



Inria

ENSTA



INSTITUT
POLYTECHNIQUE
DE PARIS



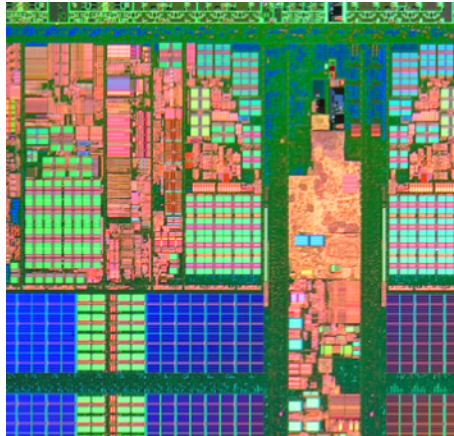
Cours SIM3

Initiation au calcul haute performance

Calcul mono-cœur (*suite*) — Vectorisation

Rappel : Vision globale de calcul haute performance ...

Cœur



Calcul séquentiel
Calcul vectoriel

Séances 1 et 2

Processeur (plusieurs cœurs)



Calcul parallèle
à **mémoire partagée**

Les cœurs peuvent travailler en parallèle sur des tâches différentes.
Ils partagent des mémoires rapides (RAM + L2 ou L3)

Séances 3 et 4

Cluster (plusieurs processeurs)



Calcul parallèle
à **mémoire distribuée**

Les processeurs peuvent travailler en parallèle sur des tâches différentes
Les données sont distribuées entre les RAM des processeurs, qui doivent communiquer (avec par ex. **MPI**).

Séance 5 (intro)

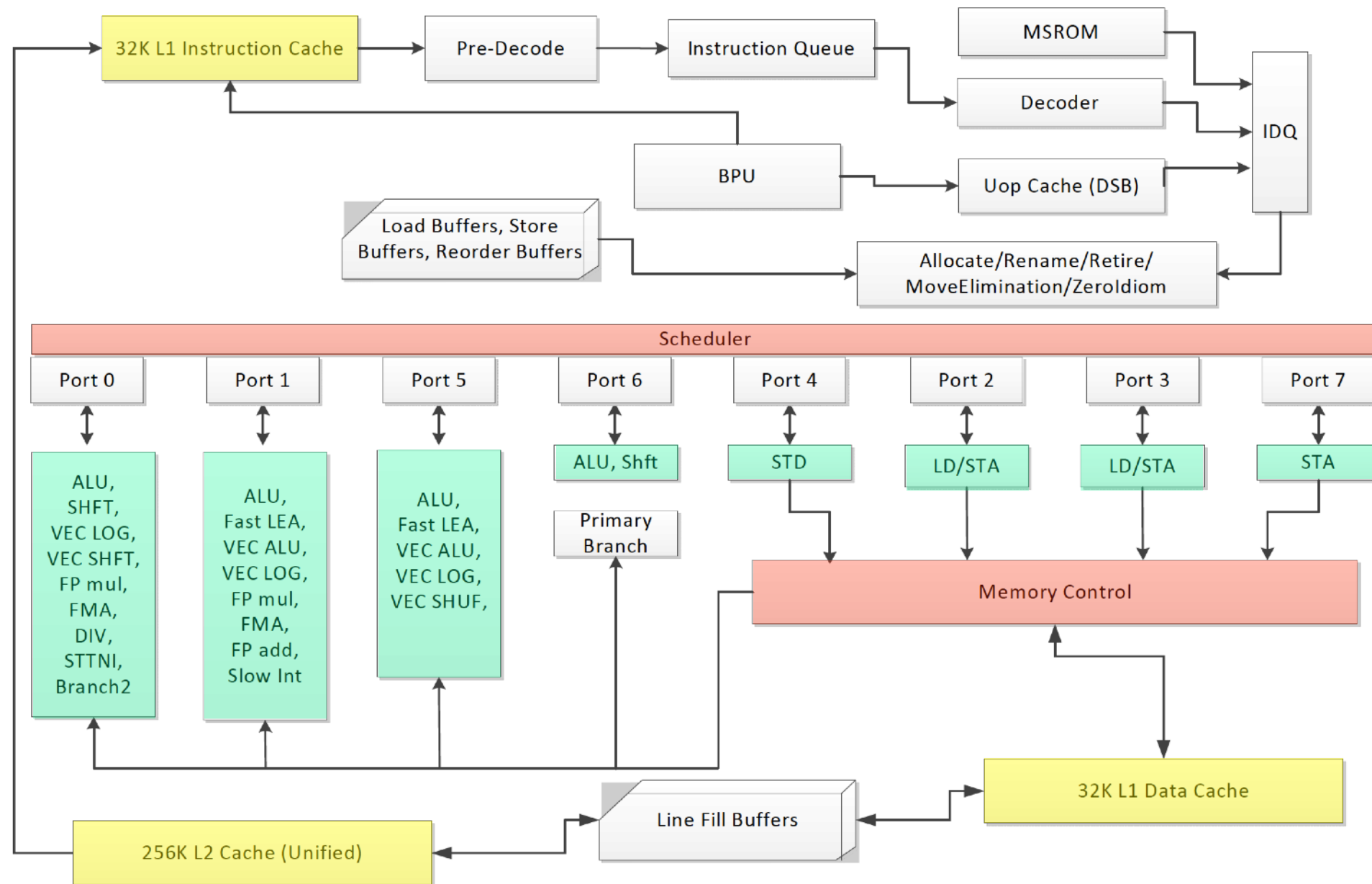
**Cours MS01 et MS-I03
(3A ModSim et M2 AMS)**

Questions importantes:

Comment **gérer les opérations** ?

Comment **gérer les données** nécessaires pour les opérations ?

