Architectures et méthodes numériques. Peut on calculer vite?

Thierry Dumont
Institut Camille Jordan, Lyon.

22 Septembre 2014.

Iris 80 $\simeq$ 40 ans.

Vitesse d'horloge 1 MhZ.

Rengaine de l'époque : « il faut se mettre au parallélisme, parce que la vitesse des horloges ne va pas pouvoir beaucoup augmenter » (sic).

Rengaine de l'époque : « il faut se mettre au parallélisme, parce que la vitesse des horloges ne va pas pouvoir beaucoup augmenter » (sic).

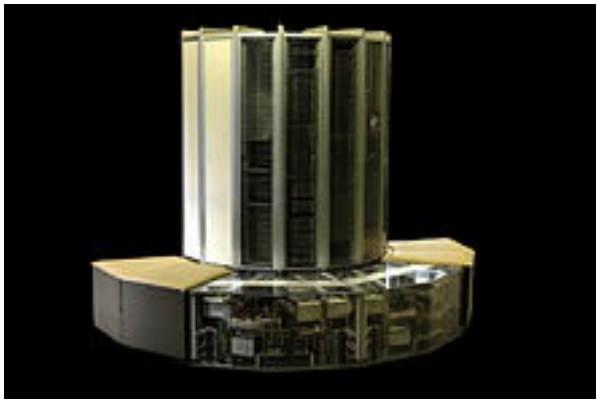
Il est alors facile de calculer « vite » :

- Aider les compilateurs (pas très sophistiqués) en sortant les valeurs constantes des boucles suffit.
- Programmes pas très sophistiqués (cartes perforées) : pas de structures de données compliquées : Fortran (IV, 66, 77) ne connaît que les tableaux.

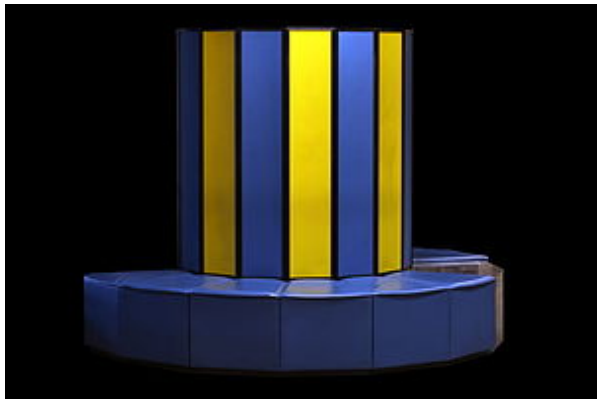
Note : il y a peu de mémoire ($\simeq$ 1 MO) !

Mais à partir de 1961 J. et P. Connes : Spectrométrie Pérot-Fabry de Jupiter. La FFT est à peine connue ! Taille mémoire limitée => développements réutilisés pour le parallélisme.

Les premières difficultés



Cray 1 1975



Cray XMP 1983 – 1985

Le Cray XMP :

- 105 MHz.
- 128 Méga Octets.

La difficulté d'écrire des programmes rapides commence avec ces machines !

Ça devient dur...

- aspect super-scalaire (2 instructions en parallèle (?)).
- pipeline.
- instructions vectorielles (taille 64) $=>$ simd.

Ça devient dur...

- aspect super-scalaire (2 instructions en parallèle (?)).
- pipeline.
- instructions vectorielles (taille 64) $=>$ simd.

Pour calculer vite, il faut qu'il y ait... des vecteurs !

for $i \in 1, n$ **do**
  $y_i = 0$
  for $j \in 1, n$ **do**
    $y_i += a_{i,j} x_j$
  end for
end for

for $i \in 1, n$ **do**
  $y_i = 0$
end for
for $j \in 1, n$ **do**
  for $i \in 1, n$ **do**
    $y_i += a_{i,j} x_j$
  end for
end for

La version de droite vectorise ! Mais pas celle de gauche.

