HAI720
Programmation Algorithmique Efficace

Pascal Giorgi

Université de Montpellier
Faculté des Sciences

Organisation

Enseignant:

Pascal Giorgi, `pascal.giorgi@umontpellier.fr`

Planning:

- CM: 8 séances 1h30 (jeudi 13h15)
- TP: 8 séance 3h: jeudi 15h (Imagine+GL); vendredi 8h ou 9h30 (Algo)

- 1 contrôle continu (TP noté)
- 1 examen terminal (avec 2nd session et max entre session)

Note module: $\max(\text{ET}, 0.7\text{ET} + 0.3\text{CC})$

Objectifs

- compréhension et exploitation fine des architectures de calcul
- voir des concepts algorithmiques en lien avec les caractéristiques des architectures modernes

Thématiques abordées

- **mémoires**: cache, calcul en-place
- ***pipeline* de calcul**: ILP, vectorisation
- **parallelisme**: calcul multithread, GPGPU

$\Rightarrow$ analyse d'algorithmes et de leurs implantations

Motivations

L'analyse de complexité classique en temps des algorithmes n'est pas suffisante car elle ne reflète pas l'exécutions en pratique.

Exemple: le produit de matrice

complexité en $O(N^3)$ opérations

Motivations

L'analyse de complexité classique en temps des algorithmes n'est pas suffisante car elle ne reflète pas l'exécutions en pratique.

Exemple: le produit de matrice

complexité en $O(N^3)$ opérations

```c
void matmul(double C[N][N], double A[N][N], double B[N][N]){
  for(size_t i=0;i<N;i++)
    for(size_t j=0;j<N;j++){
      C[i][j]=0.;
      for(size_t k=0;k<N;k++)
        C[i][j]+=A[i][k]*B[k][j];
    }
}
```

Motivations

L'analyse de complexité classique en temps des algorithmes n'est pas suffisante car elle ne reflète pas l'exécutions en pratique.

Exemple: le produit de matrice

complexité en $O(N^3)$ opérations

```
void matmul(double C[N][N], double A[N][N], double B[N][N]){
  for(size_t i=0;i<N; i++)
    for(size_t j=0;j<N; j++){
      C[i][j]=0.;
      for(size_t k=0;k<N; k++)
        C[i][j]+=A[i][k]*B[k][j];
    }
}
```

$\Rightarrow$ En pratique, on peut gagner un facteur 100 sur l'implantation de cette algorithme !!!

- algorithmes de même complexité en $O(N^3)$, mais
- mieux adaptés au calcul sur les processeurs modernes (cache, SIMD, pipeline, multi-coeur)

Motivations

analyse compléxité = nbr. opérations de calcul

machine de turing déterministe (bits) ou *Word-RAM* (mot machine)

$\Rightarrow$ Pourquoi cela ne reflète pas l'exécutions en pratique

Motivations

analyse compléxité = nbr. opérations de calcul

machine de turing déterministe (bits) ou *Word-RAM* (mot machine)

$\Rightarrow$ Pourquoi cela ne reflète pas l'exécutions en pratique

```c
int a=1;
int b=2;
int c=a+b;
return c;
```

```asm
mov a, 1        ; create variable a
mov b, 2        ; create variable b
add c, a, b     ; add into c
push c          ; put return value in place
ret             ; return
```

Question

- les instructions sont exécutées séquentiellement les unes après autres ?

Motivations

analyse compléxité = nbr. opérations de calcul

machine de turing déterministe (bits) ou *Word-RAM* (mot machine)

⇒ Pourquoi cela ne reflète pas l'exécutions en pratique

```c
int a=1;
int b=2;
int c=a+b;
return c;
```

```asm
mov a, 1          ; create variable a
mov b, 2          ; create variable b
add c, a, b       ; add into c
push c            ; put return value in place
ret               ; return
```

Question

- les instructions sont exécutées séquentiellement les unes après autres ? FAUX

  ⇒ le processeur peut changer l'ordre (*out-of-order execution*)

  ⇒ 1 coeur peut exécuter plusieurs instructions en même temps (*proc. superscalaire*)

Motivations

analyse compléxité = nbr. opérations de calcul

machine de turing déterministe (bits) ou *Word-RAM* (mot machine)

⇒ Pourquoi cela ne reflète pas l'exécutions en pratique

```c
int a=1;
int b=2;
int c=a+b;
return c;
```

```asm
mov a, 1        ; create variable a
mov b, 2        ; create variable b
add c, a, b     ; add into c
push c          ; put return value in place
ret             ; return
```

Question

- les instructions sont exécutées séquentiellement les unes après autres ? FAUX
  - ⇒ le processeur peut changer l'ordre (*out-of-order execution*)
  - ⇒ 1 coeur peut exécuter plusieurs instructions en même temps (*proc. superscalaire*)
- l'accès aux variables a toujours le même coût ?

Motivations

analyse compléxité = nbr. opérations de calcul

machine de turing déterministe (bits) ou *Word-RAM* (mot machine)

⇒ Pourquoi cela ne reflète pas l'exécutions en pratique

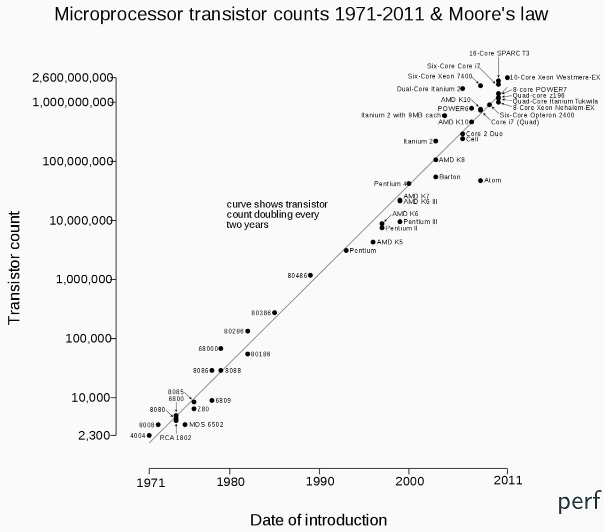
```c
int a=1;
int b=2;
int c=a+b;
return c;
```

```asm
mov a, 1        ; create variable a
mov b, 2        ; create variable b
add c, a, b     ; add into c
push c          ; put return value in place
ret             ; return
```

Question

- les instructions sont exécutées séquentiellement les unes après autres ? FAUX
  - ⇒ le processeur peut changer l'ordre (*out-of-order execution*)
  - ⇒ 1 coeur peut exécuter plusieurs instructions en même temps (*proc. superscalaire*)
- l'accès aux variables a toujours le même coût ? FAUX
  - ⇒ cela dépend d'où se trouve la donnée dans la hierarchie mémoire

Motivations

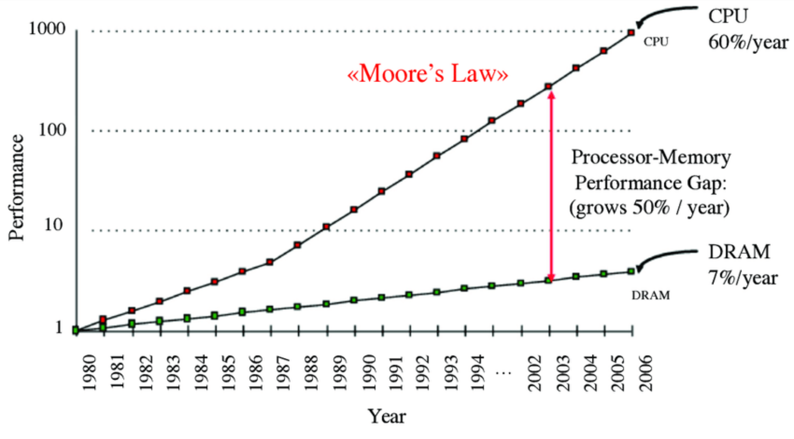


	gravure *(nm)*
1977	8000
1985	1000
⋮	⋮
2006	65
2008	45
2010	32
2012	22
2014	14
2017	10

atome Si $= 0.2\ nm$

perf énergétique $\Rightarrow$ limite à $\approx$ 4 GHz

Motivations



la mémoire est moins rapide que le calcul

$\Rightarrow$ l'analyse de la complexité spatiale des algorithmes est primordiale !!!

À avoir bien en tête

Amélioration de performance: essentiellement via du parallélisme !!!

Loi d' Amdahl

L'amélioration des performances via du parallélisme est limitée par la proportion de code séquentiel:

$$SP_{max} = \frac{1}{fs}$$

- SP_{max} représente le facteur d'accélération maximum avec une infinité de ressources
- fs représente la proportion de code séquentiel

$\Rightarrow$ un code ayant 80% d'instructions séquentielles aura un $SP_{max} = 1.25$

Plan: 1ère partie

1. Architecture matérielle et parallélisme d'instructions
2. Modèle de calcul SIMD: vectorisation sur les processeurs
3. Accès aux données: cache et complexité spatiale

Architecture matérielle et parallélisme d'instructions

Performance des programmes

Comment quantifier :

- CR= fréquence du processeur (ex. 3Ghz)
  $\hookrightarrow$ #cycles exécutés par seconde
- prog. CPU time= $\frac{\#(\text{cycle prog.})}{CR}$ en seconde

Performance des programmes

Comment quantifier :

- CR= fréquence du processeur (ex. 3Ghz)
  $\hookrightarrow$ #cycles exécutés par seconde
- prog. CPU time= $\frac{\#(\text{cycle prog.})}{CR}$ en seconde

$\Rightarrow$ améliorer performances: $\searrow$ #cycle prog ou $\nearrow$ CR.

Performance des programmes

Comment quantifier :

- CR= fréquence du processeur (ex. 3Ghz)
  $\hookrightarrow$ #cycles exécutés par seconde
- prog. CPU time= $\frac{\#(\text{cycle prog.})}{CR}$ en seconde

$\Rightarrow$ améliorer performances: $\searrow$ #cycle prog ou $\cancel{\nearrow CR}$ (limite à $\approx$ 4 GHz).

Performance des programmes

Comment quantifier :

- CR$=$ fréquence du processeur (ex. 3Ghz)
  $\hookrightarrow$ #cycles exécutés par seconde
- prog. CPU time$=$ $\frac{\#(\text{cycle prog.})}{CR}$ en seconde

$\Rightarrow$ améliorer performances: $\searrow$ #cycle prog ou $\cancel{\nearrow CR}$ (limite à $\approx$ 4 GHz).

Quantités intéressantes:

- flops: nombre d'opérations en nombre flottant (flop) par seconde
- peak performance: maximum théorique de *Gflops* (liée à CR)
- Instruction Level Parallelism (ILP): #instructions pouvant être traiter en parallèle

$\Rightarrow$ proc. 3Ghz $\rightarrow$ peak perf. $= 3 \times 10^9$ flops $=$ 3 Gflops si 1 op/cycle

Performance au niveau des instructions

$$CPI = \frac{\#(prog.cycles)}{\#(prog.instructions)}$$

CPI: Clock cycles per instruction

nombre moyen de cycles d'horloge par instruction exécutée

- chaque instruction à un nombre de cycle (latence) différent
- dépend de l'ILP de l'architecture et du programme

$\Rightarrow$ prog. CPU time$= \frac{\#(prog.instructions) \times CPI}{CR}$

Performance des programmes: dépendance aux instructions

$$\text{prog. CPU time} = \frac{\#\text{prog. instructions} \times CPI}{CR}$$

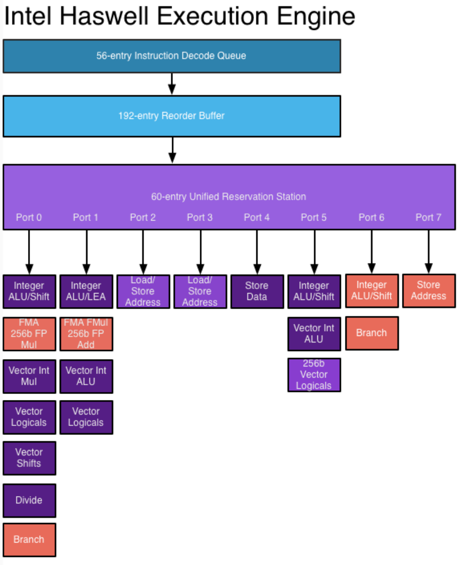
	#instructions	CPI	CR
Algorithme	X	X	
Langage Programmation	X	X	
Compilateur	X	X	
ISA	X	X	X
Design processeur		X	X

1 cœur CPU moderne:

$\Rightarrow$ peut faire plusieurs instructions en même temps ($ILP > 1$)

$\Rightarrow$ peut faire une instruction sur plusieurs données en même temps (*vectorization*)

Moteur d'exécution d'un coeur processeur



Théoriquement, on peut faire en même temps

- jusqu'a 8 instructions **différentes**
- jusqu'a 4 instruction **identiques**

$\Rightarrow$ besoin de recouvrir leur gestion

Parallelisme d'instruction

RISC: Restricted Instructions Set Computer

- instructions de taille fixe (e.g. 32 ou 64 bits)
- les opérandes sont uniquement des registres
- accès mémoire via des instructions dédiées (load/store)

Chaque instruction nécessite jusqu'a 5 cycles

- (IF) *Fetch*: charge l'instruction depuis la mémoire dédiée
- (ID) *Decode*: décode l'instruction et les registres des opérandes
- (EX) *Execute*: exécute l'instruction (ALU)
- (MEM) *Memory*: lecture/écriture des donnés en mémoire
- (WB) *Write Back*: écriture des données en registre

$\Rightarrow$ branching=2 cycles; store=4 cycles, others=5 cycles

Parallélisme instruction: pipeline matériel

- pas de pipeline: $CPI = 5 \Rightarrow$ 3 instructions en 15 cycles

Parallélisme instruction: pipeline matériel

- pas de pipeline: $CPI = 5 \Rightarrow$ 3 instructions en 15 cycles

IF	ID	EX	MEM	WB										
					IF	ID	EX	MEM	WB					
										IF	ID	EX	MEM	WB

$\downarrow i$

$\xrightarrow{t}$

- avec pipeline: $CPI = \frac{5+\#\text{instructions}-1}{\#\text{instructions}} = 1 + \epsilon$

IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

$\downarrow i$

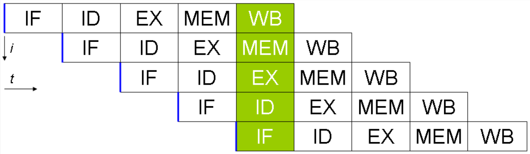
$\xrightarrow{t}$

$\Rightarrow$ 5 instructions en 9 cycles: CPI=1.8

Parallélisme instruction: pipeline matériel

- pas de pipeline: $CPI = 5 \Rightarrow$ 3 instructions en 15 cycles

- avec pipeline: $CPI = \frac{5+\#\text{instructions}-1}{\#\text{instructions}} = 1 + \epsilon$



$\Rightarrow$ 5 instructions en 9 cycles: CPI=1.8

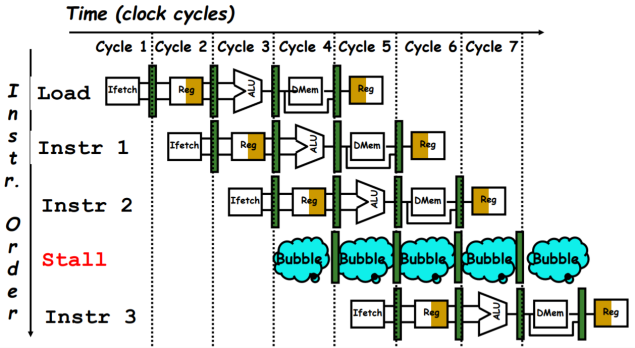
Dès que le pipeline est plein, CPI=1 $\Rightarrow$ plus compliqué en pratique

Problème avec le pipeline

Pipeline stall (blocage)

- *structural hazards*: combinaison d'instructions non supportée
- *data hazards*: utilisation d'une donnée en production dans le pipeline
- *control hazards*: décision de branchement trop hative

⇒ Le pipeline gère les blocages en décalant le cycle prévu



Gestion du parallèlisme d'instructions

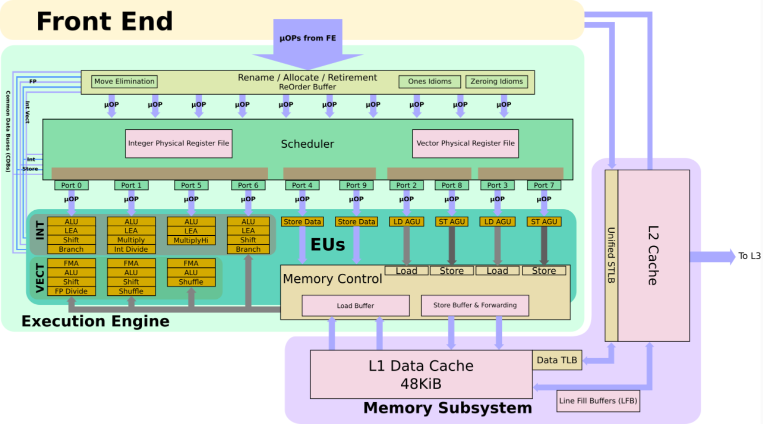
Dans le processeur

- ordonnancement dynamique *out-of-order* des instructions (en fonction des ports)
- Intel Skylake: considère 224 instructions pour réordonner

```c
int a, b, c, d;
a = 2 - 1;
b = 1 + 1;
c = a + b; // doit attendre le calcul de a et de b
d = 8 / 2; // peut être exécuter sans délai
```

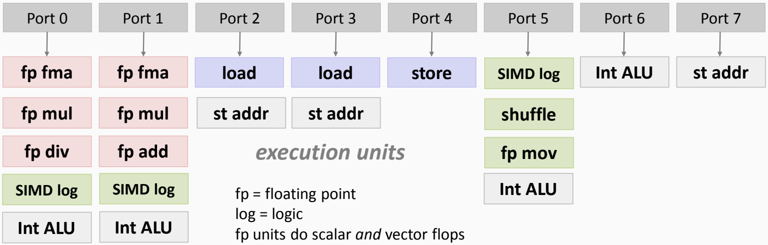
Quelles instructions en parallèle

Cela dépend de l'architecture du processeur



⇒ Holy grail: Agner Fog's website

Exemple: Haswell performance



fp = floating point
log = logic
fp units do scalar *and* vector flops
SIMD log: other, non-fp SIMD ops

Execution Unit (fp)	Latency [cycles]	Throughput [ops/cycle]	Gap [cycles/issue]
fma	5	2	0.5
mul	5	2	0.5
add	3	1	1
div (scalar) div (4-way)	14-20 25-35	1/13 1/27	13 27

- Gap = 1/throughput
- ***Intel calls gap the throughput!***
- Same exec units for scalar and vector flops
- Same latency/throughput for scalar (one double) and AVX vector (four doubles) flops, except for div

Exemple: Haswell performance

- 2 unités de calcul $FMA^1 \rightarrow a \times b + c$: $\Rightarrow$ débit de 2 FMA/cycle
- registre vectoriel de 256 bits $\Rightarrow$ 4 op. double en même temps

```
x, y are vectors of doubles of length n, alpha is a double
```

```
for (i = 0; i < n; i++)
  x[i] = x[i] + alpha*y[i];
```

$\Rightarrow$ #flop algorithme $= 2n$

- *runtime* sans vectorisation: $\frac{n}{2}$
- *runtime* avec vectorisation: $\frac{n}{8}$

[1] *fused multiply and add*

Exemple: Haswell performance

- 2 unités de calcul FMA[1] $\rightarrow a \times b + c$: $\Rightarrow$ débit de 2 FMA/cycle
- registre vectoriel de 256 bits $\Rightarrow$ 4 op. double en même temps

`x, y are vectors of doubles of length n, alpha is a double`

```
for (i = 0; i < n; i++)
    x[i] = x[i] + alpha*y[i];
```

$\Rightarrow$ #flop algorithme $= 2n$

- *runtime* sans vectorisation: $\frac{n}{2}$
- *runtime* avec vectorisation: $\frac{n}{8}$

```
for (i = 0; i < n; i++)
    alpha = x[i] + alpha*y[i];
```

$\Rightarrow$ #flop algorithme $= 2n$

[1] *fused multiply and add*

Exemple: Haswell performance

- 2 unités de calcul $FMA^1 \to a \times b + c$: $\Rightarrow$ débit de 2 FMA/cycle
- registre vectoriel de 256 bits $\Rightarrow$ 4 op. double en même temps

x, y are vectors of doubles of length n, alpha is a double

```
for (i = 0; i < n; i++)
    x[i] = x[i] + alpha*y[i];
```

$\Rightarrow$ #flop algorithme $= 2n$

- *runtime* sans vectorisation: $\frac{n}{2}$
- *runtime* avec vectorisation: $\frac{n}{8}$

```
for (i = 0; i < n; i++)
    alpha = x[i] + alpha*y[i];
```

$\Rightarrow$ #flop algorithme $= 2n$

- *runtime* sans vectorisation: n
- *runtime* avec vectorisation: n

[1] *fused multiply and add*

Exemple: Haswell performance

Analyse d'ILP $\Rightarrow$ borne inférieur sur le nbr de cycles

```
double f(double a, double b, double c){
  double r;
  r = (a + b) * (b + c) + (a * c);
  return r;
}
```

Combien de cycles pour exécuter la fonction `f` sur Haswell ?

Exemple: Haswell performance

Analyse d'ILP $\Rightarrow$ borne inférieur sur le nbr de cycles

```c
double f(double a, double b, double c){
  double r;
  r = (a + b) * (b + c) + (a * c);
  return r;
}
```

Combien de cycles pour exécuter la fonction `f` sur Haswell ?

- sans FMA :
- avec FMA :

Exemple: Haswell performance

Analyse d'ILP ⇒ borne inférieur sur le nbr de cycles

```
double f(double a, double b, double c){
  double r;
  r = (a + b) * (b + c) + (a * c);
  return r;
}
```

Combien de cycles pour exécuter la fonction `f` sur Haswell ?

- sans FMA : 12 cycles
- avec FMA :

Exemple: Haswell performance

Analyse d'ILP $\Rightarrow$ borne inférieur sur le nbr de cycles

```
double f(double a, double b, double c){
  double r;
  r = (a + b) * (b + c) + (a * c);
  return r;
}
```

Combien de cycles pour exécuter la fonction `f` sur Haswell ?

- sans FMA : 12 cycles
- avec FMA : 10 cycles

Analyse de performances *apriori*

Besoin de connaitre les complexités exactes des algorithmes pas avec des $O(\dots)$.

- besoin de compter séparément les additions, multiplications, divisions
- pas besoin de compter les opérations de contrôle (boucle, conditionnelle, ...)

Besoin de connaitre l'architecture de son processeur:

- mapping des opérations sur les ports d'éxécution
- debit et latence des opérations

Comment optimiser l'ILP dans les programmes (1/4)

ATTENTION: le compilateur peut optimiser mais pas toujours, il faut l'aider !!!

Comment optimiser l'ILP dans les programmes (1/4)

ATTENTION: le compilateur peut optimiser mais pas toujours, il faut l'aider !!!

- utiliser des variables supplémentaires

```
t4 = t0 + t1;
t4 = t4 + t2;
t4 = t4 + t3;
```

$\Rightarrow$ ILP=1

```
t4 = t0 + t1;
t5 = t2 + t3;
t4 = t4 + t5;
```

$\Rightarrow$ ILP=1.5

Comment optimiser l'ILP dans les programmes (2/4)

- appel de fonctions ⇒ le compilateur ne peut pas toujours les simplifier

```c
long f();
long f1(){ return f()+f()+f()+f() ;}
```

```c
long f();
long f2(){ return 4*f();}
```

Comment optimiser l'ILP dans les programmes (2/4)

- appel de fonctions ⇒ le compilateur ne peut pas toujours les simplifier

```
long f();
long f1(){ return f()+f()+f()+f() ;}
```

```
long f();
long f2(){ return 4*f();}
```

Problème:

les 2 codes ne sont pas identiques

```
long counter=0;
long f() { return counter++;}
```

Le compilateur conserve les appels de fonction (à cause des effets de bord)

Comment optimiser l'ILP dans les programmes (2/4)

- appel de fonctions ⇒ le compilateur ne peut pas toujours les simplifier

```c
long f();
long f1(){ return f()+f()+f()+f() ;}
```

```c
long f();
long f2(){ return 4*f();}
```

Problème:

les 2 codes ne sont pas identiques

```c
long counter=0;
long f() { return counter++;}
```

Le compilateur conserve les appels de fonction (à cause des effets de bord)

⇒ en fait, il peut *inliner* les appels avec l'option `-finline` ou à partir de `-O1`

Comment optimiser l'ILP dans les programmes (2/4)

- appel de fonctions ⇒ le compilateur ne peut pas toujours les simplifier

```
long f();
long f1(){ return f()+f()+f()+f() ;}
```

```
long f();
long f2(){ return 4*f();}
```

Problème:

les 2 codes ne sont pas identiques

```
long counter=0;
long f() { return counter++;}
```

Le compilateur conserve les appels de fonction (à cause des effets de bord)

⇒ en fait, il peut *inliner* les appels avec l'option `-finline` ou à partir de `-O1`

⇒ mais pas toujours, cf lower1.cpp lower2.cpp

Comment optimiser l'ILP dans les programmes (3/4)

- *memory aliasing* ⇒ 2 pointeurs peuvent méner à la même donnée

```
void twiddle(long *xp, long *yp){
  *xp += *yp;
  *xp += *yp;
}
```

```
void twiddle2(long *xp, long *yp){
  *xp += 2 *yp;
}
```

Comment optimiser l'ILP dans les programmes (3/4)

- *memory aliasing* ⇒ 2 pointeurs peuvent méner à la même donnée

```
void twiddle(long *xp, long *yp){
  *xp += *yp;
  *xp += *yp;
}
```

```
void twiddle2(long *xp, long *yp){
  *xp += 2 *yp;
}
```

Problème:

les 2 codes ne sont pas identiques

- *twiddle*(&*x*,&*x*) ⇒ x ← 4x
- *twiddle2*(&*x*,&*x*) ⇒ x ← 3x

le compilateur fait l'hypothèse que deux pointeurs mènent à la même donnée

Memory aliasing et performance

```c
1 /* somme des lignes de la matrice a dans le vecteur b */
2 void sum_row (double **a, double *b, int n) {
3   int i, j;
4   for (i = 0; i < n; i++) {
5     b[i] = 0;
6     for (j = 0; j < n; j++)
7       b[i] += a[i][j];
8   }
9 }
```

⇒ la ligne 7 :`b[i] += a[i][j];` impose une écriture dans la mémoire à chaque itération

En effet, on peut faire

```c
double A[2][2] = {1,2,3,4};
double *B=&(A[0][0]);
sum_row(A,B,2);
```

Memory aliasing et performance

Suppression de l'aliasing (*possible uniquement si la fonction veut l'interdire*

```
/* somme des lignes de la matrice a dans le vecteur b */
void sum_row (double **a, double *b, int n) {
  int i, j;
  double res;
  for (i = 0; i < n; i++) {
    res = 0;
    for (j = 0; j < n; j++)
      res += a[i][j];
    b[i]=res;
  }
}
```

- copie des données mémoires réutilisées dans une boucle vers des temporaires
- calcul effectué avec les temporaires et ré-écriture du résultat en mémoire à la fin

Memory aliasing et performance

Suppression de l'aliasing (*possible uniquement si la fonction veut l'interdire*

```c
/* somme des lignes de la matrice a dans le vecteur b */
void sum_row (double **a, double *b, int n) {
  int i, j;
  double res;
  for (i = 0; i < n; i++) {
    res = 0;
    for (j = 0; j < n; j++)
      res += a[i][j];
    b[i]=res;
  }
}
```

- copie des données mémoires réutilisées dans une boucle vers des temporaires
- calcul effectué avec les temporaires et ré-écriture du résultat en mémoire à la fin

⇒ améliore l'utilisation des registres CPU et favorise l'ILP

Comment optimiser l'ILP dans les programmes (4/4)

Pour exhiber plus de parallelisme on peut dérouler les boucles à la main sur quelques itérations:

```c
/* somme des lignes de la matrice a dans le vecteur b */
void sum_row (double **a, double *b, int n) {
  int i, j;
  double res1,res2;
  for (i = 0; i < n-1 ; i+=2) { // 2 lignes à la fois
    res1= res2= 0;
    for (j = 0; j < n; j++){
      res1 += a[i][j];
      res2 += a[i+1][j];
    }
    b[i]=res1;
    b[i+1]=res2;
  }
  // code pour la dernière ligne si n est impair
  res1=0;
  for (;i<n;i++)
    res1 += a[i][j];
  b[i]=res;
}
```

$\Rightarrow$ améliore l'ILP du corps de boucle

Premières optimisations: ex. réduction d'un vecteur

C'est le *reduce* dans *map/reduce*

$\Rightarrow$ réduction de n élément à un seul par application successive d'un opérateur binaire

$$v[0] \text{ OP } v[1] \text{ OP } v[2] \text{ OP } \ldots \text{ OP } v[n-1]$$

avec OP={+,*} et START= {0,1}

```
#define OP *
#define START 1
template< typename T>
void reduce(const vector<T> &V, T &res){
  res=START;
  for(size_t i=0;i< V.size(); i++)
    res= res OP V[i];
}
```

Est-ce que ce code est efficace ? et comment l'optimiser ?

