GPU for Deep Learning

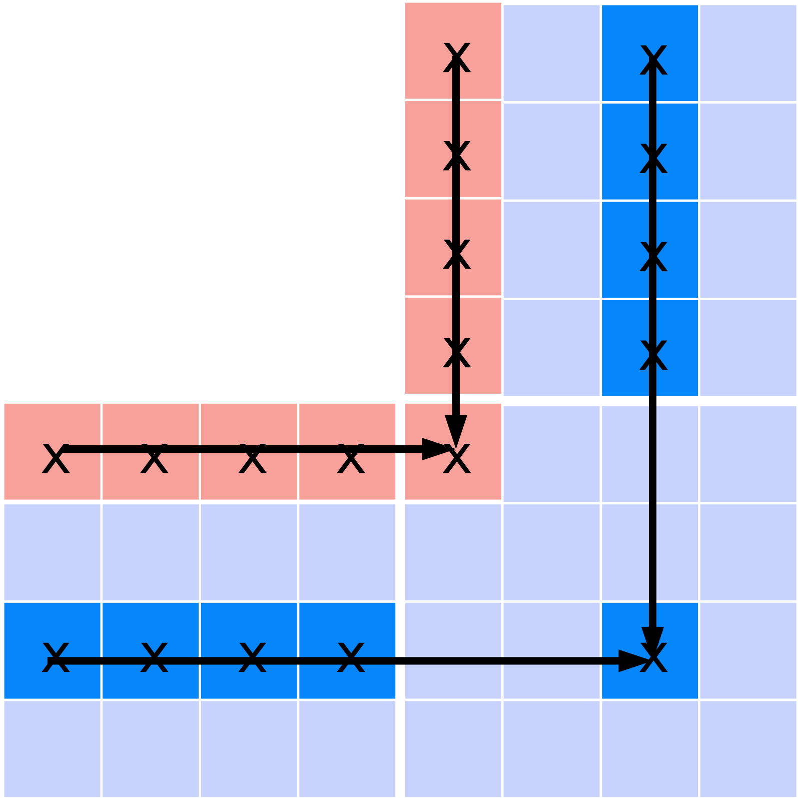
Optimized Matrix Multiplication

Elisabeth Brunet

C = AB

- **Given two matrices A and B,**

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$



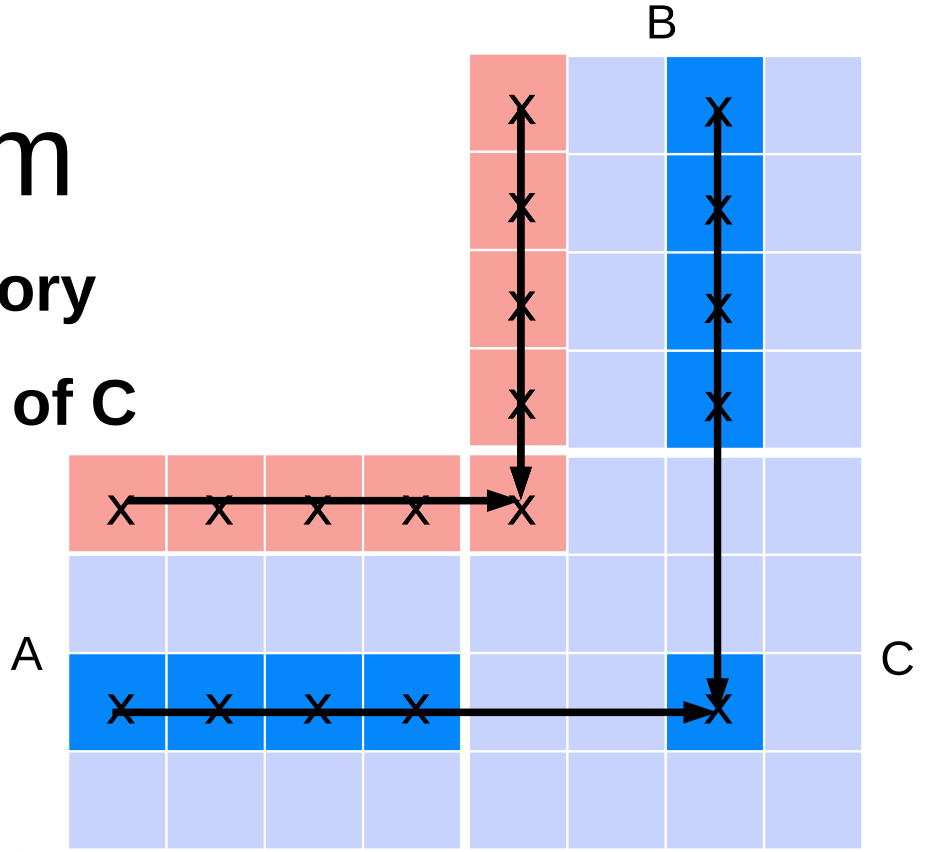
$$\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