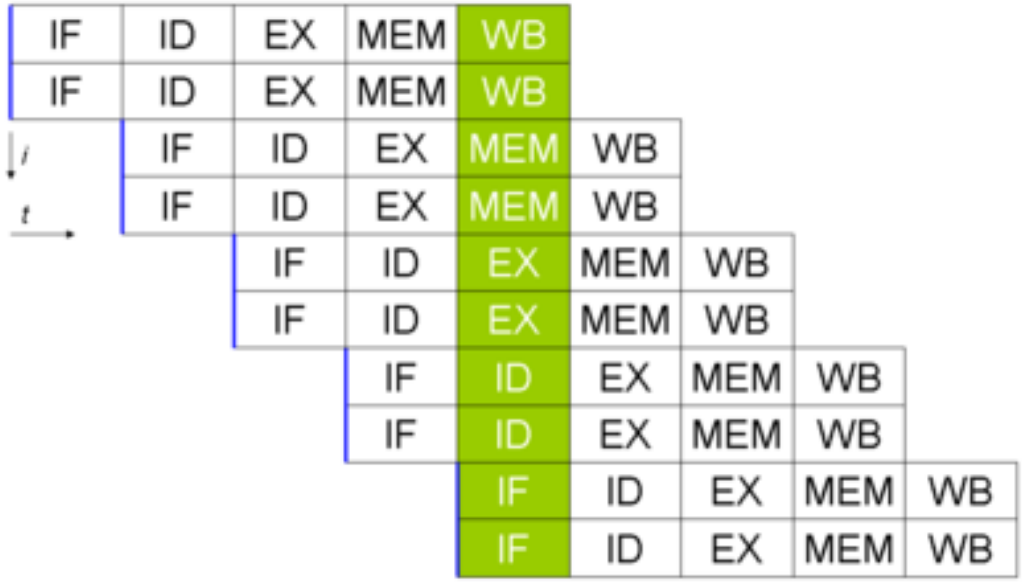
- Le jeu consiste alors à privilégier les algorithmes qui « vectorisent bien ».
- Pas de vectorisation $=>$ lent.

Autre problème : développement de codes extrêmement spécifiques de ces architectures $=>$ problèmes à long terme pour la migration vers des architectures plus modernes.

Ça devient de plus en plus dur...

Calcul sur les premiers micro processeurs (Motorola 68xx) sans grands problèmes (ni grandes performances...). Mais, rapidement :

- processeurs super-scalaires
- pipeline d'instructions.



IF : fetch, ID : decode, EX : exécute, MEM : mémoire, WB : écriture dans un registre.

Ça devient de plus en plus dur...

Ennemie du pipeline : l'instruction **if**.

Ça devient de plus en plus dur...

Ennemie du pipeline : l'instruction **if**.

Remède : l'*exécution spéculative*.

Ça devient de plus en plus dur...

Ennemie du pipeline : l'instruction **if**.

Remède : l'*exécution spéculative*.

Malgré tout : les **if** sont nocifs $=>$ plein de calculs difficiles à optimiser (parcours de graphes par exemple).

Ça devient de plus en plus dur.

Problèmes d'accès mémoire :

- augmentation sans commune mesure de la vitesse des processeurs par rapport à celle des accès mémoire.

Ça devient de plus en plus dur.

Problèmes d'accès mémoire :

- augmentation sans commune mesure de la vitesse des processeurs par rapport à celle des accès mémoire.
- augmentation de la taille mémoire des applications.

Ça devient de plus en plus dur.

Exemple : un calcul (simple) en mécanique des fluides : mouvement d'un fluide incompressible dans un cube.

- découpage du cube en n^3 sous cubes.
- pour chaque sous cube, 4 inconnues : vitesse (3 inconnues) et pression (1 inconnue).

Ça devient de plus en plus dur.

Exemple : un calcul (simple) en mécanique des fluides : mouvement d'un fluide incompressible dans un cube.

- découpage du cube en n^3 sous cubes.
- pour chaque sous cube, 4 inconnues : vitesse (3 inconnues) et pression (1 inconnue).

Soit au total $4n^3$ valeurs.
$n = 256 => \simeq 6.710^7$ inconnues $= 5.3610^8$ octets.
Mais il faut plusieurs tableaux (stocker les valeurs pour différentes temps passés et/ou des calculs intermédiaires).

Ça devient de plus en plus dur.

Exemple : un calcul (simple) en mécanique des fluides : mouvement d'un fluide incompressible dans un cube.

- découpage du cube en n^3 sous cubes.
- pour chaque sous cube, 4 inconnues : vitesse (3 inconnues) et pression (1 inconnue).

Soit au total $4n^3$ valeurs.
$n = 256 => \simeq 6.710^7$ inconnues $= 5.3610^8$ octets.
Mais il faut plusieurs tableaux (stocker les valeurs pour différentes temps passés et/ou des calculs intermédiaires).
Un problème pour les années à venir(?) : résoudre les équations de Boltzman :

- en chaque point de l'espace : la densité des particules ayant une vitesse v.

Ça devient de plus en plus dur.

