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(ET, 0.7ET + 0.3CC)

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

```
1  void matmul(double C[N][N], double A[N][N], double B[N][N]){
2    for(size_t i=0;i<N;i++)
3      for(size_t j=0;j<N;j++){
4        C[i][j]=0.;
5        for(size_t k=0;k<N;k++)
6          C[i][j]+=A[i][k]*B[k[j];
7      }
8    }
```

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

```
1  void matmul(double C[N][N], double A[N][N], double B[N][N]){
2    for(size_t i=0;i<N;i++)
3      for(size_t j=0;j<N;j++){
4        C[i][j]=0.;
5        for(size_t k=0;k<N;k++)
6          C[i][j]+=A[i][k]*B[k[j];
7      }
8    }
```

⇒ 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)

⇒ Pourquoi cela ne reflète pas l'exécutions en pratique

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

```
1  mov a, 1          ; create variable a
2  mov b, 2          ; create variable b
3  add c, a, b       ; add into c
4  push c            ; put return value in place
5  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

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

```
1  mov a, 1          ; create variable a
2  mov b, 2          ; create variable b
3  add c, a, b       ; add into c
4  push c            ; put return value in place
5  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

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

```
1  mov a, 1          ; create variable a
2  mov b, 2          ; create variable b
3  add c, a, b       ; add into c
4  push c            ; put return value in place
5  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

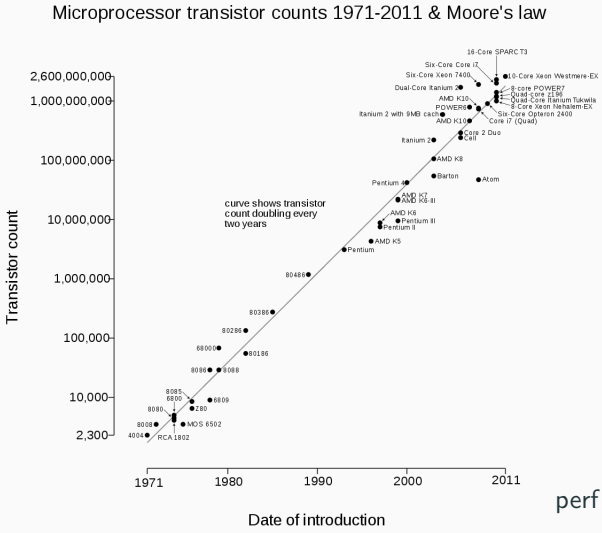
```
1  int a=1;
2  int b=2;
3  int c=a+b;
4  return c;
```

```
1  mov a, 1          ; create variable a
2  mov b, 2          ; create variable b
3  add c, a, b       ; add into c
4  push c            ; put return value in place
5  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

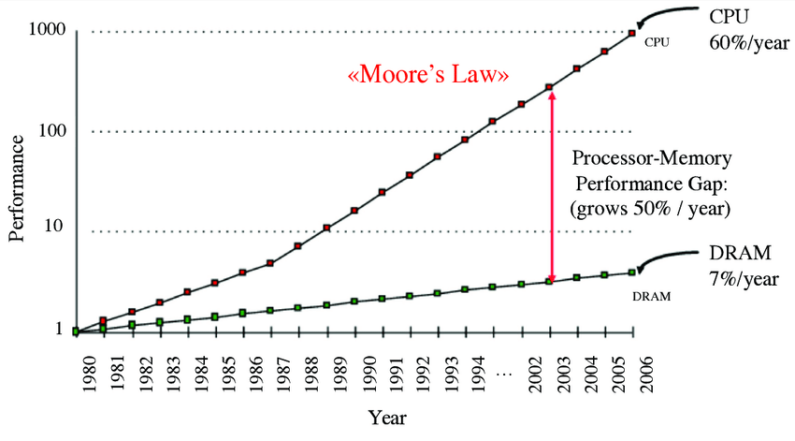


	gravure *(nm)*
1977	8000
1985	1000
⋮	⋮
2006	65
2008	45
2010	32
2012	22
2014	14
2017	10

atome Si $= 0.2$ *nm*

perf énergétique $\Rightarrow$ limite à ≈ 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 ~~$\nearrow$ 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 ~~$\nearrow$ 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