- Such that $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}.$

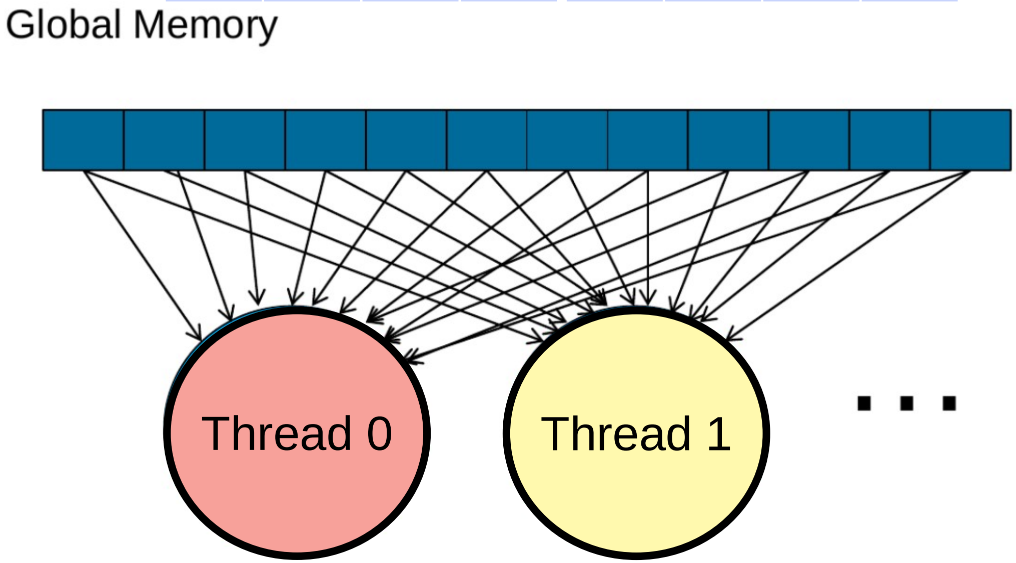
1)Basic algorithm

- **Matrices A, B and C in global memory**
- **Each thread calculates an element of C**
- **Each thread accesses**
  - to a whole line of A
  - and a whole column of B