Rappel : Architecture CPU



Différentes **zones mémoires** (*registres, caches L1 L2 L3, DRAM, HDD*)
Différentes **zones de contrôle** — Différentes **unités de calcul**

Rappel : Quelques stratégies de programmation

Localité des données

- Réutilisation des mêmes données dans des temps courts
- Utilisation de données stockées proches en mémoire
- Utilisation maximale de la ligne de cache

Utilisation des **options du compilateur**

Utilisation de **librairies**

Travailler sur l'**ordre des opérations**

Travailler sur la façon de stocker les **données en mémoire** (*en particulier les tableaux*)

Le HPC mono-coeur, c'est faire la psychanalyse du CPU et du compilateur ...

Tester des idées et des stratégies de programmation pour les comprendre ...



Stratégies pour calcul mono-cœur *(suite)*

Vectorisation

Étude d'un cas : Différences finies

Équation de la chaleur sur un domaine carré

$$\frac{\partial C}{\partial t} = \Delta C \quad \text{sur } \Omega = [0,1] \times [0,1]$$

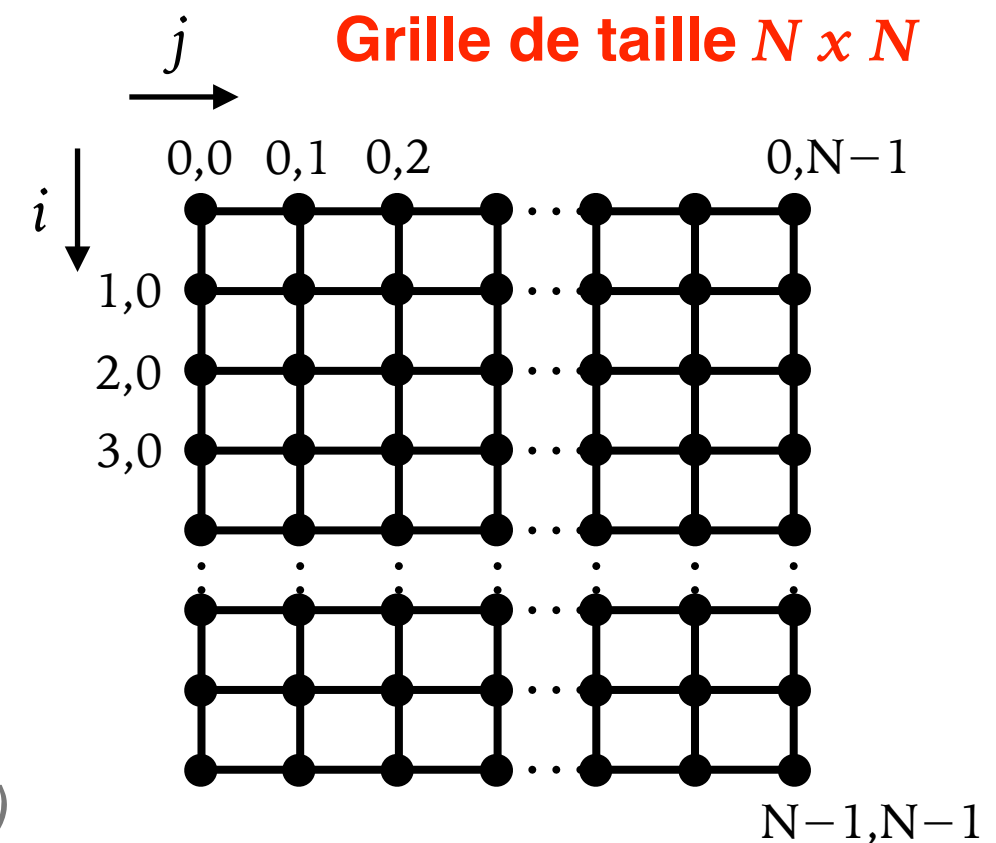
avec une condition de Dirichlet homogène $C=0$ sur $\partial\Omega$
une condition initiale $C=C_{\text{init}}$ en $t=0$

Différences finies avec schéma Euler explicite

$$\frac{C_{i,j}^{n+1} - C_{i,j}^n}{\Delta t} = \left(\frac{C_{i+1,j}^n - 2C_{i,j}^n + C_{i-1,j}^n}{\Delta x^2} + \frac{C_{i,j+1}^n - 2C_{i,j}^n + C_{i,j-1}^n}{\Delta x^2} \right)$$

$$i, j = 1 \dots N-2$$

avec $C_{i,j}^n = 0$ sur le bord ($i \in \{0, N-1\}$ et/ou $j \in \{0, N-1\}$)



Étude d'un cas : Différences finies

Implémentation 1

$$C_{i,j}^{n+1} = \left(1 - 4\frac{\Delta t}{\Delta x^2}\right) C_{i,j}^n + \frac{\Delta t}{\Delta x^2} \left(C_{i+1,j}^n + C_{i-1,j}^n + C_{i,j+1}^n + C_{i,j-1}^n\right)$$

$i, j = 1 \dots N-2$

```
int main() {  
  
    vector<double> C(N*N);  
    vector<double> Cnew(N*N);  
    ...  
  
    // Version 1  
  
    for(int n=0; n<T; n++){  
        for(int j=1; j<(N-1); j++){  
            for(int i=1; i<(N-1); i++){  
                Cnew[N*i+j]  
                    = (1 - 4*dt/(dx*dx)) * C[N*i+j]  
                      + dt/(dx*dx) * ( C[N*(i+1)+j] + C[N*(i-1)+j]  
                                         + C[N*i+(j+1)] + C[N*i+(j-1)] );  
  
                C.swap(Cnew);  
            }  
        }  
    }  
}
```