Premières optimisations: ex. réduction d'un vecteur

C'est le *reduce* dans *map/reduce*

⇒ réduction de n élément à un seul par application successive d'un opérateur binaire

$$v[0] \text{ OP } v[1] \text{ OP } v[2] \text{ OP } \ldots \text{OP } v[n\text{-}1]$$

avec OP={+,*} et START= {0,1}

```
#define OP *
#define START 1
template< typename T>
void reduce(const vector<T> &V, T &res){
  res=START;
  for(size_t i=0;i< V.size(); i++)
    res= res OP V[i];
}
```

Est-ce que ce code est efficace ? et comment l'optimiser ? `https://godbolt.org`

Optimisation de l'ILP

Règle générale:

- introduire de nouvelles variables c'est pas mauvais !!!
- supprimer les appels de fonction inutiles
- éviter les lecture/écritures en mémoire dans les boucles (pb aliasing)
- dérouler les boucles en exhibant du parallélisme d'instructions/données

Attention

L'utilisation de variables supplémentaires doit rester raisonnable

. $\Rightarrow$ le nombre de registre CPU est de l'ordre de 16 ou 32 (au dela utilisation de la pile)

Optimisation de l'ILP

Note sur les branchements conditionnels

- si le pattern des branchements est régulier $\Rightarrow$ pas de problème
- si le pattern des branchements n'est pas régulier $\Rightarrow$ vidage du pipeline

Astuce: Pensez à utiliser des affectations conditionnelles

```
// reordonne a et b tel que a[i]<b[i]
void minmax(int* a, int* b, size_t n){
  for(size_t i=0;i<n;i++)
    if (a[i]>b[i]){
      int tmp=a[i];
      a[i]=b[i]; b[i]=tmp
    }
}
```

```
// reordonne a et b tel que a[i]<b[i]
void minmax(int* a, int* b, size_t n){
  for(size_t i=0;i<n;i++){
    int min=(a[i]>b[i] ? b[i]: a[i]);
    int max=(a[i]>b[i] ? a[i]: b[i]);
    a[i]=min;
    b[i]=max;
  }
}
```

Optimisation de l'ILP et compilateur

Cas d'étude $\Rightarrow$ la somme préfixe d'un vecteur

Modèle de calcul SIMD: vectorisation sur les processeurs

Catégorisation des architectures matérielles

Taxonomy de Flint (1966)

	single instruction	multiple instruction
single data	SISD	MISD
multiple data	SIMD	MIMD

Extension vectorielle SIMD

4 way

- Extension du jeu d'instructions (ISA)
- type de données/instructions pour des calculs parallèles sur des petits vecteurs (2,4,8,. . .)
  $\Rightarrow$ disponibles pour les entiers et flottants
- Noms: SSE, SSE2, SSE3, AVX, AVX2, AVX=512, . . .

Extension vectorielle SIMD

- Extension du jeu d'instructions (ISA)
- type de données/instructions pour des calculs parallèles sur des petits vecteurs (2,4,8,...)
  $\Rightarrow$ disponibles pour les entiers et flottants
- Noms: SSE, SSE2, SSE3, AVX, AVX2, AVX=512, ...

Pourquoi ?

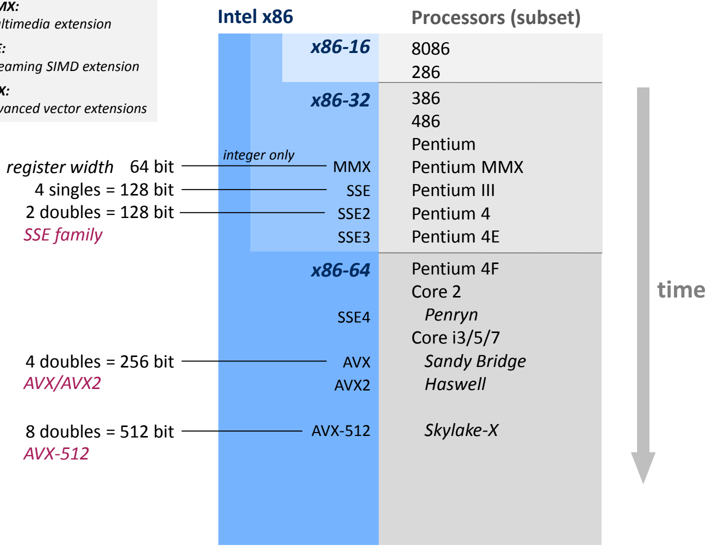
- utile dans les applications ayant un parallelisme à grain fin (multimédia)
  $\Rightarrow$ speedup de la taille du vecteur
- facile à designer dans les processeurs

Historique

MMX:
Multimedia extension

SSE:
Streaming SIMD extension

AVX:
Advanced vector extensions



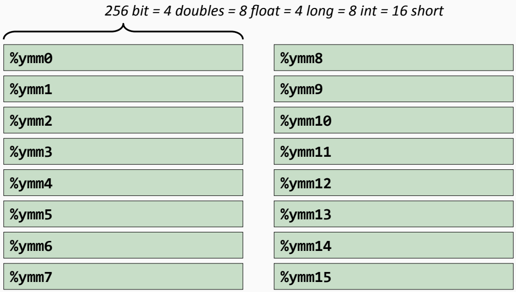
Jeux d'instructions AVX

Introduction de registres de 256 bits (%*ymm0*,...,%*ymm15*)

- avec op. flottantes uniquement (AVX)
- avec op. flottantes et entières (AVX2)

$\Rightarrow$ AVX introduit les intructions VEX à 3 opérandes ($c = a + b$ au lieu de $a = a + b$)

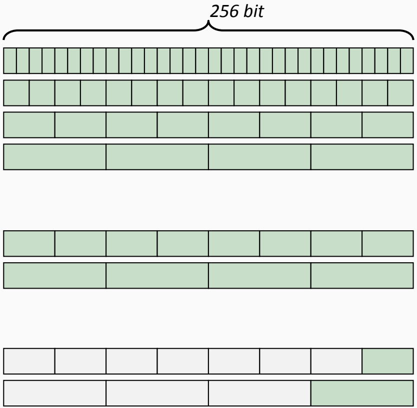
$\Rightarrow$ AVX2 introduit le FMA ($d = c + a * b$)



16 registres disponibles

Jeux d'instructions AVX

Plusieurs tailles de données et de vecteurs:



Integer vectors:

- *32-way byte*
- *16-way 2 bytes*
- *8-way 4 bytes*
- *4-way 8 bytes*

Floating point vectors:

- *8- way single*
- *4-way double*

Floating point scalars:

- *single*
- *double*

Tous les jeux d'instructions SIMD

Seule la taille, le nom des registres et les noms des instructions changent

- SSE : 16 registres de 128 bits (`%xmm0,...,%xmm15`)
- AVX : 16 registres de 256 bits (`%ymm0,...,%ymm15`)
- AVX-512 : 32 registres de 512 bits (`%zmm0,...,%zmm32`)

en pratique `%xmm0`= 128-bits LSB de `%ymm0` et `%ymm0`= 256-bits LSB de `%zmm0`

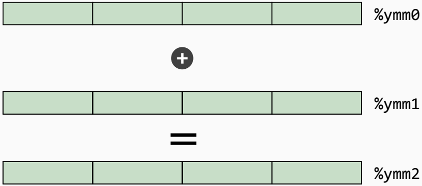
⇒ 32 registres SSE et AVX sur une architecture AVX-512

Instruction d'addition SIMD : Exemple

Sur des flottants double précision:

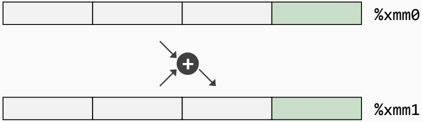
(three-operand!)

4-way vector add (in AVX) `vaddpd %ymm2 %ymm0 %ymm1`



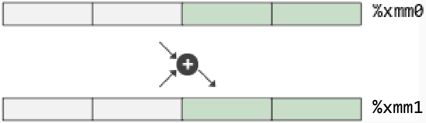
(two-operand!)

scalar add (in SSE2) `addsd %xmm1 %xmm0`



(two-operand!)

2-way vector add (in SSE2) `addpd %xmm1 %xmm0`



SIMD: convention de nommage

Les fonctions assembleur suivent une convention de nommage qui dépend:

- du type de données (`float double`, `int`, `unsigned int`, `unsigned long`)
- des registres utilisés (SSE 128 bits, AVX 256 bits ou AVX-512 512 bits)

Addition sur les nombres flottants:

	vector		scalar	
	SSE	AVX	SSE	AVX
float	addps	vaddps	addss	vaddss
double	addpd	vaddpd	addsd	vaddsd

$\Rightarrow$ il faut toujours utiliser le bon type de registres: *xmm*=128bits,*ymm*=256bits,*zmm*=512bits

Exemple produit scalaire sur `https://godbolt.org`

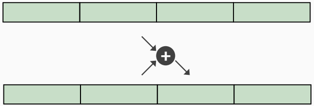
Comment tirer partie du SIMD

⇒ besoin de parallelisme à grain fin (*fine grained*)

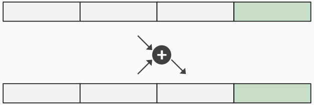
Quelles options :

- des bibliothèques vectorisées (pas toujours existant)
- auto-vectorisation par le compilateur
- utiliser des *intrinsic* dans vos programmes
- faire de l'assembleur

Comment tirer partie du SIMD



au lieu de



⇒ besoin de parallelisme à grain fin (*fine grained*)

Quelles options :

- des bibliothèques vectorisées (pas toujours existant)
- auto-vectorisation par le compilateur
- utiliser des *intrinsic* dans vos programmes
- faire de l'assembleur

Intrinsic SIMD

- exhibe les instructions SIMD au niveau C/C++
- correspond à des fonctions codées en assembleur
- *inline* à la compilation: aucun surcout

$\Rightarrow$ $\approx$ 6 000 *intrinsic*

type registres SSE (128 bits)

```
__m128 f;  // float  f0, f1, f2, f3
__m128d d; // double d0,d1
__m128i i; // 16xint8, 8xint16, 4xint32, 2xint64
```

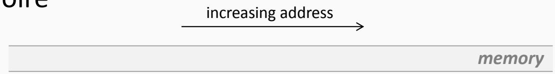
type registres AVX/AVX2 (256 bits)

```
__m256 f;  // float  f0, f1, f2, f3, f4, f5, f6, f7
__m256d d; // double d0,d1,d2,d3
__m256i i; // 32xint8, 16xint16, 8xint32, 4xint64
```

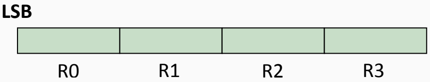
$\Rightarrow$ on ne considérera que l'AVX dans la suite car toutes les machines récentes l'ont

Convention visuelle

Mémoire



Registre



Exemple d'*intrinsinc*

```c
// t must be 32-byte aligned
double t[4]={1.0, 2.0, 3.0, 4.0};
__m256d a = _mm256_load_pd(t);
```

$\Rightarrow$ LSB

1.0	2.0	3.0	4.0	a

$\Rightarrow$ équivalent à *__m256d a* = *_m256_set_pd*(4.0, 3.0, 2.0, 1.0);

Difficultés avec les *intrinsic* SIMD

- gestion explicite des transferts de données: registre $\leftrightarrow$ mémoire
  $\hookrightarrow$ *load/compute/store*
- l'alignement des données en mémoire est important: 256 bits $=$ 32 octets
- réarrangement des données dans les registres potentiellement couteux (*shuffle op.*)
- pas de complétude dans les instructions disponibles
- plusieurs choix pour un même calcul: importance latences/débit des instructions

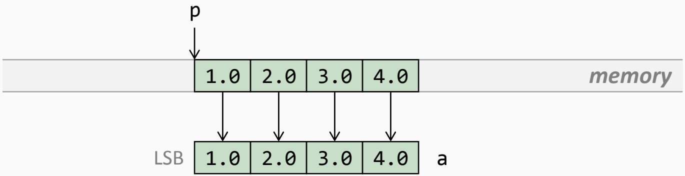
$\Rightarrow$ Holy grail: Intel's intrinsic guide

Tour d'horizon des instructions SIMD intéressantes

- load/store données
- arithmétique
- déplacement de données

Chargement de données: *load*

chargement de 4 double dans une registre de 256 bits:




```
a = _mm256_load_pd(p);   // p 32-byte aligned
```

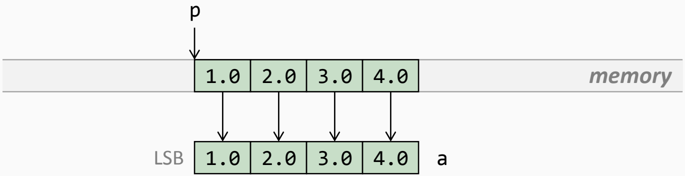
```
a = _mm256_loadu_pd(p); // p not aligned
```

potentiel ralentissement

$\Rightarrow$ *_mm256_load_ps*(*p*) ou *_mm256_loadu_ps*(*p*) : chargement de 8 float à partir de *p*

Chargement de données: *load*

chargement de 4 double dans une registre de 256 bits:




```
a = _mm256_load_pd(p);   // p 32-byte aligned
```

```
a = _mm256_loadu_pd(p); // p not aligned
```

potentiel ralentissement

$\Rightarrow$ *_mm256_load_ps*(*p*) ou *_mm256_loadu_ps*(*p*) : chargement de 8 float à partir de *p*

DANGER: *_m256_load* segfault si *p* mal aligné

Chargement de données: *load*

```
__m256d _mm256_load_pd (double const * mem_addr)
```
vmovapd

Synopsis

```
__m256d _mm256_load_pd (double const * mem_addr)
#include <immintrin.h>
Instruction: vmovapd ymm, m256
CPUID Flags: AVX
```

Description

Load 256-bits (composed of 4 packed double-precision (64-bit) floating-point elements) from memory into `dst`. `mem_addr` must be aligned on a 32-byte boundary or a general-protection exception may be generated.

Operation

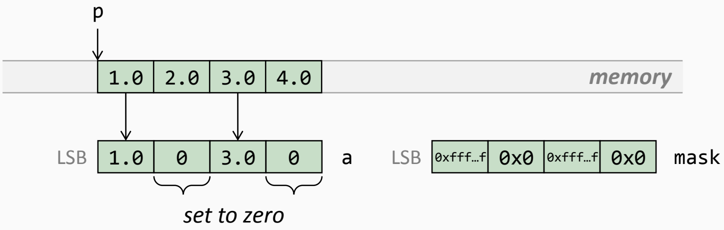
```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	7	0.5
Skylake	7	0.5
Broadwell	1	0.5
Haswell	1	0.5
Ivy Bridge	1	1

Chargement de données conditionnel: *maskload*

chargement d'une sélection parmi 4 double dans une registre de 256 bits:




```
a = _mm256_maskload_pd(p, mask); // p any alignment
```

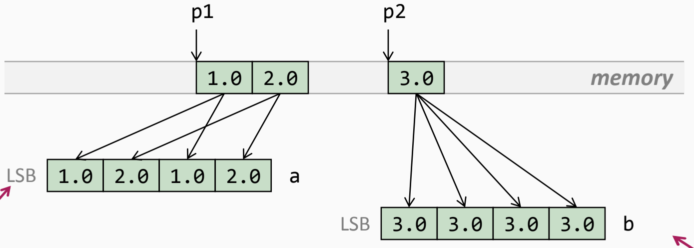
__m256i

Performance

Architecture	Latency	Throughput (CPI)
Icelake	7	0.5
Skylake	7	0.5

Chargement avec duplication de données : *broadcast*

chargement et duplication d'un ou deux double dans une registre de 256 bits:




```
a = _mm256_broadcast_pd(p1); // p1 any alignment
```

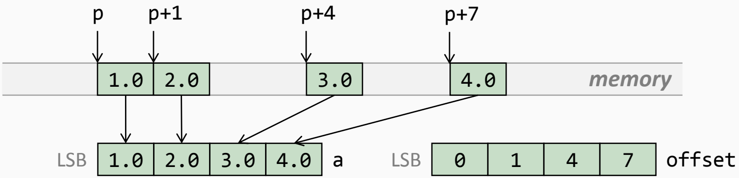
```
b = _mm256_broadcast_sd(p2); // p2 any alignment
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	7	0.5
Skylake	7	0.5

Chargement avec regroupement de données : *gather*

chargement et regroupement de 4 double dans une registre de 256 bits:



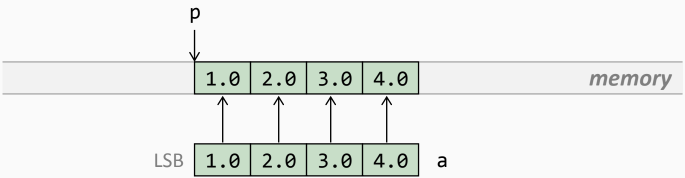

```
a = _mm256_i64gather_pd(p, offset, 8); // p any alignment
```

`scale = {1,2,4,8}`
above: scale = 8 = size of double

Performance: un peu mieux que 4 load

Enregistrement de données: *store*

Écriture d'un registre de 256 bits contenant 4 double en mémoire




```
_mm256_store_pd(p,a)     // p 32-byte aligned
```

```
_mm256_storeu_pd(p,a)    // p not aligned
```

potentiel ralentissement

DANGER: *_m256_store* segfault si *p* mal aligné

Enregistrement de données: *store*

`void _mm256_store_pd (double * mem_addr, __m256d a)` vmovapd

Synopsis

```
void _mm256_store_pd (double * mem_addr, __m256d a)
#include <immintrin.h>
Instruction: vmovapd m256, ymm
CPUID Flags: AVX
```

Description

Store 256-bits (composed of 4 packed double-precision (64-bit) floating-point elements) from `a` into memory. `mem_addr` must be aligned on a 32-byte boundary or a general-protection exception may be generated.

Operation

```
MEM[mem_addr+255:mem_addr] := a[255:0]
```

Performance

Architecture	Latency	Throughput (CPI)
Skylake	5	1
Broadwell	1	0.5
Haswell	1	0.5
Ivy Bridge	1	1

Chargement de constantes

chargement et/ou réplication de données constantes

LSB	1.0	2.0	3.0	4.0	a
LSB	1.0	1.0	1.0	1.0	b
LSB	0	0	0	0	c

```
a = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);
```

```
b = _mm256_set1_pd(1.0);
```

```
c = _mm256_setzero_pd();
```

⇒ les constantes peuvent être remplacer par des variables simples

double *a*=...; *_mm_set1_pd*(*a*);

SIMD load/store à retenir

- type d'instructions similaires : aligné ou non en mémoire
- débit pas forcément identique (load $\times 2$ par rapport au store)
- latence importante ($\approx$ 7 cycles) sur architectures récentes
- chargement de constante

Allocation alignée sur 32 octets (AVX) de 1024 `double`:

- `double * ptr32 = static_cast<double*> std::aligned_alloc(32, 1024);` en C++17
- `double * ptr32 = (double*) posix_memalign(32,1024);` en C

Exercice

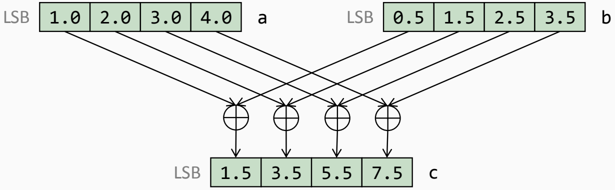
Écrire une fonction qui créé une copie d'un tableau en utilisant des instructions AVX:

- pour des `float`
- pour des `double`

Arithmétique SIMD

Intrinsic	**opération arithmétique correspondante**
_mm256_add_pd	addition
_mm256_sub_pd	soustraction
_mm256_mul_pd	multiplication
_mm256_div_pd	division
_mm256_fmadd_pd	fma
_mm256_hadd_pd	addition intra-registre
_mm256_ceil_pd	arrondi entier supérieur
_mm256_floor_pd	arrondi entier inférieur
_mm256_max_pd	maximun
_mm256_min_pd	minimum
_mm256_sqrt_pd	racine carré
⋮	⋮

Opérations classiques




```
c = _mm256_add_pd(a, b);
```

analogous:

```
c = _mm256_sub_pd(a, b);
```

```
c = _mm256_mul_pd(a, b);
```

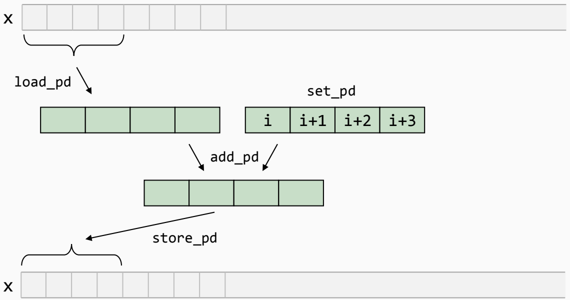
```
c = _mm256_div_pd(a, b);
```

CPI$=$ 0.5 ou 1 pour $\{+, -, \times\}$ et $9 \leq CPI \leq 28$ pour $\div$

Exemple de vectorisation

```c
void addindex(double *x, int n) {
  for (int i = 0; i < n; i++)
     x[i] = x[i] + i;
}
```

Graphiquement:



Exemple de vectorisation

```c
void addindex(double *x, int n) {
  for (int i = 0; i < n; i++)
     x[i] = x[i] + i;
}
```

Avec du code SIMD:

```c
#include <immintrin.h>
void addindex(double *x, int n) { // n multiple de 4 et x aligné sur 32 octets
  __m256d ind, x_vec;
  for (int i = 0; i < n; i+=4) {
    x_vec = _mm256_load_pd(x+i);                 // chargement des données de x+i (4 double)
    ind   = _mm256_set_pd(i+3, i+2, i+1, i);     // creation du vecteur d'indices
    x_vec = _mm256_add_pd(x_vec, ind);           // addition des deux vecteurs
    _mm256_store_pd(x+i, x_vec);                 // stockage du résultat dans x+i
  }
}
```

Exemple de vectorisation

```c
void addindex(double *x, int n) {
  for (int i = 0; i < n; i++)
     x[i] = x[i] + i;
}
```

Avec du code SIMD:

```c
#include <immintrin.h>
void addindex(double *x, int n) { // n multiple de 4 et x aligné sur 32 octets
  __m256d ind, x_vec;
  for (int i = 0; i < n; i+=4) {
    x_vec = _mm256_load_pd(x+i);                 // chargement des données de x+i (4 double)
    ind   = _mm256_set_pd(i+3, i+2, i+1, i);     // creation du vecteur d'indices
    x_vec = _mm256_add_pd(x_vec, ind);           // addition des deux vecteurs
    _mm256_store_pd(x+i, x_vec);                 // stockage du résultat dans x+i
  }
}
```

$\Rightarrow$ l'utilisation de *_mm256_set_pd* coûte trop cher !!!

