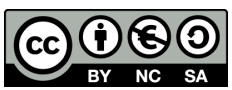
IDRIS

SIMD

Dimitri.Lecas@idris.fr

CNRS — IDRIS

v1.3 24 Octobre 2024



Plan I

Plan II

Plan

Introduction

A propos

Ce document est mis à jour régulièrement. La version la plus récente est disponible sur le site Web de l'IDRIS : http://www.idris.fr/formations/simd/

- IDRIS
  Institut du développement et des ressources en informatique scientifique
  Rue John Von Neumann
  Bâtiment 506
  BP 167
  91403 ORSAY CEDEX
  France
  http://www.idris.fr

SIMD

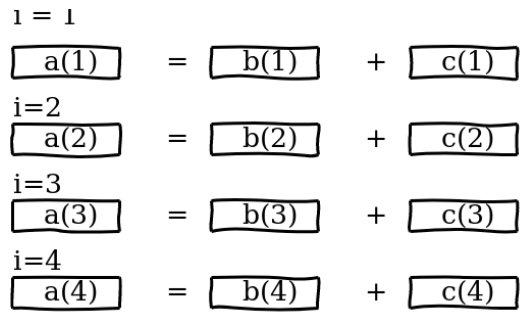
- SIMD pour Single Instruction on Multiple Data ;
- Faire une même instruction sur plusieurs données en même temps (vecteurs) ;
- C'est un niveau de parallélisme disponible dans les cœurs ;
- Seules quelques instructions ont une version vectorielle ;
- Les instructions vectorielles vont aussi vite que les instructions scalaires.

Version Scalaire

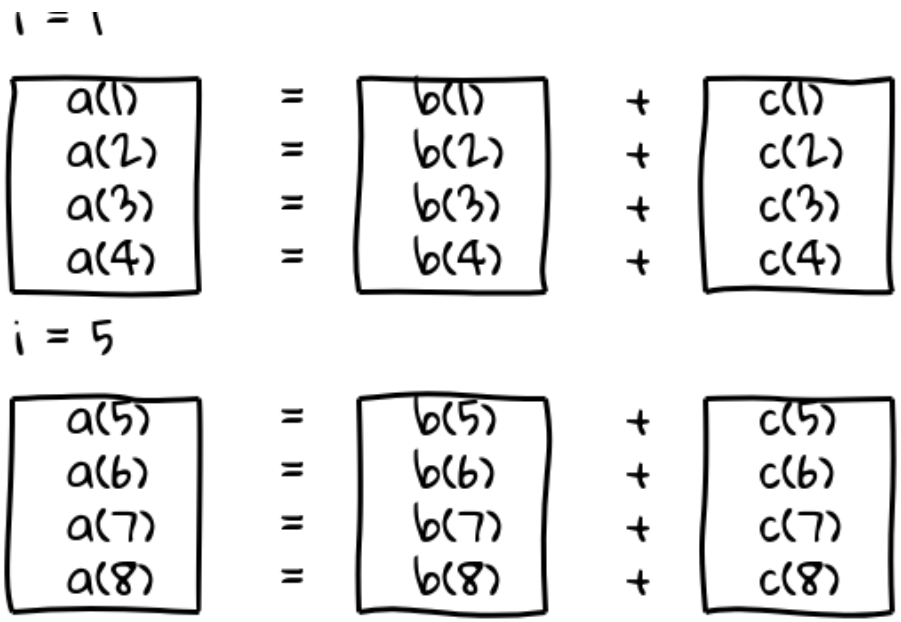
Si le code est le suivant :

```
do i=1,n
  a(i) = b(i) + c(i)
end do
```

Sans vectorisation, les instructions se font une par une :



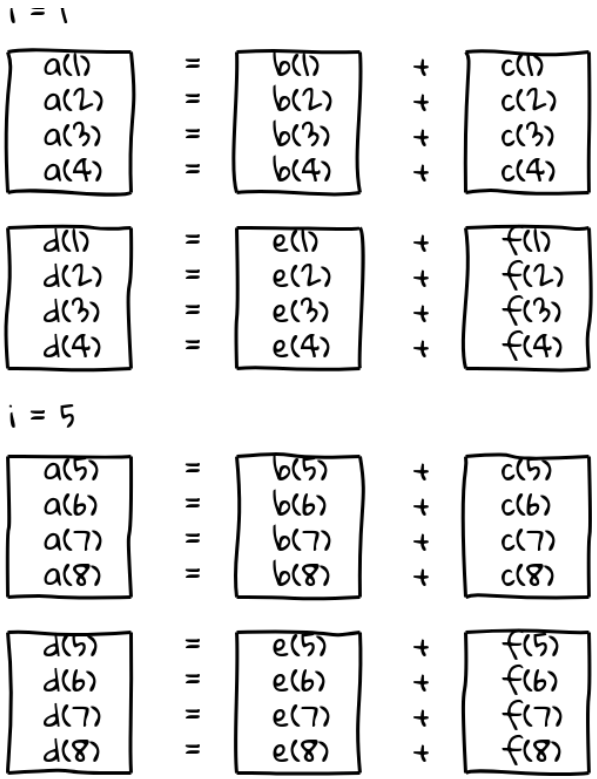
En activant la vectorisation, les instructions se feront groupées :



Version vectorisée

Si il y a plusieurs instructions :

```
do i=1,n
  a(i) = b(i)+c(i)
  d(i) = e(i)+f(i)
end do
```



La vectorisation change l'ordre d'exécution. Le compilateur s'assure que cela est sans danger.

Pourquoi la vectorisation

- Les Ghz stagnent par contrainte énergétique ;
- La loi de moore est toujours d'actualité (doublement des transistors tous les 2 ans) ;
- Plusieurs possibilités pour augmenter les performances :
  - Augmenter le nombre de coeurs ;
  - Augmenter le SIMD (tailles des registres) ;
  - Améliorer l'architecture.

Taille des registres vectoriels

- Le nombre d'éléments qui peuvent être traités en même temps dépend de la taille des registres vectoriels ;
- Cette taille s'exprime en bit car les éléments peuvent être des entiers, des flottants (simple ou double précision).

Processeurs SIMD

- x86 (Intel, AMD) avec les technologies SSE, AVX, AVX512
- Power (IBM) avec Power ISA
- AArch64 (ARM, Fujistu) avec SVE
- Nec Vector Engine Processor
- RISC V

x86 SIMD

- MMX introduit par INTEL en 1997 et 3DNow ! introduit par AMD en 1998
- SSE (Streaming SIMD Extension) introduit en 1999 par INTEL
- AVX (Advanced Vector Extension) introduit en 2011 pour les processeurs Sandy Bridge
- AVX2 introduit en 2013 pour les processeurs Haswell
- AVX2 ajoute notamment le FMA (Fused multiply add)
- AVX512 introduit en 2015 pour les processeurs Knight Landing
- AVX512 utilise des registres 512 bits et le FMA.

Techno	Taille Registres	SP	DP
MMX	64 bits		
SSE	128 bits	4	2
AVX/AVX2	256 bits	8	4
AVX512	512 bits	16	8

Puissance théorique en DP

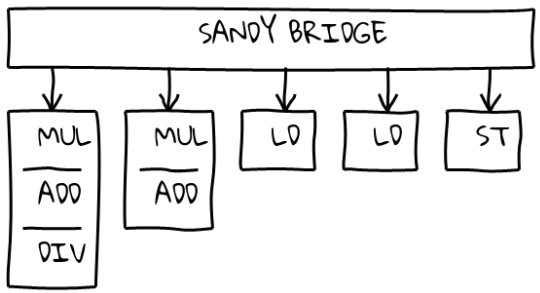
La puissance crête d'un cœur se calcul comme le produit des composants suivant :

- La fréquence vectorielle du cœur (peut être différente de la fréquence du cœur) ;
- Le nombre d'instruction flottante vectorielle par cycle (ex 1 addition et 1 multiplication) ;
- Le nombre d'opération flottante par instruction vectorielle (dépend de la taille des registres).

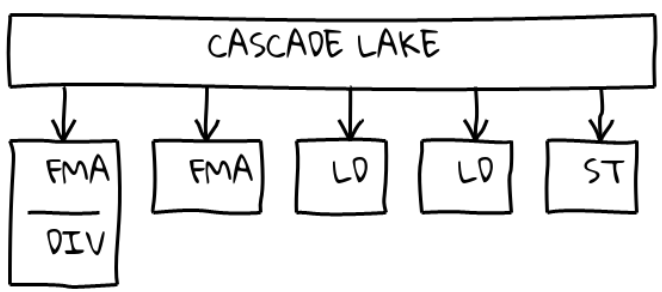
- Nehalem/Westmere (SSE) : $Ghz \times 2 \times 2$
- Sandy bridge/Ivy bridge (AVX) : $Ghz \times 2 \times 4$
- Haswell/Broadwell (AVX2) : $Ghz \times 4 \times 4$
- KNL/SkyLake/CascadeLake/...(AVX512) : $Ghz \times 4 \times 8$
- AMD EPYC Zen Naples($\sim$AVX2) : $Ghz \times 2 \times 4$
- AMD EPYC Zen 2 & 3 Rome & Milan(AVX2) : $Ghz \times 4 \times 4$
- AMD EPYC Zen 4 Genoa ($\sim$AVX512) : $Ghz \times 2 \times 8$
- Fujistu A64fx (SVE) : $Ghz \times 4 \times 8$

Puissance théorique IDRIS

- Ada était composé de SandyBridge qui supporte l'AVX ;
- On a 2.7 Ghz cycle par secondes ;
- Par cycle on peut faire 1 addition et 1 multiplication vectorielle ;
- L'AVX permet de traiter 4 opérations flottantes en double précisions ;
- Cela fait donc 2.7 $\times$ 2 $\times$ 4=21.6 GFLOPS (Double précisions).



- Jean Zay est composé de CascadeLake qui supporte l'AVX512 ;
- On a 2.5 Ghz cycle par secondes ;
- Par cycle on peut faire 2 addition et multiplication vectorielle ;
- L'AVX512 permet de traiter 8 opérations flottantes en double précisions ;
- Cela fait donc 2.5 $\times$ 4 $\times$ 8=80 GFLOPS (Double précisions).



3 méthodes pour faire de la vectorisation

- Via le compilateur :
  - Laissant faire le compilateur (les bonnes options) ;
  - En aidant le compilateur avec des directives (OpenMP ou autres) ;
- Via les intrinsic.

Compilateur INTEL ifort

- Par défaut le compilateur INTEL est en -O2 qui implique -xSSE2 ;
- Pour activer l'AVX il faut utiliser -xAVX (conseil rajoutez également -O3) ;
- -axAVX,SSE4.2 permet de faire un code optimisé pour différentes architectures, le choix se fera à l'exécution ;
- Pour activer l'AVX2, il faut utiliser -xCORE-AVX2 ;
- Pour activer l'AVX512, il faut utiliser -xCORE-AVX512 ;
- Pour activer l'AVX512 complétement, il faut utiliser -qopt-zmm-usage=high ;
- L'option -xHost permet d'utiliser l'option la plus adaptée au processeur sur lequel on compile ;
- Pour AMD EPYC Rome/Milan, il faut utiliser -mavx2, les options -x ne fonctionnent qu'avec les processeurs INTEL ;
- Pour AMD EPYC Genoa, il faut utiliser -mavx512, les options -x ne fonctionnent qu'avec les processeurs INTEL ;
- L'option -qopt-report=5 -qopt-report-phase=vec permet d'avoir un diagnostic de la vectorisation ;
- L'option -no-vec -no-simd -qno-openmp-simd désactive la vectorisation ;
- L'option -lm est a éviter car elle force l'utilisation des versions scalaires des fonctions mathématiques.

Compilateur INTEL ifx

- Par défaut le compilateur INTEL est en `-O2` qui implique `-xSSE2` ;
- Pour activer l'AVX il faut utiliser `-xAVX` (conseil rajoutez également `-O3`) ;
- `-axAVX,SSE4.2` permet de faire un code optimisé pour différentes architectures, le choix se fera à l'exécution ;
- Pour activer l'AVX2, il faut utiliser `-xCORE-AVX2` ;
- Pour activer l'AVX512, il faut utiliser `-xCORE-AVX512` ;
- Pour activer l'AVX512 complétement, il faut utiliser `-mprefer-vector-width=512` ;
- L'option `-xHost` permet d'utiliser l'option la plus adaptée au processeur sur lequel on compile ;
- Pour AMD EPYC Rome/Milan, il faut utiliser `-mavx2`, les options `-x` ne fonctionnent qu'avec les processeurs INTEL ;
- Pour AMD EPYC Genoa, il faut utiliser `-mavx512`, les options `-x` ne fonctionnent qu'avec les processeurs INTEL ;
- L'option `-qopt-report=3` permet d'avoir un diagnostic de la vectorisation ;
- L'option `-no-vec -qno-openmp-simd` désactive la vectorisation ;
- L'option `-lm` est a éviter car elle force l'utilisation des versions scalaires des fonctions mathématiques.

