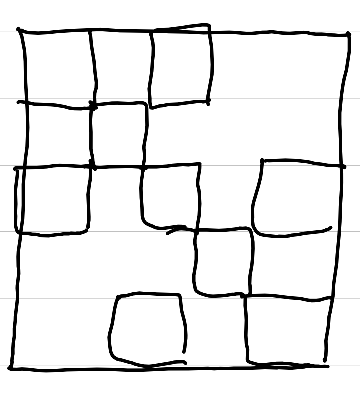
Multiplication Multithread
Matrices Sparse Symmétriques par block



- Matrices composées majoritairement de 0
- Données en blocks (taille 2, 3, 4 et symmétrique)

- On ne stock pas les 0

On stock donc pour chaque colone les block.

Forme classique:

Stockage requis

Stockage requis pour k blocks et une matrice de taille n

- $f(k, n) = (S\_Block)^2 \times k + k \times 2$ ← position block

(environ, on peut aligner les blocks sinon?)

tt

Format classique;

$f(k, n) = n^2$

$(S\_block)^2 \times k + k \times 2 \geq n^2$

$\Leftrightarrow k \geq \frac{n^2}{2+(S\_block)^2}$

nombre de block max; $\left(\frac{n}{S\_block}\right)^2$

donc $\frac{k}{\left(\frac{n}{S\_block}\right)^2}$ = densité

densité $\geq \frac{n^2}{n^2 \frac{(2+(S\_block)^2)}{(S\_block)^2}} = \frac{1}{1+\frac{2}{(S\_block)^2}}$

En terme de stockage on est gagnant pour une densité plus petite que:

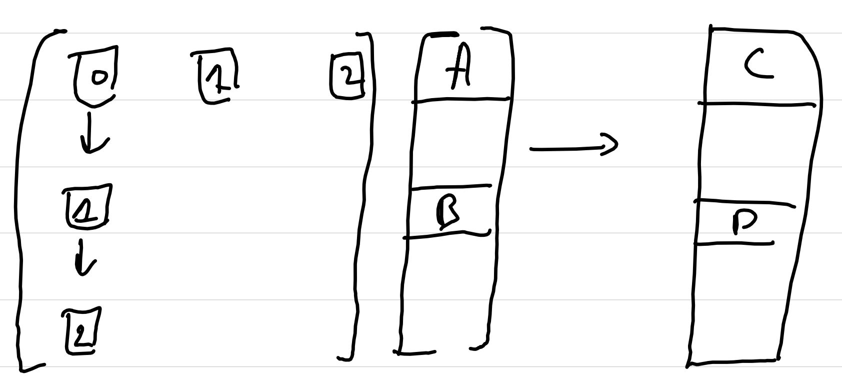
S_block = 2 ⇒ $\frac{2}{3}$

S_block = 3 ⇒ $\frac{9}{11}$

Même genre de chose en terme de calcul

Matrices sparse symmétriques:

- Systèmes physiques
  - bcp de systèmes physiques
- Multiplication vectorielle:
- Algo single thread: parcourir les blocks



Lorsque l'on parcourt on utilise la symétrie pour calculer les deux blocks en même temps

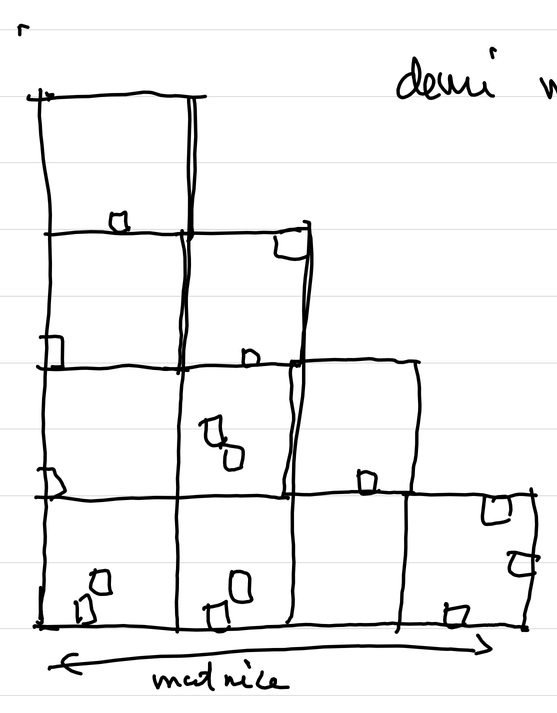
$C \pm \boxed{1} \times B$

$D \pm \boxed{1} \times A$ ⟼ On gagne déjà un facteur 2 car on ne charge pas deux fois le bloc dans le processeur