Exemple de vectorisation

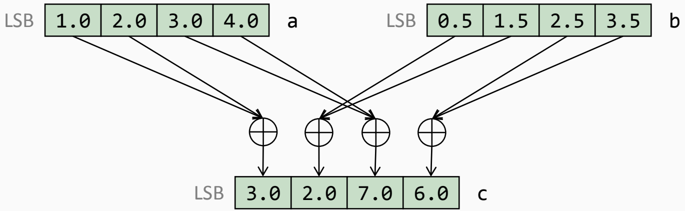
```c
void addindex(double *x, int n) {
  for (int i = 0; i < n; i++)
    x[i] = x[i] + i;
}
```

Avec du code SIMD, mais plus efficace:

```c
#include <immintrin.h>
void addindex(double *x, int n) { // n multiple de 4 et x aligné sur 32 octets
  __m256d ind, x_vec, incr;
  ind  = _mm256_set_pd(3, 2, 1, 0);    // creation du vecteur d'indices initial
  incr = _mm256_set1_pd(4);            // vecteur d'incrémentation d'indices

  for (int i = 0; i < n; i+=4) {
    x_vec = _mm256_load_pd(x+i);       // chargement des données de x+i (4 double)
    x_vec = _mm256_add_pd(x_vec, ind); // addition des deux vecteurs
    ind   = _mm256_add_pd(ind, incr);  // incrémentation du vecteur d'indices
    _mm256_store_pd(x+i, x_vec);       // stockage du résultat dans x+i
  }
}
```

$\Rightarrow$ *_mm256_set_pd* remplacé par une addition

Opération intra-registre




```
c = _mm256_hadd_pd(a, b);
```

similaire

```
c = _mm256_hsub_pd(a, b);
```

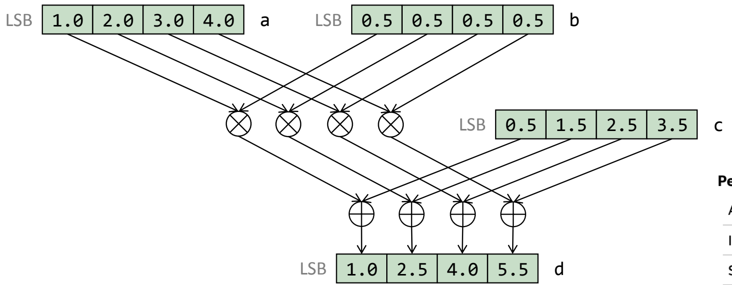
Performance

Architecture	Latency	Throughput (CPI)
Icelake	6	2
Skylake	7	2

Pas de croisement de données au-delà de 128-bits !!!

Opération de FMA

calcul de $d = a \times b + c$




```
d = _mm256_fmadd_pd(a, b, c);
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	4	0.5
Skylake	4	0.5
Knights Landing	6	0.5
Broadwell	5	0.5
Haswell	5	0.5

similaire:

```
d = _mm256_fmsub_pd(a, b, c);
```

FMA scalaire (registre 128-bits)

```
d = _mm_fmadd_sd(a, b, c);
```

Exemple avec FMA

Calcul de $y_k = a + x_k^2$ avec des nombres complexes

```
struct Complex { double Im, Re;};

void f(Complex a, Complex *x, Complex *y, size_t n){
  for (size_t i=0; i<n ; i++){
    y[i].Re = a.Re + x[i].Re * x[i].Re - x[i].Im * x[i].Im;
    y[i].Im = a.Im + 2.0* x[i].Re * x[i].Im;
  }
}
```

$\Rightarrow$ Comment introduire du SIMD dans ce code, avec du FMA ?

Exemple avec FMA

Calcul de $y_k = a + x_k^2$ avec des nombres complexes

```
struct Complex { double Im, Re;};

void f(Complex a, Complex *x, Complex *y, size_t n){
  for (size_t i=0; i<n ; i++){
    y[i].Re = a.Re + x[i].Re * x[i].Re - x[i].Im * x[i].Im;
    y[i].Im = a.Im + 2.0* x[i].Re * x[i].Im;
  }
}
```

⇒ Comment introduire du SIMD dans ce code, avec du FMA ?

pas possible, données non contigus: les parties réelles/imaginaires alternées en mémoire

Exemple avec FMA

Calcul de $y_k = a + x_k^2$ avec des nombres complexes

```
struct Complex { double Im, Re;};

void f(Complex a, Complex *x, Complex *y, size_t n){
  for (size_t i=0; i<n ; i++){
    y[i].Re = a.Re + x[i].Re * x[i].Re - x[i].Im * x[i].Im;
    y[i].Im = a.Im + 2.0* x[i].Re * x[i].Im;
  }
}
```

Solution: *Structure of Array* vs *Array of Structure*

```
struct ComplexTab { double *Im, *Re; size_t n;};

void f(Complex a, ComplexTab& x, ComplexTab& y){ // assume x and y of same size
  for (size_t i=0; i< x.n ; i++){
    y.Re[i] = a.Re + x.Re[i] * x.Re[i] - x.Im[i] * x.Im[i];
    y.Im[i] = a.Im + 2.0* x.Re[i] * x.Im[i];
  }
}
```

Exemple avec FMA

Calcul de $y_k = a + x_k^2$ avec des nombres complexes

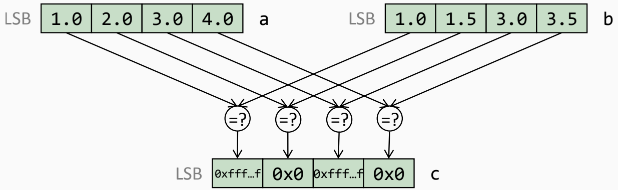
```
struct ComplexTab { double *Im, *Re; size_t n;};

// on déroule la boucle sur 4 niveau
void f(Complex a, ComplexTab& x, ComplexTab& y){ // assume x.n==y.n= 0 mod 4
  for (size_t i=0; i< x.n ; i+=4){
    y.Re[i]   = a.Re + x.Re[i]   * x.Re[i]   - x.Im[i]   * x.Im[i];
    y.Re[i+1] = a.Re + x.Re[i+1] * x.Re[i+1] - x.Im[i+1] * x.Im[i+1];
    y.Re[i+2] = a.Re + x.Re[i+2] * x.Re[i+2] - x.Im[i+2] * x.Im[i+2];
    y.Re[i+3] = a.Re + x.Re[i+3] * x.Re[i+3] - x.Im[i+3] * x.Im[i+3];
    y.Im[i]   = a.Im + 2.0* x.Re[i]   * x.Im[i];
    y.Im[i+1] = a.Im + 2.0* x.Re[i+1] * x.Im[i+1];
    y.Im[i+2] = a.Im + 2.0* x.Re[i+2] * x.Im[i+2];
    y.Im[i+3] = a.Im + 2.0* x.Re[i+3] * x.Im[i+3];
  }
}
```

Exemple avec FMA

Calcul de $y_k = a + x_k^2$ avec des nombres complexes (FMA+AVX)

```
#include <immintrin.h>

// on déroule la boucle sur 4 niveau
void f(Complex a, ComplexTab& x, ComplexTab& y){ // assume x.n==y.n== 0 mod 4
  __m256d x_re,x_im,y_re,y_im,a_re,a_im,v2;
  a_re= _mm256_set1_pd(a.Re);
  a_im= _mm256_set1_pd(a.Im);
  v2= _mm256_set1_pd(2.0);
  for (size_t i=0; i< x.n ; i+=4){
    x_re=_mm256_load_pd(x.Re+i);
    x_im=_mm256_load_pd(x.Im+i);

    y_re =  _mm256_fmadd_pd(x_re, x_re, a_re);
    y_re = _mm256_fnmadd_pd(x_im, x_im, y_re);
    y_im = _mm256_mul_pd(v2, x_re);
    y_im = _mm256_fmadd_pd(y_im, x_im, a_im);

    _mm256_store_pd(y.Re+i,y_re);
    _mm256_store_pd(y.Im+i,y_im);
}}
```

Opération de comparaison




```
c = _mm256_cmp_pd(a, b, _CMP_EQ_OQ);
```

Each field:

`0xffff…f` if true

`0x0` if false

Return type: __m256d

similaire:

```
c = _mm256_cmp_pd(a, b, _CMP_GE_OQ);
```

```
c = _mm256_cmp_pd(a, b, _CMP_LT_OQ);
```

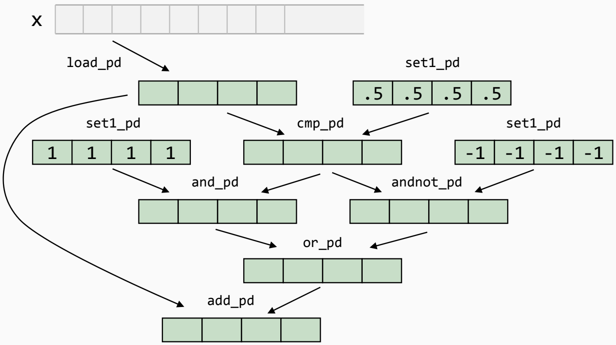
etc.

Performance

Architecture	Latency	Throughput (CPI)
Icelake	4	0.5
Skylake	4	0.5
Broadwell	3	1
Haswell	3	1
Ivy Bridge	3	1

Example

```c
void fcond(double *x, size_t n) {
  int i;
  for(i = 0; i < n; i++) {
    if(x[i] > 0.5) x[i] += 1.;
    else x[i] -= 1.;
}}
```



Example

```
void fcond(double *x, size_t n) {
  int i;
  for(i = 0; i < n; i++) {
    if(x[i] > 0.5) x[i] += 1.;
    else x[i] -= 1.;
}}
```

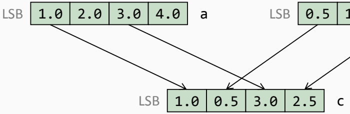
```
#include <xmmintrin.h>
void fcond_vec1(double *x, size_t n) {
  int i;
  __m256d vt, vmask, vp, vm, vr, ones, mones, thresholds;
  ones  = _mm256_set1_pd(1.);
  mones = _mm256_set1_pd(-1.);
  thresholds = _mm256_set1_pd(0.5);
  for(i = 0; i<n; i+=4) {
    vt    = _mm256_load_pd(x+i);
    vmask = _mm256_cmp_pd(vt, thresholds, _CMP_GT_OQ);
    vp    = _mm256_and_pd(vmask, ones);
    vm    = _mm256_andnot_pd(vmask, mones);
    vr    = _mm256_add_pd(vt, _mm256_or_pd(vp, vm));
    _mm256_store_pd(x+i, vr);
  }}
```

Déplacement de données dans les registres vectoriels

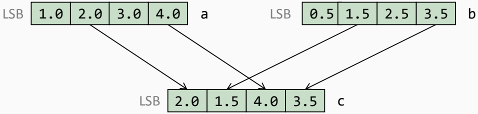
Généralement, on génére un régistre en sélectionnant des données de 2 registres vectoriels

- *unpack*: sélection par position dans les voies 128-bits
- *blendv*: sélection entre registre par masque de bit
- *shuffle* : sélection intra registre par masque de bit
- *permute*: permutation de données intra registre

Sélection de données par position




```
c = _mm256_unpacklo_pd(a, b);
```




```
c = _mm256_unpackhi_pd(a, b);
```

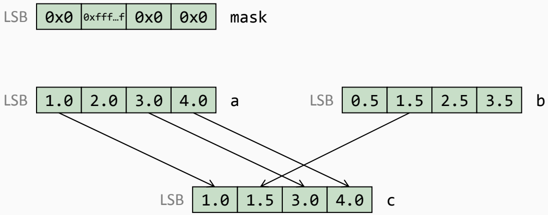
Performance

Architecture	Latency	Throughput (CPI)
Icelake	1	1
Skylake	1	1
Broadwell	1	1
Haswell	1	1
Ivy Bridge	1	1

$\Rightarrow$ Pas de croisement de données entre les lignes 128-bits !!!

Sélection de données par masquage (entre registre)

On sélectionne des données de 2 registres à partir d'un masque de bits




```
__m256d _mm256_blendv_pd(__m256d a, __m256d b, __m256d mask)
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	-	1
Skylake	2	0.66
Broadwell	2	2
Haswell	2	2
Ivy Bridge	1	1

$\Rightarrow$ Pas de croisement de données entre les lignes 128-bits !!!

Sélection de données par masquage (intra registre)

On entrecroise des données de 2 registres en ne sélectionnant qu'une donnée sur deux des registres

LSB					
LSB	1.0	2.0	3.0	4.0	a

LSB	0.5	1.5	2.5	3.5	b

LSB	c0	c1	c2	c3	c

$\underbrace{\texttt{c0}}_{a0 \text{ or } a1}$

```
c0 = mask.bit0 ? a1 : a0
c1 = mask.bit1 ? b1 : b0
c2 = mask.bit2 ? a3 : a2
c3 = mask.bit3 ? b3 : b2
```

Performance

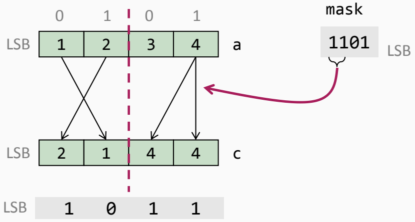
Architecture	Latency	Throughput (CPI)
Icelake	1	0.5
Skylake	1	1
Broadwell	1	1
Haswell	1	1
Ivy Bridge	1	1

```
__m256d _mm256_shuffle_pd(__m256d a, __m256d b, const int mask)
```

$\Rightarrow$ Pas de croisement de données entre les lignes 128-bits !!!

Permutation de données intra registre

On permute/réplique des données à l'intérieur d'un même registre (sans déplacer hors 128-bits)



Performance

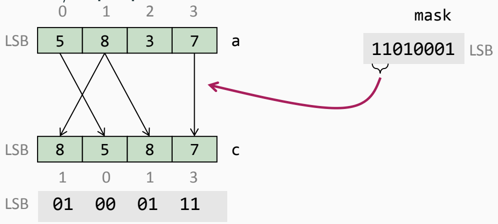
Architecture	Latency	Throughput (CPI)
Icelake	1	-
Skylake	1	1
Broadwell	1	1
Haswell	1	1
Ivy Bridge	1	1

```
__m256d _mm256_permute_pd(__m256d a, int mask)
```

⇒ Pas de croisement de données entre les lignes 128-bits !!!

Permutation de données intra registre

On permute/réplique des données à l'intérieur d'un même registre (en déplacant hors 128-bits)



Performance

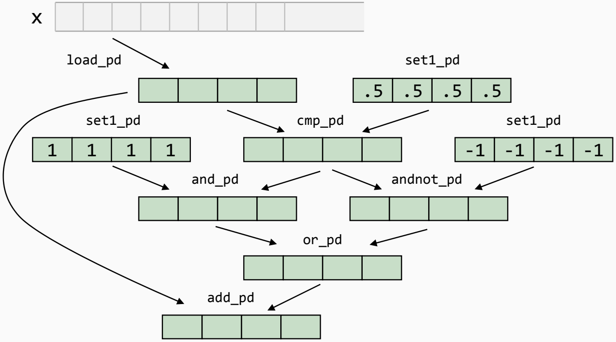
Architecture	Latency	Throughput (CPI)
Icelake	3	1
Skylake	3	1
Broadwell	3	1
Haswell	3	1

```
__m256d _mm256_permute4x64_pd(__m256d a, int mask)
```

$\Rightarrow$ ATTENTION: croisement de données entre les lignes 128-bits, un peu plus lent !!!!

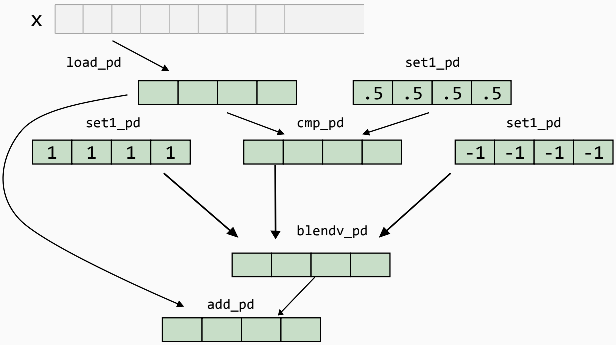
Exercice: Comment améliorer ce code en SIMD?

```c
void fcond(double *x, size_t n) {
  int i;
  for(i = 0; i < n; i++) {
    if(x[i] > 0.5) x[i] += 1.;
    else x[i] -= 1.;
}}
```

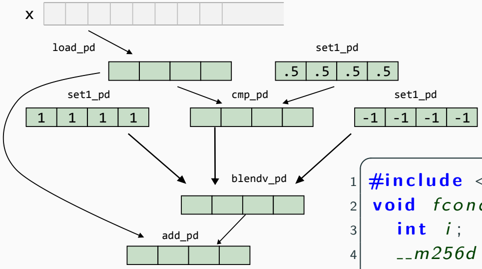


Exercice: Comment améliorer ce code en SIMD?

```
void fcond(double *x, size_t n) {
  int i;
  for(i = 0; i < n; i++) {
    if(x[i] > 0.5) x[i] += 1.;
    else x[i] -= 1.;
}}
```



Exercice: Comment améliorer ce code en SIMD?

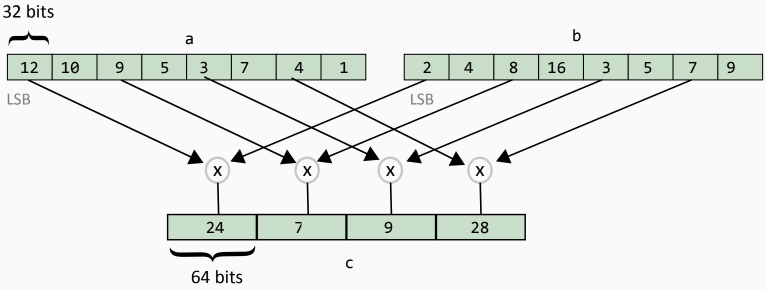



```c
#include <xmmintrin.h>
void fcond_vec2(double *x, size_t n) {
  int i;
  __m256d vt, vmask, vm, vr, ones, mones, thresholds;
  ones  = _mm256_set1_pd(1.);
  mones = _mm256_set1_pd(-1.);
  thresholds = _mm256_set1_pd(0.5);
  for(i = 0; i<n; i+=4) {
    vt    = _mm256_load_pd(x+i);
    vmask = _mm256_cmp_pd(vt, thresholds, _CMP_GT_OQ);
    vm    = _mm256_blendv_pd(ones, mones, vmask)
    vr    = _mm256_add_pd(vt, vm);
    _mm256_store_pd(x+i, vr);
  }}
```

⇒ speed-up entre 4 et 10

SIMD sur les entiers en un slide

- registre: *_m128i* ou *_m256i* ou *_m512i*
- le nom des fonctions encodent: la taille et le type des données
  - *_mm256_add_epi32* $\rightarrow$ add sur 8 *int32_t*
  - *_mm256_add_epu16* $\rightarrow$ add sur 16 *uint32_t*
- cas particulier multiplication $\Rightarrow$ 1 produit sur 2 ou résultat partiel

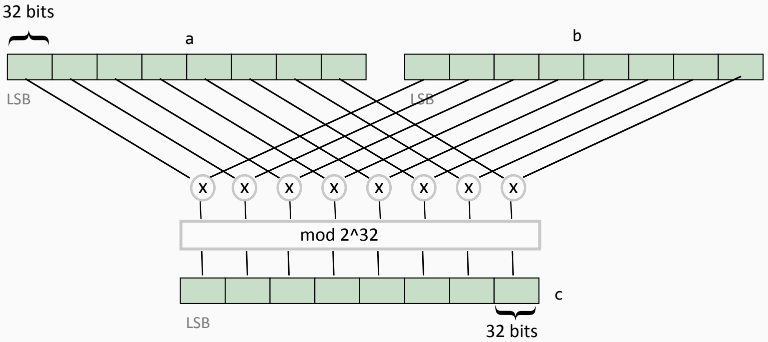


`c= _mm256_mul_epi32(a,b);`

$\Rightarrow$ (permute + mul pour les autres !!!)

SIMD sur les entiers en un slide

- registre: *_m128i* ou *_m256i* ou *_m512i*
- le nom des fonctions encodent: la taille et le type des données
  - *_mm256_add_epi32* → add sur 8 *int32_t*
  - *_mm256_add_epu16* → add sur 16 *uint32_t*
- cas particulier multiplication ⇒ 1 produit sur 2 ou résultat partiel



c= **_mm256_mullo_epi32** (a,b);

Attention: pas de version hi en 32 bits !!!

Bilan sur l'utilisation des instructions SIMD

- utilisation de *load/store* qui respectent l'alignement des registres SIMD
  $\hookrightarrow$ 32 octets en AVX2
- minimiser les opérations de déplacement de données entre registres
  $\hookrightarrow$ surtout celles qui dépassent les voies 128-bits
- minimiser les opérations arithmétiques lentes
  $\hookrightarrow$ ex. *hadd* et *div*
- consulter les informations sur les instructions existantes et leurs performances
  $\hookrightarrow$ Intel's intrinsic guide
- sur les entiers de nombreuses fonctions mais pas toujours complètes
  $\hookrightarrow$ ex. mul pas sur 64 bits et pas de FMA du tout !!!

Accès aux données: cache et compléxité spatiale

Principe de localité

Localité temporelle

si un zones mémoire d'adresse X est accédée par le programme, alors un nouvel accès mémoire d'adresse X interviendra très probablement.

$\Rightarrow$ une même données est référencée plusieurs fois dans un laps de temps assez court

Localité spatiale

si un zones mémoire d'adresse X est accédée par le programme, alors un nouvel accès mémoire proche de X interviendra très probablement.

$\Rightarrow$ 2 données proches en mémoire sont référencées dans un laps de temps assez court

Principe de localité: exemple

```c
int sum=0;
for(size_t i=0;i<n;i++)
sum+=a[i];
```

- `sum` : localité temporelle (réutiliser à chaque intruction de la boucle)
- `a[i]` : localité spatiale (incrément de 1 en mémoire)

Principe de localité: exercice

```
int sumarray_row(int A[M][N]){
  int res=0;
  for(size_t i=0;i<M; i++)
     for(size_t j=0;j<N; j++)
        res+=A[i][j];
  return res;
}
```

Bonne localité ? quel type ?

Principe de localité: exercice

```
int sumarray_row(int A[M][N]){
  int res=0;
  for(size_t i=0;i<M; i++)
     for(size_t j=0;j<N; j++)
        res+=A[i][j];
  return res;
}
```

Bonne localité ? quel type ?

- `res` : localité temporelle OK
- `a[i][j]` : localité spatiale OK (toujours un incrément de 1 en mémoire)

Principe de localité: exercice

```c
int sumarray_col(int A[M][N]){
  int res=0;
  for(size_t j=0;j<N; j++)
     for(size_t i=0;i<M; i++)
        res+=A[i][j];
  return res;
}
```

Bonne localité ? quel type ?

Principe de localité: exercice

```
int sumarray_col(int A[M][N]){
  int res=0;
  for(size_t j=0;j<N; j++)
     for(size_t i=0;i<M; i++)
        res+=A[i][j];
  return res;
}
```

Bonne localité ? quel type ?

- `res` : localité temporelle — OK
- `a[i][j]` : localité spatiale — KO (distance de O(N) en mémoire)

Principe de localité: exercice

```c
int sumarray3D(int A[M][N][N]){
  int res=0;
  for(size_t i=0;i<N; i++)
     for(size_t j=0;j<M; j++)
        for(size_t k=0;k<N; k++)
           res+=A[j][i][k];
  return res;
}
```

⇒ Comment obtenir une bonne localité spatiale ? (accés mémoire à distance 1)

Principe de localité: exercice

```
int sumarray3D(int A[M][N][N]){
  int res=0;
  for(size_t i=0;i<N; i++)
     for(size_t j=0;j<M; j++)
        for(size_t k=0;k<N; k++)
           res+=A[j][i][k];
  return res;
}
```

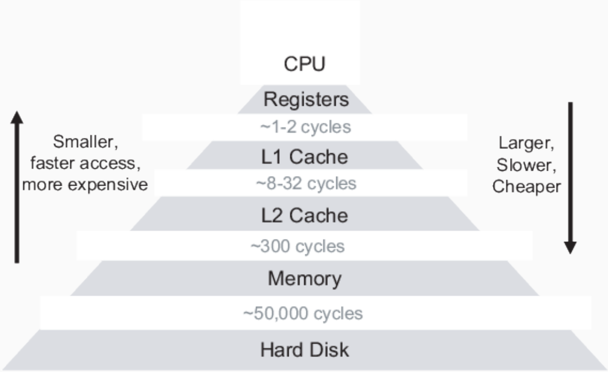
⇒ Comment obtenir une bonne localité spatiale ? (accés mémoire à distance 1)

```
  for(size_t j=0;j<M; j++)
     for(size_t i=0;i<N; i++)
        for(size_t k=0;k<N; k++)
           res+=A[j][i][k]; // ordonne boucles dans le sens des accès les plus profonds
```

Pouquoi se soucier de la localité

- les architectures exhibent des mémoires à différent débit: *RAM vs caches vs registres*
- les données sont déplacées par paquet entre chaque mémoire
  $\hookrightarrow$ besoin d'en tirer partie dans les programmes

Hierarchie mémoire des processeurs



plus une donnée est loin dans la hierachie, plus son accès est couteux !!!

prog. efficace $\Rightarrow$ minimisation des accès mémoire couteux

Rapport entre arithmétique et mémoire

Définition:

Soit $I(n)$ l'intensité opérationnelle d'un programme `P` ayant des entrées de tailles $O(n)$, alors

$$I(n) = \frac{W(n)}{Q(n)}$$

- $W(n)$ = nombre d'opérations arithmétique du programme (e.g. #flop)
- $Q(n)$ = nombre d'octets transférés entre la mémoire et les caches[2]

[2] on considère que les caches sont froids (vide) au début du programme

Rapport entre arithmétique et mémoire: exemple 1

```c
/* x, y are vectors of doubles of length n, alpha is a double */
for (i = 0; i < n; i++)
    x[i] = x[i] + alpha*y[i];
```

Rapport entre arithmétique et mémoire: exemple 1

```c
/* x, y are vectors of doubles of length n, alpha is a double */
for (i = 0; i < n; i++)
    x[i] = x[i] + alpha*y[i];
```

- flops: $W(n) = 2n$

Rapport entre arithmétique et mémoire: exemple 1

```c
/* x, y are vectors of doubles of length n, alpha is a double */
for (i = 0; i < n; i++)
    x[i] = x[i] + alpha*y[i];
```

- flops: $W(n) = 2n$
- accès mémoire $\geq 2n$ (lecture des données à minima)

Rapport entre arithmétique et mémoire: exemple 1

```
/* x, y are vectors of doubles of length n, alpha is a double */
for (i = 0; i < n; i++)
    x[i] = x[i] + alpha*y[i];
```

- flops: $W(n) = 2n$
- accès mémoire $\geq 2n$ (lecture des données à minima)
- nbr octets chargés $Q(n) \geq 2n * 8 = 16n$

Rapport entre arithmétique et mémoire: exemple 1

```c
/* x, y are vectors of doubles of length n, alpha is a double */
for (i = 0; i < n; i++)
   x[i] = x[i] + alpha*y[i];
```

- flops: $W(n) = 2n$
- accès mémoire $\geq 2n$ (lecture des données à minima)
- nbr octets chargés $Q(n) \geq 2n * 8 = 16n$

$$I(n) = \frac{W(n)}{Q(n)} \leq \frac{1}{8}$$

Rapport entre arithmétique et mémoire: exemple 2

```
/* matrix multiplication; A, B, C are n x n matrices of doubles */
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            C[i*n+j] += A[i*n+k]*B[k*n+j];
```

Rapport entre arithmétique et mémoire: exemple 2

```
/* matrix multiplication; A, B, C are n x n matrices of doubles */
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i*n+j] += A[i*n+k]*B[k*n+j];
```

- flops: $W(n) = 2n^3$

Rapport entre arithmétique et mémoire: exemple 2

```c
/* matrix multiplication; A, B, C are n x n matrices of doubles */
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            C[i*n+j] += A[i*n+k]*B[k*n+j];
```

- flops: $W(n) = 2n^3$
- accès mémoire $\geq 3n^2$ (lecture des données à minima)

Rapport entre arithmétique et mémoire: exemple 2

```
/* matrix multiplication; A, B, C are n x n matrices of doubles */
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
         C[i*n+j] += A[i*n+k]*B[k*n+j];
```

- flops: $W(n) = 2n^3$
- accès mémoire $\geq 3n^2$ (lecture des données à minima)
- nbr octets chargés $Q(n) \geq 3n^2 * 8 = 24n^2$

Rapport entre arithmétique et mémoire: exemple 2

```
/* matrix multiplication; A, B, C are n x n matrices of doubles */
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
         C[i*n+j] += A[i*n+k]*B[k*n+j];
```

- flops: $W(n) = 2n^3$
- accès mémoire $\geq 3n^2$ (lecture des données à minima)
- nbr octets chargés $Q(n) \geq 3n^2 * 8 = 24n^2$

$$I(n) = \frac{W(n)}{Q(n)} \leq \frac{n}{12}$$

Rapport entre arithmétique et mémoire

Un programme est caractérisé de :

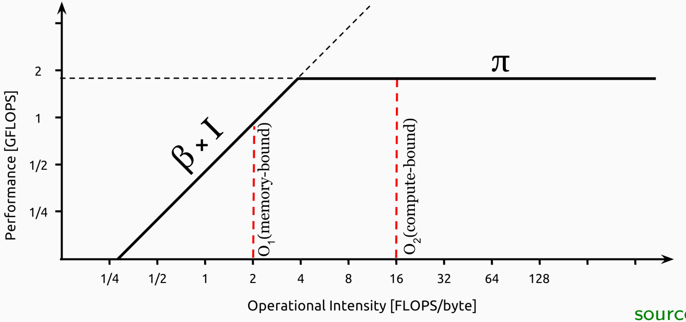
- *memory bound*: quand $I(n)$ est petit e.g. O(1)
- *compute bound*: quand $I(n)$ est grand e.g. O(f(n))

Impact sur les performances:

- *memory bound* $\Rightarrow$ limité par le débit mémoire du CPU en octet/s (*Byte/s*)
- *compute bound* $\Rightarrow$ limité par le peak arithmétique du CPU en GFlop/s

Roofline model

Modèle théorique permettant de caractériser *a priori* les performances d'un code.