Étude d'un cas : Différences finies

Implémentation 2

$$C_{i,j}^{n+1} = \left(1 - 4\frac{\Delta t}{\Delta x^2}\right) C_{i,j}^n + \frac{\Delta t}{\Delta x^2} \left(C_{i+1,j}^n + C_{i-1,j}^n + C_{i,j+1}^n + C_{i,j-1}^n\right)$$

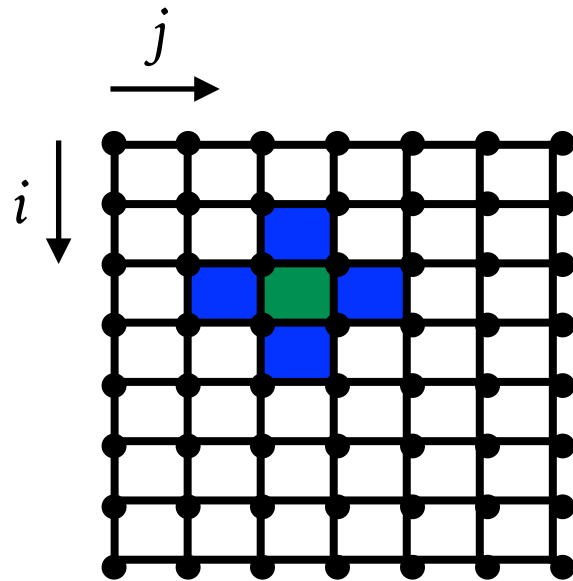
$i, j = 1 \dots N-2$

```
int main() {  
  
    vector<double> C(N*N);  
    vector<double> Cnew(N*N);  
    ...  
  
    // Version 2  
  
    for(int n=0; n<T; n++){  
        for(int i=1; i<(N-1); i++)  
            for(int j=1; j<(N-1); j++)  
                Cnew[N*i+j]  
                    = (1 - 4*dt/(dx*dx)) * C[N*i+j]  
                      + dt/(dx*dx) * ( C[N*(i+1)+j] + C[N*(i-1)+j]  
                                       + C[N*i+(j+1)] + C[N*i+(j-1)] );  
  
        C.swap(Cnew);  
    }  
}
```

Étude d'un cas : Différences finies

Implémentations 1 & 2

Éléments nécessaires
pour une itération :



$C[N * i + j]$

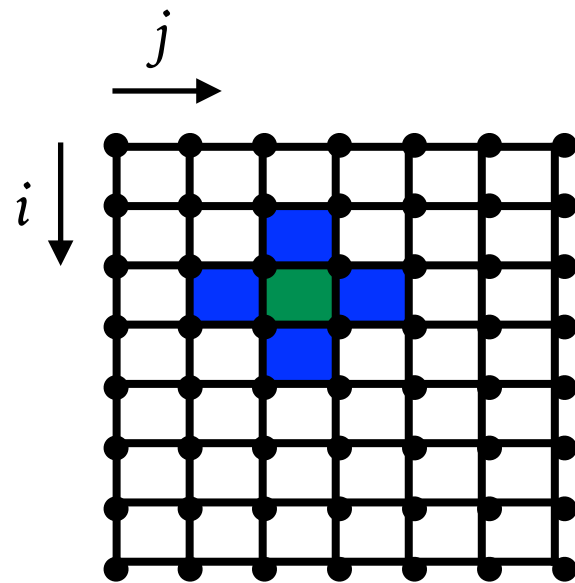
$$\begin{aligned} C_{\text{new}}[N * i + j] &= (1 - 4 * dt / (dx * dx)) * C[N * i + j] \\ &+ dt / (dx * dx) * (C[N * (i + 1) + j] + C[N * (i - 1) + j] \\ &+ C[N * i + (j + 1)] + C[N * i + (j - 1)]); \end{aligned}$$

Ici, les éléments d'une même ligne
sont contigus en mémoire.

Étude d'un cas : Différences finies

Implémentations 1 & 2

Éléments nécessaires
pour une itération :



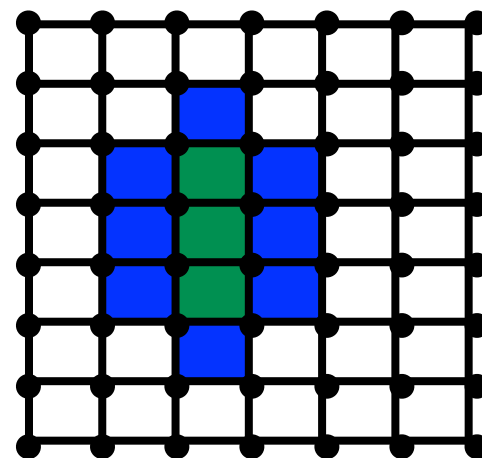
$C[N * i + j]$

Ici, les éléments d'une même ligne
sont contigus en mémoire.