Les chargements mémoires sont le facteur limitant et l'arithmétique est moins coûteuse

vaut mieux 32op 1 load
que 16 op 1 load

- Multithread sans contrôle d'accès mémoire thread-safe ex: 4 threads

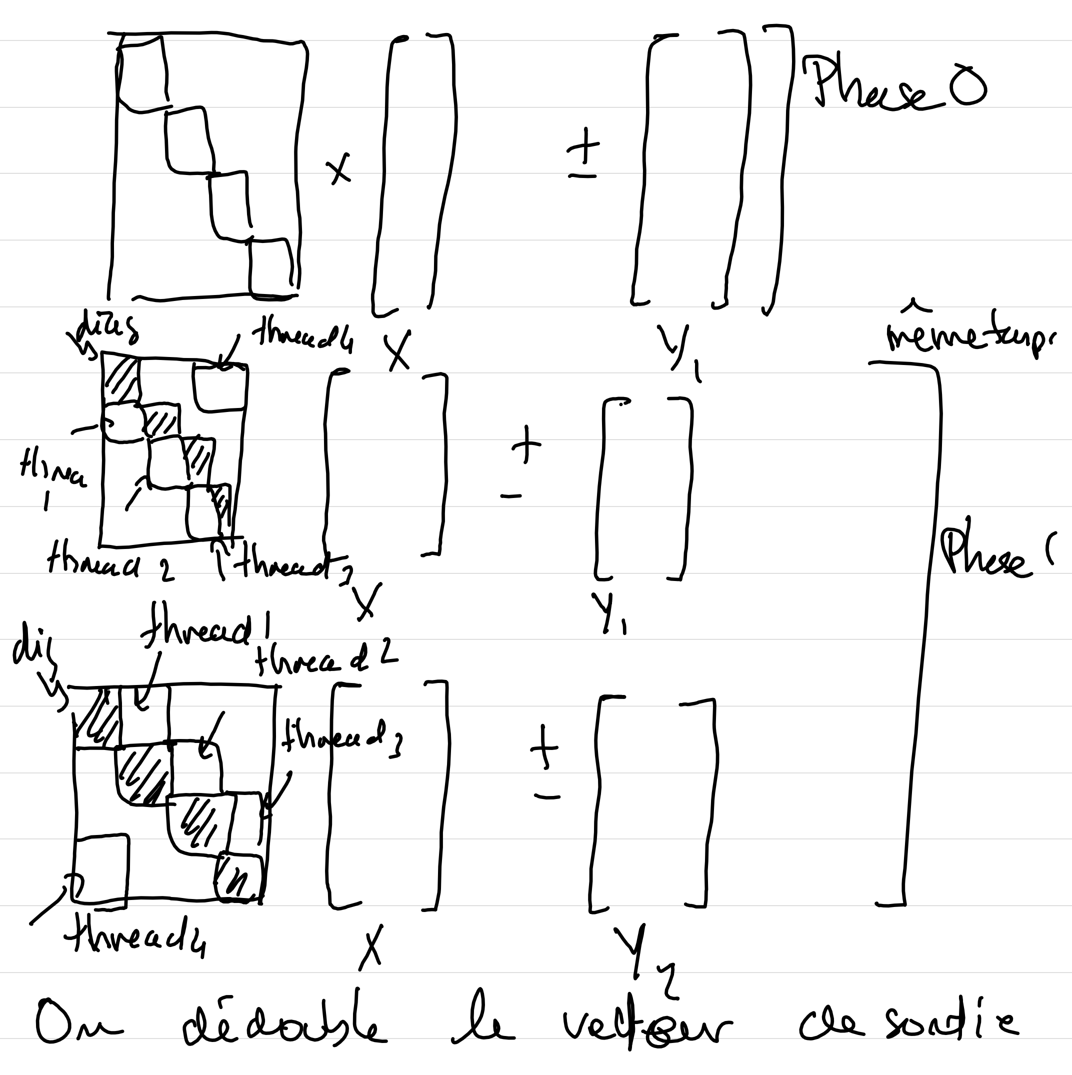


demi matrice stockée (symétrie?)