Exemple : un calcul (simple) en mécanique des fluides : mouvement d'un fluide incompressible dans un cube.

- découpage du cube en n^3 sous cubes.
- pour chaque sous cube, 4 inconnues : vitesse (3 inconnues) et pression (1 inconnue).

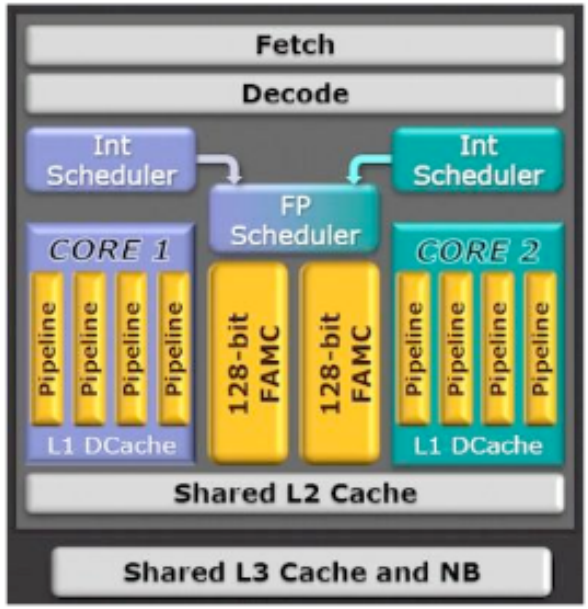
Soit au total $4n^3$ valeurs.
$n = 256 => \simeq 6.710^7$ inconnues $= 5.3610^8$ octets.
Mais il faut plusieurs tableaux (stocker les valeurs pour différentes temps passés et/ou des calculs intermédiaires).
Un problème pour les années à venir(?) : résoudre les équations de Boltzman :

- en chaque point de l'espace : la densité des particules ayant une vitesse v.Espace + vitesse : dimension 6. Dans le même cube : n^6 inconnues !

Ce n'est pas vraiment la quantité de mémoire qui pose problème (quoique), mais la vitesse d'accès.

Caches

Hiérarchie de mémoires de rapidités décroissantes : L1, L2, L3, RAM, mais de taille croissante.

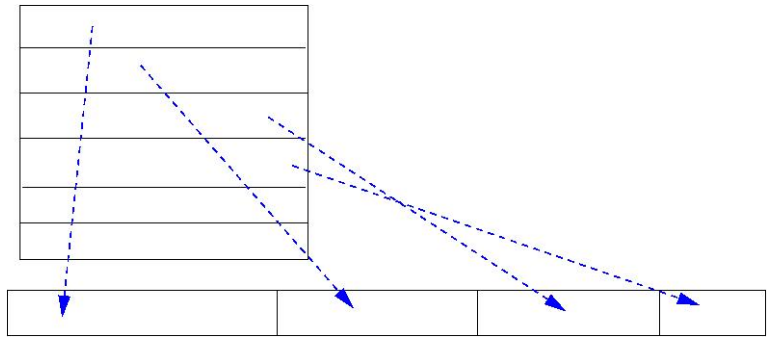


Pour calculer vite :

- favoriser les accès à des adresses proches.
- favoriser le réemploi de données amenées dans les caches.

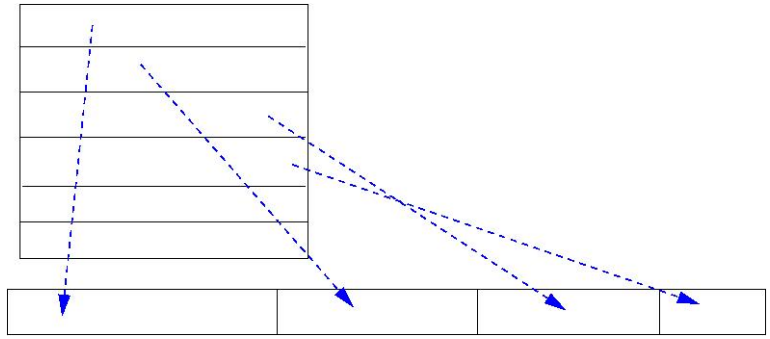
Exemple : parcours d'un tableau

Les lignes (C) ou les colonnes (Fortran) sont stockées les unes après les autres dans un vecteur.



Exemple : parcours d'un tableau

Les lignes (C) ou les colonnes (Fortran) sont stockées les unes après les autres dans un vecteur.



for $i \in 1, n$ **do**
  for $j \in 1, n$ **do**
    $a_{i,j} = 0.$
  end for
end for

for $j \in 1, n$ **do**
  for $i \in 1, n$ **do**
    $a_{i,j} = 0$
  end for
end for

Une version est « bonne », l'autre pas !