Éléments nécessaires pour 3 itérations :

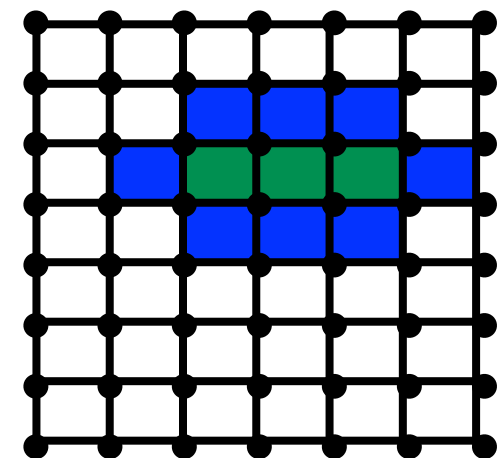
Implémentation 1

(itération sur j puis i)



Implémentation 2

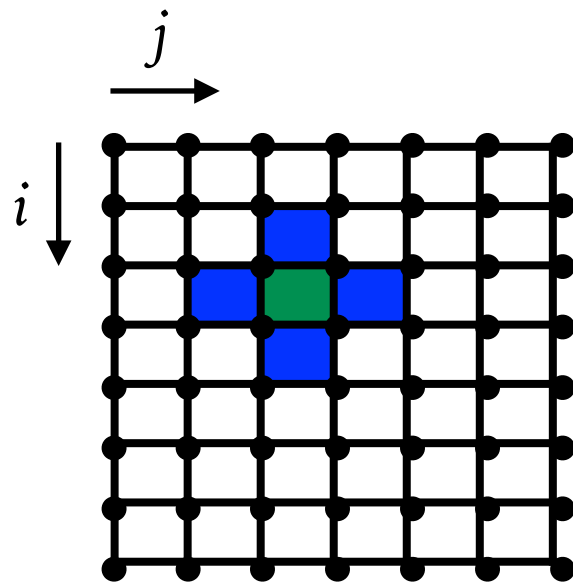
(itération sur i puis j)



Étude d'un cas : Différences finies

Implémentations 1 & 2

Éléments nécessaires
pour une itération :



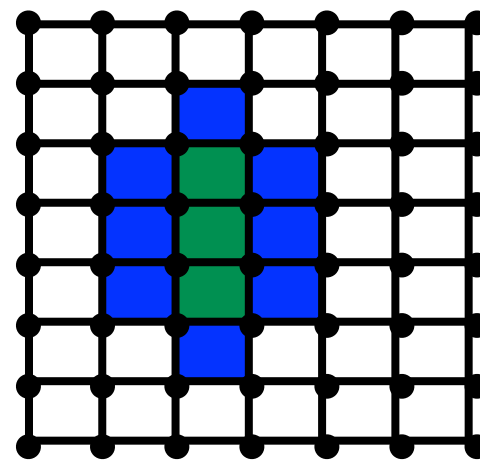
$C[N * i + j]$

Ici, les éléments d'une même ligne
sont contigus en mémoire.

Éléments nécessaires pour 3 itérations :

Implémentation 1

(itération sur j puis i)

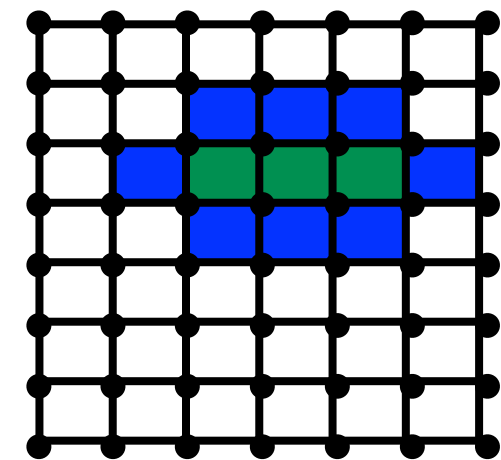


Les données utiles
sont sur 5 lignes.

~ 62 sec

Implémentation 2

(itération sur i puis j)



Les données utiles
sont sur 3 lignes.

**Meilleure utilisation
des caches !**

~ 43 sec

```
>> icpx -O0 diffusion.cpp
>> ./a.out 1 10000 512
>> ./a.out 2 10000 512
```

Étude d'un cas : Différences finies

Implémentation 3

Pré-calcul et réutilisation des coefficients !

```
int main() {  
    vector<double> C(N*N);  
    vector<double> Cnew(N*N);  
    ...  
    double coef1 = dt/(dx*dx);  
    double coef2 = 1 - 4*coef1;  
  
    // Version 3  
  
    for(int n=0; n<T; n++){  
        for(int i=1; i<(N-1); i++){  
            for(int j=1; j<(N-1); j++){  
                Cnew[N*i+j]  
                    = coef2 * C[N*i+j]  
                    + coef1 * ( C[N*(i+1)+j] + C[N*(i-1)+j]  
                               + C[N*i+(j+1)] + C[N*i+(j-1)] );  
  
                C.swap(Cnew);  
            }  
        }  
    }  
}
```

Étude d'un cas : Différences finies

Implémentation 3

Pré-calcul et réutilisation des coefficients !

```
int main() {  
    vector<double> C(N*N);  
    vector<double> Cnew(N*N);  
    ...  
    double coef1 = dt/(dx*dx);  
    double coef2 = 1 - 4*coef1;  
  
    // Version 3  
    for(int n=0; n<T; n++){  
        for(int i=1; i<(N-1); i++){  
            for(int j=1; j<(N-1); j++){  
                Cnew[N*i+j]  
                    = coef2 * C[N*i+j]  
                    + coef1 * ( C[N*(i+1)+j] + C[N*(i-1)+j]  
                               + C[N*i+(j+1)] + C[N*i+(j-1)] );  
            }  
            C.swap(Cnew);  
        }  
    }  
}
```

```
>> icpc -O0 diffusion.cpp  
>> ./a.out 3 10000 512
```

~ 40 sec

Options du compilateur

Temps de calcul (en secondes)

Machines de l'ENSTA

Compilateur d'Intel
(propriétaire)

```
>> icpx ...
```

	Implémentation 1	Implémentation 2	Implémentation 3
-O0	62	43	40
-O1	19	2.8	3.2
-O2	16	2.8	2.8
-O3	16	2.8	2.8

Compilateur g++
(libre)

```
>> g++ ...
```

	Implémentation 1	Implémentation 2	Implémentation 3
-O0	64	42	40
-O1	20	3.4	4.0
-O2	20	3.4	3.5
-O3	36	1.9	2.1

Les options de compilations ne font pas tout ...
Ils n'ont pas permuté les boucles de la première implémentation

Le pré-calcul n'est pas forcément significatif.