- **Data access**

  non-aligned and scattered
  - Coalescing problem
- **Repeated data access**

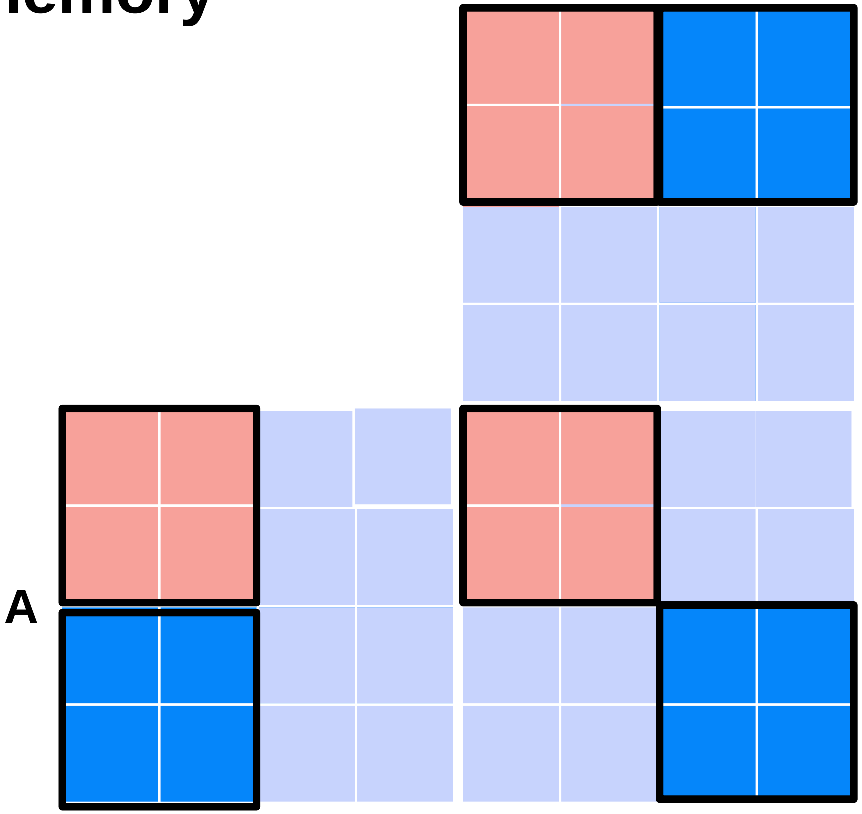


2) Tiled algorithm

- **Iterative algorithm**

  on sub-matrixes multiplication

  treated by a block fiting in shared memory

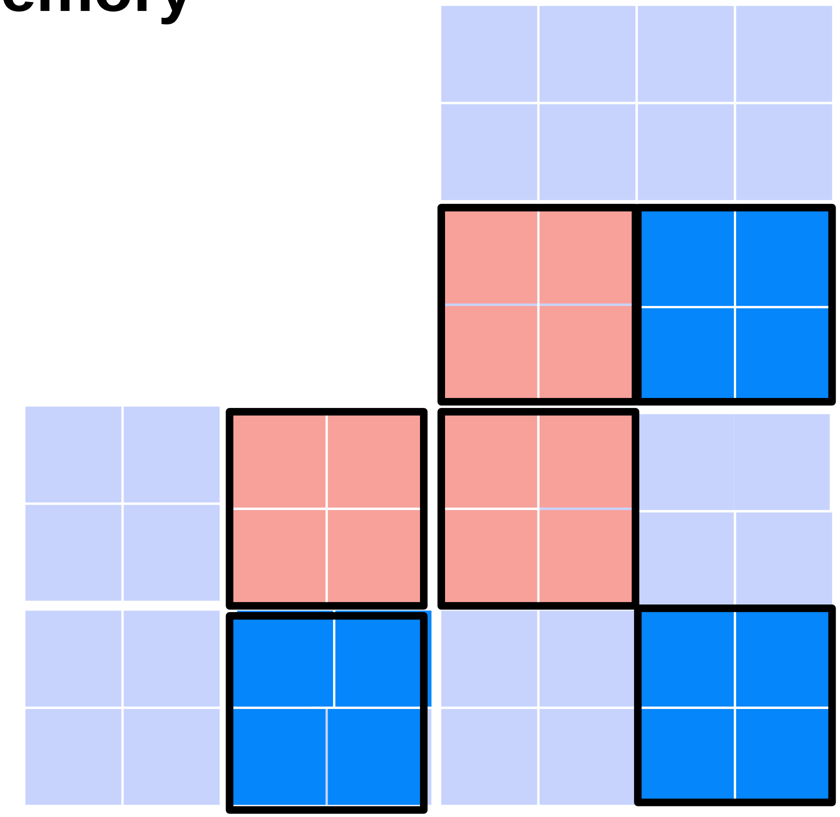


2) Tiled algorithm

- **Iterative algorithm**

  on sub-matrixes multiplication

  treated by a block fiting in shared memory



2) Tiled algorithm

- **Iterative algorithm**

  on sub-matrixes multiplication

  treated by a block fiting in shared memory

- **Iteration in 4 steps :**

  (a) Cooperative loading of a data in tiles in shared memory

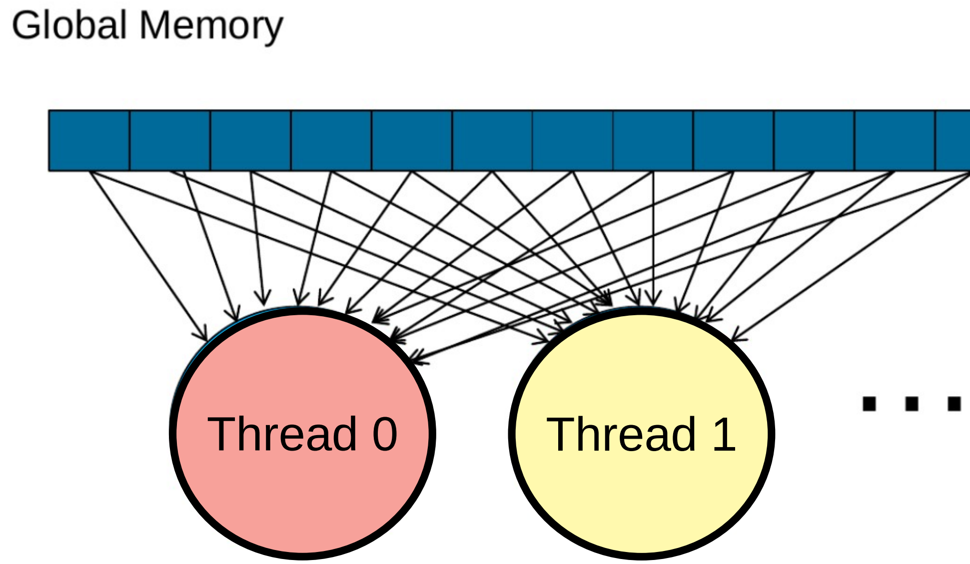
  (b) Synchronization to ensure that data are loaded