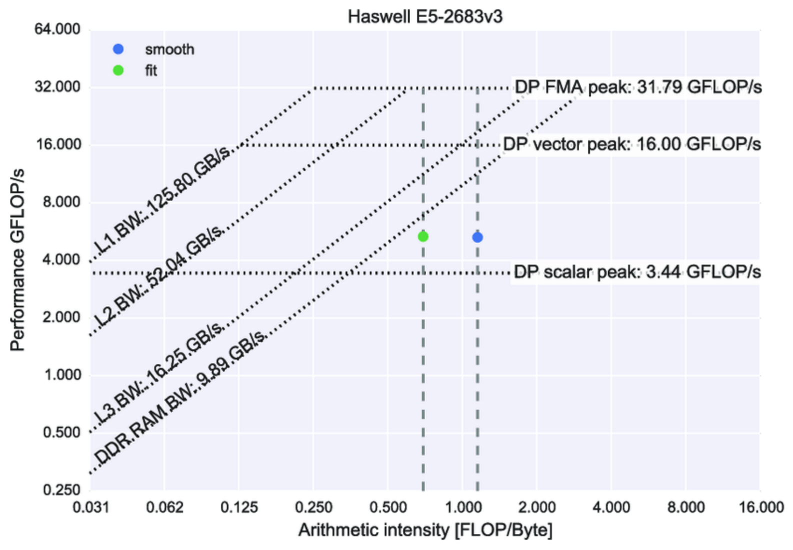
source image: wikipédia

- I= intensité opérationnelle
- β = débit en GByte/s de la mémoire
- π = peak performance du processeur en GFlops/s

$\Rightarrow$ performance $= \min(\beta \times \mathrm{I}, \pi)$

Roofline model (en vrai)

Daniel Hugo and Cámpora Pérez 2017 J. Phys.: Conf. Ser. 898 032052



Cache processeur

Définition (Wikipédia)

Un cache de processeur est une mémoire plus petite et plus rapide, située au plus près d'une unité centrale de traitement qui stocke des copies des données à partir d'emplacements de la mémoire principale qui sont fréquemment utilisés avant leurs transmissions aux registres du processeur.

Il exploite :

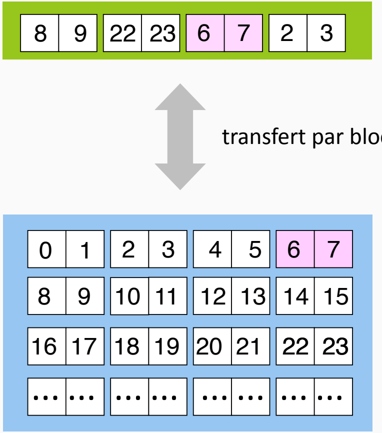
- localité temporelle par nature
- localité spatiale par conception: copie des données en bloc (ex. 64 octets sur Intel core)

Concept général d'un cache

Cache

petit, rapide

contient un sous-ensemble des blocs mémoire



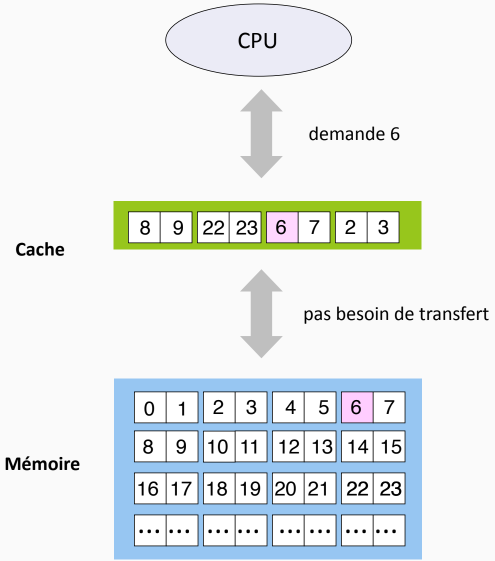
transfert par bloc

Mémoire

large, lent

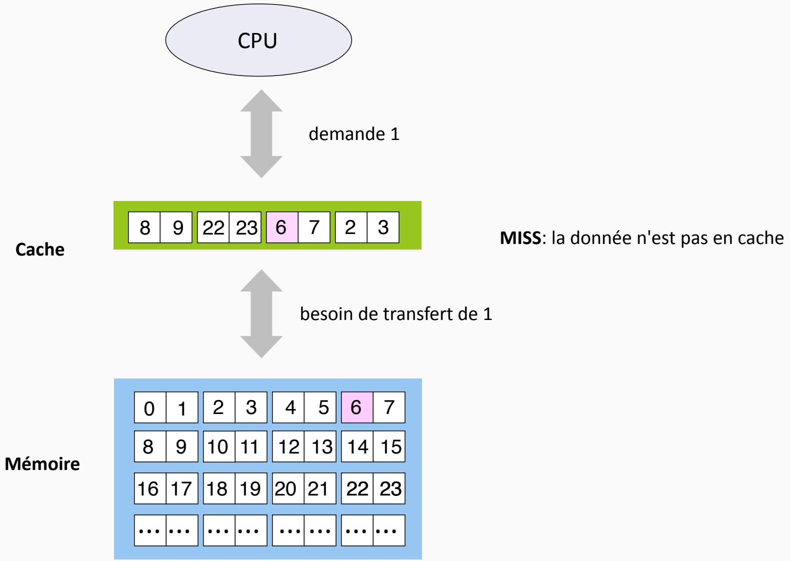
contient les données par bloc

Concept général d'un cache: *hit*

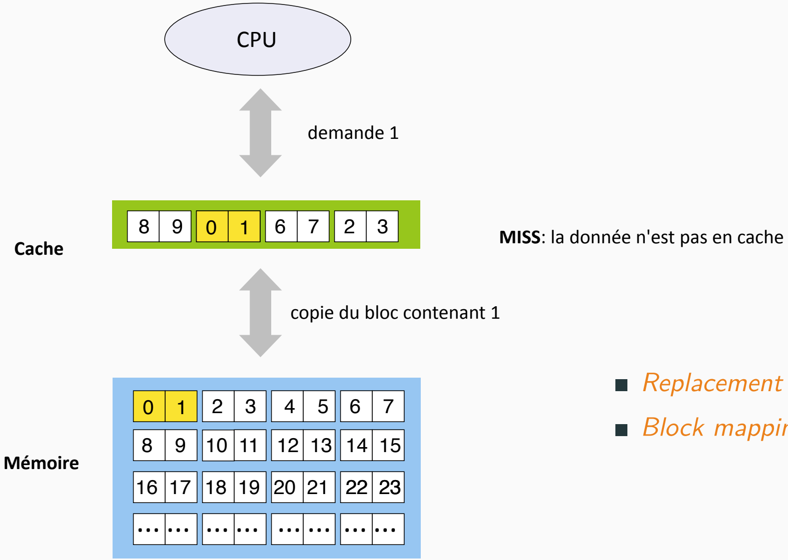


HIT: la donnée est en cache

Concept général d'un cache: *miss*



Concept général d'un cache: *miss*



- *Replacement policy*: quel bloc on remplace
- *Block mapping*: où on place le bloc

Différent défaut de cache (*cache miss*)

cache hit

$\Rightarrow$ accès à une donnée présente dans le cache (ex. L1, L2 ou L3)

cache miss (défauts de cache)

$\Rightarrow$ accès à une donnée non-présente dans le cache (ex. L1, L2 ou L3)

Différent défaut de cache (*cache miss*)

cache hit

$\Rightarrow$ accès à une donnée présente dans le cache (ex. L1, L2 ou L3)

cache miss (défauts de cache)

$\Rightarrow$ accès à une donnée non-présente dans le cache (ex. L1, L2 ou L3)

classification des défauts de cache:

- obligatoire (*cold miss*)
  $\hookrightarrow$ 1er chargement des blocs de données
- capacité (*capacity miss*)
  $\hookrightarrow$ jeu de données supérieur à la taille du cache
- conflit (*conflict miss*)
  $\hookrightarrow$ pas de problème de taille mais de placement des blocs au même endroit

Fonctionnement d'un cache

Différentes structures possibles pour le placement des blocs dans le cache

- adressage direct (*direct mapped cache*)
  $\hookrightarrow$ 1 seule possibilité de placement

Fonctionnement d'un cache

Différentes structures possibles pour le placement des blocs dans le cache

- adressage direct (*direct mapped cache*)
  $\hookrightarrow$ 1 seule possibilité de placement
- adressage associatif à E-voie(*E-way set-associative cache*)
  $\hookrightarrow$ E possibilités de placement

Fonctionnement d'un cache

Différentes structures possibles pour le placement des blocs dans le cache

- adressage direct (*direct mapped cache*)
  $\hookrightarrow$ 1 seule possibilité de placement
- adressage associatif à E-voie(*E-way set-associative cache*)
  $\hookrightarrow$ E possibilités de placement
- adressage associatif complet (*fully associative cache*)
  $\hookrightarrow$ toutes les possibilités de placement

Fonctionnement d'un cache

Différentes structures possibles pour le placement des blocs dans le cache

- adressage direct (*direct mapped cache*)
  $\hookrightarrow$ 1 seule possibilité de placement
- adressage associatif à E-voie(*E-way set-associative cache*)
  $\hookrightarrow$ E possibilités de placement
- adressage associatif complet (*fully associative cache*)
  $\hookrightarrow$ toutes les possibilités de placement

$\Rightarrow$ via le codage binaire des adresses des données

Direct mapped cache

direct mapped cache

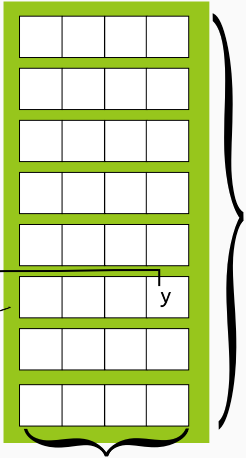
(E=1, B= 4 double, S=8)

adresse d'un double (64 bits)

tag	set idx	block offset
56 bits	3 bits	5 bits

LSB

y at : 0x0...0C8 = (0....0 101 11000)



S = 8 = nbr bloc

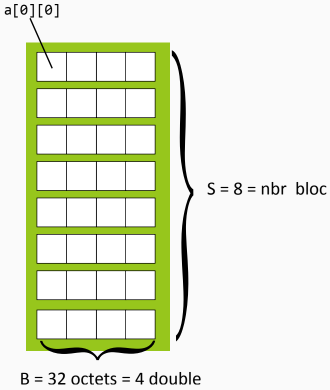
B = 32 octets = 4 double

B = taille d'un bloc mémoire

Direct mapped cache: exemple 1

On considère (E=1, S=8, B= 4 `double`), des caches froids et la place de `a[0][0]`.

```c
double sumarray_row(double a[8][8]){
  double res=0;
  for(size_t i=0;i<8;i++)
     for(size_t j=0;j<8;j++)
        res+=a[i][j];
  return res;
}
```

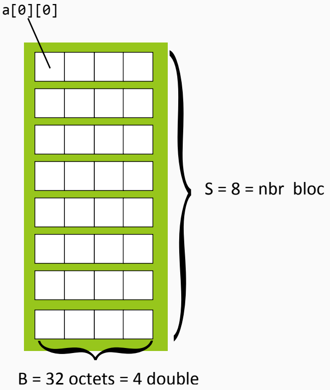


⇒ Comment se remplit le cache ? combien de cache miss ?

Direct mapped cache: exemple 2

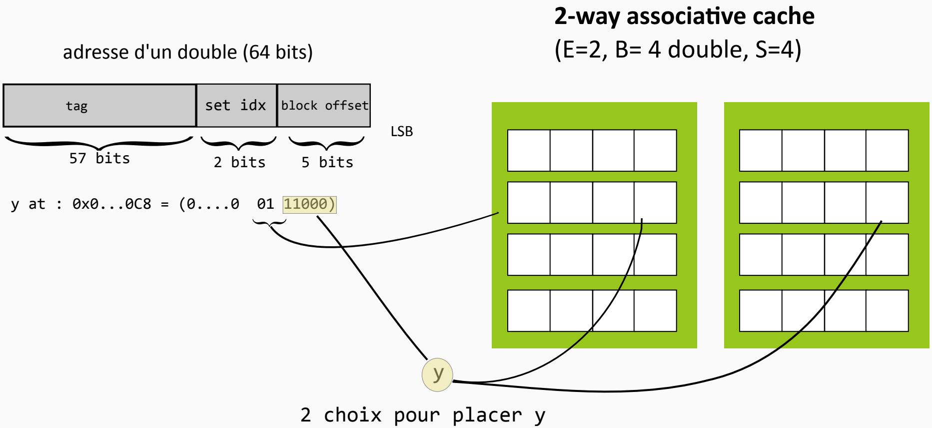
On considère (E=1, S=8, B= 4 `double`), des caches froids et la place de `a[0][0]`.

```c
double  sumarray_col(double a[8][8]){
  double res=0;
  for(size_t j=0;j<8;j++)
     for(size_t i=0;i<8;i++)
        res+=a[i][j];
  return res;
}
```



$\Rightarrow$ Comment se remplit le cache ? combien de cache miss ?

Cache associatif



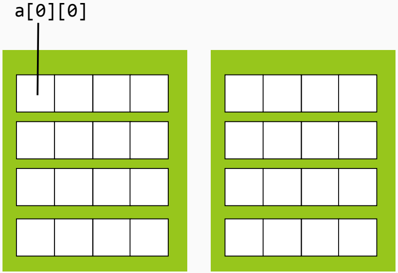
Différentes politiques de choix/remplacement:

- *Least Recently Used (LRU),*
- *Least Frequently Used (LFU),*
- *Random*

Associative cache: exemple

On considère (E=2, S=4, B= 4 `double`), des caches froids et la place de `a[0][0]`.

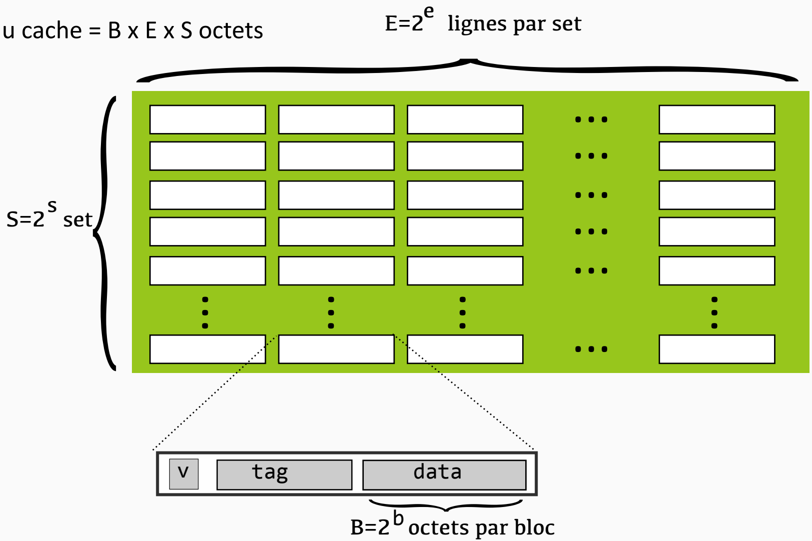
```c
double  sumarray_col(double a[8][8]){
  double res=0;
  for(size_t j=0;j<8;j++)
     for(size_t i=0;i<8;i++)
        res+=a[i][j];
  return res;
}


double  sumarray_row(double a[8][8]){
  double res=0;
  for(size_t i=0;i<8;i++)
     for(size_t j=0;j<8;j++)
        res+=a[i][j];
  return res;
}
```



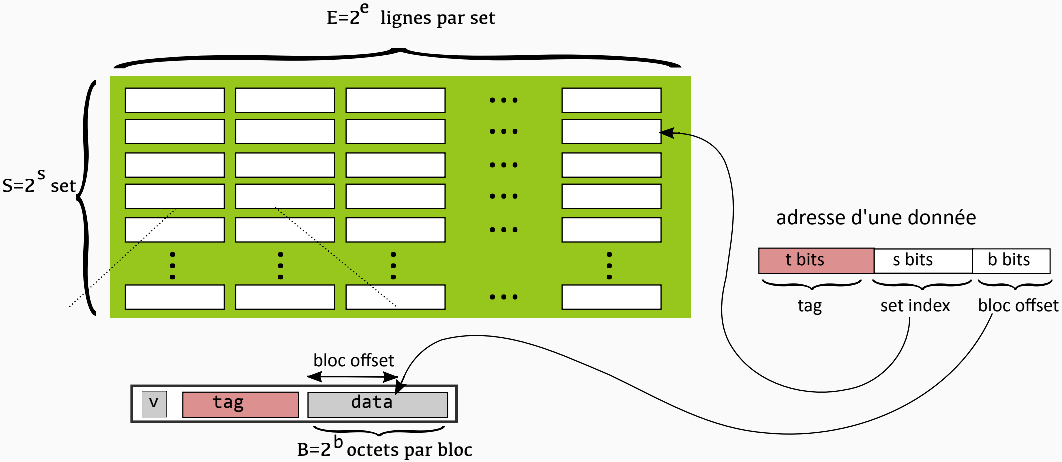
⇒ Comment se remplit le cache ? combien de cache miss ? avec politique *LRU*

Design general d'un cache

taille du cache = B x E x S octets



Lecture en cache



lecture d'une donnée en cache:

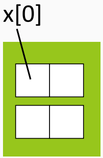
$\Rightarrow$ trouver set; trouver tag dans le set; si valid lire donnée à l'offset sinon aller niveau supérieur

Lecture en cache: exemple

double lecture du tableau

`x= x[0],x[1], ..., x[7]`

```
for(j=0;j<2;j++)
    for (i=0;i<8;i++)
        cout<<x[i];
```



cache

- E=1
- S=2
- B=16 (2 `double`)

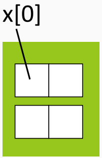
- pattern d'accès à x : 0123456701234567
- Hit/Miss en cache :

Lecture en cache: exemple

double lecture du tableau

`x= x[0],x[1], ..., x[7]`

```
for(j=0;j<2;j++)
   for (i=0;i<8;i++)
      cout<<x[i];
```



cache

- E=1
- S=2
- B=16 (2 `double`)

- pattern d'accès à x : 0123456701234567
- Hit/Miss en cache : `MHMHMHMHMHMHMHMH`

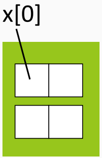
Résultats: 8 miss et 8 hit $\Rightarrow$ **localité spatiale**: OUI, **localité temporelle**: NON

Lecture en cache: exemple

double lecture du tableau

`x= x[0],x[1], ..., x[7]`

```
for(j=0;j<2;j++)
    for (i=0;i<8;i++)
        cout<<x[i];
```



cache

- E=1
- S=2
- B=16 (2 `double`)

- pattern d'accès à x : 0123456701234567
- Hit/Miss en cache : MHMHMHMHMHMHMHMH

Résultats: 8 miss et 8 hit ⇒ **localité spatiale**: OUI, **localité temporelle**: NON

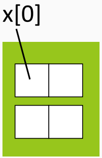
Peut-on faire mieux comme pattern d'accès ?

Lecture en cache: exemple

double lecture du tableau

`x= x[0],x[1], ..., x[7]`

```
for(j=0;j<2;j++)
   for (i=0;i<8;i++)
      cout<<x[i];
```



cache

- E=1
- S=2
- B=16 (2 `double`)

- pattern d'accès à x : 0123456701234567
- Hit/Miss en cache : `MHMHMHMHMHMHMHMH`

Résultats: 8 miss et 8 hit ⇒ **localité spatiale**: OUI, **localité temporelle**: NON

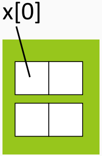
Peut-on faire mieux comme pattern d'accès ? oui !!!

Lecture en cache: exemple

double lecture du tableau

`x= x[0],x[1], ..., x[7]`

```
for(j=0;j<2;j++)
    for (i=0;i<8;i++)
        cout<<x[i];
```



cache

- E=1
- S=2
- B=16 (2 `double`)

- pattern d'accès à x : 0123456701234567
- Hit/Miss en cache : `MHMHMHMHMHMHMHMH`

Résultats: 8 miss et 8 hit $\Rightarrow$ **localité spatiale**: OUI, **localité temporelle**: NON

Peut-on faire mieux comme pattern d'accès ? oui !!!

0123012345674567 $\Rightarrow$ `MHMHHHHHMHMHHHHH`

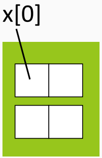
Résultats: 4 miss et 12 hit $\Rightarrow$ **localité spatiale**: OUI, **localité temporelle**: OUI

Lecture en cache: exemple

double lecture du tableau

`x= x[0],x[1], ..., x[7]`

```
for(j=0;j<2;j++)
   for (i=0;i<8;i++)
      cout<<x[i];
```



cache

- E=1
- S=2
- B=16 (2 `double`)

- pattern d'accès à x : 0123456701234567
- Hit/Miss en cache : `MHMHMHMHMHMHMHMH`

Résultats: 8 miss et 8 hit $\Rightarrow$ **localité spatiale**: OUI, **localité temporelle**: NON

Peut-on faire mieux comme pattern d'accès ? oui !!!

0123012345674567 $\Rightarrow$ `MHMHHHHHMHMHHHHH`

Résultats: 4 miss et 12 hit $\Rightarrow$ **localité spatiale**: OUI, **localité temporelle**: OUI

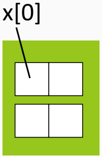
Pourquoi résultat optimal ?

Lecture en cache: exemple

double lecture du tableau

`x= x[0],x[1], ..., x[7]`

```
for(j=0;j<2;j++)
   for (i=0;i<8;i++)
      cout<<x[i];
```



cache

- E=1
- S=2
- B=16 (2 `double`)

- pattern d'accès à x : 0123456701234567
- Hit/Miss en cache : `MHMHMHMHMHMHMHMH`

Résultats: 8 miss et 8 hit $\Rightarrow$ **localité spatiale**: OUI, **localité temporelle**: NON

Peut-on faire mieux comme pattern d'accès ? oui !!!

0123012345674567 $\Rightarrow$ `MHMHHHHHMHMHHHHH`

Résultats: 4 miss et 12 hit $\Rightarrow$ **localité spatiale**: OUI, **localité temporelle**: OUI

Pourquoi résultat optimal ? 8 `double` = 4 miss $\times$ 2 `double` (seulement des cold miss)

Écriture en cache

Gestion plus compliquée que la lecture: cohérence des données

Quoi faire après un *write hit*:

- write-through: écriture directe dans la mémoire
- write-back: délègue l'écriture à l'éviction du bloc en cache

Quoi faire après un *write miss*:

- write-allocate: charge la donnée en cache et simule un *write hit*
- no write-allocate: écriture directe dans la mémoire

Mesure de performance avec les caches

- **miss rate**: taux d'échec des accès en cache

  ⇒ #cache miss / #accès mémoire
- **hit rate**: taux de succès des accès en cache

  ⇒ 1 - **miss rate**
- **hit time**: temps de transfert d'une donné du cache au CPU

  ⇒ quelques cycles pour le cache L1
- **miss penalty**: temps additionnel à cause d'un miss

  ⇒ L1 miss: 10 cycles; L2 miss: 50 cycles; L3 miss: 200 cycles

Et dans la vrai vie les caches c’est comment ?