Compilateur gfortran

- La vectorisation s'active avec `-ftree-vectorize` ;
- Est activé par défaut avec `-O3` ;
- `-ffast-math` est conseillé pour gérer les réductions ;
- Pour activer l'AVX, il faut utiliser `-mavx` ;
- Pour activer l'AVX2, il faut utiliser `-mavx2` ;
- Pour activer l'AVX512, il faut utiliser
  `-mavx512f -mavx512dq -mavx512bw -mavx512vbmi -mavx512vbmi2 -mavx512vl` ;
- Pour activer l'AVX512 complétement, il faut utiliser en plus `-mprefer-vector-width=512` ;
- L'option `-march=native` permet d'utiliser l'option la plus adaptée au processeur sur lequel on compile ;
- L'option `-fopt-info-vec-all` donne des informations sur la vectorisation des boucles ;
- L'option `-fno-tree-vectorize` désactive la vectorisation.

Plan

Problème lors de la vectorisation par le compilateur

Le compilateur va vectoriser une boucle (la plus interne) si il peut s'assurer qu'il peut traiter les instructions par bloc (la taille des blocs dépendant de la taille des registres vectorielles).
Il faut donc éviter :

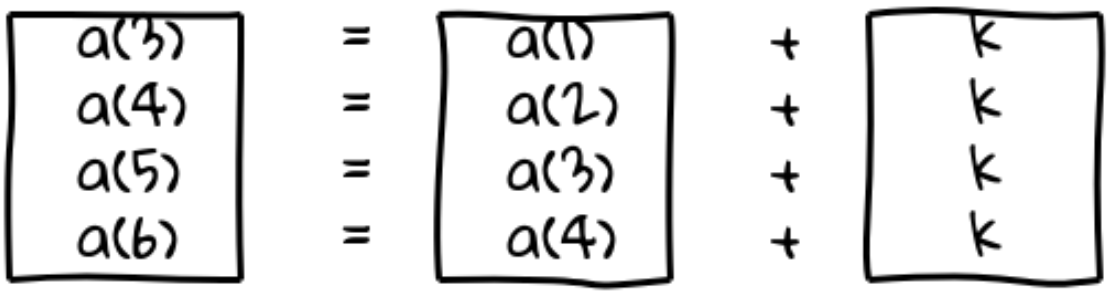
- les dépendances ;
- que le nombre d'itérations ne soit pas connu lors de l'exécution ;
- les tests.

Dépendances

Voici un exemple de dépendance :

```
do i=1,n-2
  a(i+2) = a(i)+k
end do
```

Si on vectorise on va avoir les instructions suivantes qui sont calculées en même temps.



Or a(3) et a(5) ne peuvent être calculées en même temps. Le compilateur ne vectorisera pas ou seulement partiellement. Un exemple de diagnostic du compilateur :

```
remark #15344: loop was not vectorized: vector dependence prevents vectorization
```

Dépendances - Définition

- Il y a dépendance quand dans une boucle deux instructions accèdent une même variable et l'une des instructions est une écriture pour cette variable.
- Read after write ou flow dependancy ex : A(i) = A(i−1)+1
- Write after read ou anti dependency ex : A(i) = A(i+1)+1
- Write after write ex : sum = sum+A(i)*B(i)
- Problème de l'aliasing

Aliasing

On parle d'aliasing lorsque plusieurs identifiants peuvent modifier une même donnée.
En Fortran c'est possible d'avoir ce genre de situation avec les pointeurs

```
subroutine f1(x,y)
  double precision, pointer, dimension(:), intent(in) :: x,y
  integer :: i
  do i=1, n
    x(i) = x(i)+y(i)
  end do
end subroutine

call f1(a(2:n),a(1:n-1))
```

En C il faut utiliser le mot clef restrict pour indiquer qu'il n'y a pas d'aliasing

```
void f1(double *restrict x, double *restrict y) {
  int i;
  for (i=0; i<n; i++) {
    x[i] = x[i]+y[i]; }
}
```

Appel de fonction

- Les appels aux routines empêchent la vectorisation ;
- Les fonctions empêchent aussi la vectorisation ;
- Le compilateur peut vectoriser quand il connaît le contenu de la fonction et qu'elle est simple.

```
function f1_dbl(x)
  double precision, intent(in) :: x
  double precision :: f1_dbl
  f1_dbl = cos(x*x+1.)/(x*x+1.)
end function

do i = 1, n
  Y(i) = f1_dbl(X(i))
end do
```

	Visible	Pas Visible
novec	11.25s	12.91s
sse	4.86s	12.82s
avx2	2.23s	11.99s
avx512	1.35s	12.14s

Nombre d'itération fixe lors de l'exécution

- Pas de while
- Pas d'exit dans une boucle
- Pas de goto

```
do i = 1, n
  if(X(i) < 0.) exit
  X(i) = sqrt(X(i))
end do
```

```
loop was not vectorized: loop with multiple exits cannot be vectorized
```

Test

Les tests introduisent des dépendances.

```
do i = 1, length
  c=0.0d0
  do j = 1, length
    if (j == i) then
      c = c+0.01d0*X(i)
    else
      v = X(j)-X(i)
      c = c+X(i)*(0.01d0*exp(-2.0d0*v*v)+0.02d0*v)
    end if
  end do
  Y(i) = Y(i)-c
end do
```

no vec	4.18s
SSE	4.17s
AVX2	3.88s
AVX512	3.89s

```
loop was not vectorized: vector dependence prevents vectorization
vector dependence: assumed FLOW dependence between c line 8 and c line 5
vector dependence: assumed ANTI dependence between c line 5 and c line 5
vector dependence: assumed FLOW dependence between c line 5 and c line 5
vector dependence: assumed ANTI dependence between c line 5 and c line 8
```

Test

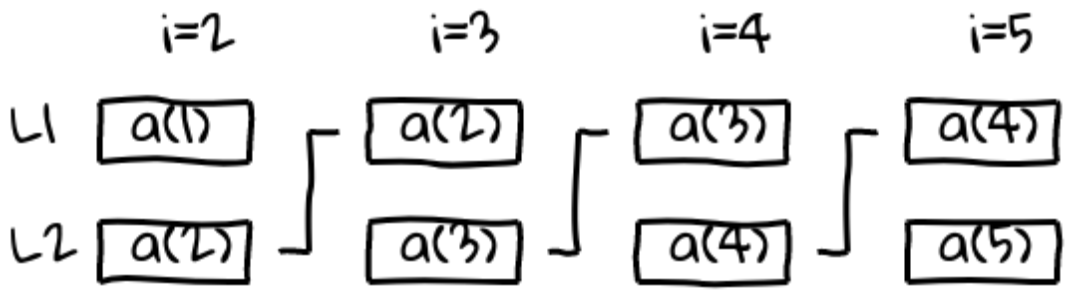
En enlevant le test, le code vectorise.

```
do i = 1, length
  c=0.0d0
  do j = 1, length
    v = X(j)-X(i)
    c = c+X(i)*(0.01d0*exp(-2.0d0*v*v)+0.02d0*v)
  end do
  Y(i) = Y(i)-c
end do
```

no vec	4.18s
SSE	4.13s
AVX2	2.24s
AVX512	1.01s

Contournement des dépendances

```
do i = 2, length
  A(i-1) = B(i)
  C(i) = A(i)
end do
```

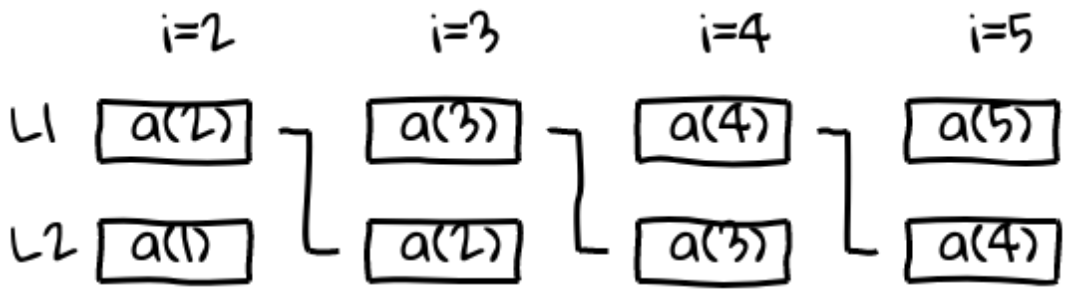


On a ici une dépendance qui empêche la vectorisation.

Contournement des dépendances

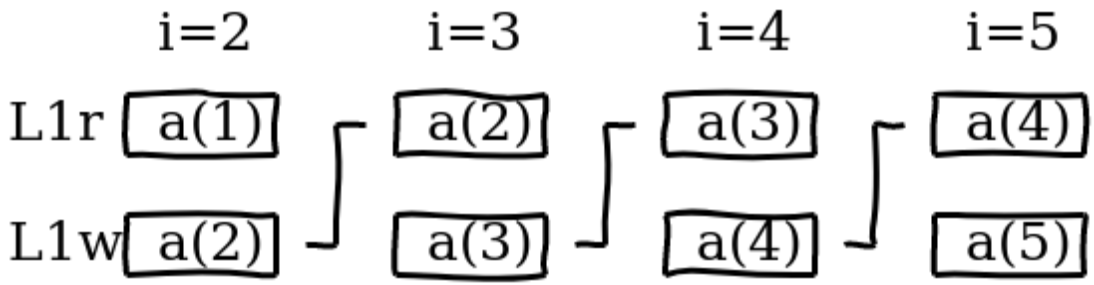
On peut inverser les lignes puisqu'il n'y a pas de dépendances à i fixé

```
do i = 2, length
  C(i) = A(i)
  A(i-1) = B(i)
end do
```

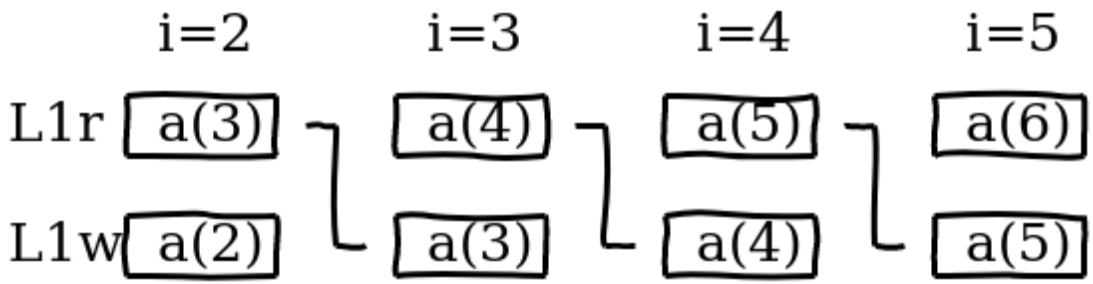


On a plus de dépendance, cette version vectorise.

```
do i=2, length
  a(i) = a(i-1)+c(i-1)
end do
```

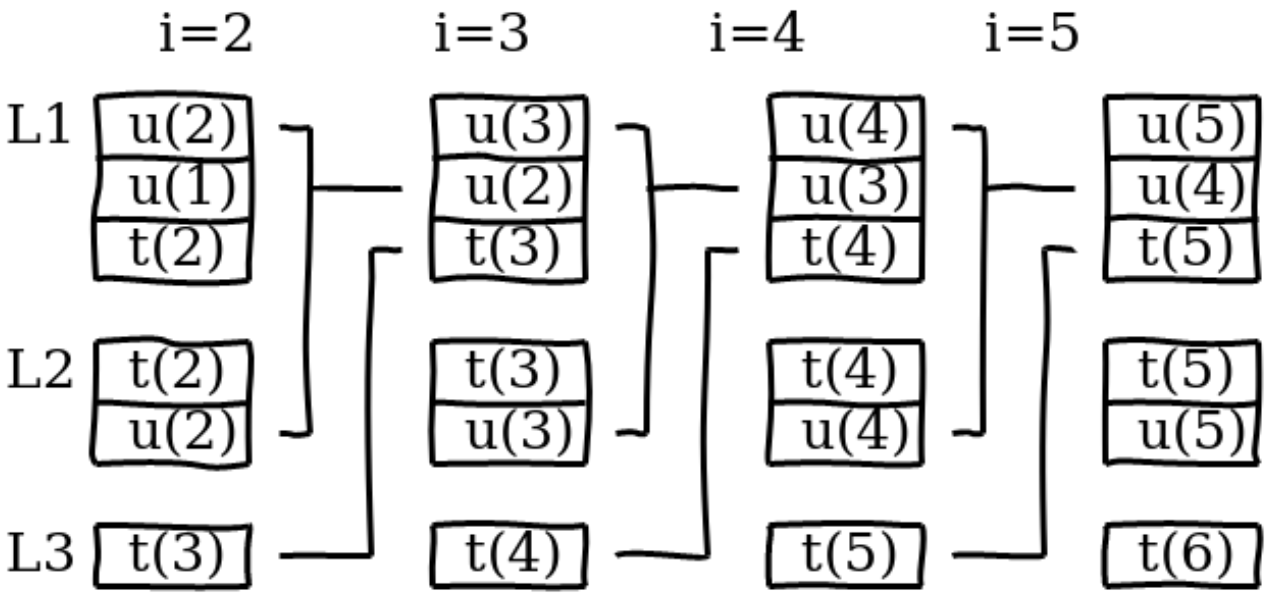



```
do i=2, length-1
  a(i) = a(i+1)+c(i-1)
end do
```



