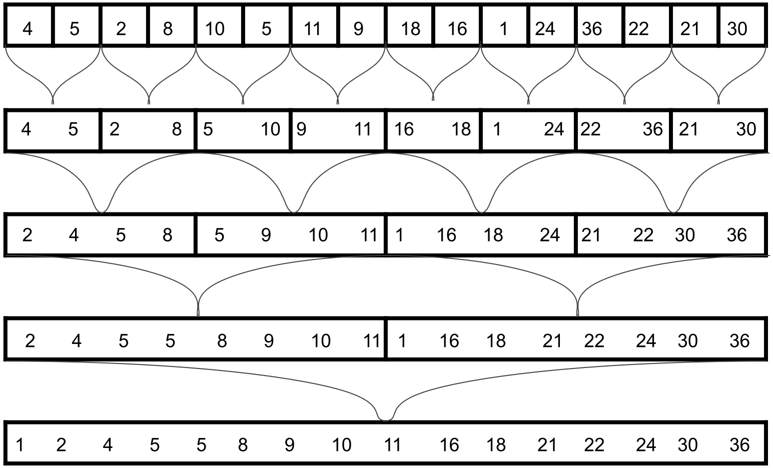


Question 3 : Sort array

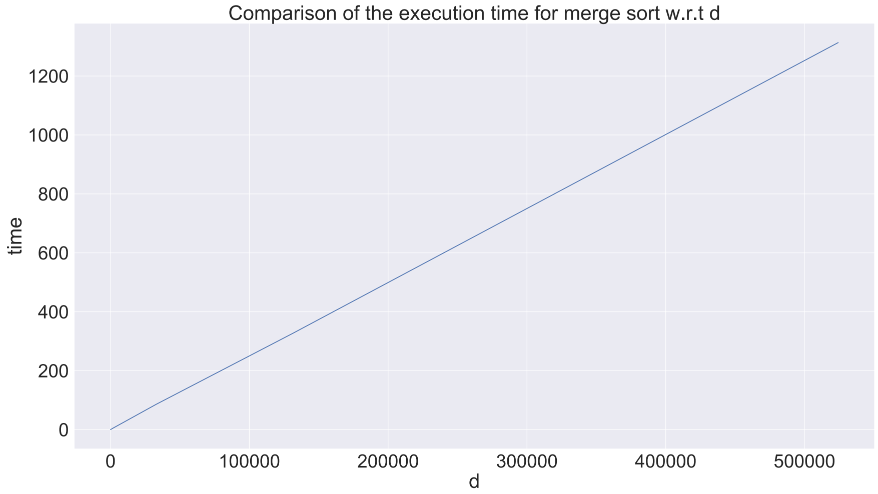
Concept

- Sort any array M of size d
- Using the two previous questions

Question 3 : Method



Question 3 : The execution time for different sizes of M



Batch merge

Batch merge : Concept

Concept

- A large number N of arrays A_i and B_i with $|A_i| + |B_i| = d \leq 1024$ for each $i = 1, ..., N$.
- Merges two by two, each i, A_i and B_i.

Batch merge : structure

Structure used

- One big array M of size $N \ \ d$
- $M = [A_1|B_1|...|A_i|B_i|...|A_N|B_N]$
- $|A_i| + |B_i| = d \leq 1024$ for each $1 \leq i \leq N$
- $A_i = a_{i1}, a_{i2}, ..., a_{il}$ and $B_i = b_{i1}, b_{i2}, ..., b_{ik}$ with $l + k = d$
- Store sizes of A_i and B_i

Batch merge : index

```
int tidx = threadIdx.x%d;
int Qt = (threadIdx.x-tidx)/d;
int gbx = Qt + blockIdx.x*(blockDim.x/d);
```