Core Cache Size/ Latency/ Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	**64 Bytes/ cycle**
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	**32 Bytes/ cycle**
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	**64 Bytes/ cycle**
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	**4K+2M shared: 1024, 8-way**

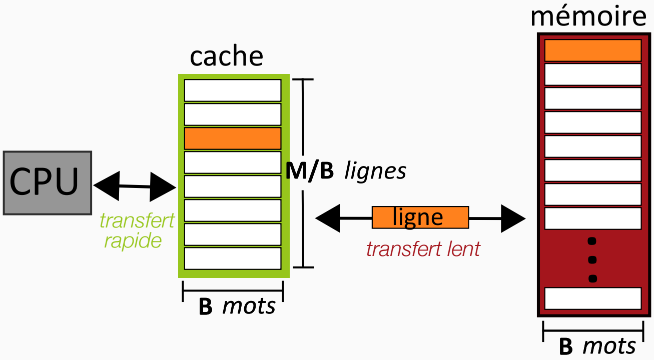
All caches use 64-byte lines

15 Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Sandy Bridge); Intel® Microarchitecture (Nehalem)

[2] source: nextimpact.com

Approche théorique: *Ideal Cache model*

- seulement 2 niveaux de mémoire: cache (rapide), mémoire (lent)
- transfert de données par ligne (bloc) de B mots

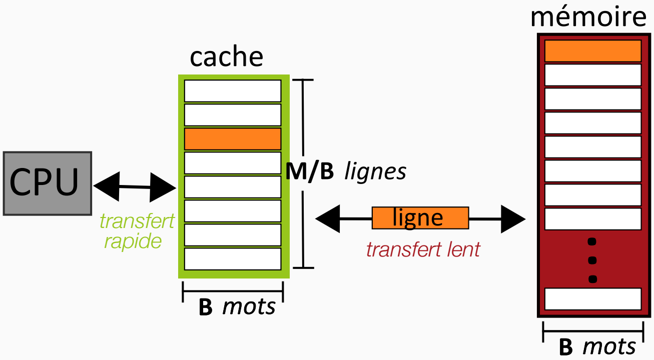


Analyse d'algorithmes:

nombre de transferts de lignes avec la mémoire (avec un cache vide au début)

$\Rightarrow$ cache complexity $\mathrm{MT}(n)$

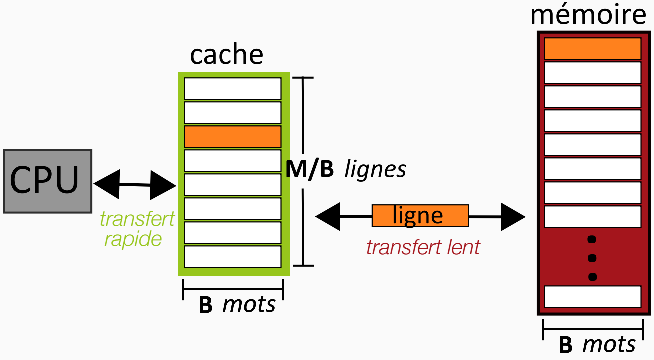
Ideal Cache model



Hypothèses fortes:

- le cache est *Fully Associative* $\rightarrow$ placement libre des lignes
- le cache est haut (*Tall cache asumption*) $\rightarrow$ $M \in \Omega(B^2)$
- la politique d'éviction des lignes est optimale $\rightarrow$ ligne la plus tardive dans le futur

Ideal Cache model

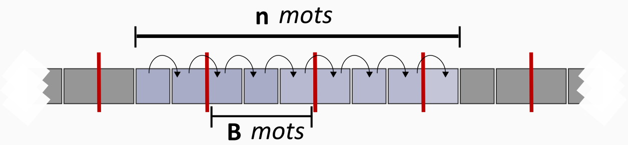


Objectif: minimisation de la complexité en cache MT(n)

- *cache aware algorithm*: adaptation du comportement en fonction des valeurs de B et M
- *cache oblivious algorithm*: ne connait pas les valeurs de B et M

Parcours séquentiel d'un tableau

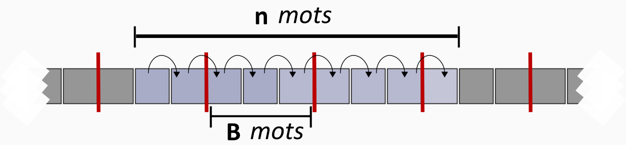
Parcourir un tableau de n mots contigüs en mémoire



$\Rightarrow$ complexité en cache: $\mathrm{MT}(n) = \lceil n/B \rceil + 1$

Parcours séquentiel d'un tableau

Parcourir un tableau de n mots contigüs en mémoire



$\Rightarrow$ complexité en cache: $\mathrm{MT}(n) = \lceil n/B \rceil + 1$

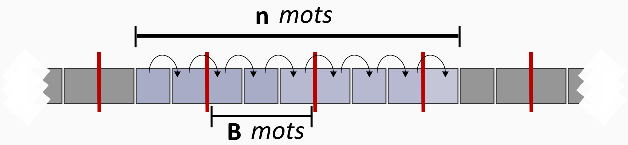
Remarques

- $\mathrm{MT}(n)$ correspond au nombre de cache miss

  $\Rightarrow$ complexité optimale(*cold miss* uniquement)

Parcours séquentiel d'un tableau

Parcourir un tableau de n mots contigüs en mémoire



$\Rightarrow$ complexité en cache: $\mathrm{MT}(n) = \lceil n/B \rceil + 1$

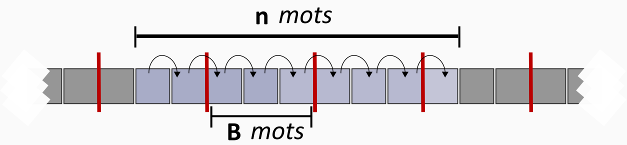
Remarques

- $\mathrm{MT}(n)$ correspond au nombre de cache miss

  $\Rightarrow$ complexité optimale(*cold miss* uniquement)
- $\mathrm{MT}(n) = \lceil n/B \rceil$ si cache aware $\rightarrow$ en alignant le tableau sur les limites des lignes de cache

Parcours séquentiel d'un tableau

Parcourir un tableau de n mots contigüs en mémoire



$\Rightarrow$ complexité en cache: $\mathrm{MT}(n) = \lceil n/B \rceil + 1$

Remarques

- $\mathrm{MT}(n)$ correspond au nombre de cache miss
  $\Rightarrow$ complexité optimale(*cold miss* uniquement)
- $\mathrm{MT}(n) = \lceil n/B \rceil$ si cache aware $\rightarrow$ en alignant le tableau sur les limites des lignes de cache

$\Rightarrow$ ex: fonction `reduce` : `for i in range(n): sum +=T[i]`

Renversement d'un tableau

En considérant un tableau de n mots contigüs en mémoire

```
void reverse(int *T, int n){
  for(int i=0;i<n/2;i++)
     swap(T[i],T[n-i-1]);
}
```

$\Rightarrow$ complexité en cache: $\mathrm{MT}(n) = \lceil n/B \rceil + 1$ si $M/B \geq 2$

Renversement d'un tableau

En considérant un tableau de n mots contigüs en mémoire

```c
void reverse(int *T, int n){
  for(int i=0;i<n/2;i++)
     swap(T[i],T[n-i-1]);
}
```

$\Rightarrow$ complexité en cache: $\mathrm{MT}(n) = \lceil n/B \rceil + 1$ si $M/B \geq 2$

Preuve:

- `T[i]` et `T[n-i-1]` tiennent dans au plus deux lignes de caches diférentes
- le tableau T tiens dans au plus $\lceil n/B \rceil + 1$ blocs

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
# histogramme des clés
Count= [0]*k
for x in T :
    Count[key(x)] +=1
# position de la première valeur pour chaque clés
Count.insert(0,0)
Count=prefixsum(Count[:-1])
# positionnement des valeurs dans le résultat
Output= [0]*n
for x in T :
    Output[Count[key(x)]] = x
    Count[key(x)] +=1
return Output
```

$\Rightarrow$ Complexité en temps: $O(n+k)$

Tri linéaire par comptage

tri d'un tableau T de n éléments ayant une clé entière dans $[0, k[$

Complexité en cache

```
# histogramme des clés
Count= [0]*k
for x in T :
    Count[key(x)] +=1
# position de la première valeur pour chaque clés
Count.insert(0,0)
Count=prefixsum(Count[:-1])
# positionnement des valeurs dans le résultat
Output= [0]*n
for x in T :
    Output[Count[key(x)]] = x
    Count[key(x)] +=1
return Output
```

$\Rightarrow$ Complexité en temps: $O(n+k)$

Tri linéaire par comptage

tri d'un tableau T de n éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2: $\lceil k/B \rceil + 1$

$\Rightarrow$ Complexité en temps: $O(n + k)$

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2: $\lceil k/B \rceil + 1$

ligne 3: $\lceil n/B \rceil + 1$

$\Rightarrow$ Complexité en temps: $O(n+k)$

Tri linéaire par comptage

tri d'un tableau T de n éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2: $\lceil k/B \rceil + 1$

ligne 3: $\lceil n/B \rceil + 1$

ligne 4: n

$\Rightarrow$ Complexité en temps: $O(n+k)$

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2:	$\lceil k/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 4:	n
ligne 6,7:	$\lceil k/B \rceil + 1$

$\Rightarrow$ Complexité en temps: $O(n + k)$

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2:	$\lceil k/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 4:	n
ligne 6,7:	$\lceil k/B \rceil + 1$
ligne 9:	$\lceil n/B \rceil + 1$

$\Rightarrow$ Complexité en temps: $O(n+k)$

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2:	$\lceil k/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 4:	n
ligne 6,7:	$\lceil k/B \rceil + 1$
ligne 9:	$\lceil n/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$

$\Rightarrow$ Complexité en temps: $O(n + k)$

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2:	$\lceil k/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 4:	n
ligne 6,7:	$\lceil k/B \rceil + 1$
ligne 9:	$\lceil n/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$
ligne 11,12:	$2n$

$\Rightarrow$ Complexité en temps: $O(n + k)$

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2:	$\lceil k/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 4:	n
ligne 6,7:	$\lceil k/B \rceil + 1$
ligne 9:	$\lceil n/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$
ligne 11,12:	$2n$

total: $3n + 3\lceil n/B \rceil + 2\lceil k/B \rceil + 5$

$\Rightarrow$ Complexité en temps: $O(n+k)$

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2:	$\lceil k/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 4:	n
ligne 6,7:	$\lceil k/B \rceil + 1$
ligne 9:	$\lceil n/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$
ligne 11,12:	$2n$

total: $3n + 3\lceil n/B \rceil + 2\lceil k/B \rceil + 5$

$\Rightarrow$ Complexité en temps: $O(n+k)$ Complexité en cache: $\mathrm{MT}(n,k) = O(n + \lceil k/B \rceil)$

Tri linéaire par comptage

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