Contournement des dépendances

```
do i = 2, length-1
  t(i) = 0.5 * (u(i)-u(i-1))
  u(i) = t(i) * c(i)
  b(i) = EXP(-t(i+1)**2)
end do
```




```
do i = 2, length-1
  b(i) = EXP(-t(i+1)**2)
end do
do i = 2, length-1
  t(i) = 0.5 * (u(i)-u(i-1))
  u(i) = t(i) * c(i)
end do
```

Contournement des dépendances

Ici, il y a une dépendance (V1)

```
do i = 1, length
  x(i) = x(length/2)+y(i)
end do
```

On peut enlever la dépendance (V2)

```
do i = 1, length/2
  x(i) = x(length/2)+y(i)
end do
do i = length/2+1, length
  x(i) = x(length/2)+y(i)
end do
```

	V1	V2
novec	5.49s	6.85s
sse	5.47s	2.51s
avx2	5.49s	1.41s
avx512	5.49s	0.93s

Techniques de contournement

- Inversion de lignes
- Inversion de boucles
- Séparation des boucles
- Tableaux temporaires
- Restructuration des boucles

Conclusion

Pour écrire un code vectorisable :

- Préférer les boucles do ;
- Éviter les appels de fonctions ;
- Éviter les dépendances ;
- Éviter les pointeurs.

tp0 - Options compilateur

- Il faut regarder si le code vectorise.
- Faire module load intel−oneapi−all/2023.1 pour avoir le compilateur intel
- Pour compiler make

Organisation des répertoires des travaux pratiques

- `v00` contient les sources de départ (à ne pas modifier)
- `v01` contient le répertoire de travail
- `v99` contient la solution

Plan

OpenMP

- Le programmeur peut indiquer au compilateur qu'il est sans danger de vectoriser ;
- Depuis la version 4.0 de la norme OpenMP est disponible une nouvelle construction pour gérer la vectorisation ;
- Ces directives ne sont pas des conseils mais des ordres ;
- Avec le compilateur Intel, on peut activer uniquement ces directives SIMD avec l'option `-qopenmp-simd` (activé par défaut depuis la version 19.0) ;
- Avec le compilateur Gnu, c'est l'option `-fopenmp-simd`..

OpenMP SIMD

```
!$omp simd [clauses]
  do
    ...
  end do
[ !$omp end simd]
```

Cette construction va regrouper plusieurs itérations et les vectoriser.
Les clauses utilisables sont :

- private, lastprivate, reduction et collapse ;
- aligned(a:8,b:16) indique que les tableaux a et b sont alignés. a sur un multiple de 8 et b sur un multiple de 16 ;
- linear (j :pas) indique que j dépend linéairement du nombre d'itérations ;
- safelen(l) limite le nombre d'itérations vectorielles maximum à regrouper ;
- simdlen(l) indique le nombre d'itérations vectorielles souhaitées (OpenMP 4.5) ;
- if (test) active la vectorisation si test est vrai (OpenMP 5.0) ;
- nontemporal(a) desactive la mise en cache de a (OpenMP 5.0).

OpenMP Thread et SIMD

La construction omp do simd permet de fusionner les constructions omp do et omp simd en une fois :

```
!$omp do simd [clauses]
  do
  ...
  end do
[!$omp end do simd [nowait]]
```

- Les threads vont se distribuer les boucles, puis chaque thread va traiter les boucles de manière vectorielle.
- Attention au mode de répartition et à la taille des paquets : un schedule(static,1) est souvent un mauvais choix.
- Il est préférable d'utiliser le modificateur de clause simd qui permet d'arrondir les chunks au prochain multiple de la taille du registre vectorielle, ex : schedule(simd:static,chunk) (OpenMP 4.5).

Boucle interne

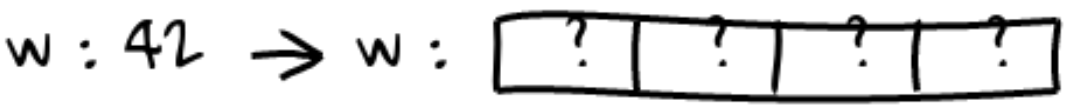
Pour vectoriser une boucle qui n'est pas la plus interne :

```
do j = 1, j_max
  y_0 = y_offset + dy_dj * j
  !$omp simd private(x_0,zx,zy,n,zx2,zy2)
  do i = 1, i_max
    x_0 = x_offset + dx_di * i

    zx = 0
    zy = 0
    n=0

    do
      zx2=zx*zx
      zy2=zy*zy
      if ( (zx2+zy2 > 4.0d+0) .or. (n == n_max) ) then
        image(i,j) = 255*(n*1.d+0)/n_max
        exit
      end if

      zy = y_0 + 2.0d+0 * zx*zy
      zx = x_0 + zx2 - zy2
      n= n+1
    end do
  end do
end do
```

OpenMP SIMD private

Exemple d'utilisation de la clause private :

```
!$omp simd private(w)
do i=1,n-1
  w=y(i)
  x(i) = w*y(i+1)
end do

! Alternative
do i=1,n-1
  block
    real(kind=dp) :: w
    w=y(i)
    x(i) = w*y(i+1)
  end block
end do
```



L'indice de boucle est automatiquement lastprivate

OpenMP SIMD reduction

Exemple d'utilisation des clauses reduction et collapse

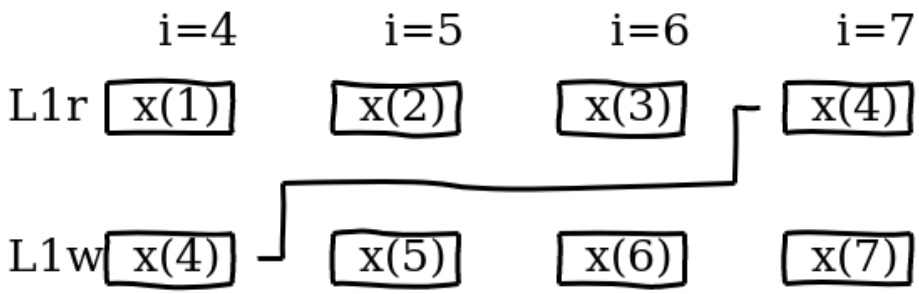
```
!$omp simd reduction(+:t) collapse(2)
do i=1,n
  do j=1,m
    t = t+a(i)*b(j)
  end do
end do
```

Les deux indices de boucles sont automatiquement lastprivate
Attention : La clause collapse augmente la complexité du code vectoriel

OpenMP SIMD safelen

Exemple d'utilisation de la clause safelen :

```
!$omp simd safelen(3)
do i=4,n
  x(i) = x(i-3)+y(i)
end do
```



OpenMP SIMD linear

Exemple d'utilisation de la clause linear :

```
!$omp simd linear(j:2)
do i=1,n
  a(i) = b(j) + c(i)
  j = j+2
end do
```

OpenMP SIMD nontemporal

Exemple d'utilisation de la clause nontemporal :

```
!$omp simd nontemporal(z)
do i=1,n
  z(i) = a*x(i)+y(i)
end do
```

z ne sera pas mis dans les caches de premier niveau

OpenMP Fonctions vectorisables

L'instruction omp declare simd permet d'indiquer qu'une fonction est vectorisable. Avec cette instruction, on permet la vectorisation même si on fait des appels de fonctions.

```
!$ omp declare simd[(function)] [clauses]
```

Les clauses utilisables sont :

- aligned.
- simdlen indique le nombre d'itérations souhaité.
- uniform(var) indique que var est constant dans les appels.
- inbranch, notinbranch indique si la fonction est utilisée ou pas dans des tests.
- linear.

Il est possible d'utiliser plusieurs déclarations avec des clauses différentes.

OpenMP Fonctions vectorisables exemple

```
function compsq(x,i) result(y)
!$omp declare simd uniform(x) linear(i) notinbranch
  double precision, dimension(:), contiguous :: x
  integer :: i
  double precision :: y

  y = x(i)*x(i)

end function compsq

...

!$omp simd
do i=1,n
  y(i) = compsq(x,i)
end do
```

OpenMP Fonctions vectorisables

- Le compilateur génère une version vectorisée de la fonction ;
- Les paramètres scalaires deviennent des vecteurs ;
- Les opérations scalaires deviennent des opérations vectorielles ;
- Si la fonction est appelée en dehors d'une région simd, c'est la variante scalaire qui est utilisé, sauf si la fonction est déclarée elemental et est utilisée avec des tableaux ;
- Pour avoir des fonctions vectorisées longues avec le compilateur Intel, il faut utiliser l'option –vecabi=cmdtarget

OpenMP Fonctions vectorisables - linear

- La clause `linear` a une signification différente pour les fonctions ;
- Elle indique une relation linéaire entre les appels de fonctions ;
- `linear(val(variable))` indique que la valeur est linéaire (passage par valeur) (OpenMP 4.5) ;
- `linear(uval(variable))` indique que la valeur est linéaire mais l'adresse est constante (passage d'indice de boucle) (OpenMP 4.5) ;
- `linear(ref(variable))` indique que l'adresse est linéaire (passage classique de tableau) (OpenMP 4.5) ;

OpenMP Fonctions vectorisables - linear exemple

```
elemental subroutine compsin(x,y)
!$omp declare simd linear(ref(x,y))
  double precision, intent(in) :: x
  double precision, intent(out) :: y

  y = 1.+sin(x)**3
end subroutine compsin

...

!$omp simd
do i=1,n
  call compsin(x(i),y(i))
end do

call compsin(x,y)
```

Fortran SIMD Explicit

- Il est possible de programmer la vectorisation de manière plus explicite ;
- Permet de palier des difficultés du compilateur ;
- Apporte une complexité plus grande du code sans garantie de performance.

```
do i = 1, n
  Y(i) = f(X(i))
end do
...
function f(x)
  double precision, intent(in) :: x
  double precision :: f
  f = cos(x*x+1.)/(x*x+1.)
end function
```

Fortran SIMD Explicit

```
integer, parameter :: SIMDNBDP=8
double precision, dimension(SIMDNBDP) :: v1,v2
simdloop = SIMDNBDP*(n/SIMDNBDP)
do i=1, simdloop,SIMDNBDP
  v1(1:SIMDNBDP) = X(i:i+SIMDNBDP-1)
  v2(1:SIMDNBDP) = Y(i:i+SIMDNBDP-1)
  call vf(v1,v2)
  Y(i:i+SIMDNBDP-1) = v2(1:SIMDNBDP)
end do
do i = simdloop,n
  Y(i) = f(X(i))
end do

subroutine vf(x,y)
  double precision, dimension(SIMDNBDP),intent(inout) :: x,y
  integer :: i
  !$omp simd
  do i=1,SIMDNBDP
    y(i) = cos(x(i)*x(i)+1.)/(x(i)*x(i)+1.)
  end do
end subroutine
```

	Inline	Declare SIMD	Explicit
novec	10.56s	11.61s	11.70s
sse	4.86s	6.73s	5.85s
avx2	2.34s	3.58s	3.03s
avx512	1.25s	2.12s	1.62s

tp1 - Pi SIMD

- Il s'agit de calculer π par intégration numérique $\pi = \int_0^1 \frac{4}{1+x^2} dx$.
- On utilise la méthode des rectangles (point milieu).
- La fonction à intégrer est $f(x) = \frac{4}{1+x^2}$.
- *nbbloc* est le nombre de points.
- $largeur = \frac{1}{nbbloc}$ est le pas de discrétisation et la largeur de chaque rectangle.
- Il faut regarder si le code vectorise.
- Vectoriser le code si cela n'est pas le cas.