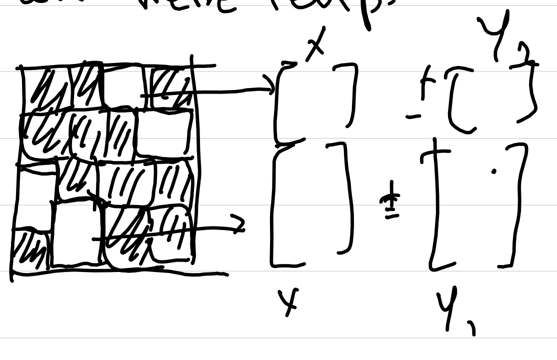
On veut 1: profiter de la symétrie
2: calculer un max de block en même temps

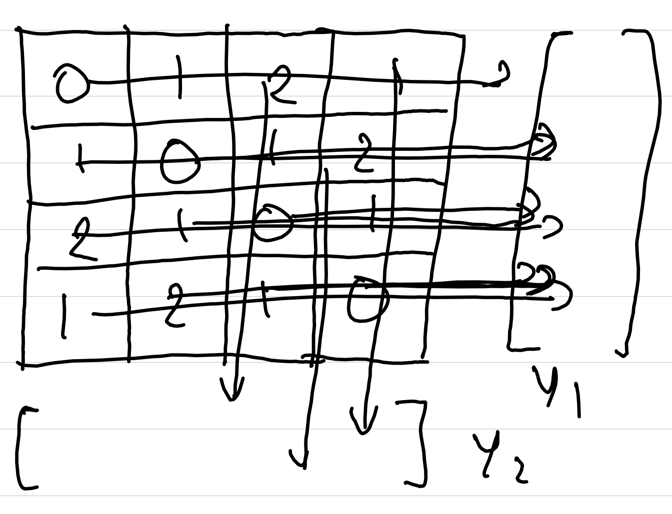
On coupe en zone come si dessus (ex threads = 4)

On fait les calculs dans cet ordre



On dédouble le vecteur de sortie pour pouvoir faire ces deux calculs en même temps





Avec cette méthode on peut répartir sur chaque phase le travail parmis tous les threads

Pour chaque phase et chaque threads

On a une quantité de blocks à calculer

On stock tout dans un tableau par phase

[16 | 16 | 16 | 16 | 16 | . . .] Phase 1
thread1 thread 2 th3 th4 th1

afin d'optimiser le cache grâce à la localité et les positions des blocks

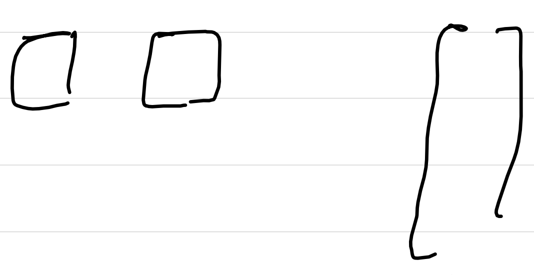
[2 | 2 | 2 | 2 | . . .]
thread1 thread2 thread3

(ix, iy)

On a également une multiplication optimisé pour calculer AX_1 et ${}^tAX_2$ en même temps pour minimiser les loads

- Améliorations possibles:

Profiter de l'alignement des blocks ex:



Ces deux blocks sont alignés et affecteront la même région de Y.
Chaque modifications de Y1 crée un load
On pourrait accumuler les résultats et ne modifier qu'une fois qu'un block non alignée apparait
(Alignement sur X pour un block implique alignement selon Y pour son symétrique)

Si 50% de blocks alignés (ce qui arrive souvent quasiment tjrs
gain de 50%.