Rapport de compilation

```
>> icpx -O3 -qopt-report=1 -qopt-report-phase=loop,vec diffusion.cpp  
icc: remark #10397: optimization reports are generated in *.optrpt files in  
the output location
```

Quelques options de compilation :

<code>-qopt-report=1</code>	Rapport avec moins de détails (<i>de 1 à 5 — par défaut: 2</i>)
<code>-qopt-report-phase=loop</code>	Rapport spécialisé sur les boucles (loop)
<code>-qopt-report-phase=vec</code>	Rapport spécialisé sur la vectorisation (vec)
<code>-qopt-report-phase=loop,vec</code>	Rapport spécialisé sur les boucles et la vectorisation

Options de compilations possibles :
icpx -help

Quelques optimisations du compilateur

Déroutage de boucle [Loop unrolling ; iterations unrolled]

Le déroulage de boucle est une technique d'optimisation des boucles visant à en augmenter la rapidité d'exécution. Il s'agit de **dupliquer le corps de la boucle** de manière à **éviter de répéter l'instruction de saut**. Il est possible ensuite d'appliquer d'autres optimisations (allocation de registre, ordonnancement des instructions) au code après duplication.

```
for (i=0; i<100; i++)  
    g();
```

Déroutage de boucle

```
for (int i=0; i<100; i+=4)  
{  
    g();  
    g();  
    g();  
    g();  
}
```

Cette technique est fréquemment utilisée par les compilateurs optimisants, et permet aussi de générer du code vectoriel à partir d'une boucle.

Quelques optimisations du compilateur

Inlining

En informatique, l'extension inline, ou inlining, est une optimisation d'un compilateur qui **remplace un appel de fonction par le code de cette fonction**. Cette optimisation vise à réduire le temps d'exécution ainsi que la consommation mémoire. Toutefois, l'extension inline peut augmenter la taille du programme (par la répétition du code d'une fonction).

```
int add(int x, int y) { return x + y; }  
int sub(int x, int y) { return add(x, -y); }
```

```
int sub(int x, int y) { return x + -y; }
```

Simplification par inlining →

```
int sub(int x, int y) { return x - y; }
```

Dans le code, sub(x, y) sera sans-doute remplacé par (x-y)
(plusieurs fois si plusieurs appels)

Options de compilations possibles :
icpx -help
g++ --help=optimizers

Stratégies pour calcul mono-cœur *(suite)*

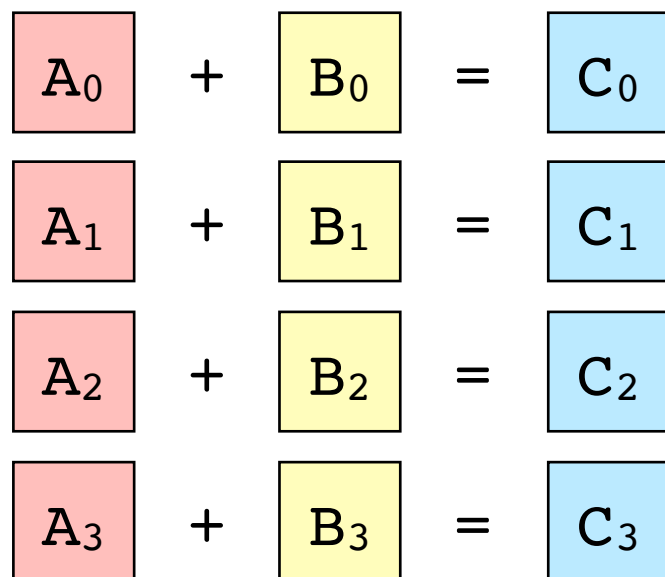
Vectorisation

Vectorisation : Principe de base

La **vectorisation** est la conversion d'un algorithme d'une **implémentation scalaire** à une **implémentation vectorielle**.

SISD : Single Instruction Single Data

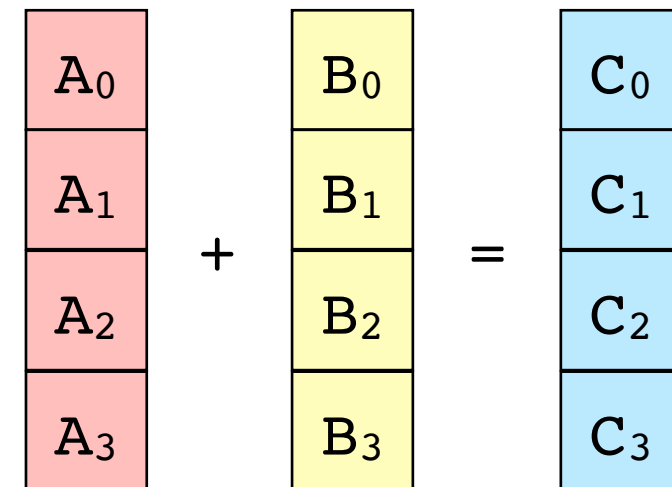
Les instructions s'appliquent à une **seule paire** de valeurs à la fois



```
for (i=0; i<4; i++)  
    C[i] = A[i] + B[i];
```