tp1 - Pi SIMD

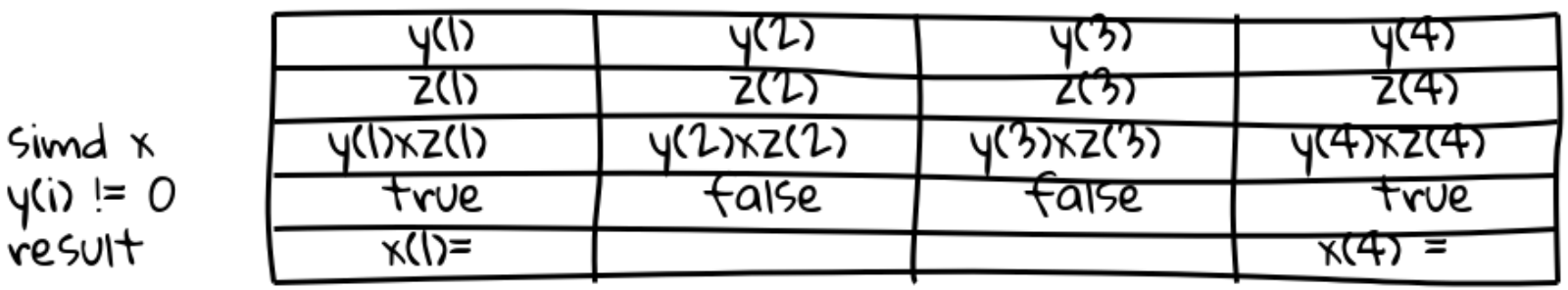
- Faire module load intel−oneapi−all/2023.1 pour avoir le compilateur intel.
- Pour compiler make
- Pour exécuter sbatch zay.slurm ou ./main
- Pour voir la liste des travaux en cours d'exécution squeue −u $USER

Plan

Test

Les tests à l'intérieur peuvent empêcher la vectorisation. Toutefois lorsque le corps du test est court et simple, la vectorisation peut se faire à l'aide d'instruction avec masque. Un masque est le résultat d'une comparaison vectorielle, c'est un vecteur de booléens. Le calcul sera vectoriel, l'affectation aussi mais en utilisant le masque (l'affectation se produira pour les indices qui sont a vrai dans le masque).

```
do i=1,n
  if (Y(i) .ne. 0.d0) then
    X(i) = Y(i)*Z(i)
  end if
end do
```



Test et Fortran SIMD Explicit

```
integer, parameter :: SIMDNBDP=8
double precision, dimension(SIMDNBDP) :: v1,v2
logical, dimension(SIMDNBDP) :: m

do i=1, n,SIMDNBDP
  !$omp simd
  do ii=0,SIMDNBDP-1
    m(ii+1) = ((i+ii<=n))
    if (m(ii+1)) then
      v1(ii+1) = X(i+ii)
      v2(ii+1) = Y(i+ii)
    end if
  end do
  call vf(v1,v2,m)
  !$omp simd
  do ii=0,SIMDNBDP-1
    if (m(ii+1)) then
      Y(i+ii) = v2(ii+1)
    end if
  end do
end do

subroutine vf(x,y,m)
  double precision, dimension(SIMDNBDP),intent(inout) :: x,y
  logical, dimension(SIMDNBDP) :: m
  integer :: i
  !$omp simd
  do i=1,SIMDNBDP
    if (m(i)) then
      y(i) = cos(x(i)*x(i)+1.)/(x(i)*x(i)+1.)
    end if
  end do
end subroutine
```

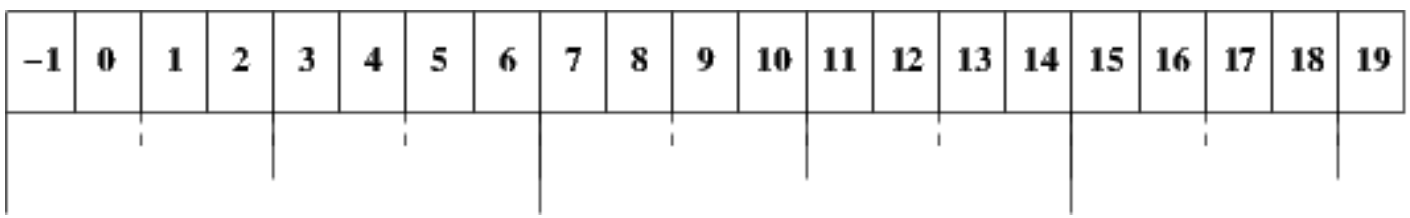
Alignement mémoire

Les processeurs transfèrent plus rapidement les données quand l'adresse de ces données sont un multiple d'un certain nombre d'octets.
Ce nombre correspond en général à la taille des registres SIMD.
Dans le cas où les données ne sont pas alignées le compilateur va faire la boucle en trois parties (peel loop, vector loop et remainder loop).

```
allocate(X(-1:n+2), Y(-1:n+2))
do i = 1, n
  X(i) = Y(i)+1
end do
```

Par exemple, si n = 19, le nombre d'indices traités dans chaque partie :

	Peel	Vector	Remainder	Iteration SIMD	Accélération
sse	0	18	1	10	1.9
avx2	2	16	1	6	3.2
avx512	6	8	5	5	3.8



Alignement mémoire

- L'optimal est que l'adresse de départ des données soit un multiple de la taille des registres vectoriels. (SSE 16 octets, AVX 32 octets, AVX512 64 octets) ;
- L'option `-align arraynbyte` du compilateur intel permet d'aligner les tableaux alloués dynamiquement ;
- Il peut être utile d'ajouter des éléments en début pour éviter les parties peel, surtout pour les tableaux à plusieurs dimensions ;
- La clause `aligned(a:64,b:32)` de la directive simd d'OpenMP permet d'indiquer les alignements.

Accès contiguës

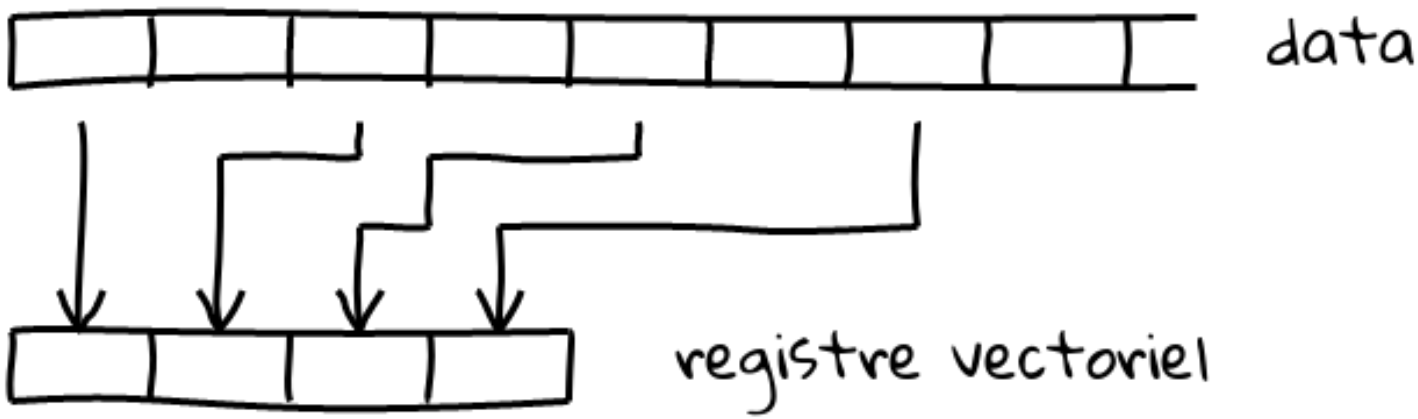
La vectorisation ne sera efficace que si on fait les mêmes calculs sur des données contiguës.

```
do i = 1, length, k
  X(i) = Y(i)+1
end do
```

k	1	2	3	4
novec	2093.73 GFLOPS	2051.61 GFLOPS	2006.83 GFLOPS	1970.60 GFLO
sse	4769.22 GFLOPS	2048.82 GFLOPS	2000.59 GFLOPS	1968.78 GFLO
avx2	8780.66 GFLOPS	1805.81 GFLOPS	1771.70 GFLOPS	1749.47 GFLO
avx512	16465.67 GFLOPS	5667.48 GFLOPS	1802.80 GFLOPS	1944.85 GFLO

Accès contiguës

Quand les données ne sont pas contiguës, des transferts mémoires sont nécessaires pour remplir les registres vectoriels.



Assume Shape Array

Les tableaux passés en argument avec dimension(:) peuvent empêcher la vectorisation car ces tableaux peuvent être des sections de tableau.

```
subroutine sub(X,Y)
  double precision, dimension(:) :: X,Y
  integer :: i

  do i=1, length
    X(i) = Y(i)+1
  end do
end subroutine
```

Le compilateur doit faire attention car il est possible de faire un appel call sub(X(1:100),Y(1:400:4)). Si on n'utilise pas de section de tableau, on peut utiliser l'attribut contiguous introduit dans la norme Fortran 2008.

```
subroutine sub(X,Y)
  double precision, dimension(:), contiguous :: X,Y
```

Attention, dans ce cas un appel call sub(X(1:100),Y(1:400:4)) générera des copies temporaires.

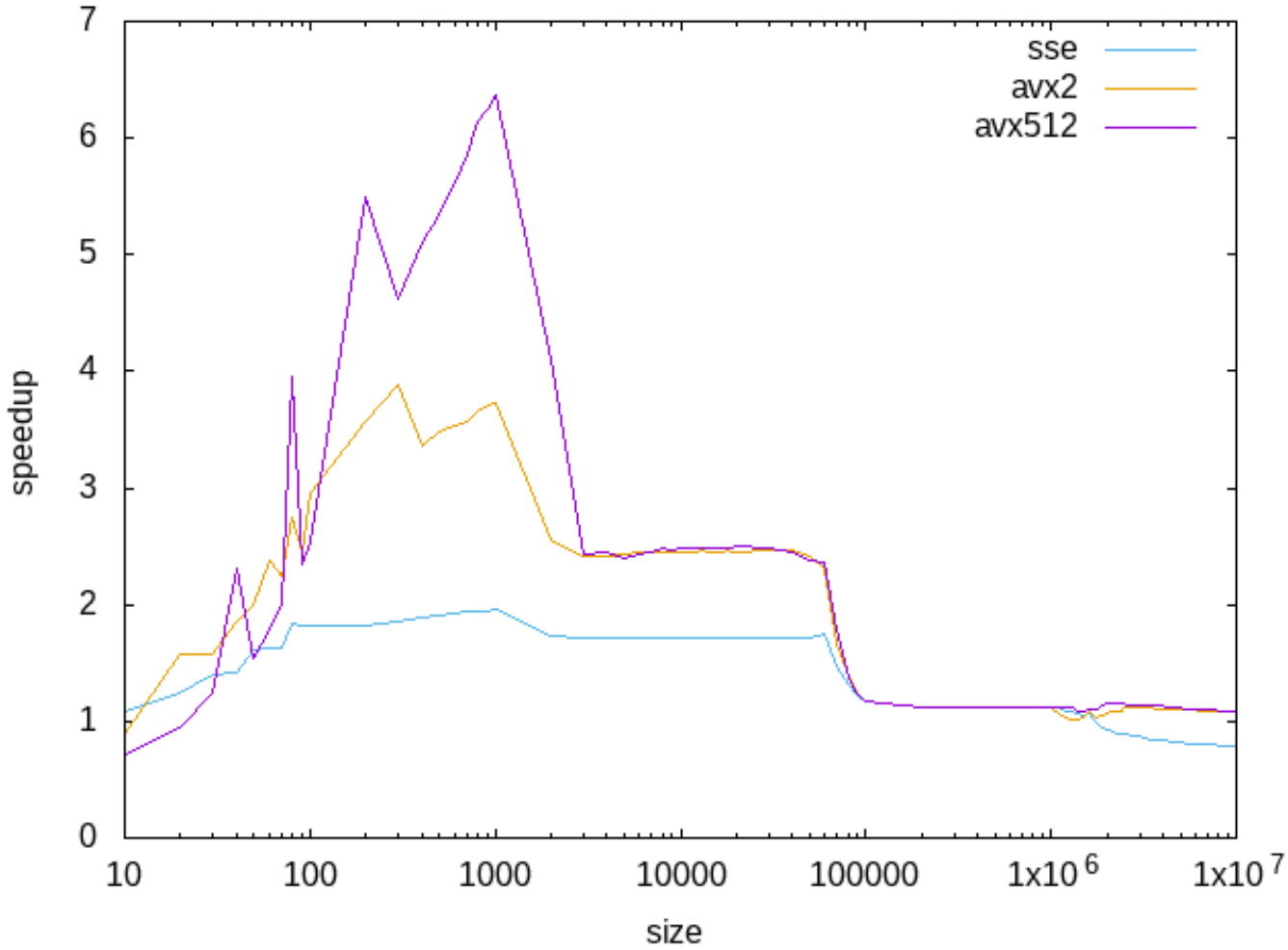
Gain - Test01

```
do i = 1, n
  X(i) = Y(i)+1
end do
```