```
1  # histogramme des clés
2  Count= [0]*k
3  for x in T :
4      Count[key(x)] +=1
5  # position de la première valeur pour chaque clés
6  Count.insert(0,0)
7  Count=prefixsum(Count[:-1])
8  # positionnement des valeurs dans le résultat
9  Output= [0]*n
10 for x in T :
11     Output[Count[key(x)]] = x
12     Count[key(x)] +=1
13 return Output
```

Complexité en cache

ligne 2:	$\lceil k/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 4:	n
ligne 6,7:	$\lceil k/B \rceil + 1$
ligne 9:	$\lceil n/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$
ligne 11,12:	$2n$

total: $3n + 3\lceil n/B \rceil + 2\lceil k/B \rceil + 5$

$\Rightarrow$ Complexité en temps: $O(n + k)$

Complexité en cache: $\mathrm{MT}(n, k) = O(n + \lceil k/B \rceil)$

loin de l'optimal: $O(\frac{n+k}{B})$

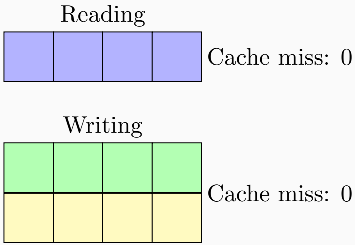
Tri linéaire par comptage: mauvaise localité spatiale

Input Array

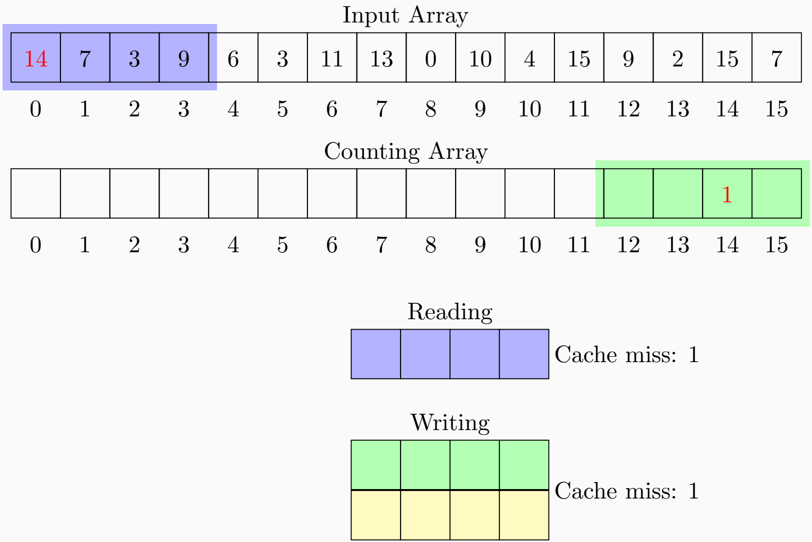
14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Counting Array

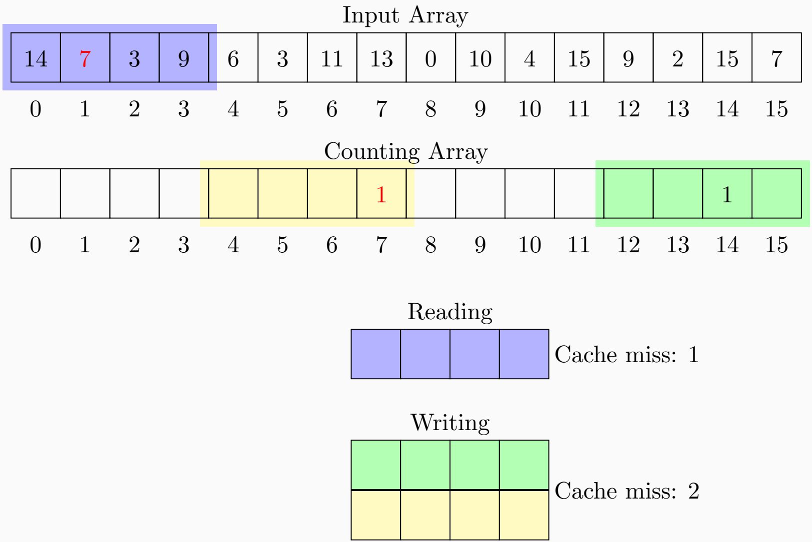
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



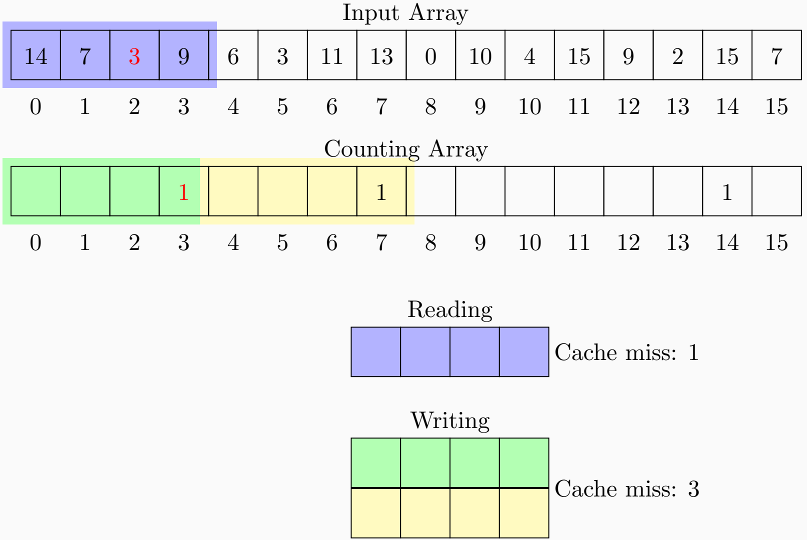
Tri linéaire par comptage: mauvaise localité spatiale



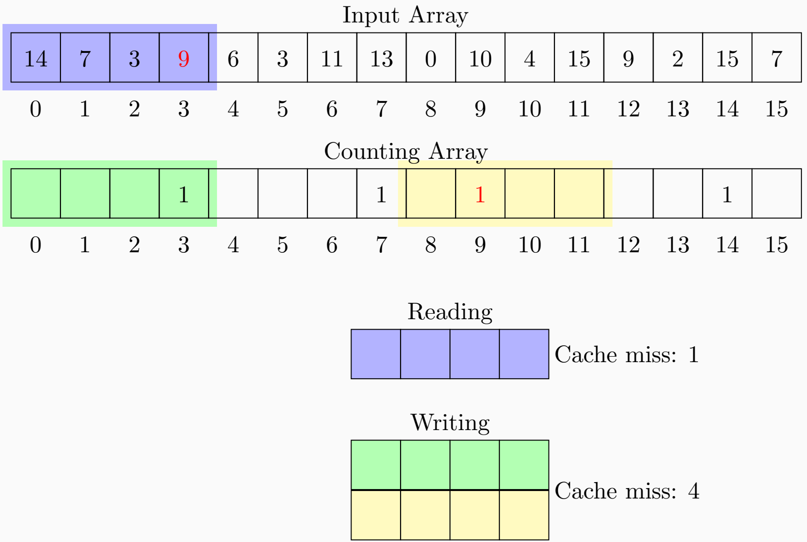
Tri linéaire par comptage: mauvaise localité spatiale



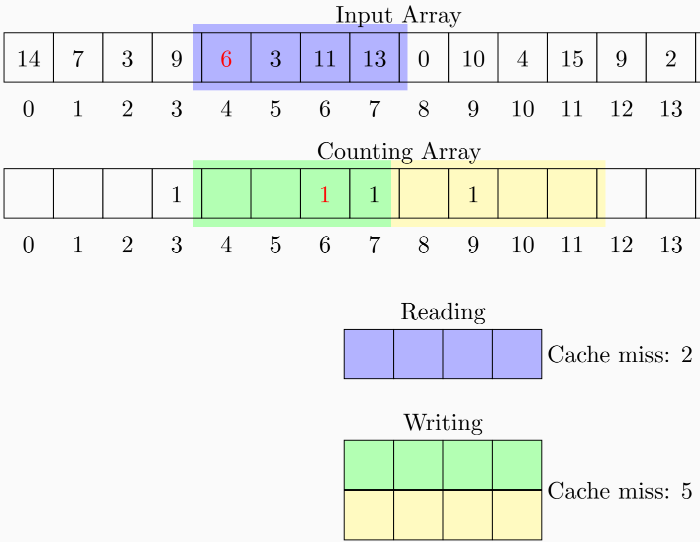
Tri linéaire par comptage: mauvaise localité spatiale



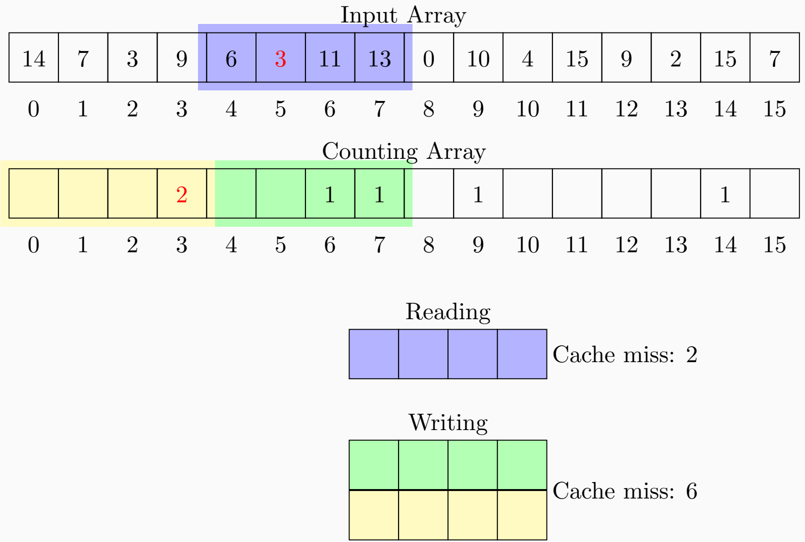
Tri linéaire par comptage: mauvaise localité spatiale



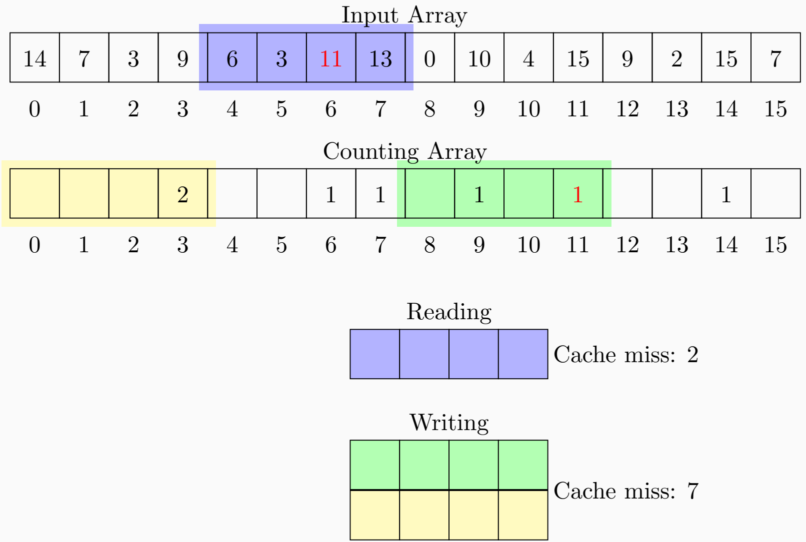
Tri linéaire par comptage: mauvaise localité spatiale



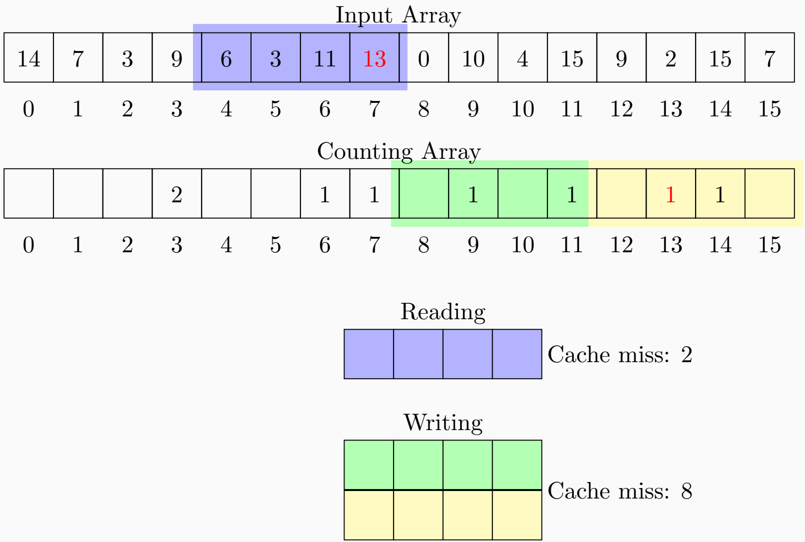
Tri linéaire par comptage: mauvaise localité spatiale



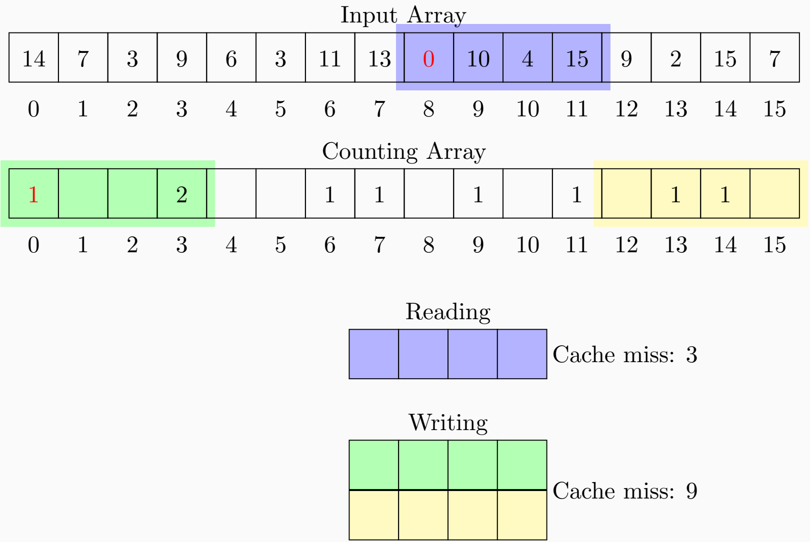
Tri linéaire par comptage: mauvaise localité spatiale



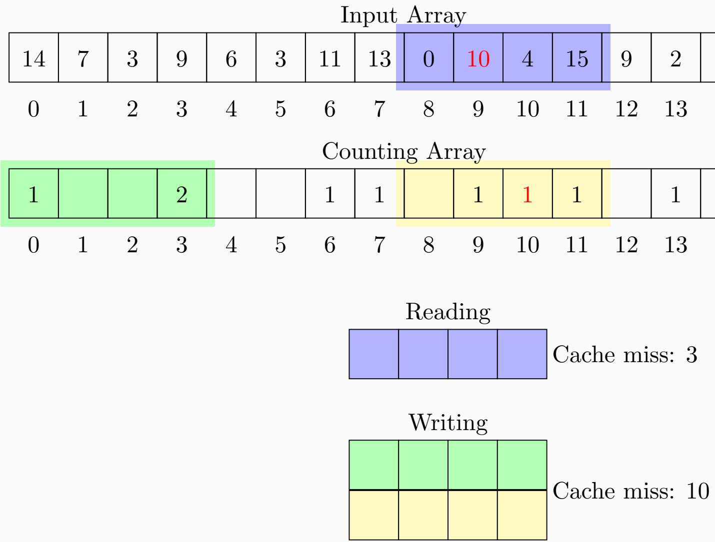
Tri linéaire par comptage: mauvaise localité spatiale



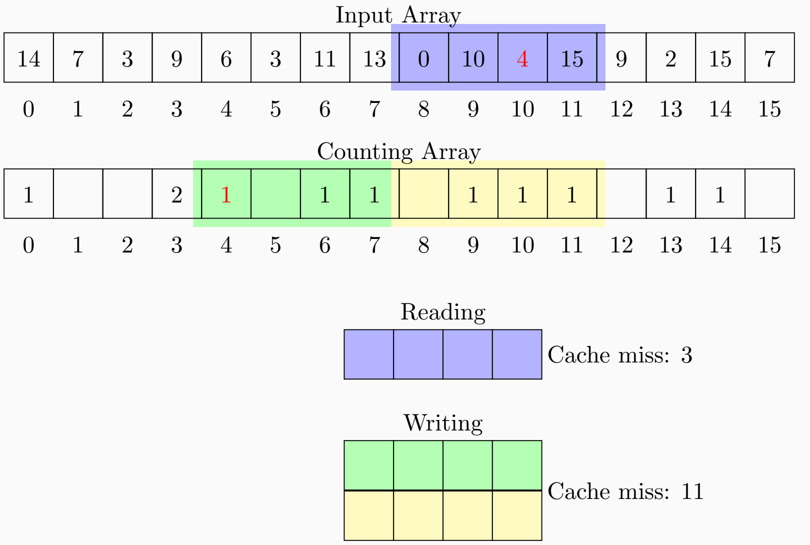
Tri linéaire par comptage: mauvaise localité spatiale



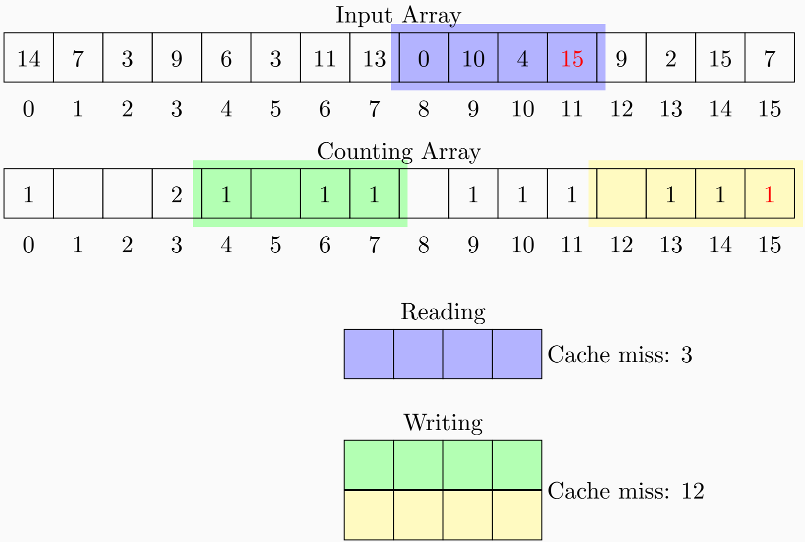
Tri linéaire par comptage: mauvaise localité spatiale



Tri linéaire par comptage: mauvaise localité spatiale



Tri linéaire par comptage: mauvaise localité spatiale



Tri linéaire par comptage: mauvaise localité spatiale

Input Array

Value	14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Counting Array

Count	1			2	1		1	1		2	1	1		1	1	1
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

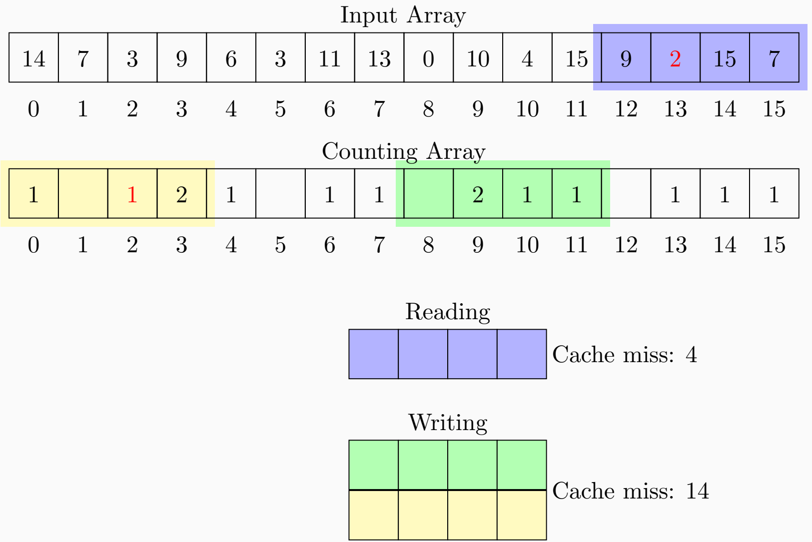
Reading

Cache miss: 4

Writing

Cache miss: 13

Tri linéaire par comptage: mauvaise localité spatiale



Tri linéaire par comptage: mauvaise localité spatiale

Input Array

14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Counting Array

1		1	2	1		1	1		2	1	1		1	1	2
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

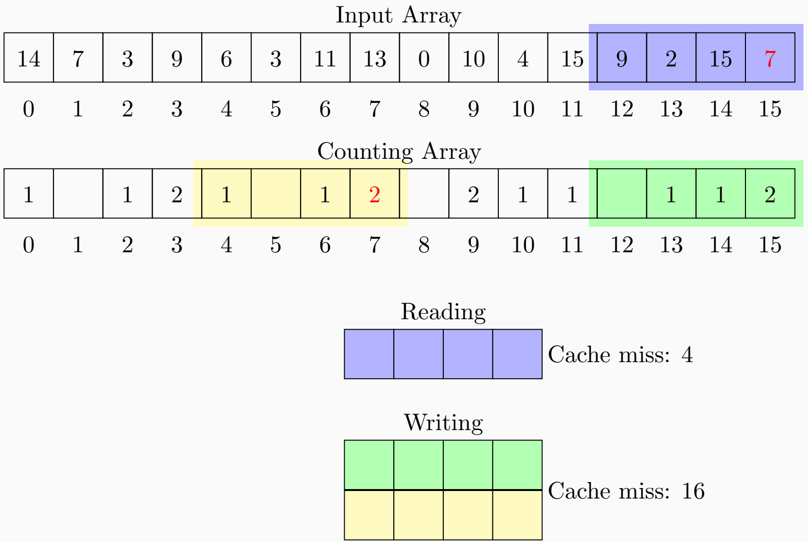
Reading

Cache miss: 4

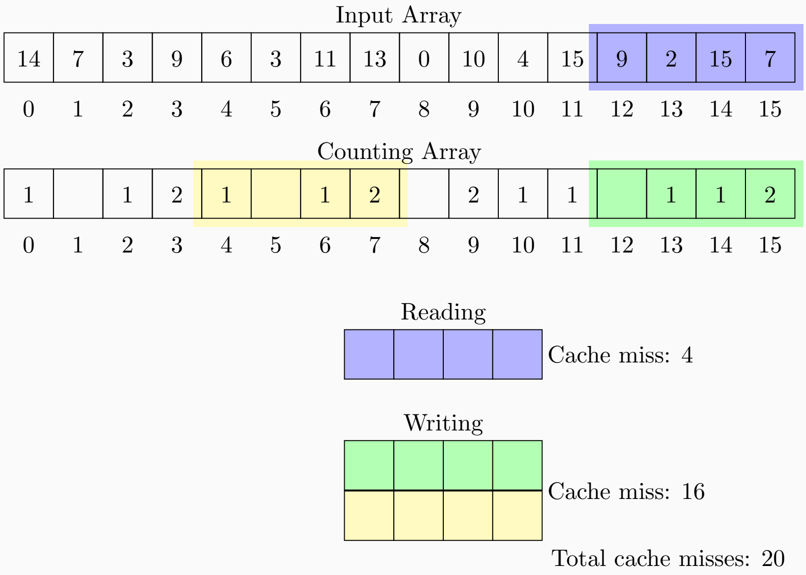
Writing

Cache miss: 15

Tri linéaire par comptage: mauvaise localité spatiale



Tri linéaire par comptage: mauvaise localité spatiale



Comment obtenir une meilleure localité spatiale

Technique de découpage en blocs

on décompose en plusieurs problèmes plus petits ayant des bonnes propriétés pour les caches:

- soit les sous-problèmes tiennent dans le cache
- soit les sous-problèmes n'engendrent que des *cold miss*

Comment obtenir une meilleure localité spatiale

Technique de découpage en blocs

on décompose en plusieurs problèmes plus petits ayant des bonnes propriétés pour les caches:

- soit les sous-problèmes tiennent dans le cache
- soit les sous-problèmes n'engendrent que des *cold miss*

Pour le tri par comptage, on découpe le problème sur m plages de valeurs de clé

$$[0, k[= [k_0, k_1[+ [k_1, k_2[+ [k_2, k_3] + \cdots + [k_{m-1}, k_m[$$

en séparant l'entrée en m tableaux de taille $n_1, n_2, \ldots n_m$

$\Rightarrow$ si $\max(n_i) + \max(k_i - k_{i-1}) < M$ uniquement des *cold miss*

Tri linéaire par comptage: meilleure localité spatiale

Input Array

14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pivot: 8

Bucket 1

0	1	2	3	4	5	6	7

Bucket 2

0	1	2	3	4	5	6	7

Counting Array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

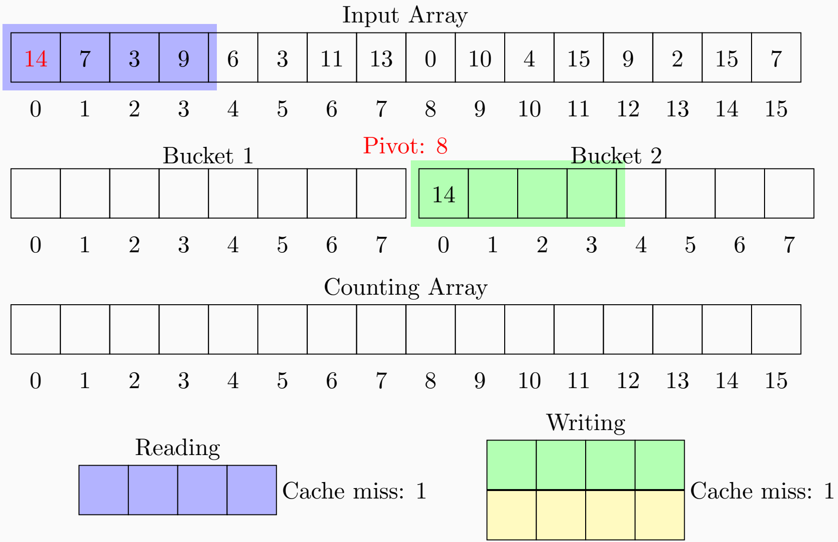
Reading

Cache miss: 0

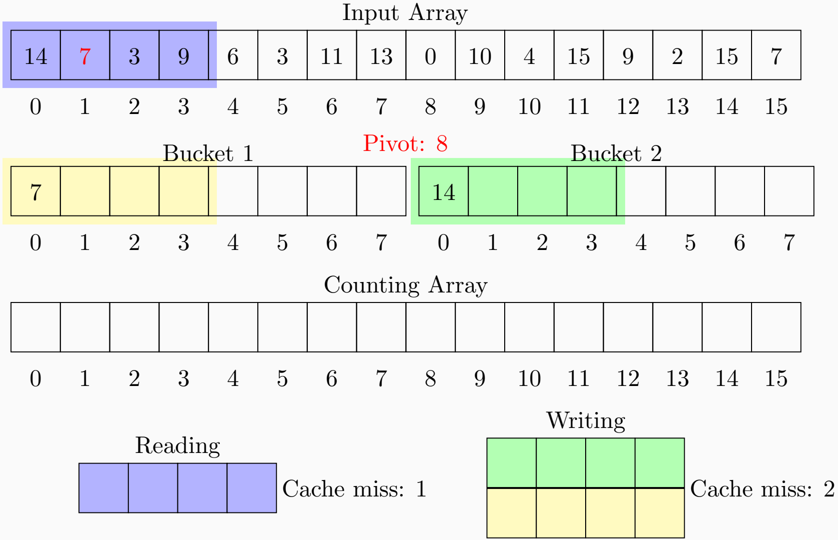
Writing

Cache miss: 0

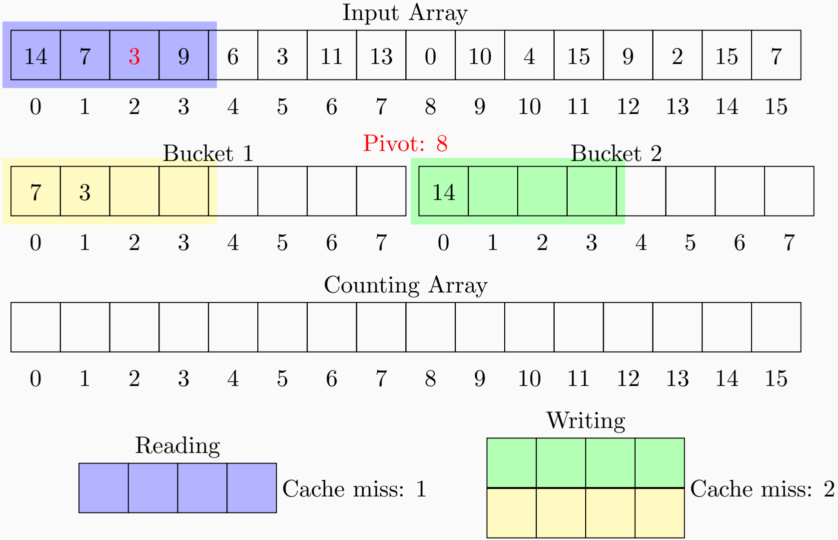
Tri linéaire par comptage: meilleure localité spatiale



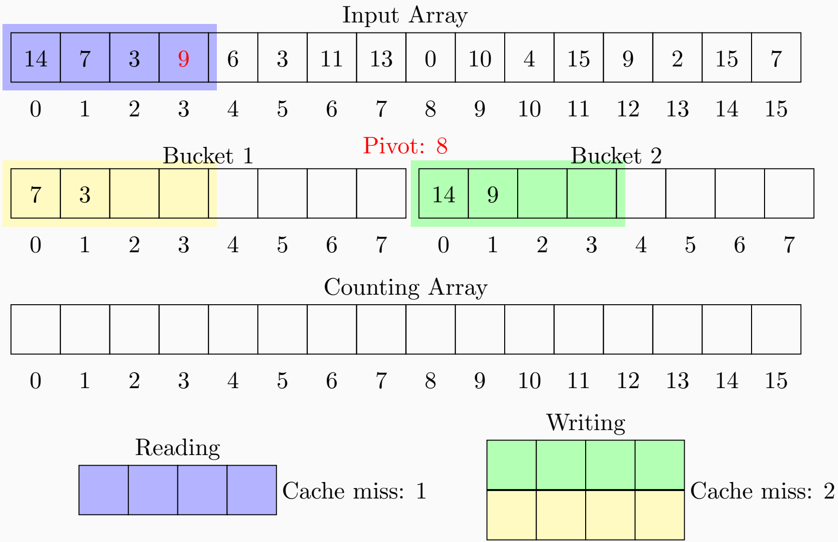
Tri linéaire par comptage: meilleure localité spatiale



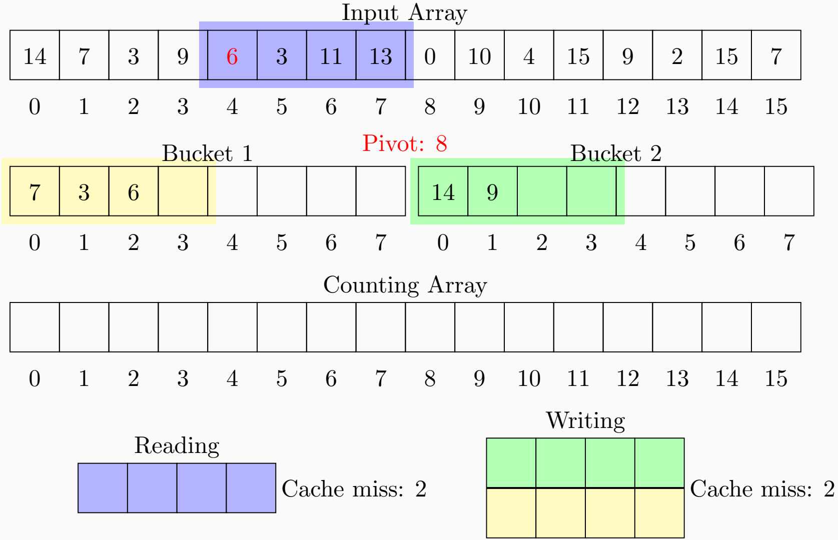
Tri linéaire par comptage: meilleure localité spatiale



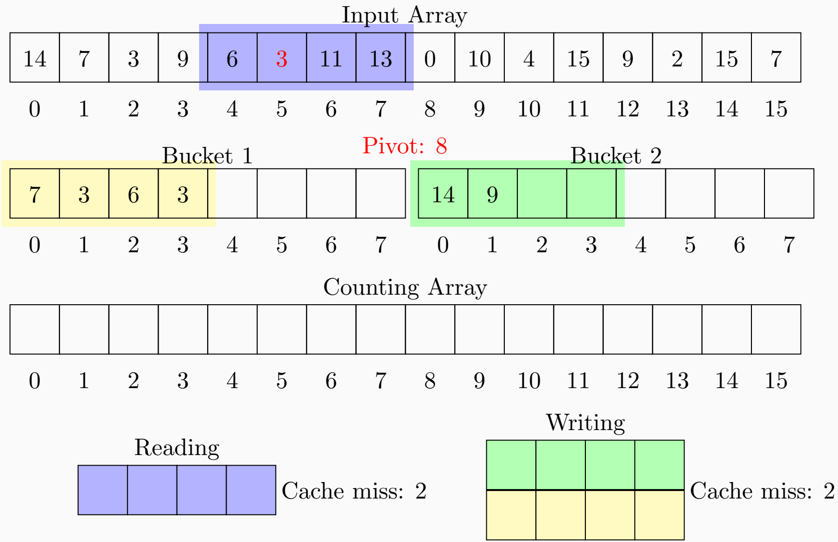
Tri linéaire par comptage: meilleure localité spatiale