SIMD : Single Instruction Multiple Data

Les instructions s'appliquent à des **vecteurs** de valeurs adjacentes

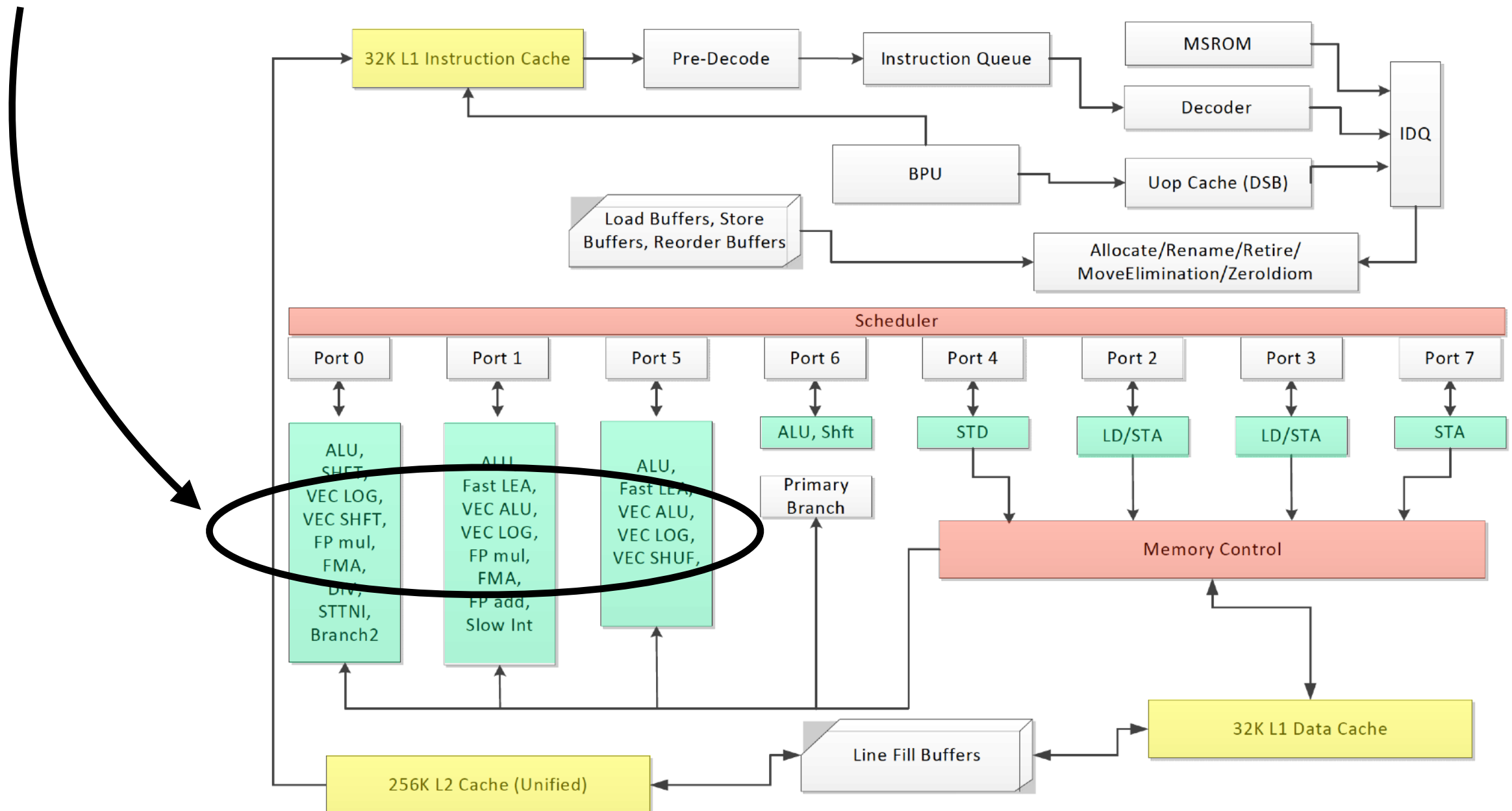


“ $C[0:4] = A[0:4] + B[0:4];$ ”

→
Meilleures performances
Utilisation des unités de calcul vectoriel

Vectorisation : Au niveau matériel

On utilise les **capacités vectorielles** des unités de calcul.
Les capacités disponibles dépendent du processeur !



Vectorisation : Au niveau matériel

Comment savoir quelles capacités sont disponibles ?

Machines de l'ENSTA

```
>> less /proc/cpuinfo
processor          : 0
vendor_id         : GenuineIntel
model name        : Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz
...
flags             : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2
ssse3 sdbg fma cx16 xtpr pdcml pcid sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
cpuid_fault epb pti ssbd ibrs ibpb stibp tpr_shadow flexpriority ept vpid ept_ad
fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap
clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp
hwp_notify hwp_act_window hwp_epp vnmi md_clear flush_lld arch_capabilities
ibpb exit to user
...
```

Quelques flags qui correspondent à des capacités vectorielles :

MMX (1996) **3DNow!** (1998) **SSE** (1999) **SSE2** (2001) **SSE3** (2004) **SSSE3** (2006) **SSE4** (2006)
AVX (2008) **F16C** (2009) **XOP** (2009) **FMA** (FMA4: 2011, FMA3: 2012) **AVX2** (2013) **AVX-512** (2015)

Signification :

SSE = Streaming SIMD Extensions
AVX = Advanced Vector Extension

Capacités plus récentes
que les machines de l'ENSTA

Vectorisation : Au niveau matériel

Quelques différences entre les capacités ...