L'intensité arithmétique

q= Nombre d'opérations flottantes / nombre d'accès mémoire.

Exemple : le produit scalaire de 2 vecteurs de taille n :

$$s = \sum_{i=1}^{n} x_i . y_i .$$

a une mauvaise intensité arithmétique : $q = 1/2$!

L'intensité arithmétique

q= Nombre d'opérations flottantes / nombre d'accès mémoire.

Exemple : le produit scalaire de 2 vecteurs de taille n :

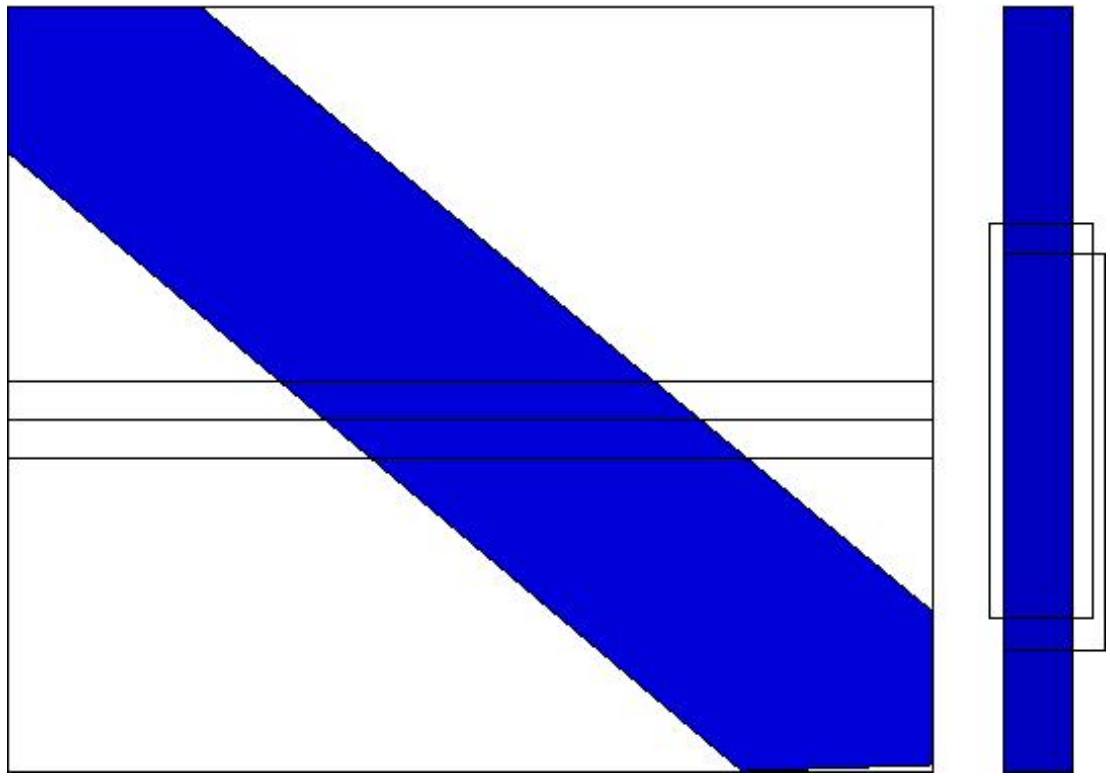
$$s = \sum_{i=1}^{n} x_i . y_i .$$

a une mauvaise intensité arithmétique : $q = 1/2$!

Un des moyens pour augmenter l'intensité arithmétique consiste à favoriser le réemploi des données.