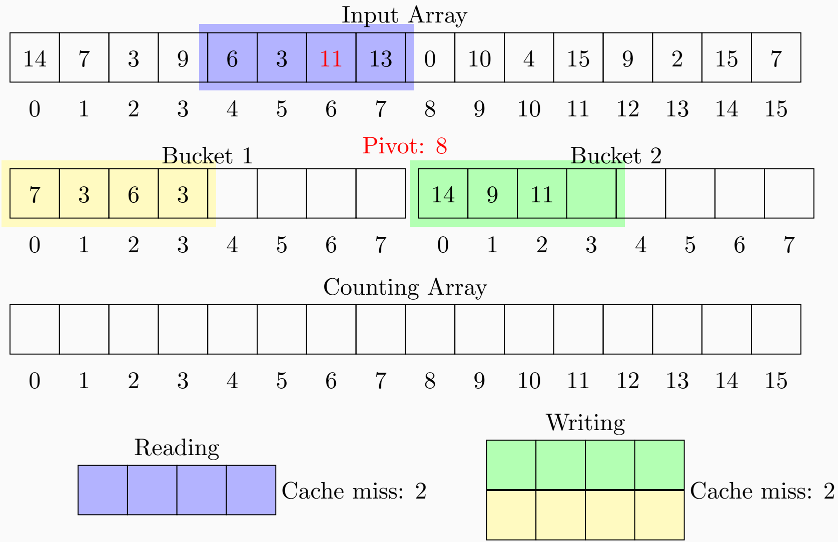
Tri linéaire par comptage: meilleure localité spatiale



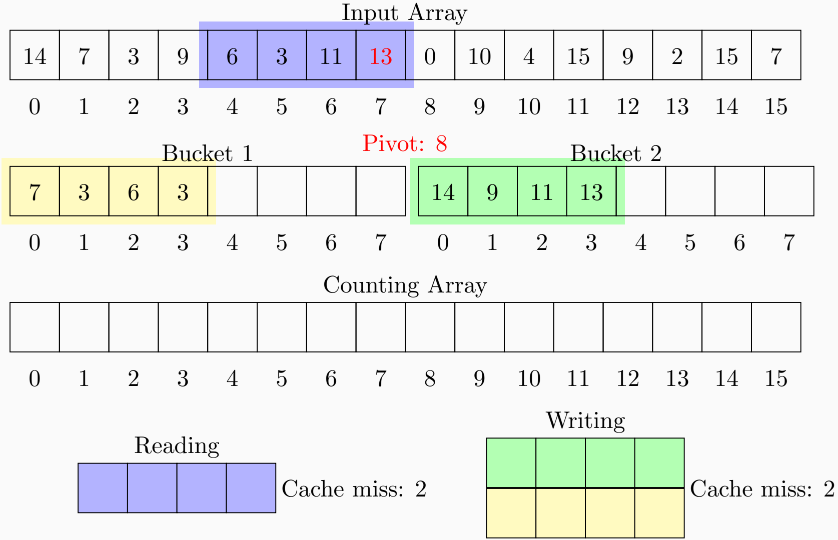
Tri linéaire par comptage: meilleure localité spatiale



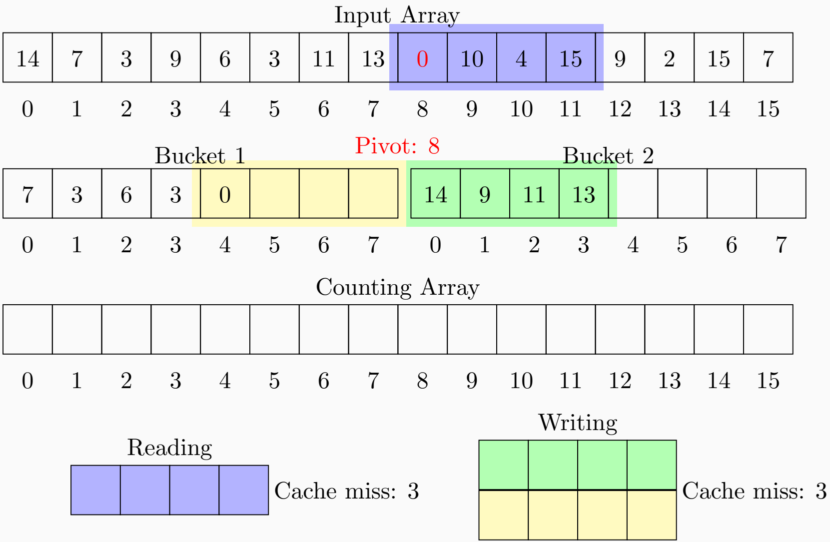
Tri linéaire par comptage: meilleure localité spatiale



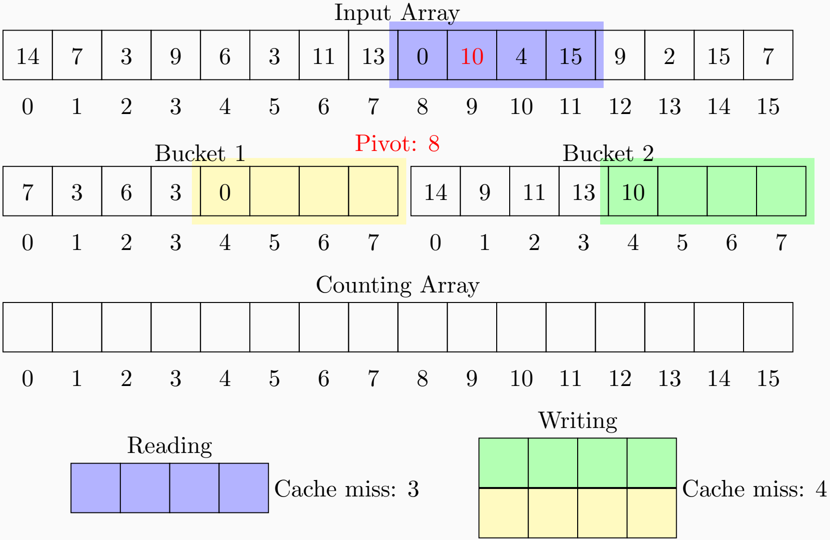
Tri linéaire par comptage: meilleure localité spatiale



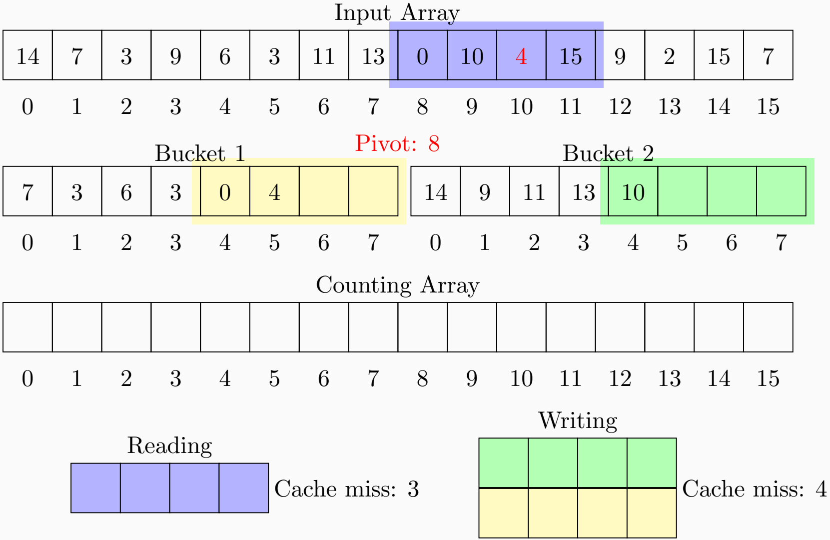
Tri linéaire par comptage: meilleure localité spatiale



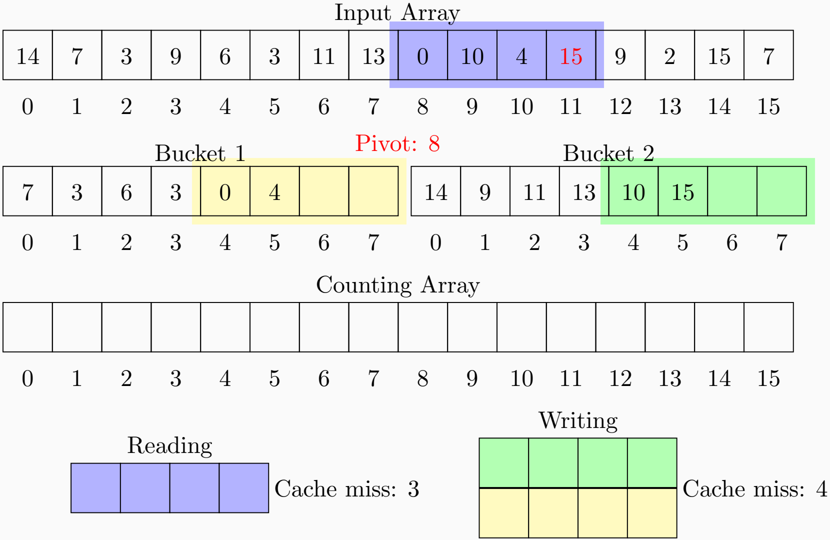
Tri linéaire par comptage: meilleure localité spatiale



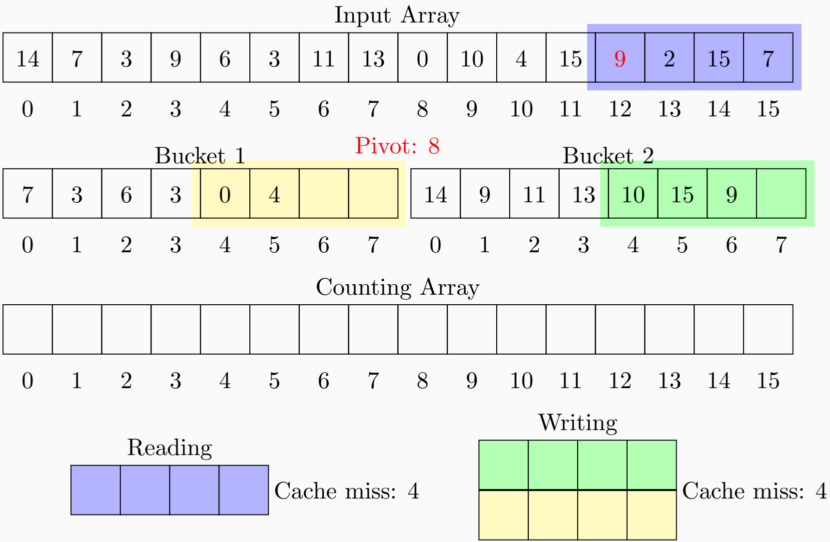
Tri linéaire par comptage: meilleure localité spatiale



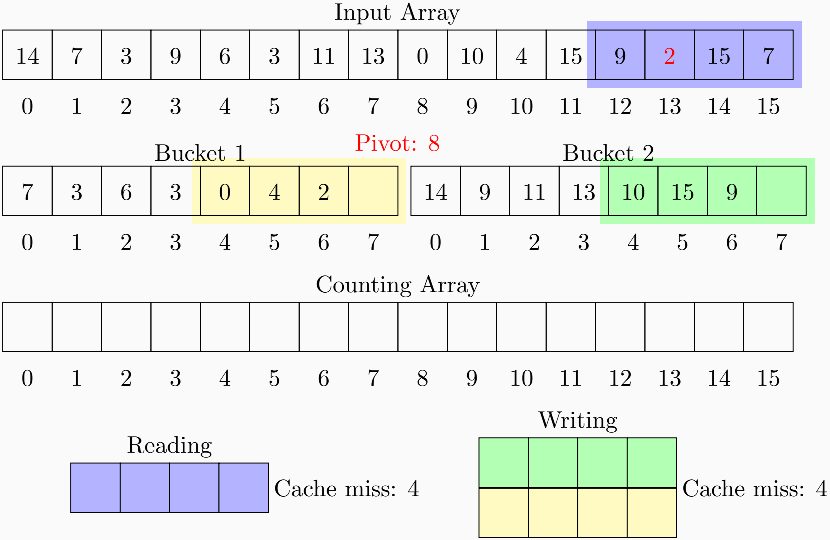
Tri linéaire par comptage: meilleure localité spatiale



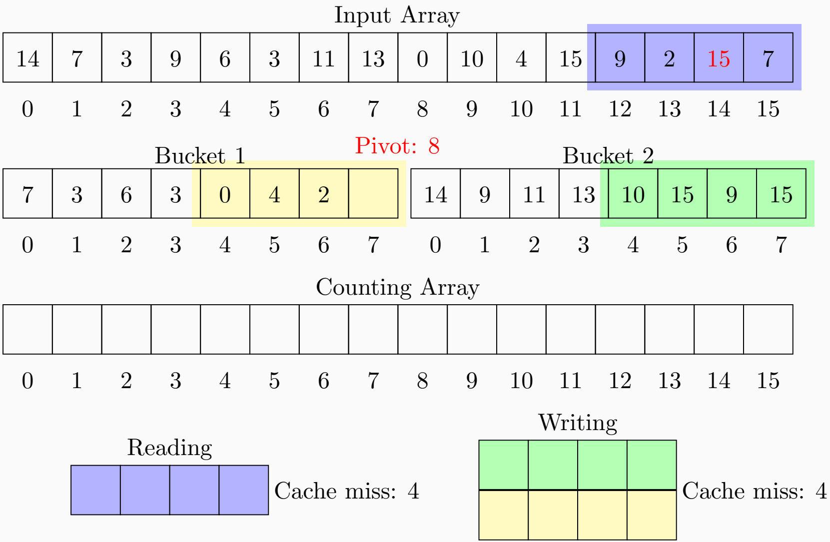
Tri linéaire par comptage: meilleure localité spatiale



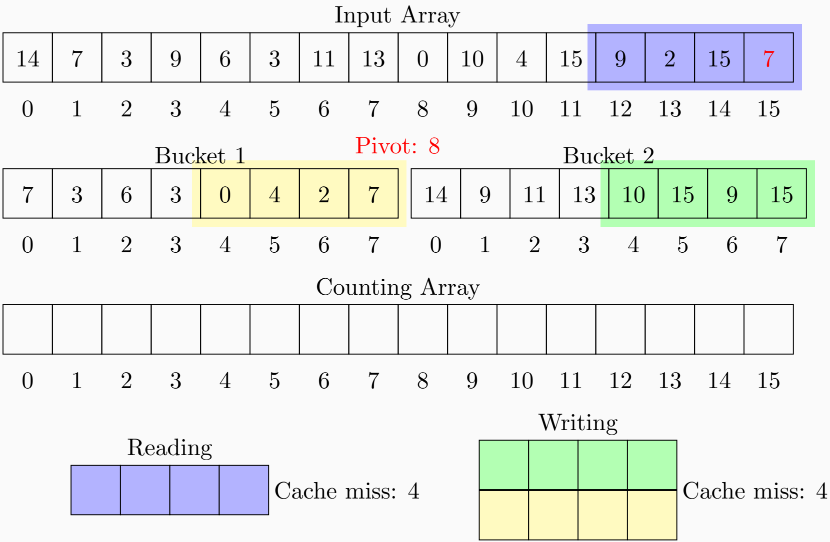
Tri linéaire par comptage: meilleure localité spatiale



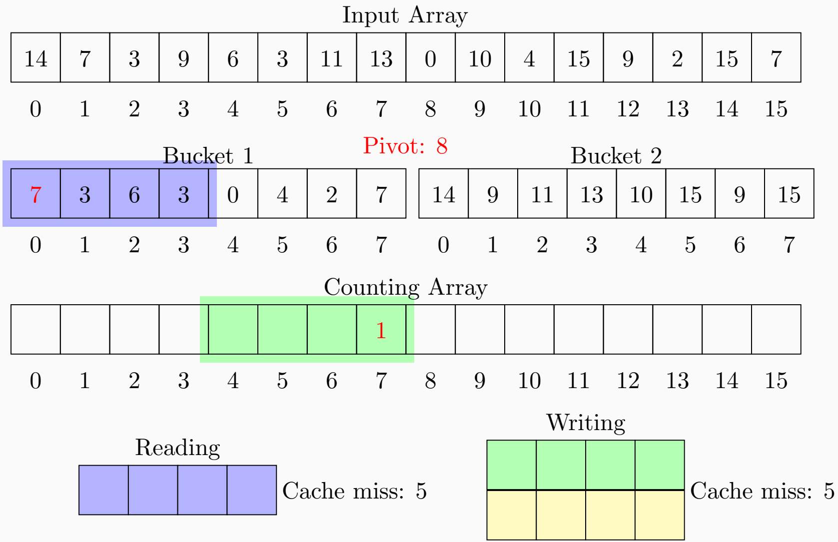
Tri linéaire par comptage: meilleure localité spatiale



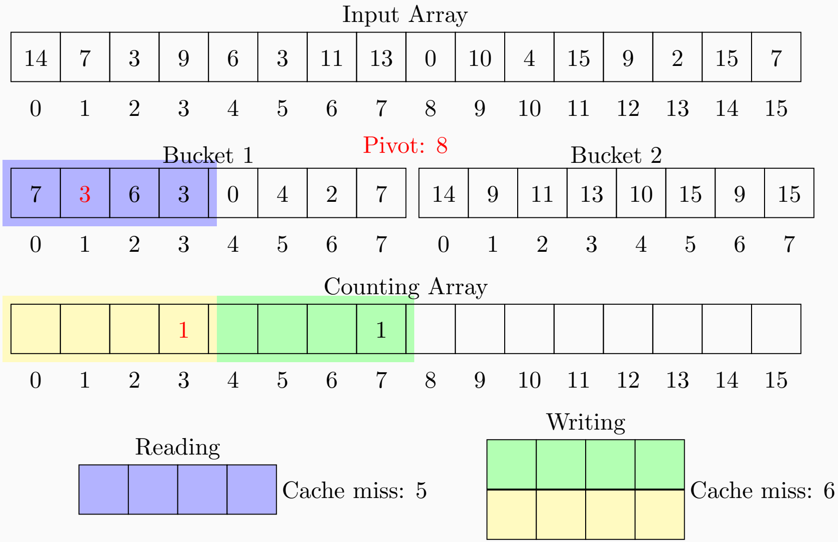
Tri linéaire par comptage: meilleure localité spatiale



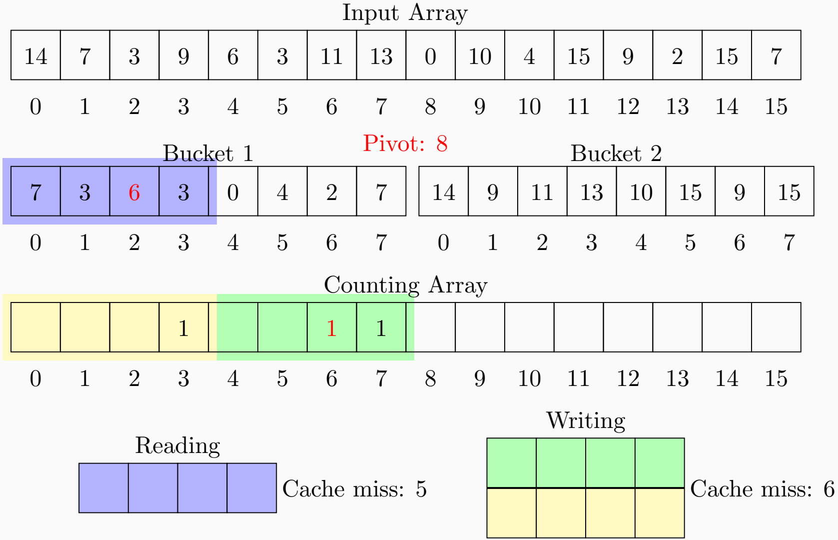
Tri linéaire par comptage: meilleure localité spatiale



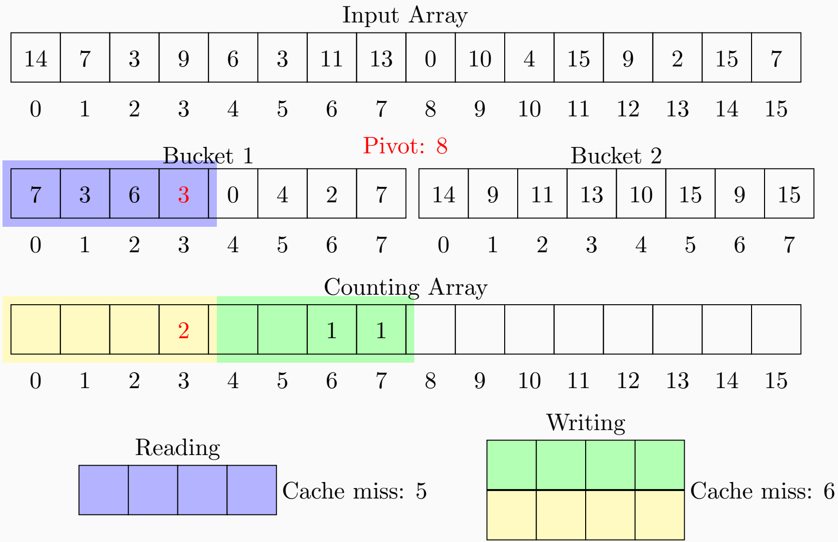
Tri linéaire par comptage: meilleure localité spatiale



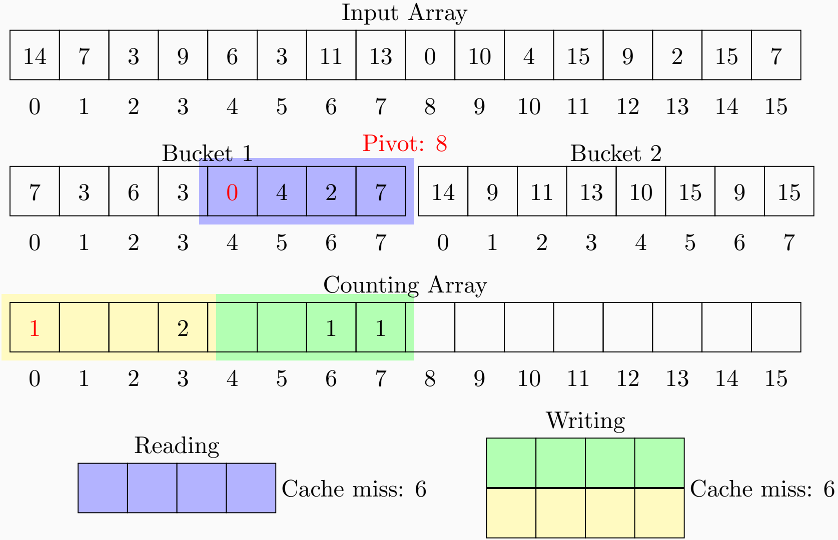
Tri linéaire par comptage: meilleure localité spatiale



Tri linéaire par comptage: meilleure localité spatiale



Tri linéaire par comptage: meilleure localité spatiale



Tri linéaire par comptage: meilleure localité spatiale

Input Array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7

Pivot: 8

Bucket 1

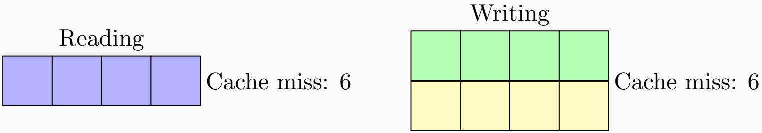
0	1	2	3	4	5	6	7
7	3	6	3	0	4	2	7

Bucket 2

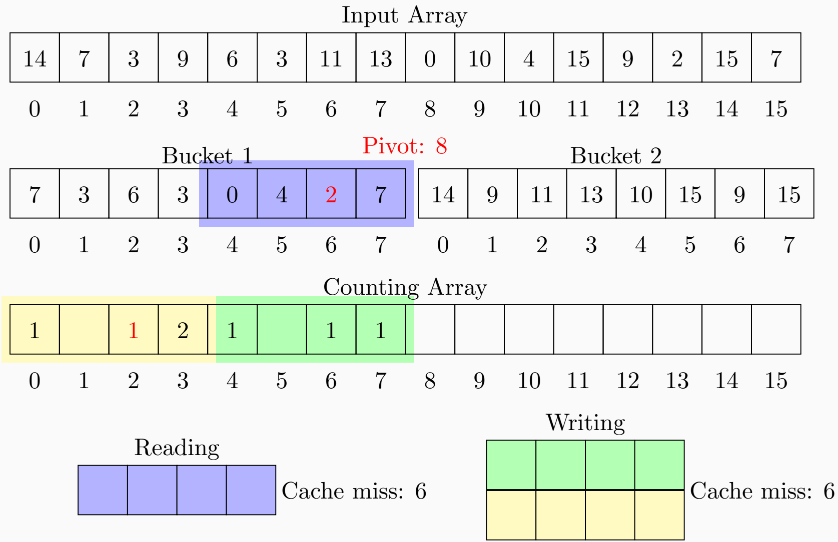
0	1	2	3	4	5	6	7
14	9	11	13	10	15	9	15

Counting Array

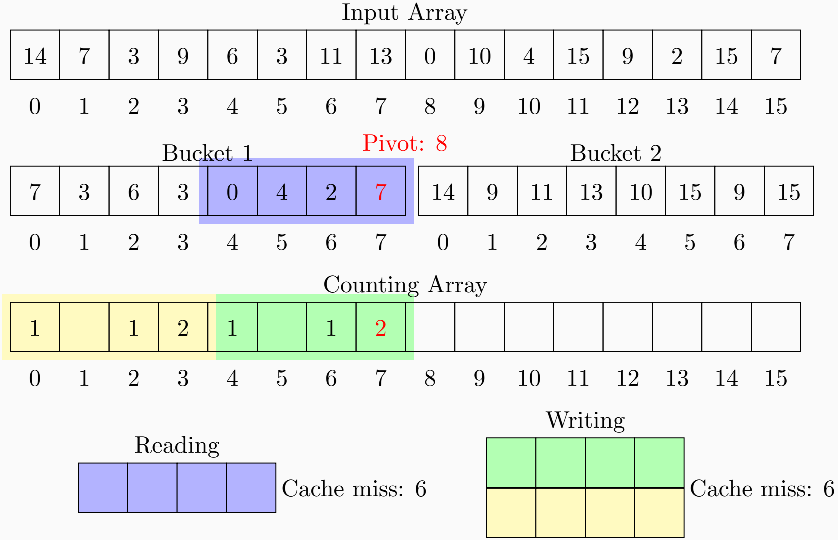
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1			2	1		1	1								



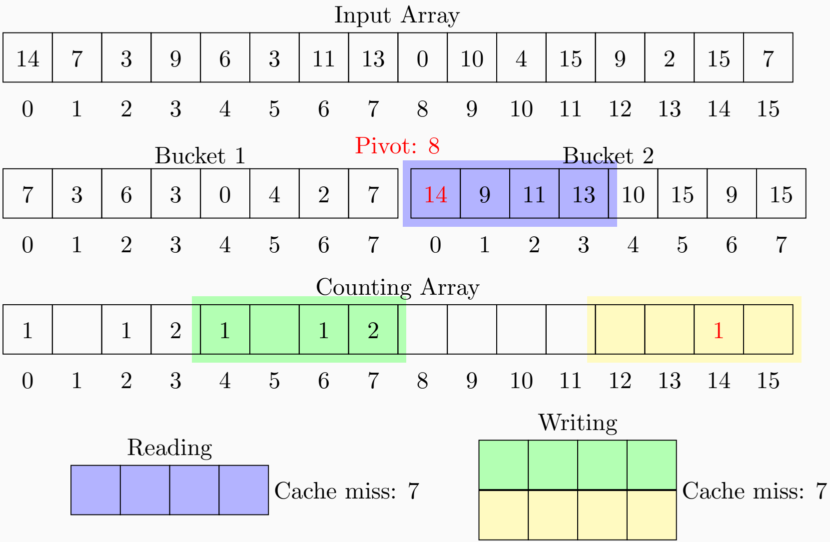
Tri linéaire par comptage: meilleure localité spatiale



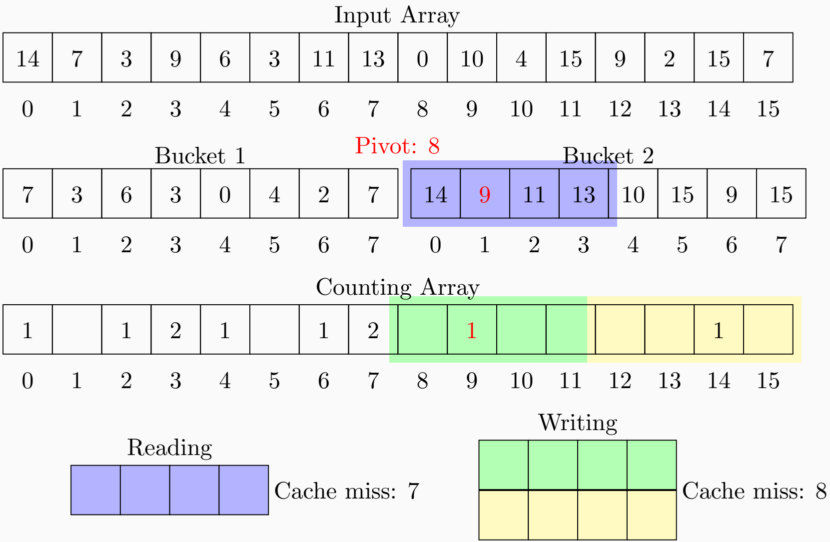
Tri linéaire par comptage: meilleure localité spatiale



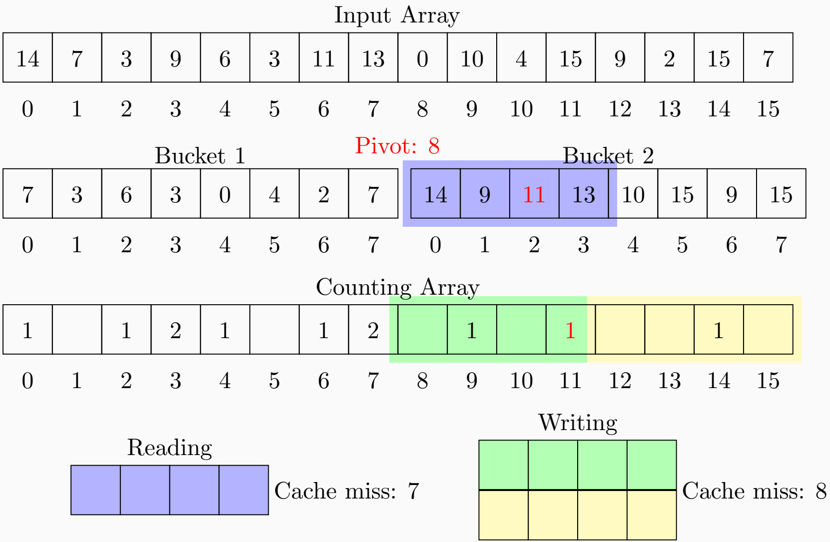
Tri linéaire par comptage: meilleure localité spatiale



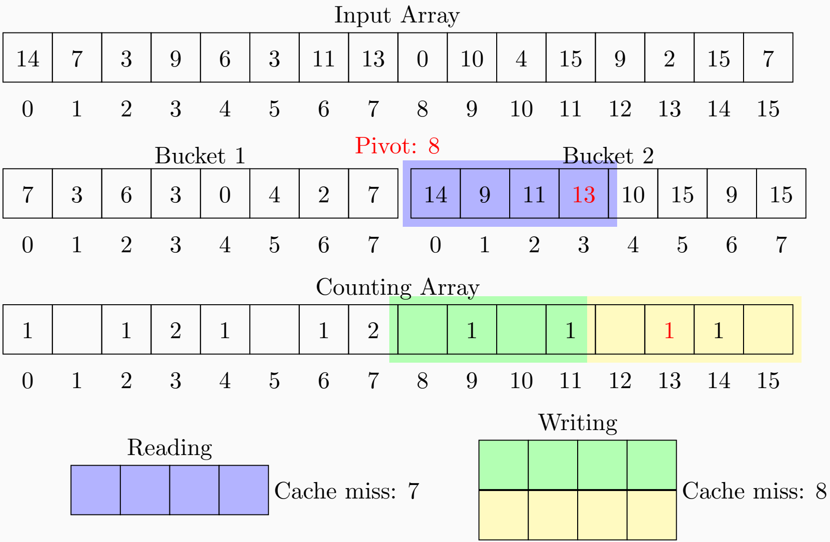
Tri linéaire par comptage: meilleure localité spatiale



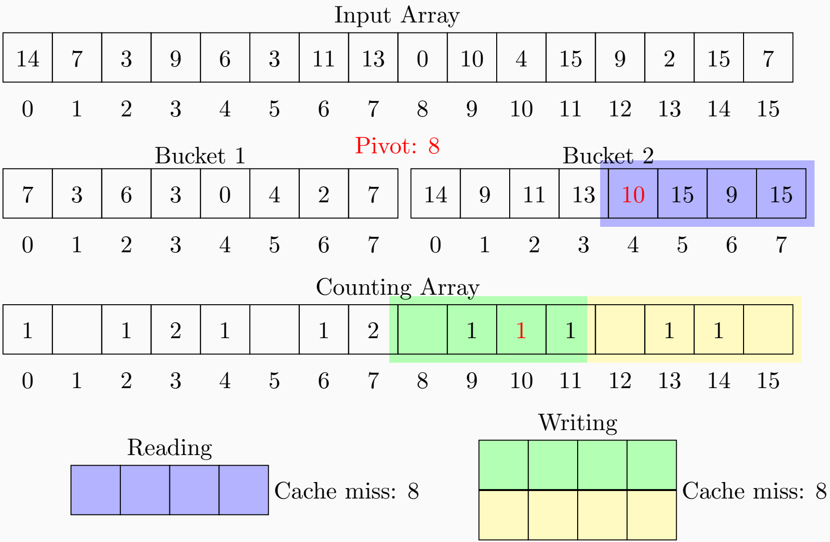
Tri linéaire par comptage: meilleure localité spatiale



Tri linéaire par comptage: meilleure localité spatiale



Tri linéaire par comptage: meilleure localité spatiale



Tri linéaire par comptage: meilleure localité spatiale

Input Array

14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pivot: 8

Bucket 1

7	3	6	3	0	4	2	7
0	1	2	3	4	5	6	7

Bucket 2

14	9	11	13	10	15	9	15
0	1	2	3	4	5	6	7

Counting Array

1		1	2	1		1	2		1	1	1		1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Reading

Cache miss: 8

Writing

Cache miss: 8

Tri linéaire par comptage: meilleure localité spatiale

Input Array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7

Pivot: 8

Bucket 1

0	1	2	3	4	5	6	7
7	3	6	3	0	4	2	7

Bucket 2

0	1	2	3	4	5	6	7
14	9	11	13	10	15	9	15

Counting Array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1		1	2	1		1	2		2	1	1		1	1	1

Reading

Cache miss: 8

Writing

Cache miss: 8

Tri linéaire par comptage: meilleure localité spatiale

Input Array

14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pivot: 8

Bucket 1

7	3	6	3	0	4	2	7
0	1	2	3	4	5	6	7

Bucket 2

14	9	11	13	10	15	9	15
0	1	2	3	4	5	6	7

Counting Array

1		1	2	1		1	2		2	1	1		1	1	2
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Reading

Cache miss: 8

Writing

Cache miss: 8

Tri linéaire par comptage: meilleure localité spatiale

Input Array

14	7	3	9	6	3	11	13	0	10	4	15	9	2	15	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pivot: 8

Bucket 1

7	3	6	3	0	4	2	7
0	1	2	3	4	5	6	7

Bucket 2

14	9	11	13	10	15	9	15
0	1	2	3	4	5	6	7

Counting Array

1		1	2	1		1	2		2	1	1		1	1	2
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Reading

Cache miss: 8

Writing

Cache miss: 8

Total cache misses: 16

Tri linéaire par comptage: bucketing

tri d'un tableau T de n éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
# histogramme nbr de clés dans [(i-1)k/m, ik/m[
Bucket=[0]*m
for x in T:
    q=m*key(x)//k
    Bucket[q]+=1 # à la fin on a ni= Bucket[i]
# calcul des indices de début de chaque bucket
Bucket.insert(0,0)
Bucket=prefixsum(Bucket)
# placement des éléments de T dans les buckets
BucketedInput=[0]*n
for x in T:
    q=m*key(x)//k
    BucketedInput[Bucket[q]]=x
    Bucket[q]+=1
```