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

Performance des programmes: dépendance aux instructions

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

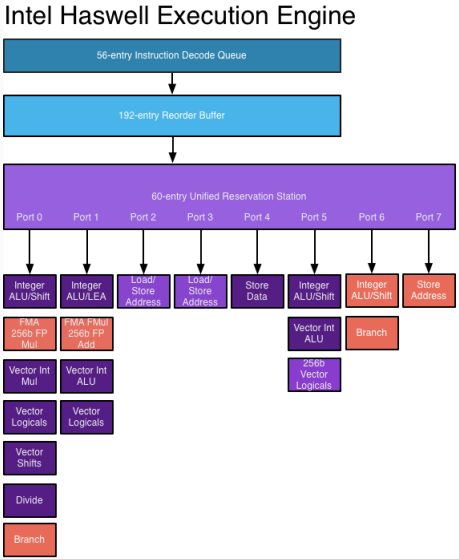
	#instructions	CPI	CR
Algorithme	X	X	
Langage Programmation	X	X	
Compilateur	X	X	
ISA	X	X	X
Design processeur		X	X

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

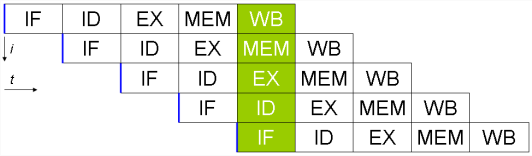
Parallélisme instruction: pipeline matériel

- 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

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

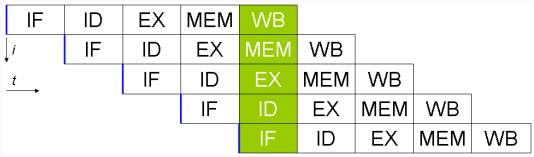


$\Rightarrow$ 5 instructions en 9 cycles: CPI=1.8

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

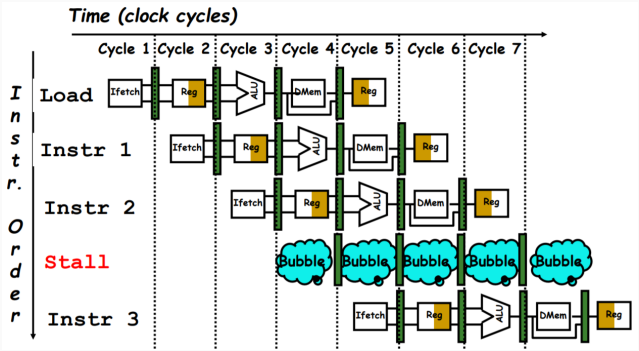
Dès que le pipeline est plein, CPI=1 $\Rightarrow$ plus compliqué en pratique