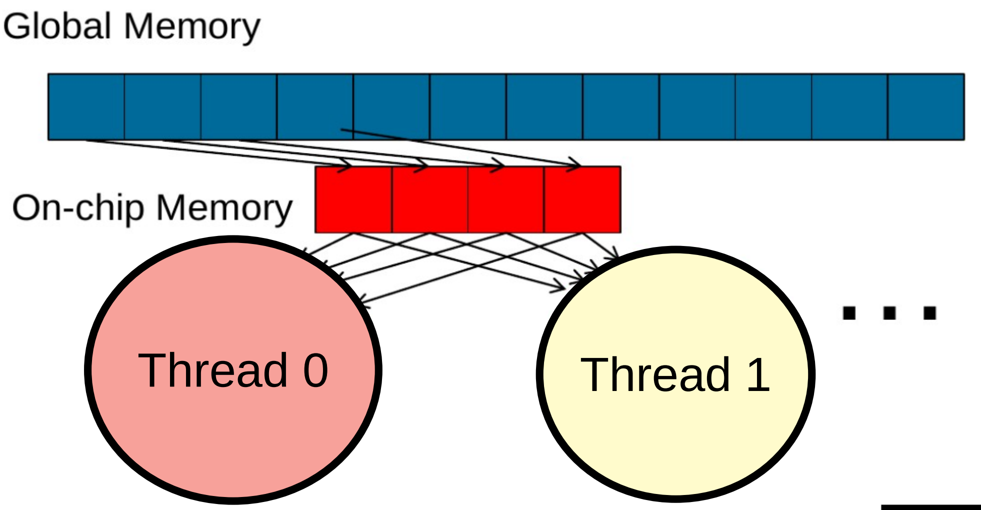
  (c) Calculation of partial results by threads on loaded data

  (d) Synchronization before changing the data of the tiles for the next iteration

(a) Cooperative data loading $_B$

- **Objective : change the access pattern**





→ Regular access to global memory
+ Quick access once in shared memory !!!

(a) Cooperative data loading

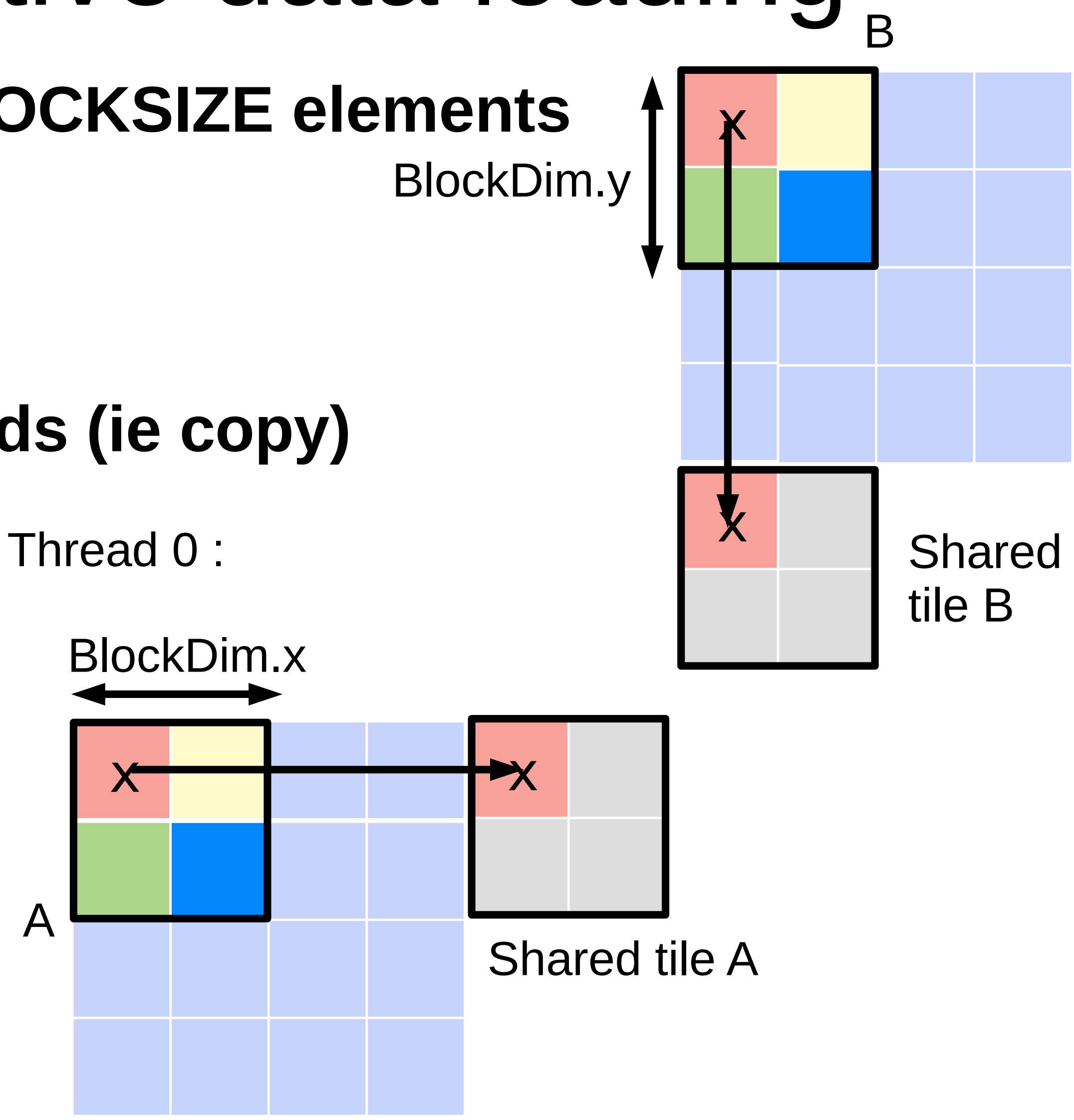
- **Sub-tiles of BLOCKSIZE*BLOCKSIZE elements**
  - BLOCKSIZE = sqrt(1024) = 32