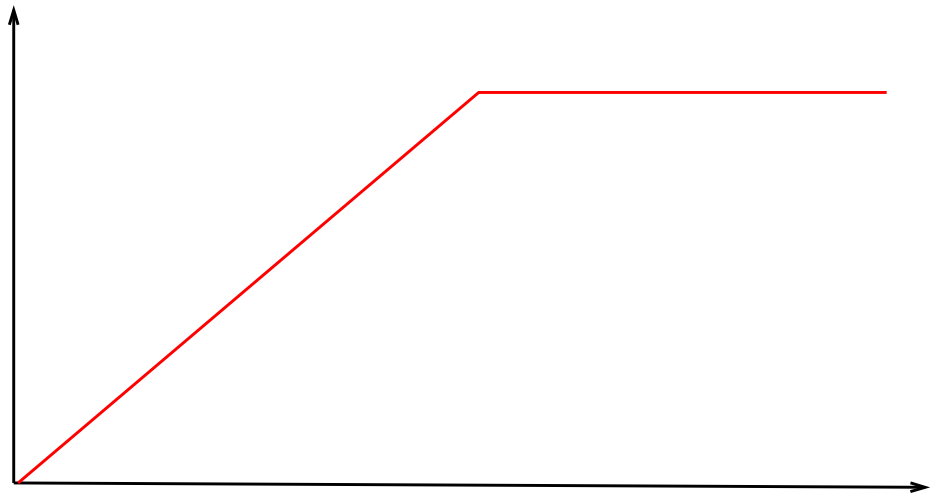
L'intensité arithmétique

Exemple : produit matrice x vecteur avec une matrice bande.



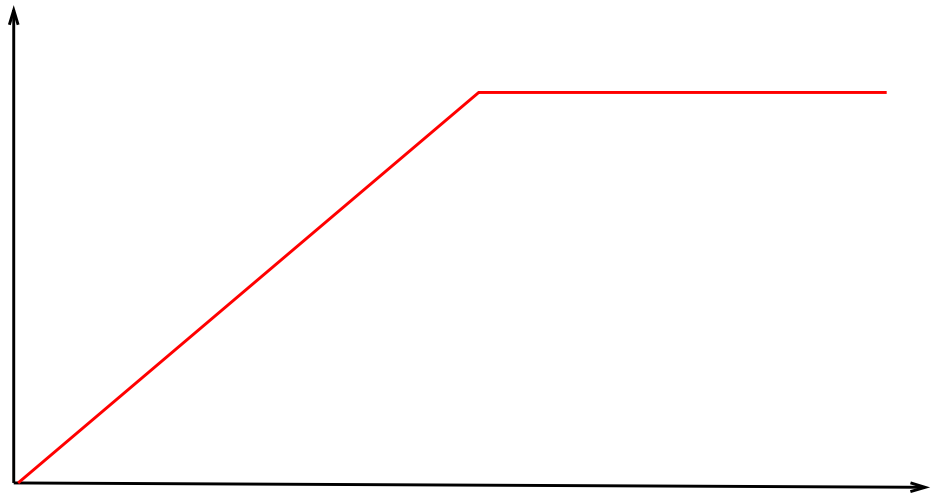
Il *peut* y avoir du réemploi de données => meilleure intensité arithmétique.

Intensité arithmétique le « rooftop model »



En abscisse, l'intensité arithmétique.
En ordonnée, les performances (flops).

Intensité arithmétique le « rooftop model »



En abscisse, l'intensité arithmétique.

En ordonnée, les performances (flops).

Deux parties :

1. partie croissante : limitation des performances par les accès mémoire.
2. partie constante : limitation par le cpu.

Intensité arithmétique

Opérations classiques et intensité arithmétique.
En fonction de la taille n du problème.

- opérations entre vecteurs, (matrice, vecteur), stencils : q indépendant de n ;
- Transformées de Fourier Rapides : $O(\log n)$.
- Produits matrices matrices (pleines) : $O(n)$ (avec une bonne implantation, par exemple `atlas`.

Bibliothèques optimisées : deux exemples

Pour améliorer les performances, on peut développer des programmes optimisés pour des architectures spécifiques (exemple : mkl de Intel) ou écrire des programmes qui s'adaptent aux architectures (au moyen de batteries de tests) : exemples emblématiques : atlas et fftw.

1. Atlas :

Opérations matrices x vecteurs et matrices x matrices.

Accès aux données par blocs. Quelle tailles pour les blocs ?

L'installation (longue) consiste à expérimenter toutes sortes de blocs.

Il faut compiler atlas !

1. Atlas :
Opérations matrices x vecteurs et matrices x matrices.
Accès aux données par blocs. Quelle tailles pour les blocs ?
L'installation (longue) consiste à expérimenter toutes sortes de blocs.
Il faut compiler atlas !

2. FFTW : Implantation optimisée de la Transformée de Fourier rapide .
Transformée de Fourier discrète :

$$f_j = \sum_{k=0}^{n-1} x_k e^{\frac{-2\pi i}{n} jk}.$$

On peut voir ça comme : $\vec{f} = A\vec{x}$ avec $A_{jk} = e^{\frac{-2\pi i}{n} jk}$.