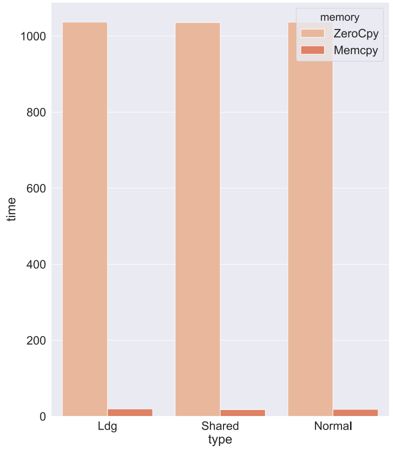
Index

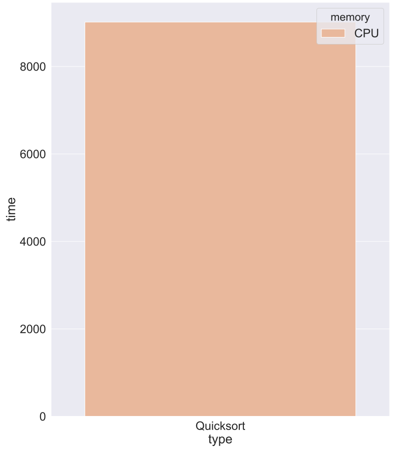
- tidx : which diagonal does the thread takes care of.
- gbx : index of the value of the size of the sub-array in the global array of sizes. Allows us to retrieve the addresses of the sub-array A and B in M.

With gbx, each block takes care of a sub-array.
With tidx, each thread takes care of each diagonal.

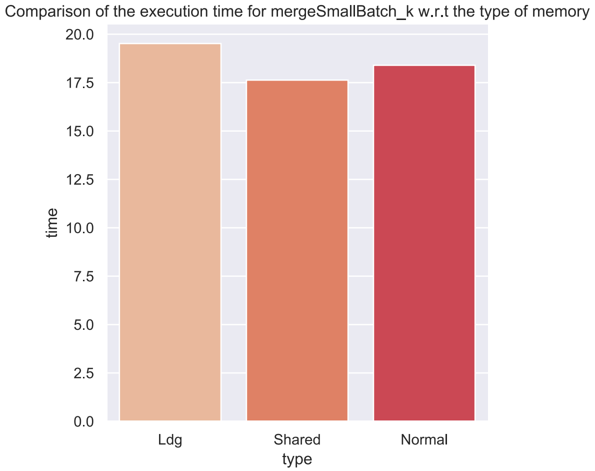
Batch merge : Different type of memory $d = 500$ and $N = 1000000$

Comparison of the execution time for mergeSmallBatch_k w.r.t the type of memory

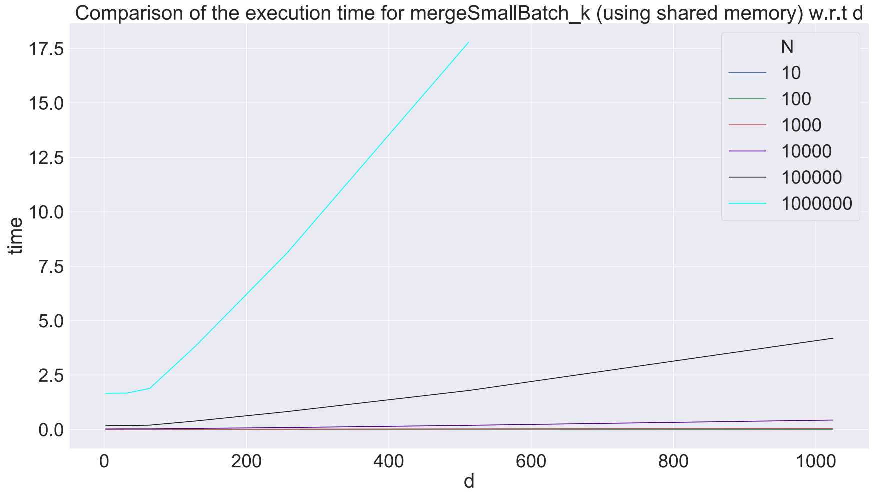




Batch merge : Different type of memory $d = 500$ and $N = 1000000$



Batch merge : The execution time for different d and N



Merge sort applications

A first Idea

Improving the parallel merge sort

Based on both questions, we came up with a new version of a fast merge sort.

It is based on the observation that in question 3, we have a nested loop which is very slow. Moreover, not every thread is used.