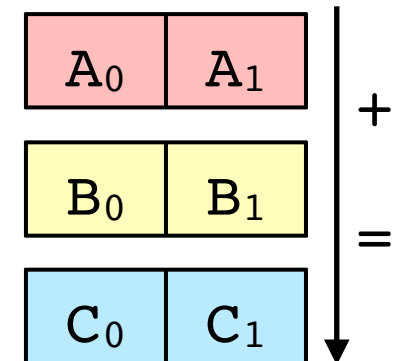
MMX

Taille du vecteur : **64 bits**

Types de donné : entiers 8, 16 et 32 bits

Longueur du vecteur : 8, 4 ou 2

Dans l'exemple : 2 entiers (**int**) de 32 bits



SSE

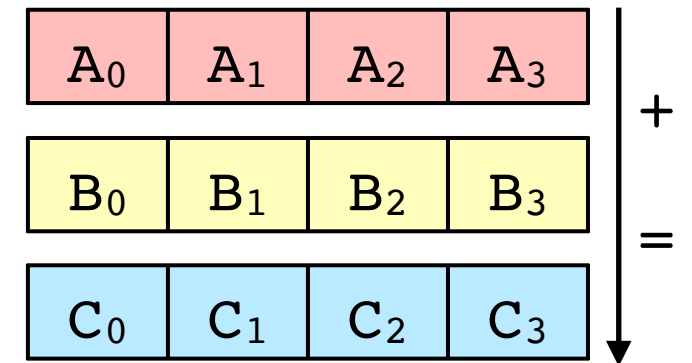
Taille du vecteur : **128 bits**

Types de donné : entiers 8, 16, 32 et 64 bits

ou nombre à virgule flottante (FL) de 32 et 64 bits

Longueur du vecteur : 16, 8, 4 ou 2

Dans l'exemple : 4 entiers (**int**) ou 4 FL (**float**) de 32 bits



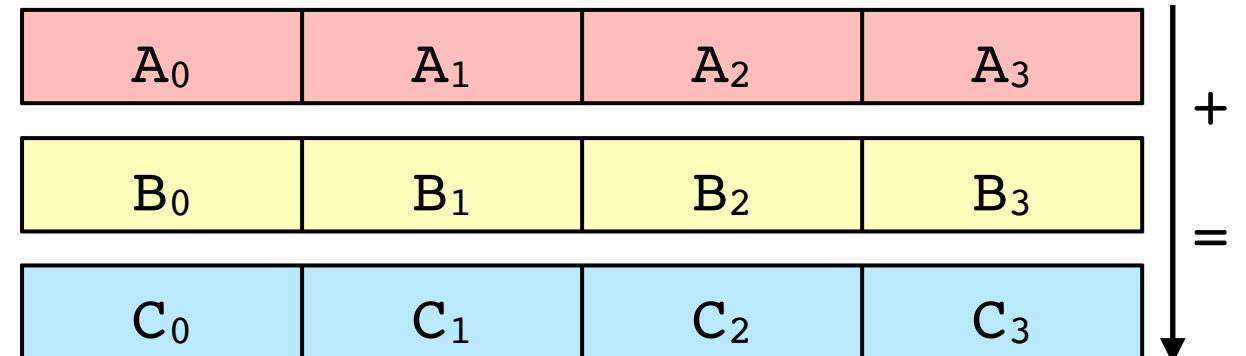
AVX

Taille du vecteur : **256 bits**

Types de donné : FL de 32 ou 64 bits

Longueur du vecteur : 8 ou 4

Dans l'exemple : 4 FL (**double**) de 64 bits



Vectorisation : Comment l'utiliser ?

1. Vectorisation automatique

- Pas de changement nécessaire dans le code
- Le compilateur cherche automatiquement les parties de code vectorisables
- Option de compilation `-O2`

```
>> icpx -O2 code.cpp
```

2. Utilisation de fonctions vectorisées provenant de bibliothèques

- Routines BLAS
- Routines “vector mathematical functions” de la bibliothèque mkl ([lien](#))

```
cblas_dgemm(..., M, N, K, alpha, A, ...)
```

3. #pragma et options de compilation avancées

- Le programmeur donne des indications au compilateurs pour aider/forcer/empêcher la vectorisation.

```
#pragma ivdep  
for (int i=0; i<n; i++)  
    a[i] = b[i] + c[i];
```

4. Fonctions intrinsèques (propres à chaque capacité)

- Très spécifiques !
- Exemple de fonction pour des compilateurs Intel : ([lien](#))

```
__m128i _mm_add_epi16(__m128i a, __m128i b)
```

Fonctionnalités de + en + spécifiques,
adaptées au code et à l'architecture

Performance ↗ Portabilité ↘

Conditions pour la vectorisation automatique d'une boucle

```
for (int i=0; i<4; i++)  
    C[i] = A[i] + B[i];
```

Vectorisation
de la boucle

“ C[0:4] = A[0:4] + B[0:4]; ”

- Des itérations successives travaillent sur des données contigües en mémoire.

```
for (int i=0; i<N; i++)  
    C[i] = A[i]+B[i];
```

OK

Vectorisation efficace

```
for (int i=0; i<N; i+=8)  
    C[i] = A[i]+B[i];
```

KO

```
for (int i=0; i<N; i++)  
    C[i] = A[i]+B[index[i]];
```

KO

Vectorisation inefficace ou impossible

Ici, on suppose que A, B et C
sont tableaux différents :

```
vector<double> A(N);  
vector<double> B(N);  
vector<double> C(N);
```

Conditions pour la vectorisation automatique d'une boucle

- Une itération ne doit pas dépendre du résultat d'une autre itération.

```
for (int i=1; i<N; i++)  
  A[i-1] = A[i]+B[i];
```

OK

```
A[0] = A[1]+B[1];  
A[1] = A[2]+B[2];  
A[2] = A[3]+B[3];  
A[3] = A[4]+B[4];  
...
```

Les itérations peuvent être effectuées en même temps.

```
for (int i=0; i<N-1; i++)  
  A[i+1] = A[i]+B[i];
```

KO

```
A[1] = A[0]+B[0];  
A[2] = A[1]+B[1];  
A[3] = A[2]+B[2];  
A[4] = A[3]+B[3];  
...
```

Chaque itération a besoin du résultat de l'itération précédente ! Les itérations ne peuvent pas être effectuées en même temps.