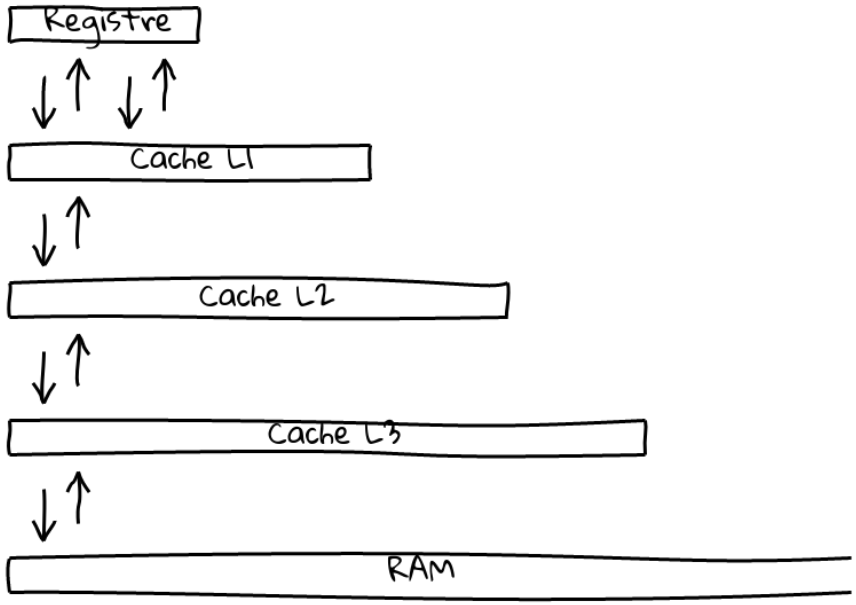
n	NoVec	SSE2	AVX2	AVX512
1024	4.09s	2.11s (x1.94)	1.12s (x3.65)	0.63s (x6.49)
10485760	13.54s	12.87s (x1.05)	12.44s (x1.09)	12.29s (x1.10)

Avec SSE2 on s'attend a avoir un x2 en double, avec AVX2 c'est normalement un x4 et avec AVX512 un x8.

Gain - Test02


sse
avx2
avx512
speedup
size
0
1
2
3
4
5
6
7
10
100
1000
10000
100000
1x10 6
1x10 7

Cache et Performance



Niveau	Taille	Latence
L1	32 ko	4 c
L2	1 Mo	14 c
L3	1,5 Mo	50-70 c
RAM	4 Go	200 c

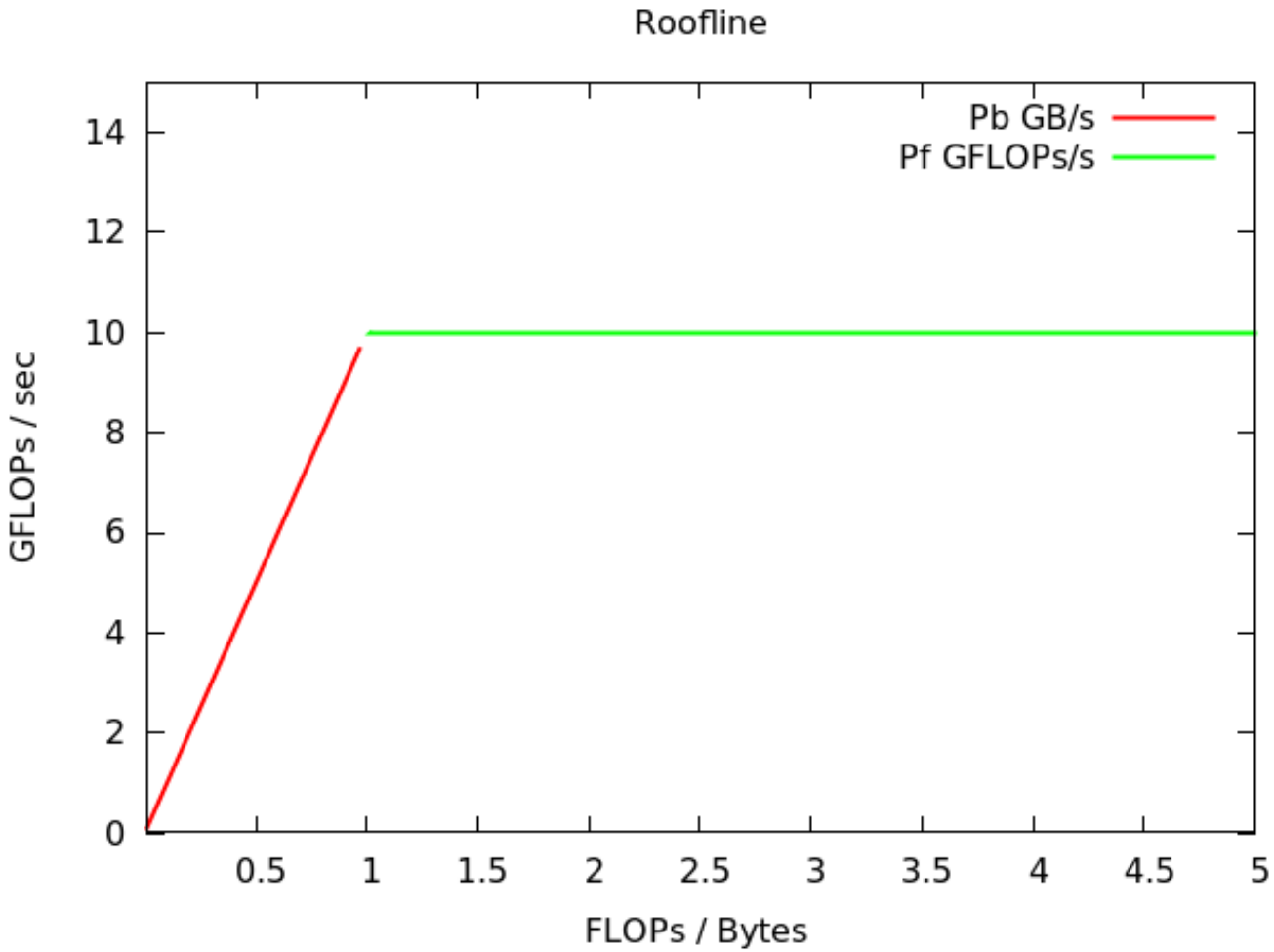
Roofline model - Présentation

- Modèle introduit en 2009 par Samuel Williams, Andrew Waterman, and David Patterson. *Roofline : an insightful visual performance model for multicore architectures*
- Modèle pour comprendre les performances limites.
- Basé sur la notion d'intensité arithmétique :

$$AI = \frac{FLOPS}{BYTES}$$

- Utilise le pic de la bande passante (Pb) et le pic FLOPS (Pf).
- $$GFLOPS = \frac{FLOP}{max(tmem, tcpu)} = \frac{FLOP}{max(\frac{BYTES}{pb}, \frac{FLOP}{pf})} = min(AI \cdot pb, pf)$$

Roofline


Roofline
Pb GB/s
Pf GFLOPs/s
GFLOPs / sec
FLOPs / Bytes
0
2
4
6
8
10
12
14
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

Roofline Jean Zay

- Sur Zay le pic FLOPS peut se calculer :

$$Pf = 2.5Ghz \cdot 4\frac{inst}{cycle}(fma + fma) \cdot 8\frac{DP}{inst}(avx512) = 80GFLOPS$$

  Mais la fréquence AVX2 et la fréquence AVX512 sont différentes :
  - 1,9 Ghz pour l'AVX2 (accélération de 3 plutôt que 4) ;
  - 1,6 Ghz pour l'AVX512 (accélération de 5.1 plutôt que 8).

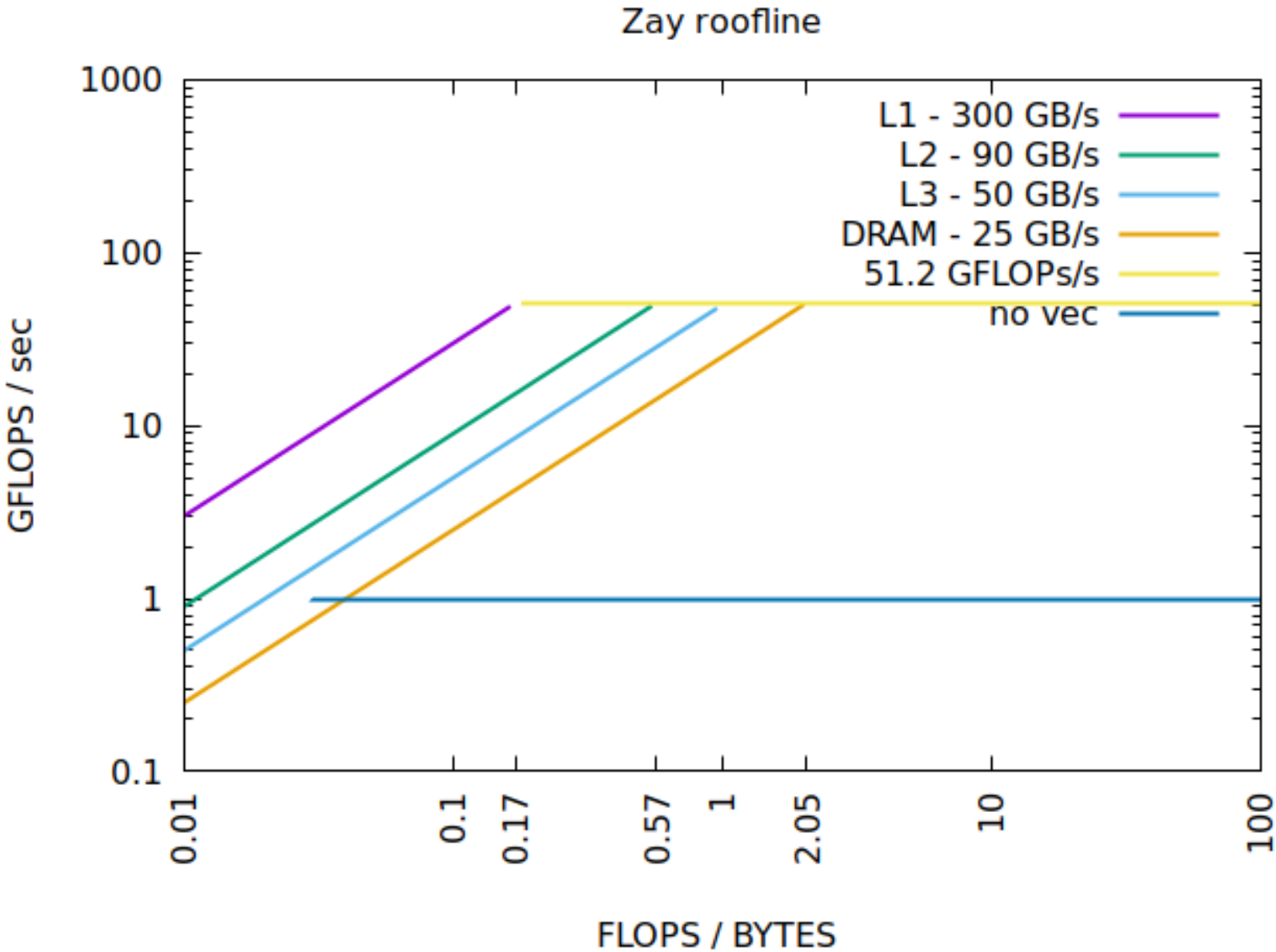
  Ce qui donne un pic FLOPS plutôt de l'ordre de 51.2 GFLOPS.
- Pour la bande passante, il faut utiliser des kernels :

$$Pb = 298.9GB/s(L1)$$

  et

$$Pb = 25.2GB/s(RAM)$$

Roofline Zay



Roofline Zay

- Pour Zay, les points d'intersection sont à 0.17 FLOPS/BYTES pour L1, ou 1.37 FLOPS/DP.
- Pour L2 c'est 0.57 FLOPS/BYTES ou 4.55 FLOPS/DP
- Pour L3 c'est 1.024 FLOPs/BYTES ou 8.192 FLOPS/DP
- Pour la DRAM c'est 2.048 FLOPs/BYTES ou 16.384 FLOPS/DP