- **Each thread of the block loads (ie copy)**

  an element from A

  and an element from B

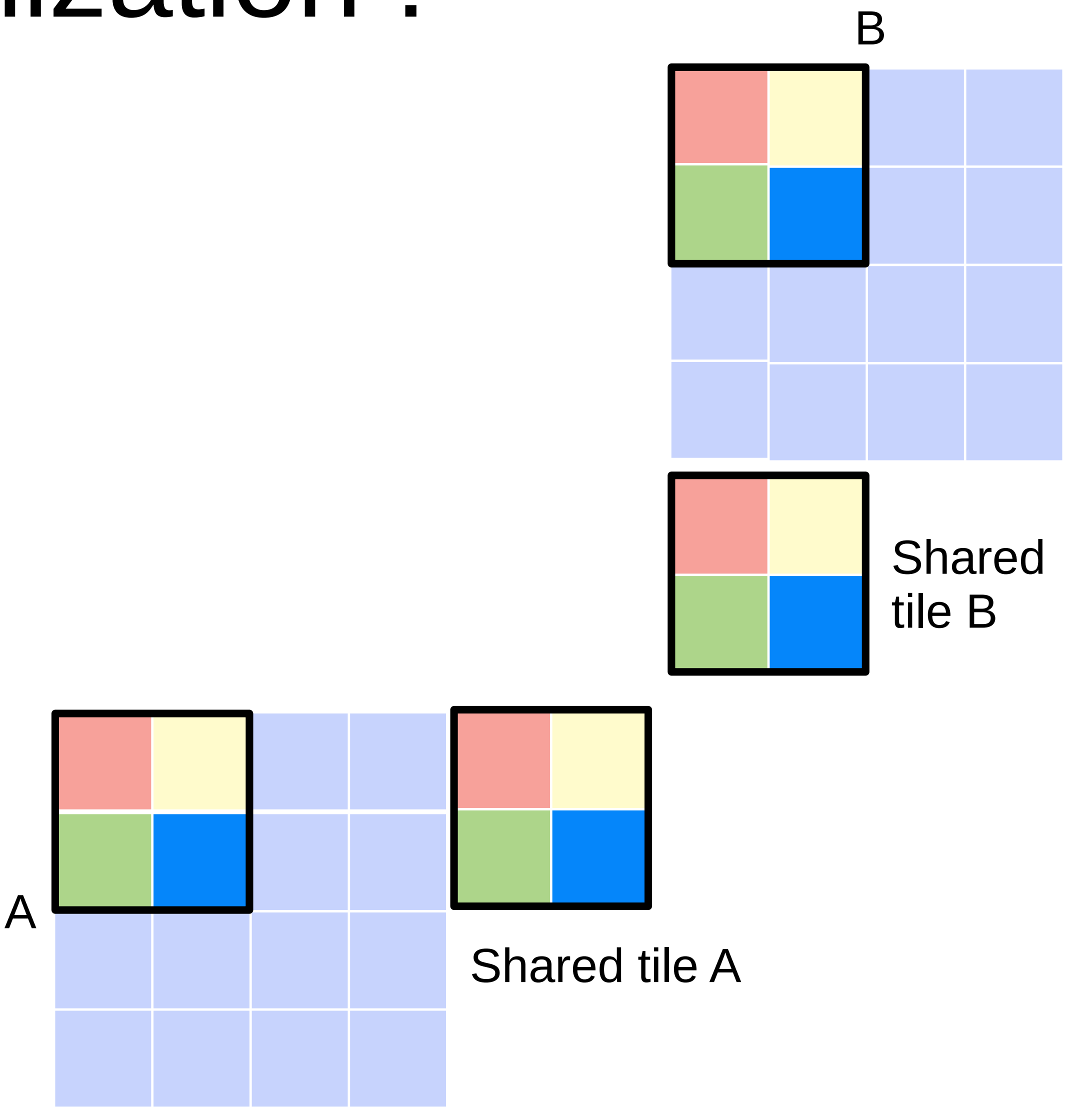
  into the shared tiles



(b) Synchronization !