Exception importante ...

```
float sum=0.;  
for (int i=0; i<N; i++)  
  sum += A[i];
```

Même si chaque itération dépend du résultat de l'itération précédente, c'est une situation classique où le compilateur peut utiliser la vectorisation !

Conditions pour la vectorisation automatique d'une boucle

- Les **mêmes opérations** sont effectuées à chaque itération (*sur des données diff.*).

Souvent, les instructions “if” sont permises, même si le traitement dépend des données !
Les instructions sont alors effectuées sur toutes les données (*même si la condition “if” est fausse*), mais les résultats ne seront écrits que si la condition “if” est vérifiée.

```
for (int i=0; i<length; i++) {  
    float s = B[i]*B[i] - 4*A[i]*C[i];  
    if (s >= 0) {  
        s = sqrt(s);  
        x2[i] = (-B[i]+s)/(2.*A[i]);  
        x1[i] = (-B[i]-s)/(2.*A[i]);  
    }  
    else {  
        x2[i] = 0.;  
        x1[i] = 0.;  
    }  
}
```

Calcul de la racine de polynômes ...

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

OK, cette boucle
sera vectorisée !

- Pas d'appels à des fonctions, sauf des **fonctions mathématiques intrinsèques**.

fmax fmin fabs sqrt pow cos sin tan exp log log2 log10 ...

Conditions pour la vectorisation automatique d'une boucle

- Le **nombre d'itérations** doit être **constant** pendant la durée de la boucle.
La condition sortie de la boucle ne doit pas dépendre des données !

```
int n=0, N;  
cin >> N;  
while (n<N) {  
    C[n] = A[n]+B[n]; n++;  
}
```

OK

Cette boucle sera vectorisée
par le compilateur.

```
int n=0, N;  
cin >> N;  
while (n<N && C[n]<1e3) {  
    C[n] = A[n]+B[n]; n++;  
}
```

KO

```
int n=0, N;  
cin >> N;  
while (n<N) {  
    if(C[n]<1e3) break;  
    C[n] = A[n]+B[n]; n++;  
}
```

KO

Ces boucles ne seront pas
vectorisées par le compilateur!

Cas 1

```
float myFunc(float x, float xp, float r) {
    float val;
    val = (x-xp)*(x-xp)/(r*r);
    val = exp(-val);
    return val;
}

float trapInt(float x0, float xn, int nx, float xp, float r) {
    float x, h, sum;
    int i;
    h = (xn-x0)/nx;
    sum = 0.5*( myFunc(x0,xp,r) + myFunc(xn,xp,r) );
    for (i=1;i<nx;i++) {
        x = x0 + i*h;
        sum = sum + myFunc(x,xp,r);
    }
    sum = sum * h;
    return sum;
}
```

Cas 2

```
void myFunc(int N, double* A, double* B, double* C)
    for (int i=1; i<N; i++)
        C[i] = A[i]+B[i];
    return;
}
```

Cas 2

```
void myFunc(int N, double* A, double* B, double* C)
    for (int i=1; i<N; i++)
        C[i] = A[i]+B[i];
    return;
}
```

Sans info supplémentaire,
pas de vectorisation !!!

Deux cas de figure ...

Si ...

```
void main() {
    ...
    double *A = new double[N];
    double *B = new double[N];
    init(A,B);
    double* C;
    C = &(A[0])-1
    myFunc(N,A,B,C)
    ...
}
```

```
void main() {
    ...
    double *A = new double[N];
    double *B = new double[N];
    init(A,B);
    double* C;
    C = &(A[0])+1
    myFunc(N,A,B,C)
    ...
}
```

La boucle
équivalent à ...

```
void myFunc(...) {
    for (int i=1, i<N, i++)
        A[i-1] = A[i]+B[i];
    ...
}
```

```
void myFunc(...)
    for (int i=1, i<N, i++)
        A[i+1] = A[i]+B[i];
    ...
}
```

OK

KO

Directive de compilation `#pragma` pour aider la vectorisation

Les **directives de compilation** donne des informations pour guider le compilateur. Elles sont ignorées si le compilateur ne les connaît pas.

```
void myFunc(int N, double* A, double* B, double* C)
#pragma ivdep
    for (int i=1, i<N, i++)
        C[i] = A[i]+B[i];
    return;
}
```

- La directive **`#pragma ivdep`** signale au compilateur qu'il peut ignorer les dépendances entre les données ...
- La boucle sera vectorisée (*pour le meilleur ... ou pour le pire s'il y a des dépendances*)

D'autres `#pragma` utiles au prochain cours !

Quelques directives possibles

<code>#pragma ivdep</code>	Tell the compiler that it may safely ignore any potential data
<code>#pragma novector</code>	Asks the compiler not to vectorize a particular loop.
<code>#pragma vector</code> <i>[opt.]</i>	Indicates to the compiler that the loop should be vectorized according to the argument keywords.

Résumé ...

Ce qu'on veut ...

Localité des données

- Réutilisation des mêmes données dans des temps courts [localité temporelle]
- Utilisation de données stockées proches en mémoire [localité spatiale]
- Utilisation maximale de la ligne de cache

Utilisation des **capacités vectorielles**

Ce qu'on peut faire ...

Travailler sur l'**ordre des opérations**

Travailler sur la **façon de stocker les données** en mémoire (*en particulier les tableaux*)

Utiliser des **bibliothèques**, des **options du compilateur** et des **directives de compilation**

Le HPC mono-cœur, c'est faire la psychanalyse du CPU et du compilateur ...

Tester des idées et des stratégies de programmation pour les comprendre ...