Calcul de l'intensité arithmétique

- A la main
- Compteur hardware
- Intel Vector Advisor

Intensité arithmétique

```
do i = 1, n
  X(i) = Y(i)*Y(i)+Y(i)
end do
```

$$AI = \frac{2}{2 * 8} = 0.125$$

```
do i = 1, n
  X1(i) = Y1(i)+1
end do
```

$$AI = \frac{1}{2 * 8} = 0.0625$$

```
do i = 1, n
  X(perm(i)) = Y(perm(i))*Y(perm(i))+Y(perm(i))
end do
```

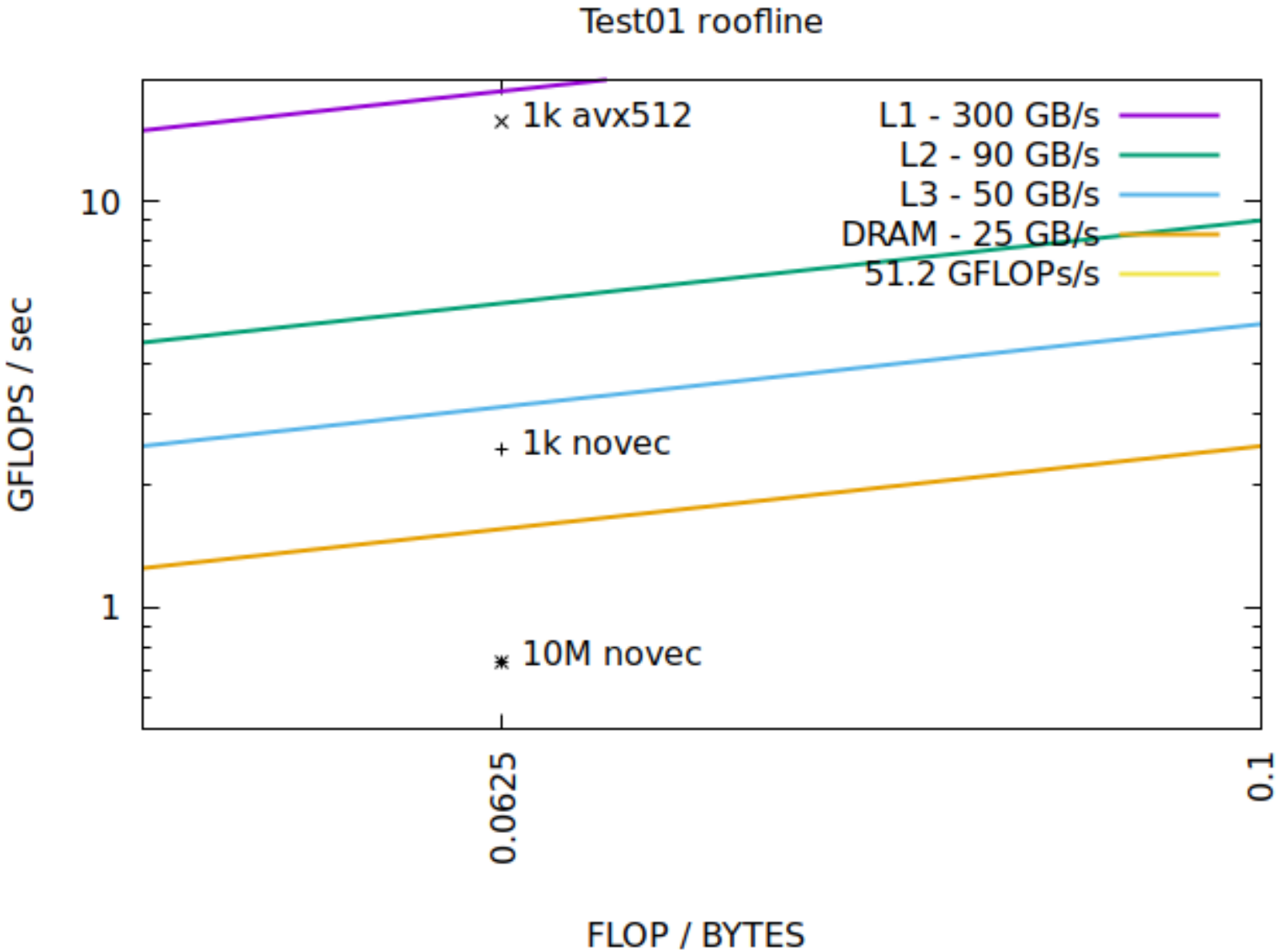
Au mieux

$$AI = \frac{2}{2 * 8} = 0.125$$

au pire

$$AI = \frac{2}{2 * 8 * 8} = 0.0156$$

Roofline Test01



Roofline : Raison qui éloigne du plafond

- Problèmes de vectorisation (mask, remainder, unaligned, no fma, division, sqrt)
- Instructions non flottante
- Accès non régulier à la mémoire
- Défauts de cache

Intensité arithmétique

On peut améliorer l'intensité arithmétique par la fusion de boucles

```
do i = 1, n
  a(i) = b(i)+c(i)
end do
do i = 1, n
  d(i) = b(i)+e(i)
end do
```

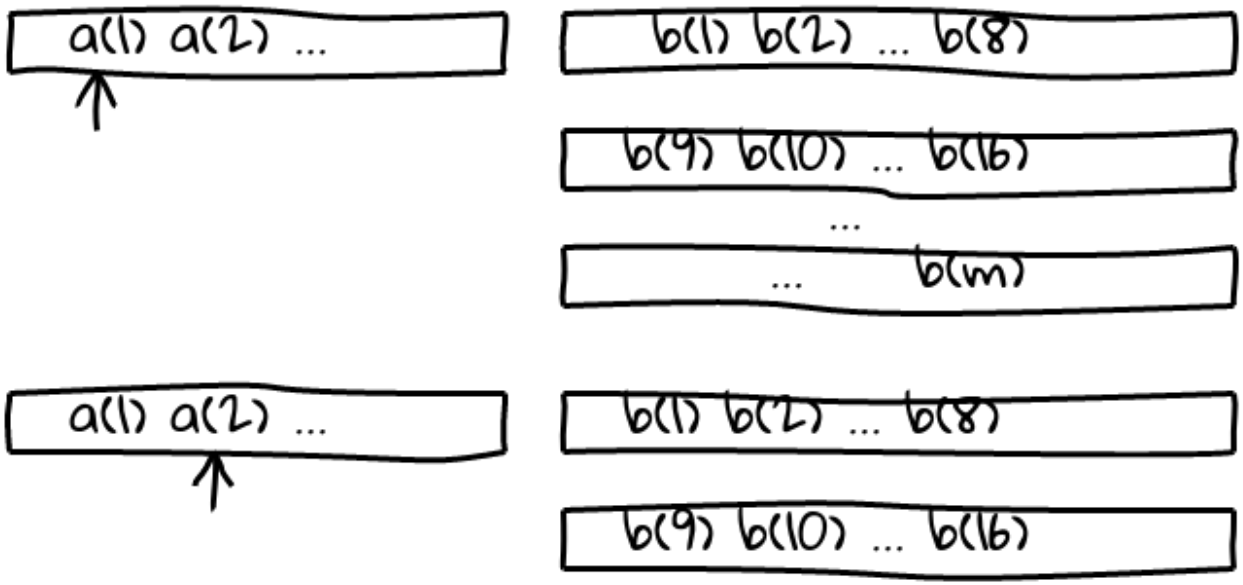
$$AI = \frac{2}{2 * 3 * 8} = 0.04167$$

```
do i = 1, n
  a(i) = b(i)+c(i)
  d(i) = b(i)+e(i)
end do
```

$$AI = \frac{2}{5 * 8} = 0.05$$

Optimisation des accès mémoires

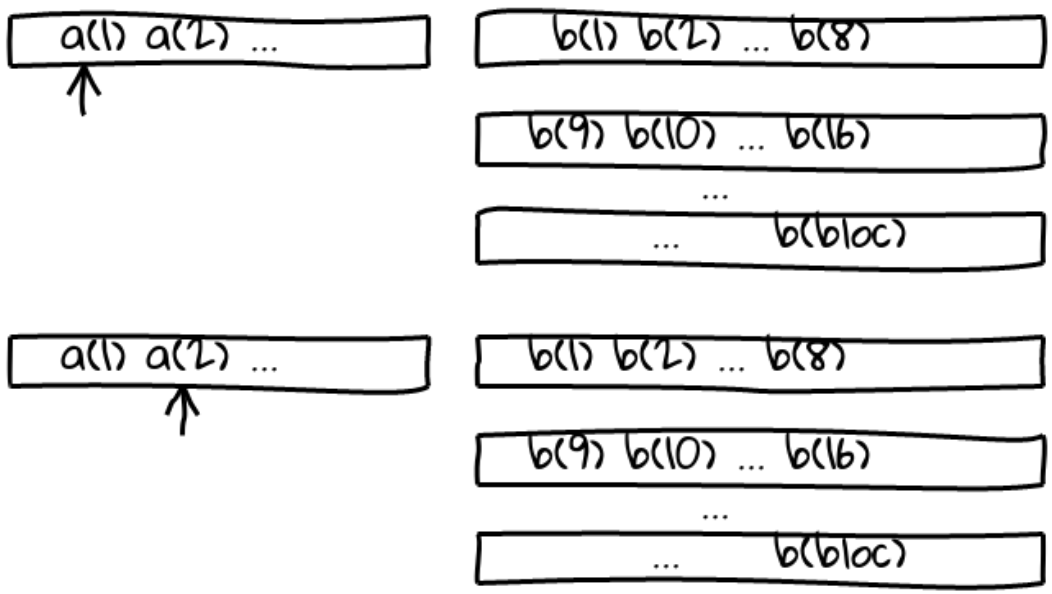
```
do i = 1, n
  do j = 1, m
    c = c + a(i) + b(j)
  end do
end do
```



$$AI = \frac{2 * m * n}{8 * (n + n * m)}$$

Optimisation des accès mémoires

```
do jb = 1,m,bloc
  do i = 1, n
    do j = jb, jb+bloc
      c = c + a(i) + b(j)
    end do
  end do
end do
```



$$AI = \frac{2 * m * n}{8 * (n + bloc) * (m/bloc)}$$

Optimisation des accès mémoires

Il existe de nombreuses techniques pour limiter les défauts de caches

- Tri, tableau temporaire, changement de structure de données
- Blocking et Loop tiling
- Cache oblivous algorithm

Gain : Autres limitations

- Fréquence vectorielle baisse avec la taille des registres utilisés
- Arithmétique des complexes

GFLOPS vs temps

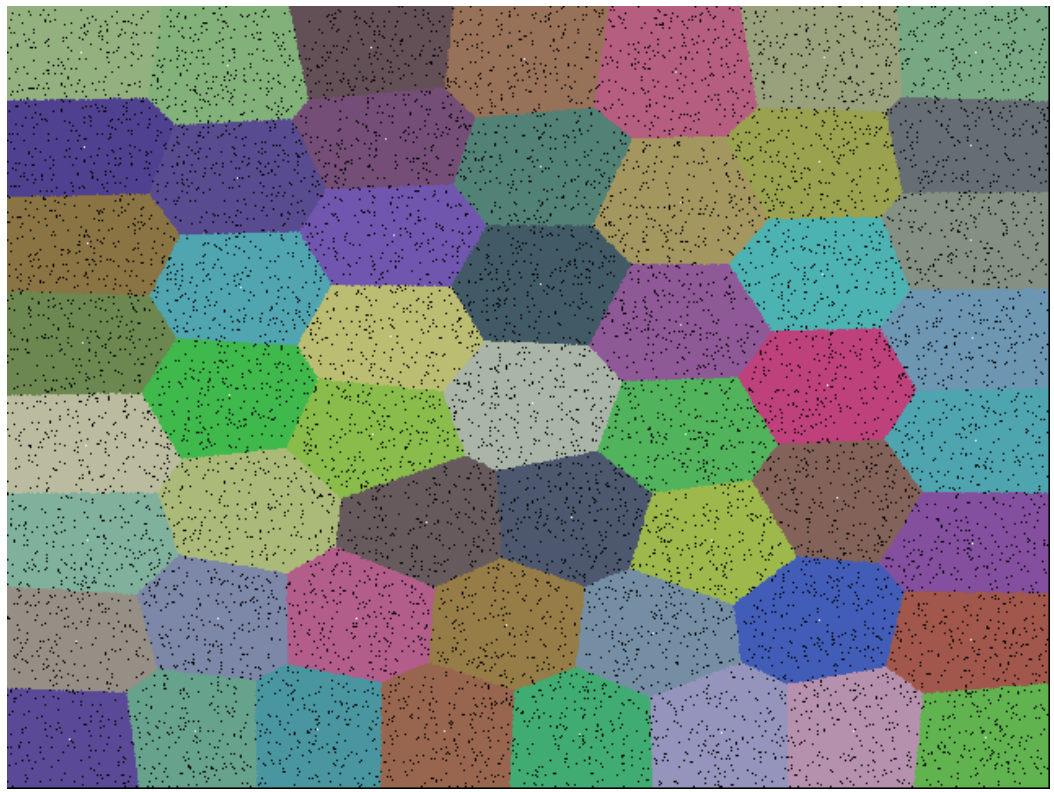
Attention : le roofline se focalise sur les GFLOPS
Ces deux codes ont quasiment les mêmes performances en GFLOPS et font le même calcul, mais le deuxième est bien plus rapide.

```
do j=1, n
  do i=1, n
    d(i) = d(i)+(a(i)*b(i))*c(j)
  end do
end do
```

```
do j=1, n
  sumc = sumc+c(j)
end do
do i=1, n
  d(i) = d(i)+(a(i)*b(i))*sumc
end do
```

tp2 - kmeans

- Le code implémente le partitionnement en k-moyennes.
- Le code vectorise mais les accès ne sont pas contigües.
- Comment faire pour minimiser les transferts mémoires.
- Attention à respecter l'ordre initial dans les initialisations avec le générateur aléatoire.