1. Atlas :
Opérations matrices x vecteurs et matrices x matrices.
Accès aux données par blocs. Quelle tailles pour les blocs ?
L'installation (longue) consiste à expérimenter toutes sortes de blocs.
Il faut compiler atlas !

2. FFTW : Implantation optimisée de la Transformée de Fourier rapide .
Transformée de Fourier discrète :

$$f_j = \sum_{k=0}^{n-1} x_k e^{\frac{-2\pi i}{n} jk}.$$

On peut voir ça comme : $\vec{f} = A\vec{x}$ avec $A_{jk} = e^{\frac{-2\pi i}{n} jk}$.
Donc, le coût (nombre d'opérations) est de l'ordre de n^2.

Cooley et Tuckey montrent en 1965 que si $n = n_1.n_2$ on peut se ramener à deux calculs de transformées de longueur n_1 et n_2. Si $n = 2^p$, on a finalement p transformées de taille 2.

Coût en $n \log_2 n$!

Cooley et Tuckey montrent en 1965 que si $n = n_1.n_2$ on peut se ramener à deux calculs de transformées de longueur n_1 et n_2. Si $n = 2^p$, on a finalement p transformées de taille 2.

Coût en $n \log_2 n$!

Découverte fondamentale, énorme ! pas de FFT => pas de traitement du signal, par de DVDs, pas de Scanner de RMN, pas de MP3.

Cooley et Tuckey montrent en 1965 que si $n = n_1.n_2$ on peut se ramener à deux calculs de transformées de longueur n_1 et n_2. Si $n = 2^p$, on a finalement p transformées de taille 2.

Coût en $n \log_2 n$!

Découverte fondamentale, énorme ! pas de FFT => pas de traitement du signal, par de DVDs, pas de Scanner de RMN, pas de MP3.

Oui, mais pas optimal du point de vue de la gestion de la mémoire ! si $n = 2^p$ (par exemple), il faut (peut-être) mieux travailler sur des vecteurs de longueur 4 ou 8 ou...

Cooley et Tuckey montrent en 1965 que si $n = n_1.n_2$ on peut se ramener à deux calculs de transformées de longueur n_1 et n_2. Si $n = 2^p$, on a finalement p transformées de taille 2.

Coût en $n \log_2 n$!

Découverte fondamentale, énorme ! pas de FFT => pas de traitement du signal, par de DVDs, pas de Scanner de RMN, pas de MP3.

Oui, mais pas optimal du point de vue de la gestion de la mémoire ! si $n = 2^p$ (par exemple), il faut (peut-être) mieux travailler sur des vecteurs de longueur 4 ou 8 ou...

FFTW fait une phase de test pour n donné => déduction de paramètres « optimaux ».

Intensité arithmétique : comment l'améliorer

- trouver de meilleurs algorithmes.
- la partie croissante : améliorer l'accès à la mémoire (blocages, prefetch...)
- le « toit » : améliorer l'utilisation des cpu (vectorisation, réordonancement des données).

Le retour en force de la vectorisation

SSEx et AVX

- 2007 Intel et AMD : SSE2 (Streaming SIMD Extensions), 128 bits.
- 2011 AVX (Advanced Vector Extensions), 256 bits.
  Permettent par exemple de faire :

  for $i \in 1, n$ **do**

  $$y_i += a.x_i + b_i.d_i$$

  end for

  en 1 tour d'horloge pour $n = 4$ doubles ou $n = 8$ float.
  Mais avec des contraintes d'alignement.

L'avenir à courte échéance.

- GPU ?
- multiplication des cœurs :
  - plus de cœurs mais plus lents,
  - vectorisation plus poussée.

Consommer moins d'électricité !

L'avenir à courte échéance.

- GPU ?
- multiplication des cœurs :
  - plus de cœurs mais plus lents,
  - vectorisation plus poussée.

Consommer moins d'électricité !

Exemple : le Xeon-Phi

- 60 cœurs à 1 Ghz.
- Bande passante : 240 GB/sec.
- AVX 512 : vecteurs de 8 doubles ou 16 floats.
- 4 threads virtuels par cœur.
- processeur simple et pas très évolué.
- jeu d'instructions Intel classique (+AVX 512).

Projets à base de processeurs arm.
Plusieurs centaines de cœurs, taille mémoire faible.
Influence sur les algorithmes ?

Projets à base de processeurs arm.
Plusieurs centaines de cœurs, taille mémoire faible.
Influence sur les algorithmes ?

Super calculateurs : nœuds de x cœurs (x?), connectés.

Quelles méthodes seront portables ?