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

Complexité en cache

```
# histogramme nbr de clés dans [(i-1)k/m, ik/m[
Bucket=[0]*m
for x in T:
    q=m*key(x)//k
    Bucket[q]+=1 # à la fin on a ni= Bucket[i]
# calcul des indices de début de chaque bucket
Bucket.insert(0,0)
Bucket=prefixsum(Bucket)
# placement des éléments de T dans les buckets
BucketedInput=[0]*n
for x in T:
    q=m*key(x)//k
    BucketedInput[Bucket[q]]=x
    Bucket[q]+=1
```

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2: $\lceil m/B \rceil + 1$

Tri linéaire par comptage: bucketing

tri d'un tableau T de n éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2: $\lceil m/B \rceil + 1$

ligne 3: $\lceil n/B \rceil + 1$

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2:	$\lceil m/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 5:	$\lceil m/B \rceil + 1$

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2:	$\lceil m/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 5:	$\lceil m/B \rceil + 1$
ligne 7,8:	$\lceil m/B \rceil + 1$

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2:	$\lceil m/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 5:	$\lceil m/B \rceil + 1$
ligne 7,8:	$\lceil m/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2: $\lceil m/B \rceil + 1$

ligne 3: $\lceil n/B \rceil + 1$

ligne 5: $\lceil m/B \rceil + 1$

ligne 7,8: $\lceil m/B \rceil + 1$

ligne 10: $\lceil n/B \rceil + 1$

ligne 11: $\lceil n/B \rceil + 1$

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2:	$\lceil m/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 5:	$\lceil m/B \rceil + 1$
ligne 7,8:	$\lceil m/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$
ligne 11:	$\lceil n/B \rceil + 1$
ligne 14:	$\lceil m/B \rceil + 1$

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2:	$\lceil m/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 5:	$\lceil m/B \rceil + 1$
ligne 7,8:	$\lceil m/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$
ligne 11:	$\lceil n/B \rceil + 1$
ligne 14:	$\lceil m/B \rceil + 1$

ligne 13:

$\sum_i (\lceil n_i/B \rceil + 1) < \lceil n/B \rceil + 2m$

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i-1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2:	$\lceil m/B \rceil + 1$
ligne 3:	$\lceil n/B \rceil + 1$
ligne 5:	$\lceil m/B \rceil + 1$
ligne 7,8:	$\lceil m/B \rceil + 1$
ligne 10:	$\lceil n/B \rceil + 1$
ligne 11:	$\lceil n/B \rceil + 1$
ligne 14:	$\lceil m/B \rceil + 1$

ligne 13:

$\sum_i (\lceil n_i/B \rceil + 1) < \lceil n/B \rceil + 2m$

---

total: $4\lceil m/B \rceil + 3\lceil n/B \rceil + 2m$

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

$\Rightarrow$ génération de m buckets en supposant $m + mB < M$

```
1  # histogramme nbr de clés dans [(i−1)k/m, ik/m[
2  Bucket=[0]*m
3  for x in T:
4      q=m*key(x)//k
5      Bucket[q]+=1 # à la fin on a ni= Bucket[i]
6  # calcul des indices de début de chaque bucket
7  Bucket.insert(0,0)
8  Bucket=prefixsum(Bucket)
9  # placement des éléments de T dans les buckets
10 BucketedInput=[0]*n
11 for x in T:
12     q=m*key(x)//k
13     BucketedInput[Bucket[q]]=x
14     Bucket[q]+=1
```

Complexité en cache

ligne 2: $\lceil m/B \rceil + 1$

ligne 3: $\lceil n/B \rceil + 1$

ligne 5: $\lceil m/B \rceil + 1$

ligne 7,8: $\lceil m/B \rceil + 1$

ligne 10: $\lceil n/B \rceil + 1$

ligne 11: $\lceil n/B \rceil + 1$

ligne 14: $\lceil m/B \rceil + 1$

ligne 13:

$\sum_i (\lceil n_i/B \rceil + 1) < \lceil n/B \rceil + 2m$

---

total: $4\lceil m/B \rceil + 3\lceil n/B \rceil + 2m$

$\Rightarrow$ Complexité en temps: $O(n + m)$ Complexité en cache: $O(m + \lceil n/B \rceil)$

plus le coût de trier les m sous-tableaux de taille $\approx n/m$ avec des clés d'amplitude k/m

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

- génération de m buckets en supposant $m + mB < M$

  $\Rightarrow$ Complexité en temps: $O(n + m)$

  $\Rightarrow$ Complexité en cache: $O(m + \lceil n/B \rceil)$

- trier les m buckets de taille $\approx n/m$ avec des clés d'amplitude k/m

[3] valeur réaliste avec la taille des caches en vrai

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

- génération de m buckets en supposant $m + mB < M$

  $\Rightarrow$ Complexité en temps: $O(n + m)$

  $\Rightarrow$ Complexité en cache: $O(m + \lceil n/B \rceil)$

- trier les m buckets de taille $\approx n/m$ avec des clés d'amplitude k/m

  si on suppose[3] $n/m < M$ et $k/m < M$ on obtient:

  $\Rightarrow$ Complexité en temps: $m \times O(n/m + k/m) = O(n + k)$

  $\Rightarrow$ Complexité en cache: $m \times O(n/(Bm) + \lceil k/(Bm) \rceil) = O(\lceil n/B \rceil + \lceil k/B \rceil)$

[3] valeur réaliste avec la taille des caches en vrai

Tri linéaire par comptage: bucketing

tri d'un tableau `T` de `n` éléments ayant une clé entière dans $[0, k[$

- génération de m buckets en supposant $m + mB < M$
  $\Rightarrow$ Complexité en temps: $O(n + m)$
  $\Rightarrow$ Complexité en cache: $O(m + \lceil n/B \rceil)$
- trier les m buckets de taille $\approx n/m$ avec des clés d'amplitude k/m
  si on suppose[3] $n/m < M$ et $k/m < M$ on obtient:
  $\Rightarrow$ Complexité en temps: $m \times O(n/m + k/m) = O(n + k)$
  $\Rightarrow$ Complexité en cache: $m \times O(n/(Bm) + \lceil k/(Bm) \rceil) = O(\lceil n/B \rceil + \lceil k/B \rceil)$

Au total: $MT(n, k, m) = O(m + \lceil n/B \rceil + \lceil k/B \rceil)$ au lieu de $O(n + \lceil k/B \rceil)$

[3] valeur réaliste avec la taille des caches en vrai

Multiplication de matrices

$Z = X \times Y$ avec des matrices de taille $n \times n$ stockées par lignes (Z initialisée avec des zéros)

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            Z[i,j]+=X[i,k]*Y[k,j]
```

- $\mathrm{MT}(n) = O(n^2/B)$ si $3n^2 < M$
  $\hookrightarrow$ X,Y,Z tiennent dans le cache
- $\mathrm{MT}(n) = O(n^3/B)$ si $n(1+B) < M \leq 3n^2$
  $\hookrightarrow$ une ligne de X et B colonnes de Y tiennent dans le cache
- $\mathrm{MT}(n) = O(n^3)$ si $3 < M/B \leq n$
  $\hookrightarrow$ une colonne de Y ne tient pas dans le cache

Multiplication de matrices: amélioration via le stockage

$Z = X \times Y$ avec des matrices de taille $n \times n$ (Z initialisée avec des zéros)

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            Z[i,j]+=X[i,k]*Y[k,j]
```

- X et Z stockées par ligne
- Y stockée par colonne

- $\mathrm{MT}(n) = O(n^2/B)$ si $3n^2 < M$
  $\hookrightarrow$ X,Y,Z tiennent dans le cache
- $\mathrm{MT}(n) = O(n^3/B + n^2)$ si $n(1+B) < M \leq 3n^2$
  $\hookrightarrow$ une ligne de X et B colonnes de Y tiennent dans le cache
- $\mathrm{MT}(n) = O(n^3/B + n^2)$ si $3 < M/B \leq 3n^2/B$
  $\hookrightarrow$ une seule ligne de cache par matrice

Multiplication de matrices: amélioration via les boucles

$Z = X \times Y$ avec des matrices de taille $n \times n$ stockées par lignes (Z initialisée avec des zéros)

```
for i in range(n):
    for k in range(n):
        for j in range(n):
            Z[i,j]+=X[i,k]*Y[k,j]
```

- X, Y, Z stockées en ligne
- on inverse les boucles `k` et `j`

- $\mathrm{MT}(n) = O(n^2/B)$ si $3n^2 < M$
  $\hookrightarrow$ X,Y,Z tiennent dans le cache
- $\mathrm{MT}(n) = O(n^3/B + n^2)$ si $n(1+B) < M \leq 3n^2$
  $\hookrightarrow$ une ligne de X et B colonnes de Y tiennent dans le cache
- $\mathrm{MT}(n) = O(n^3/B + n^2)$ si $3 < M/B \leq 3n^2/B$
  $\hookrightarrow$ une seule ligne de cache par matrice

Multiplication de matrices: amélioration via les boucles

$Z = X \times Y$ avec des matrices de taille $n \times n$ stockées par lignes (Z initialisée avec des zéros)

```
for i in range(n):
    for k in range(n):
        for j in range(n):
            Z[i,j]+=X[i,k]*Y[k,j]
```

- X, Y, Z stockées en ligne
- on inverse les boucles `k` et `j`

- $\mathrm{MT}(n) = O(n^2/B)$ si $3n^2 < M$
  $\hookrightarrow$ X,Y,Z tiennent dans le cache
- $\mathrm{MT}(n) = O(n^3/B + n^2)$ si $n(1+B) < M \leq 3n^2$
  $\hookrightarrow$ une ligne de X et B colonnes de Y tiennent dans le cache
- $\mathrm{MT}(n) = O(n^3/B + n^2)$ si $3 < M/B \leq 3n^2/B$
  $\hookrightarrow$ une seule ligne de cache par matrice

Toutes ces améliorations exploitent peu la localité temporelle !!!

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

- complexité en temps identique: $(\frac{n}{b})^3 * 2b^3 = O(n^3)$

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

- complexité en temps identique: $(\frac{n}{b})^3 * 2b^3 = O(n^3)$
- les données de `blockMatrixMul`
  - tiennent en cache
  - mais sont contigües par paquet $\Rightarrow$ b tableaux contigüs de b données

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

- complexité en temps identique: $(\frac{n}{b})^3 * 2b^3 = O(n^3)$
- les données de `blockMatrixMul`
    - tiennent en cache
    - mais sont contigües par paquet $\Rightarrow$ b tableaux contigüs de b données

Combien de cache miss pour lire un bloc $b \times b$?

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits ($b \times b$)

- complexité en temps identique: $(\frac{n}{b})^3 * 2b^3 = O(n^3)$
- les données de `blockMatrixMul`
  - tiennent en cache
  - mais sont contigües par paquet $\Rightarrow$ b tableaux contigüs de b données

Combien de cache miss pour lire un bloc $b \times b$? $O(b^2/B)$ cache miss

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

- complexité en temps identique: $(\frac{n}{b})^3 * 2b^3 = O(n^3)$
- les données de `blockMatrixMul`
  - tiennent en cache
  - mais sont contigües par paquet $\Rightarrow$ b tableaux contigüs de b données

Combien de cache miss pour lire un bloc $b \times b$? $O(b^2/B)$ cache miss

$\Rightarrow$ $B^2 < cM$ (Tall cache) et $cM \leq b^2 \leq M/3$ pour une constante $c \leq 1/3$

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b,j:j+b], X[i:i+b,k:k+b], Y[k:k+b,j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

- complexité en temps identique: $(\frac{n}{b})^3 * 2b^3 = O(n^3)$
- les données de `blockMatrixMul`
  - tiennent en cache
  - mais sont contigües par paquet $\Rightarrow$ b tableaux contigüs de b données

Combien de cache miss pour lire un bloc $b \times b$? $O(b^2/B)$ cache miss

$\Rightarrow$ $B^2 < cM$ (Tall cache) et $cM \leq b^2 \leq M/3$ pour une constante $c \leq 1/3$

$\Rightarrow$ $b(\lceil b/B \rceil + 1)$ lignes de cache, donc $\mathrm{MT}(b) = b^2/B + 2b \leq b^2/B + 2bB/B < 3b^2/B$

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

- complexité en temps identique: $(\frac{n}{b})^3 \times 2b^3 = O(n^3)$

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

- complexité en temps identique: $(\frac{n}{b})^3 \times 2b^3 = O(n^3)$
- complexité en cache: $\mathrm{MT}(n) = (\frac{n}{b})^3 \times O(\frac{b^2}{B}) = O(\frac{n^3}{B\sqrt{M}})$

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits ($b \times b$)

- complexité en temps identique: $(\frac{n}{b})^3 \times 2b^3 = O(n^3)$
- complexité en cache: $\mathrm{MT}(n) = (\frac{n}{b})^3 \times O(\frac{b^2}{B}) = O(\frac{n^3}{B\sqrt{M}}) \Rightarrow$ optimal [Hong, Kung 1981]

Multiplication de matrices: meilleure localité temporelle

Découpage en bloc de taille $b \times b$ qui tiennent dans le cache: $3b^2 \leq M$

```
for i in range(0,n,b):
    for k in range(0,n,b):
        for j in range(0,n,b):
            blockMatrixMul(Z[i:i+b, j:j+b], X[i:i+b, k:k+b], Y[k:k+b, j:j+b])
```

On a $(\frac{n}{b})^3$ produits de matrices plus petits $(b \times b)$

- complexité en temps identique: $(\frac{n}{b})^3 \times 2b^3 = O(n^3)$
- complexité en cache: $\mathrm{MT}(n) = (\frac{n}{b})^3 \times O(\frac{b^2}{B}) = O(\frac{n^3}{B\sqrt{M}}) \Rightarrow$ optimal [Hong, Kung 1981]

Remarques

- nécessite un cache haut (tall cache): c'est le cas en pratique
- nécessite de connaitre la taille du cache: cache aware
- et si on ne connait pas la taille du cache ? $\Rightarrow$ cache oblivious

Multiplication de matrices: *cache oblivious*

Cache oblivious: aucune connaissance des caches

$\Rightarrow$ l'objectif est de tirer partie des caches de manière intrinsèque au niveau algorithmique

Multiplication de matrices: *cache oblivious*

Cache oblivious: aucune connaissance des caches

$\Rightarrow$ l'objectif est de tirer partie des caches de manière intrinsèque au niveau algorithmique

Diviser pour régner

- permet souvent d'avoir les meilleures complexités en temps: ex. tri de tableau
- la plupart du temps donne des algorithmes *cache oblivious*
  $\Rightarrow$ la taille décroit durant les appels récursifs et rentre en cache à un certain moment

Multiplication de matrices: *cache oblivious*

Cache oblivious: aucune connaissance des caches

$\Rightarrow$ l'objectif est de tirer partie des caches de manière intrinsèque au niveau algorithmique

Diviser pour régner

- permet souvent d'avoir les meilleures complexités en temps: ex. tri de tableau
- la plupart du temps donne des algorithmes *cache oblivious*
  $\Rightarrow$ la taille décroit durant les appels récursifs et rentre en cache à un certain moment

Ex: produit de matrice

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \times \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix} = \begin{pmatrix} X_{11}Y_{11} + X_{12}Y_{21} & X_{11}Y_{12} + X_{12}Y_{22} \\ X_{21}Y_{11} + X_{22}Y_{21} & X_{21}Y_{12} + X_{22}Y_{22} \end{pmatrix}$$

$\Rightarrow$ 8 produits de matrice de taille moitiée

Multiplication de matrices: *cache oblivious*

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \times \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix} = \begin{pmatrix} X_{11}Y_{11} + X_{12}Y_{21} & X_{11}Y_{12} + X_{12}Y_{22} \\ X_{21}Y_{11} + X_{22}Y_{21} & X_{21}Y_{12} + X_{22}Y_{22} \end{pmatrix}$$

```
def MultRecAdd(Z,X,Y):
    n=Z.nrows()
    if n==1 :
        Z[0,0] += X[0,0]*Y[0,0]
    else :
        S1, S2 = slice(0,n//2), slice(n//2,n)
        X11,X12,X21,X22 = X[S1,S1], X[S1,S2], X[S2,S1], X[S2,S2]
        Y11,Y12,Y21,Y22 = Y[S1,S1], Y[S1,S2], Y[S2,S1], Y[S2,S2]
        Z11,Z12,Z21,Z22 = Z[S1,S1], Z[S1,S2], Z[S2,S1], Z[S2,S2]
        MultRecAdd(Z11 , X11, Y11), MultRecAdd(Z11 , X12, Y21)
        MultRecAdd(Z12 , X11, Y12), MultRecAdd(Z12 , X12, Y22)
        MultRecAdd(Z21 , X21, Y11), MultRecAdd(Z21 , X22, Y21)
        MultRecAdd(Z22 , X21, Y12), MultRecAdd(Z22 , X22, Y22)
        Z[S1,S1], Z[S1,S2], Z[S2,S1], Z[S2,S2]= Z11,Z12,Z21,Z22
```

Multiplication de matrices: *cache oblivious*

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \times \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix} = \begin{pmatrix} X_{11}Y_{11} + X_{12}Y_{21} & X_{11}Y_{12} + X_{12}Y_{22} \\ X_{21}Y_{11} + X_{22}Y_{21} & X_{21}Y_{12} + X_{22}Y_{22} \end{pmatrix}$$

```
def MultRecAdd(Z,X,Y):
    n=Z.nrows()
    if n==1 :
        Z[0,0] += X[0,0]*Y[0,0]
    else :
        S1, S2 = slice(0,n//2), slice(n//2,n)
        X11,X12,X21,X22 = X[S1,S1], X[S1,S2], X[S2,S1], X[S2,S2]
        Y11,Y12,Y21,Y22 = Y[S1,S1], Y[S1,S2], Y[S2,S1], Y[S2,S2]
        Z11,Z12,Z21,Z22 = Z[S1,S1], Z[S1,S2], Z[S2,S1], Z[S2,S2]
        MultRecAdd(Z11 , X11, Y11), MultRecAdd(Z11 , X12, Y21)
        MultRecAdd(Z12 , X11, Y12), MultRecAdd(Z12 , X12, Y22)
        MultRecAdd(Z21 , X21, Y11), MultRecAdd(Z21 , X22, Y21)
        MultRecAdd(Z22 , X21, Y12), MultRecAdd(Z22 , X22, Y22)
        Z[S1,S1], Z[S1,S2], Z[S2,S1], Z[S2,S2]= Z11,Z12,Z21,Z22
```

Complexité en temps: $T(n) = 8T(n/2)$ avec $T(1) = 2$

$\Rightarrow \quad T(n) = 8^{\log_2(n)} \times T(1) = 2n^3$

Multiplication de matrices: *cache oblivious*

$$\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \times \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix} = \begin{pmatrix} X_{11}Y_{11} + X_{12}Y_{21} & X_{11}Y_{12} + X_{12}Y_{22} \\ X_{21}Y_{11} + X_{22}Y_{21} & X_{21}Y_{12} + X_{22}Y_{22} \end{pmatrix}$$

```
def MultRecAdd(Z,X,Y):
    n=Z.nrows()
    if n==1 :
        Z[0,0] += X[0,0]*Y[0,0]
    else :
        S1, S2 = slice(0,n//2), slice(n//2,n)
        X11,X12,X21,X22 = X[S1,S1], X[S1,S2], X[S2,S1], X[S2,S2]
        Y11,Y12,Y21,Y22 = Y[S1,S1], Y[S1,S2], Y[S2,S1], Y[S2,S2]
        Z11,Z12,Z21,Z22 = Z[S1,S1], Z[S1,S2], Z[S2,S1], Z[S2,S2]
        MultRecAdd(Z11 , X11, Y11), MultRecAdd(Z11 , X12, Y21)
        MultRecAdd(Z12 , X11, Y12), MultRecAdd(Z12 , X12, Y22)
        MultRecAdd(Z21 , X21, Y11), MultRecAdd(Z21 , X22, Y21)
        MultRecAdd(Z22 , X21, Y12), MultRecAdd(Z22 , X22, Y22)
        Z[S1,S1], Z[S1,S2], Z[S2,S1], Z[S2,S2]= Z11,Z12,Z21,Z22
```

Complexité en temps: $T(n) = 8T(n/2)$ avec $T(1) = 2$

$\Rightarrow$ $T(n) = 8^{\log_2(n)} \times T(1) = 2n^3$ comme l'algo iteratif !!!

Multiplication de matrices: *cache oblivious*

Comme avec le modèle *cache aware*, on sait que si $n^2 \leq M/3$ alors $\mathrm{MT}(n) = O(n^2/B)$.
Par conséquent on a la récurrence suivante:

$$\mathrm{MT}(n) = \begin{cases} O(n^2/B) & \text{si } n^2 \leq M/3 \\ 8 \times \mathrm{MT}(n/2) & \text{sinon} \end{cases}$$

Multiplication de matrices: *cache oblivious*

Comme avec le modèle *cache aware*, on sait que si $n^2 \leq M/3$ alors $\mathrm{MT}(n) = O(n^2/B)$.
Par conséquent on a la récurrence suivante:

$$\mathrm{MT}(n) = \begin{cases} O(n^2/B) & \text{si } n^2 \leq M/3 \\ 8 \times \mathrm{MT}(n/2) & \text{sinon} \end{cases}$$

$\Rightarrow \quad \mathrm{MT}(n) = 8^{\log_2(n) - \log_2(\sqrt{M/3})}$

Multiplication de matrices: *cache oblivious*

Comme avec le modèle *cache aware*, on sait que si $n^2 \leq M/3$ alors $\mathrm{MT}(n) = O(n^2/B)$.
Par conséquent on a la récurrence suivante:

$$\mathrm{MT}(n) = \begin{cases} O(n^2/B) & \text{si } n^2 \leq M/3 \\ 8 \times \mathrm{MT}(n/2) & \text{sinon} \end{cases}$$

$$\Rightarrow \quad \mathrm{MT}(n) = 8^{\log_2(n) - \log_2(\sqrt{M/3})} = \frac{2^{3\log_2(n)}}{2^{3/2\log_2(M/3)}}$$

Multiplication de matrices: *cache oblivious*

Comme avec le modèle *cache aware*, on sait que si $n^2 \leq M/3$ alors $\mathrm{MT}(n) = O(n^2/B)$.
Par conséquent on a la récurrence suivante:

$$\mathrm{MT}(n) = \begin{cases} O(n^2/B) & \text{si } n^2 \leq M/3 \\ 8 \times \mathrm{MT}(n/2) & \text{sinon} \end{cases}$$

$$\Rightarrow \quad \mathrm{MT}(n) = 8^{\log_2(n) - \log_2(\sqrt{M/3})} = \frac{2^{3\log_2(n)}}{2^{3/2\log_2(M/3)}} = O\left(\frac{n^3}{B\sqrt{M}}\right)$$

Multiplication de matrices: *cache oblivious*

Comme avec le modèle *cache aware*, on sait que si $n^2 \leq M/3$ alors $\mathrm{MT}(n) = O(n^2/B)$.
Par conséquent on a la récurrence suivante:

$$\mathrm{MT}(n) = \begin{cases} O(n^2/B) & \text{si } n^2 \leq M/3 \\ 8 \times \mathrm{MT}(n/2) & \text{sinon} \end{cases}$$

$$\Rightarrow \quad \mathrm{MT}(n) = 8^{\log_2(n) - \log_2(\sqrt{M/3})} = \frac{2^{3\log_2(n)}}{2^{3/2\log_2(M/3)}} = O\left(\frac{n^3}{B\sqrt{M}}\right)$$

Remarques

- même complexité qu'avec le modèle *cache aware* (optimal)
- pas besoin de connaître la taille du cache, l'algorithme est auto-adaptatif
- la gestion de plusieurs niveaux de cache est implicite