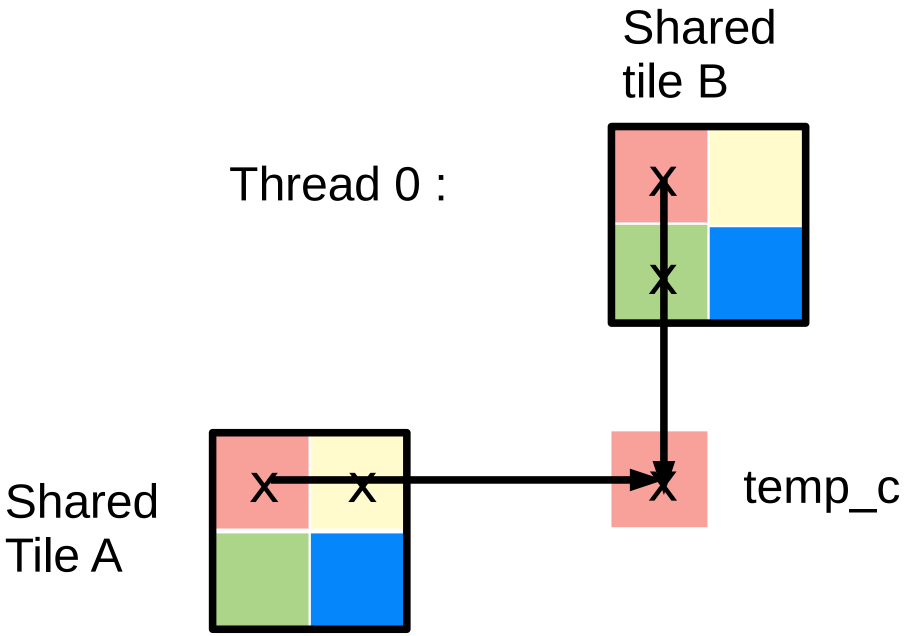
- Once the barrier passed,

  the two sub-tiles are complete



(c) Partial result computation

- **Multiplication of matrices on current tiles**
- **Accumulation in a scalar variable stored in a register**