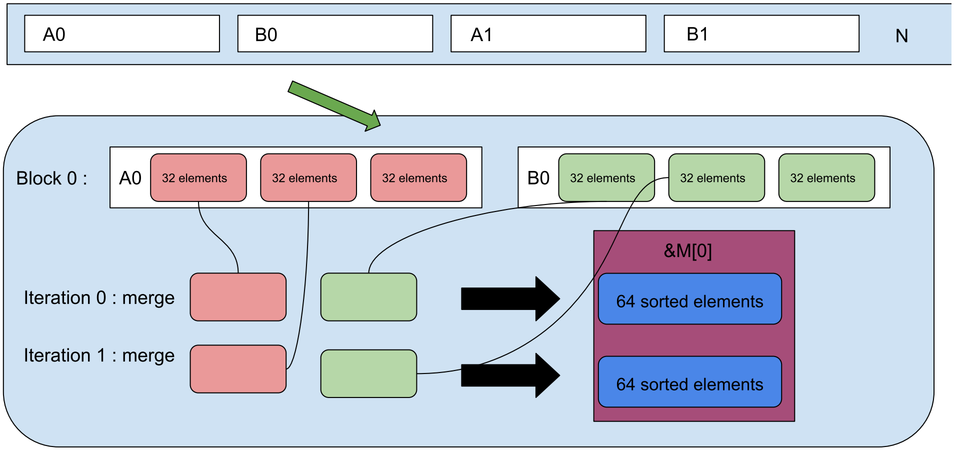
1. When $d \leq 1024$, `mergeSmallBatch` can be used in a single loop to get an array of sorted arrays of size 1024. Thus, the occupancy of the GPU is optimal since it is well schedule.
2. Then, a modified version of pathBig_k and mergeSmall_k will try to use the same strategy.

A first Idea

The strategy

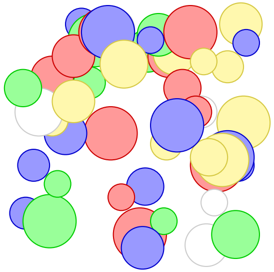
1. Use block of size 32 or 64
2. Each block will merge start with a window of size `BlockDim`
3. `BlockDim` elements of A and B will be loaded sequentially in the shared memory
4. The path is computed and the merge is also done
5. Then, the action is repeated for every sub-arrays of size `BlockDim` in each arrays of size > 1024.

Representation



Studying species of bacteria

Question 5 can be used to sort energy values of different "cells" that can be studied in a modelling system.



We can for example consider simple cells, interacting with each other. Each cell will have an **energy** counter, which will vary depending on the collisions and movements of the cell We could want to know the cumulative distribution of the energy with regard to the cell kind.

Studying species of bacteria

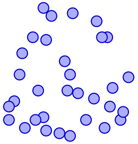
We thus need to compute the table containing the energy of each cell. We suppose that each cell's energy is automatically modified in the table. Then, we would end up with something close to :

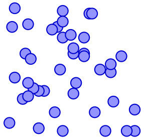
Species/energy	0-50	50-100	> 100
A	10	5	0
B	60	1	1
⋮	⋮	⋮	⋮
C	30	30	30

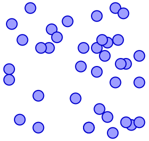
Questions such as "Which species survived the best" or "What makes a species strive ?" can arise.

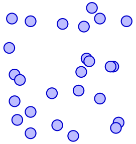
Launching different simulations in parallel

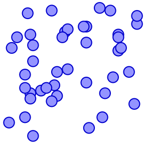
We could also want to study the same dynamics but on different populations set with different types of parameters. Again, if we study groups of size ≤ 1024, we can compute every histogram at a fast pace.

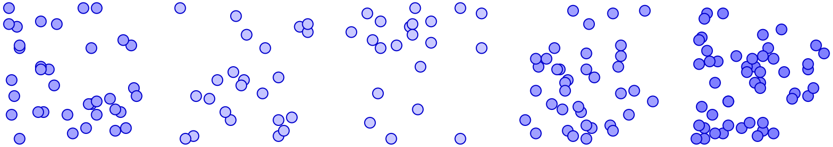

(a) first


(b) second


...


...


...



Launching different simulations in parallel

`mergeSmallBatch` will be used on the following arary

Speed/energy	0-50	50-100	> 100
4	10	5	0
6	60	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
10	30	30	30

We would loop over calls of `mergeSmallBatch`, changing *d* in order to merge and sort using the strategy from question 3. This is very efficient : at every step of the loop, each thread works on the same amount of values

Thanks for your attention

If you have any questions feel free to ask !