Plan

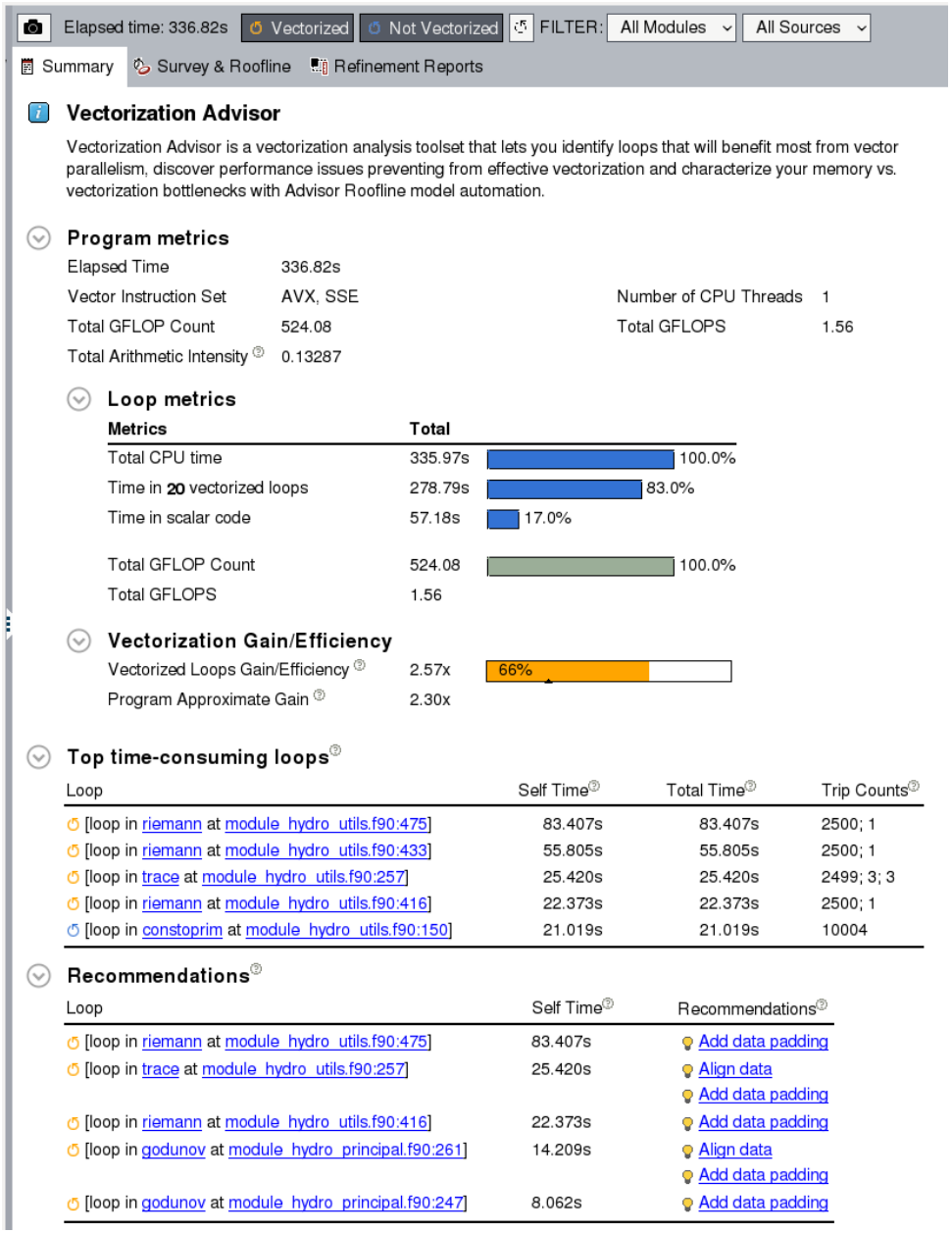
Intel Vector Advisor - Introduction

- Permet de voir si les boucles les plus importantes sont vectorisées ;
- Indique les problèmes éventuels de vectorisation ;
- Donne des conseils sur la résolution des problèmes ;
- Peut détecter les faux problèmes de dépendances ;
- Trace le roofline et la position des boucles.

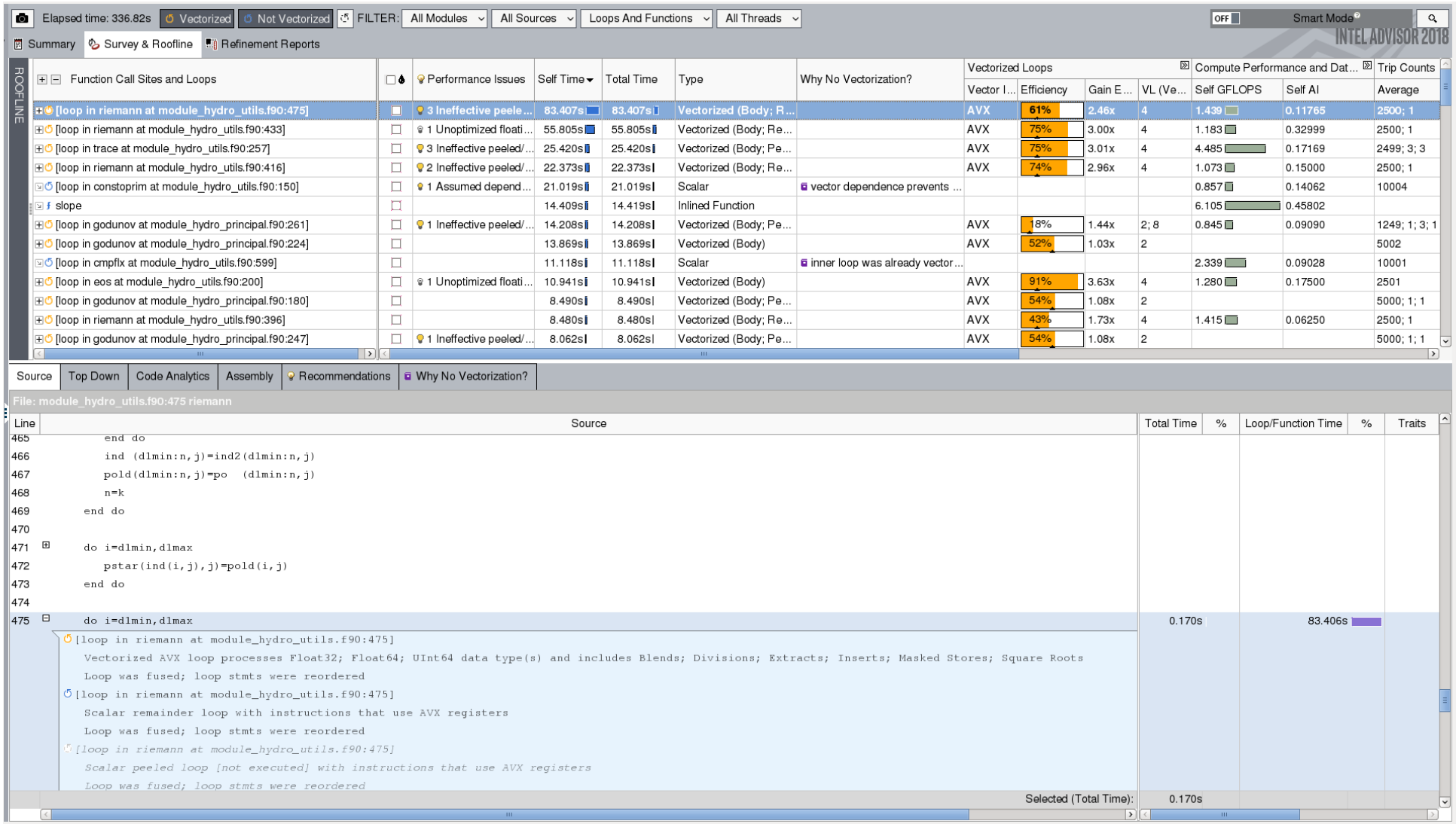
Intel Vector Advisor - Utilisation

- Compiler votre code avec l'option -g ;
- L'analyse se fait avec la commande advixe-cl --collect= ... exe ;
- --collect=survey fait une analyse générale rapide ;
- --collect=tripcounts fait une analyse profonde et permet d'afficher le roofline ;
- --collect=roofline fait les deux opérations précedente ;
- --collect=dependencies analyse les dépendances ;
- --collect=map analyse les accès mémoires ;
- L'analyse des résultats se fait avec l'interface graphique advixe-gui