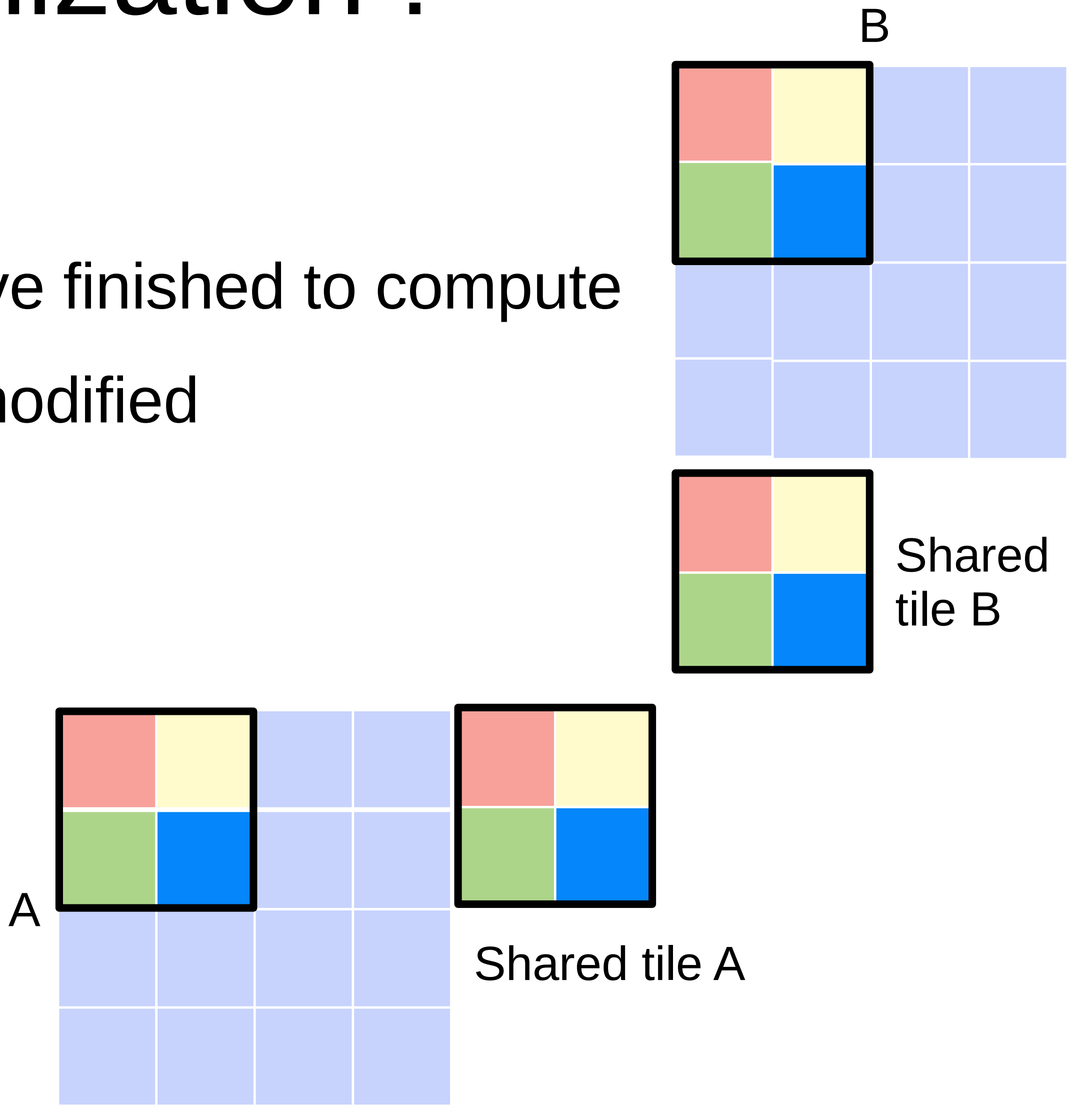


(d) Synchronization !

- Once the barrier passed,

  all the threads of the block have finished to compute

  and the two sub-tiles can be modified



Switch to the next tile

(a) Cooperative data loading

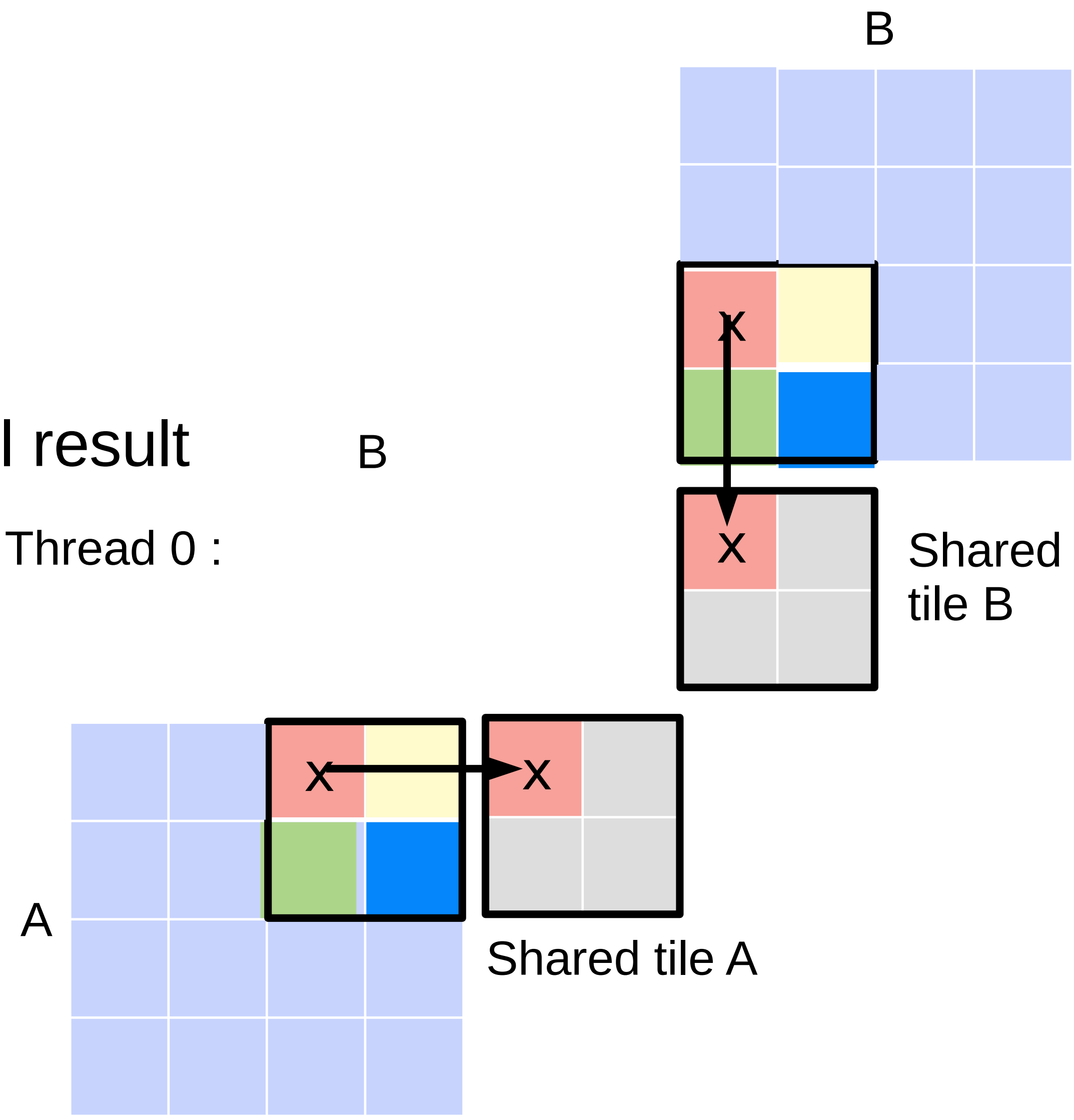
in the **SAME** shared tile

(b) Synchronization

(c) Accumulation of the partial result

in local variable temp_c

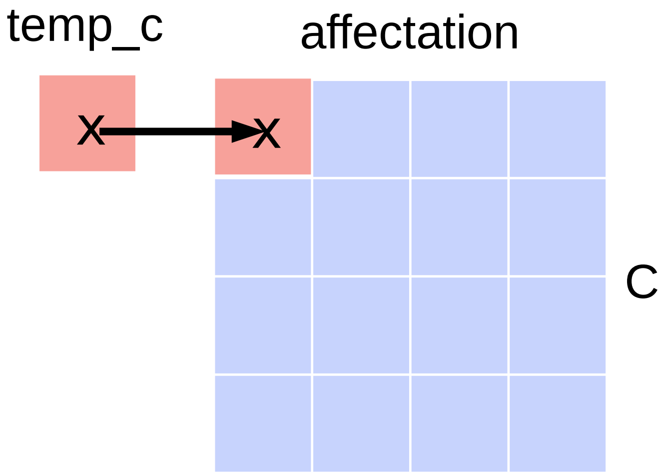
(d) Synchronization



Final result

- **Assignment of the result in the C matrix remaining in global memory**

Thread 0 :



temp_c

affectation

x

x

C

To go further,

- **Each partial result could be computed by another block**
  - Additional dimension on the grid to identify the tile to be treated
  - Reduction of the partial results :
    - Atomic operation for accumulation: atomicAdd
    - Reduction on the GPU thanks to a kernel
    - Reduction on the CPU

To go further,

- **Each partial result could be computed by another block**
  - Additional dimension on the grid to identify the tile to be treated
  - Reduction of the partial results :
    - Atomic operation for accumulation: atomicAdd
    - Reduction on the GPU thanks to a kernel
    - Reduction on the CPU
- A lot of different optimizations and strategies

→ Field of research in itself

To go further,

- **Each partial result could be computed by another block**
  - Additional dimension on the grid to identify the tile to be treated
  - Reduction of the partial results :
    - Atomic operation for accumulation: atomicAdd
    - Reduction on the GPU thanks to a kernel
    - Reduction on the CPU
- A lot of different optimizations and strategies

  → Field of research in itself
- **Best algorithms in the cuBLAS library**

Last detail :
access to a matrix element

- **Matrix organization**

  = one-dimensional vector column major

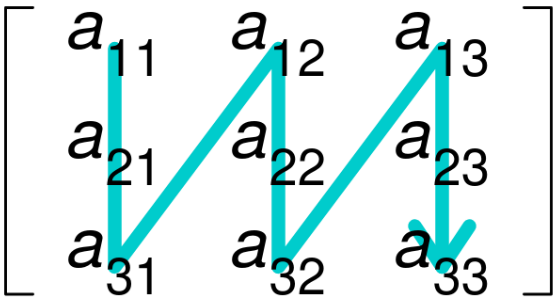
  - Colum major required by cuBLAS

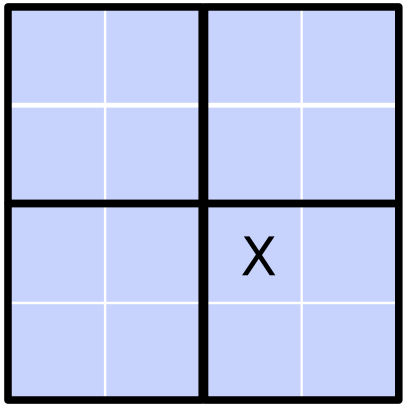
- **Access to an element in 2 steps**

  1) Projection of the grid of threads on the matrix
     - Line = blockIdx.y * blockDim.y + threadIdx.y
     - Column = blockIdx.x * blockDim.x + threadIdx.x
  2) Linearization in the data structure

     in one column-major dimension
     - A[column * nbLineOfA+ line]
     - Given macro IDX2C : #define IDX2C(i,j,nb_rows) (((j)*(nb_rows))+(i))

Column-major order





Let's go to practise now !