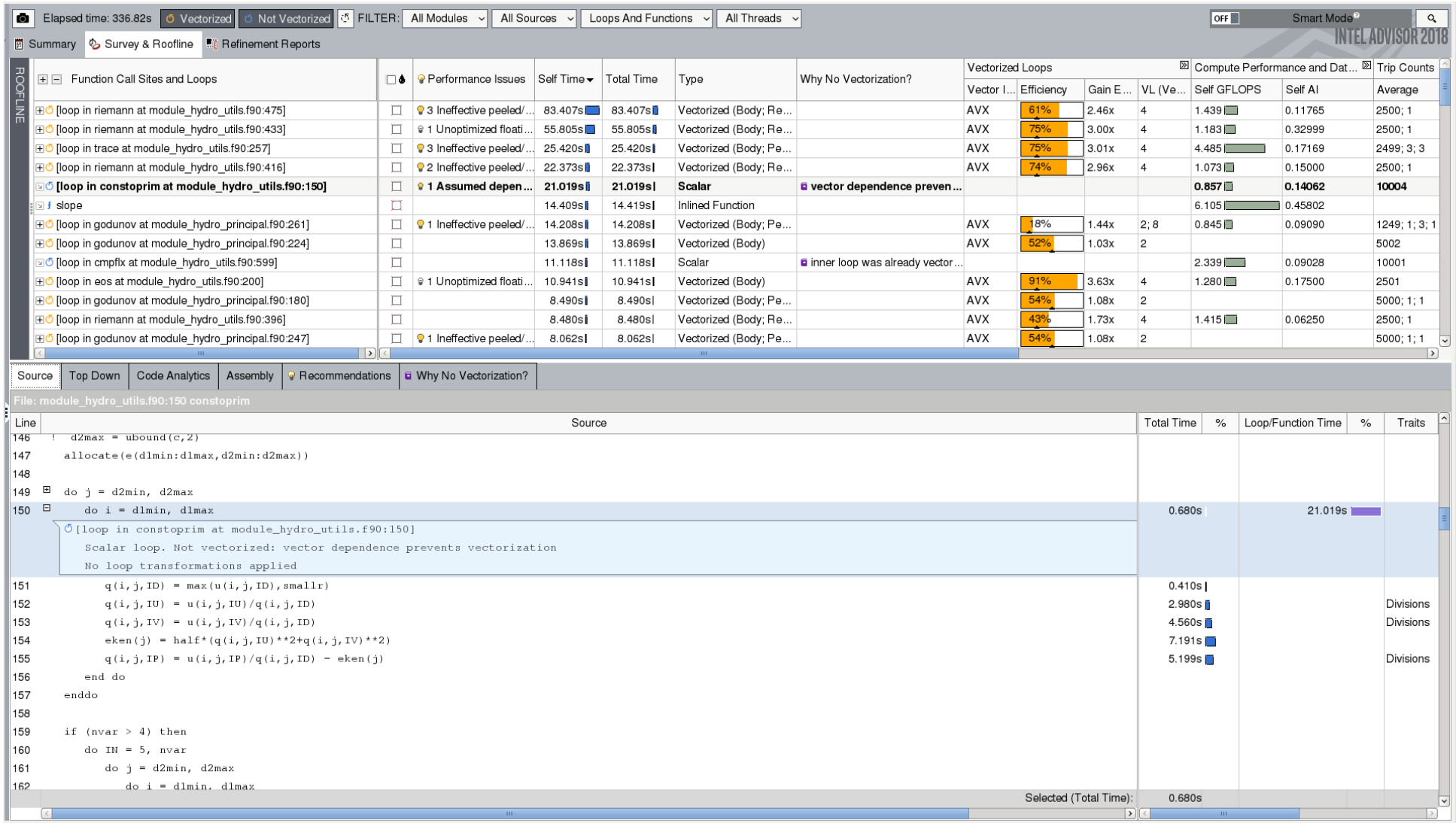
Intel Vector Advisor



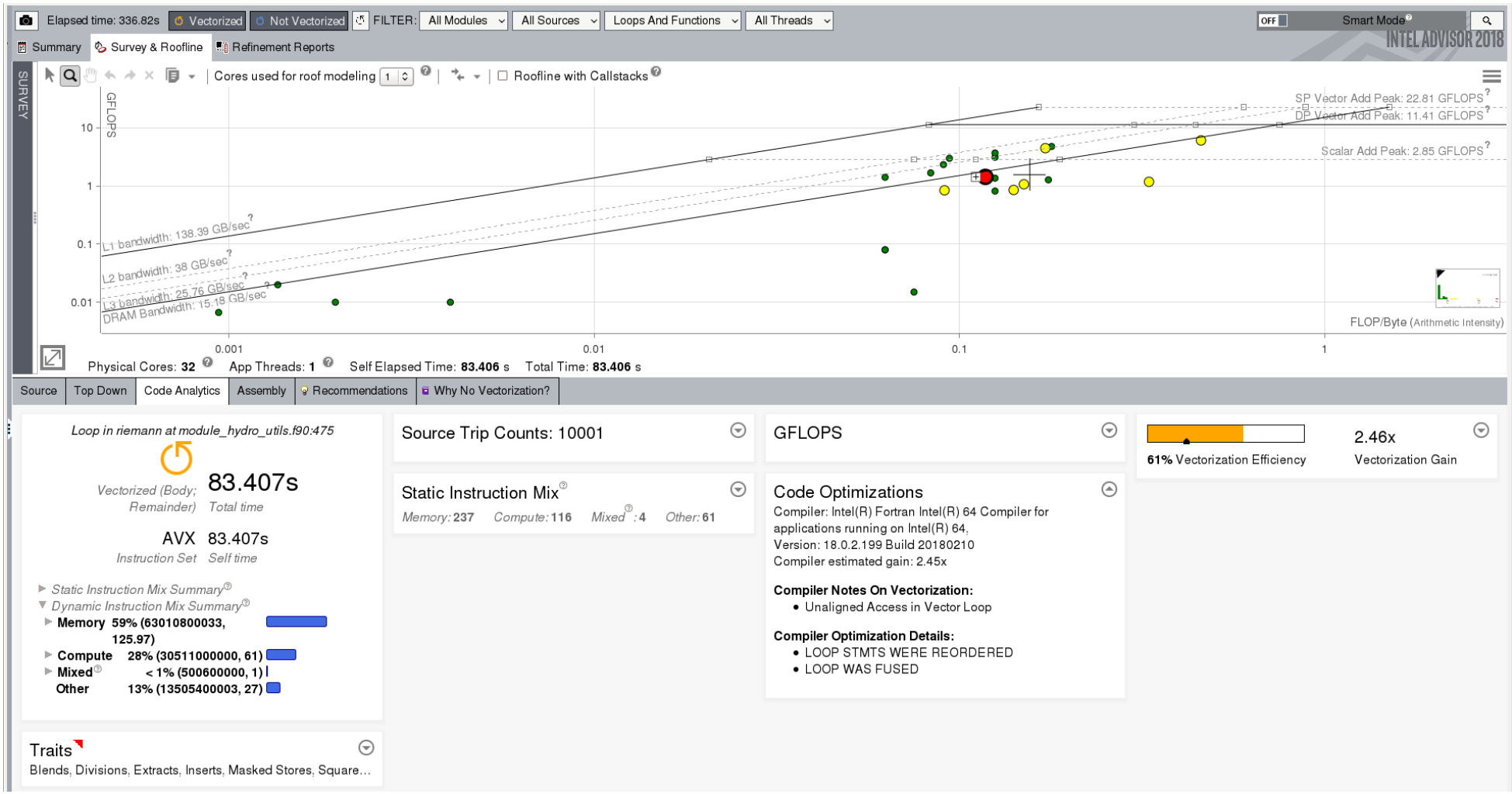
Intel Vector Advisor



Intel Vector Advisor



Intel Vector Advisor



Maqao

- Maqao (Modular Assembly Quality Analyzer and Optimizer)
- Analyse l'assembleur et indique l'efficacité de la vectorisation
- Uniquement pour architecture x86 et Xeon Phi
- Opensource
- Disponible `http://www.maqao.org`
- Équipe de l'université de Versailles Saint Quentin

Maqao - Utilisation

```
$maqao lprof -- ./exe input

....
....
[MAQAO] ANALYZING EXECUTABLE ././hydro
[MAQAO] EXECUTABLE ././hydro DONE
[MAQAO] Your experiment path is maqao_lprof
[MAQAO] To display your profiling results:
#############################################################
#      LEVEL      |    REPORT    |COMMAND                    #
#############################################################
#   Functions     |   Summary    |  maqao lprof -df xp=maqao_lprof #
#   Functions     |   Complete   |  maqao lprof -df xp=maqao_lprof #
#   Loops         |   Summary    |  maqao lprof -dl xp=maqao_lprof #
#   Loops         |   Complete   |  maqao lprof -dl xp=maqao_lprof #
#   Fcts+Loops    |   HTML       |  maqao lprof     xp=maqao_lprof #
#############################################################
```

Maqao - Utilisation

```
$ maqao lprof -dl xp=maqao_lprof/

                                    HOTSPOTS SUMMARY
#########################################################################
#  Loop  |    Function    |         Source Info        |   %    |   Time [TID]   #
#########################################################################
#  46    |  riemann       |  utils.f90:475-558         |  24.40 |  82.14 [49439]  #
#  59    |  riemann       |  utils.f90:433-448         |  16.52 |  55.62 [49439]  #
#  89    |  trace         |  utils.f90:257-306         |  7.27  |  24.48 [49439]  #
#  69    |  riemann       |  utils.f90:416-421         |  7.21  |  24.28 [49439]  #
#  121   |  constoprim    |  utils.f90:150-155         |  6.44  |  21.68 [49439]  #
#  181   |  godunov       |  principal.f90:261-265     |  4.15  |  13.98 [49439]  #
#  166   |  godunov       |  principal.f90:224-228     |  4.14  |  13.94 [49439]  #
#  94    |  trace         |  utils.f90:237-362         |  4.10  |  13.80 [49439]  #
#  79    |  cmpflx        |  utils.f90:599-616         |  3.04  |  10.22 [49439]  #
#  108   |  constoprim    |  utils.f90:200-203         |  2.88  |  9.70 [49439]   #
#  153   |  godunov       |  principal.f90:180-182     |  2.78  |  9.36 [49439]   #
#  76    |  riemann       |  utils.f90:396-402         |  2.51  |  8.44 [49439]   #
#  187   |  godunov       |  principal.f90:247-249     |  2.38  |  8.02 [49439]   #
#  161   |  godunov       |  principal.f90:157-161     |  2.32  |  7.82 [49439]   #
#  136   |  cmpdt         |  principal.f90:106-112     |  1.60  |  5.38 [49439]   #
#  72    |  riemann       |  utils.f90:408-410         |  1.36  |  4.58 [49439]   #
#  148   |  godunov       |  principal.f90:195-199     |  1.30  |  4.36 [49439]   #
#  56    |  riemann       |  utils.f90:459-463         |  1.12  |  3.76 [49439]   #
#  32    |  eos           |  utils.f90:200-203         |  0.83  |  2.80 [49439]   #
```

Maqao - Utilisation

```
$ maqao cqa ./hydro -loop=59

Target processor is: Intel Xeon processor E5 Family based on Intel
 microarchitecture code name Sandy Bridge, Intel Core i7-39xx Processor Extreme
  Edition (x86_64 architecture).

Info: Assuming lines 466 and 467 correspond to the same source loop

Section 1: Function: riemann
============================

Code for this function has been specialized for the machine where it has been
 compiled. For execution on another machine, recompile on it or with explicit
 target (example for a Haswell machine: use -xCORE-AVX2, see compiler manual
 for full list).
These loops are supposed to be defined in: utils.f90


Section 1.1: Binary loop #59
============================

The loop is defined in utils.f90:433-448.

It is main loop of related source loop which is unrolled by 4 (including
 vectorization).
3% of peak computational performance is used
 (0.29 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))
```

Maqao - Utilisation

```
Vectorization
-------------
Your loop is partially vectorized.
Only 67% of vector register length is used (average across all SSE/AVX
 instructions).

66% of SSE/AVX instructions are used in vector version (process two or more data
 elements in vector registers):
 - 7% of SSE/AVX loads are used in vector version.


Execution units bottlenecks
---------------------------
Performance is limited by execution of divide and square root operations
 (the divide/square root unit is a bottleneck).

By removing all these bottlenecks, you can lower the cost of an iteration from
 438.00 to 30.00 cycles (14.60x speedup).

Workaround(s):
 - Reduce the number of division or square root instructions:
   * If denominator is constant over iterations, use reciprocal
    (replace x/y with x*(1/y)). Check precision impact. This will be done by your
     compiler with no-prec-div or Ofast
 - Check whether you really need double precision. If not, switch to single
   precision to speedup execution



All innermost loops were analyzed.

Info: Rerun CQA with conf=hint,expert to display more advanced reports or
 conf=all to display them with default reports.
```

Maqao - Oneview

Pour générer un fichier de configuration

```
$ maqao oneview --create-config

$ edit config.lua

binary          = "./hydro"
dataset         = "./"
run_command     = "<binary> <dataset>/input_sedov_noio_10000x10000.nml"
batch_script     = "./oneview.slurm"
batch_command    = "sbatch <batch_script>"
```

Maqao - Oneview

Après modification du fichier de configuration

```
$ maqao oneview --create-report=one --config=config.lua
Info: Experiment directory created: maqao
Info: START THE APPLICATION PROFILING
Info: -> RUNNING THE PROFILER...
Info: -> FORMAT PROFILER RESULTS
Info: STOP THE APPLICATION PROFILING
Info:
Info: START FUNCTIONS AND LOOPS ANALYSIS ...
Info: -> OPEN THE MAIN APPLICATION BINARY ...
Info: --> ALL LOOPS HAVE BEEN ANALYZED
Info: --> ALL FUNCTIONS HAVE BEEN ANALYZED
Info: STOP FUNCTIONS AND LOOPS ANALYSIS ...
Info:
Info: START THE REPORT GENERATION
Info: -> ONE-VIEW EXPERIMENT DIRECTORY: maqao
Info: -> HTML REPORT GENERATED        : maqao/RESULTS/hydro_one_html/
Info: -> HTML INDEX FILE              : maqao/RESULTS/hydro_one_html/index.html
Info: STOP THE REPORT GENERATION
Info:
Info: If your application produces files, they can be found in directory maqao/dataset
Info:

* Warning: SOME WARNINGS OCCURRED DURING THE RUN. MORE DETAILS ARE AVAILABLE IN maqao/logs/log.txt
```

tp3 - Hydro - Advixe et Maqao

- Tester les deux outils sur le code Hydro
- D'abord compiler avec la commande make
- Pour Intel Advisor, il faut le faire en deux étapes :
  - d'abord l'exécution avec le script adv.slurm
  - puis l'analyse avec l'interface graphique avec advixe–gui
- Pour maqao :
  - Pour le profiling, il faut utiliser le script maqao.slurm
  - Pour Oneview, le config.lua est déja fait. Il faut utiliser la commande maqao oneview ––create–report=one ––config=config.lua

Plan

Conclusion

Pour utiliser la vectorisation de manière efficace, il faut donc :

- Éviter les dépendances ;
- Aider le compilateur avec l'OpenMP SIMD ;
- Rester le plus souvent dans les premiers niveaux de mémoire.

tp4 - HeatNum

- HeatNum est un code avec un schéma à 9 points ;
- La version disponible tourne en 238 secondes ;
- Il est possible de l'améliorer pour arriver à 50 secondes.

tp5 - Argmin

- Argmin est un noyau de calcul de l'indice du minimum ;
- La version ne vectorise pas ;
- Il est possible d'obtenir un x6 en faisant de la vectorisation explicite.