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

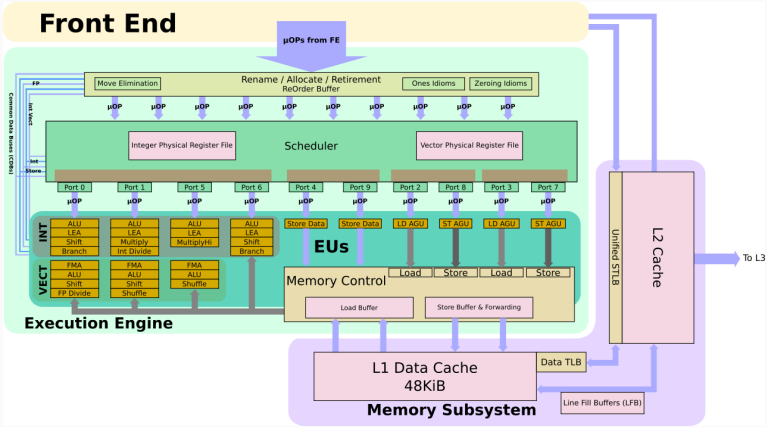
Dans le processeur

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

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

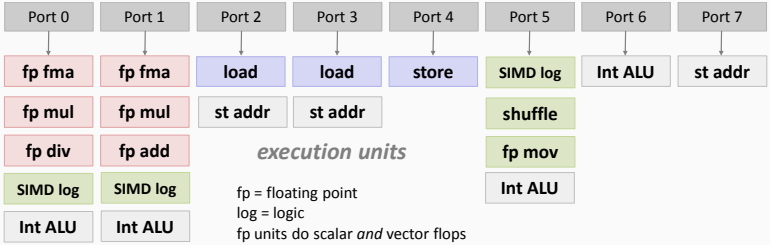
Quelles instructions en parallèle

Cela dépend de l'architecture du processeur



⇒ Holy grail: Agner Fog's website

Exemple: Haswell performance



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)	14-20	1/13	13
div (4-way)	25-35	1/27	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`

```
1 for (i = 0; i < n; i++)
2     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`

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

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

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

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

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

[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`

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

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

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

```
1 for (i = 0; i < n; i++)
2     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

```
1 double f(double a, double b, double c){
2   double r;
3   r = (a + b) * (b + c) + (a * c);
4   return r;
5 }
```

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

Exemple: Haswell performance

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

```
1 double f(double a, double b, double c){
2   double r;
3   r = (a + b) * (b + c) + (a * c);
4   return r;
5 }
```

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

- sans FMA :
- avec FMA :

Exemple: Haswell performance

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

```
1 double f(double a, double b, double c){
2   double r;
3   r = (a + b) * (b + c) + (a * c);
4   return r;
5 }
```

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

```
1 double f(double a, double b, double c){
2   double r;
3   r = (a + b) * (b + c) + (a * c);
4   return r;
5 }
```

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

```
1    t4 = t0 + t1;
2    t4 = t4 + t2;
3    t4 = t4 + t3;
```

$\Rightarrow$ ILP=1

```
1    t4 = t0 + t1;
2    t5 = t2 + t3;
3    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

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

```
1  long f();
2  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

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

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

Problème:

les 2 codes ne sont pas identiques

```
1  long counter=0;
2  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

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

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

Problème:

les 2 codes ne sont pas identiques

```
1  long counter=0;
2  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

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

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

Problème:

les 2 codes ne sont pas identiques

```
1  long counter=0;
2  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 mener à la même donnée

```
1  void twiddle(long *xp, long *yp){
2    *xp += *yp;
3    *xp += *yp;
4  }
```

```
1  void twiddle2(long *xp, long *yp){
2    *xp += 2 *yp;
3  }
```

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

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

```
1  void twiddle(long *xp, long *yp){
2    *xp += *yp;
3    *xp += *yp;
4  }
```

```
1  void twiddle2(long *xp, long *yp){
2    *xp += 2 *yp;
3  }
```

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

```
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 }
```

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

En effet, on peut faire

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

Memory aliasing et performance

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

```
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    double res;
5    for (i = 0; i < n; i++) {
6      res = 0;
7      for (j = 0; j < n; j++)
8        res += a[i][j];
9      b[i]=res;
10   }
11 }
```

- 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*

```
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    double res;
5    for (i = 0; i < n; i++) {
6      res = 0;
7      for (j = 0; j < n; j++)
8        res += a[i][j];
9      b[i]=res;
10   }
11 }
```

- 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

$\Rightarrow$ 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:

```
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    double res1,res2;
5    for (i = 0; i < n-1 ; i+=2) { // 2 lignes à la fois
6      res1= res2= 0;
7      for (j = 0; j < n; j++){
8        res1 += a[i][j];
9        res2 += a[i+1][j];
10     }
11     b[i]=res1;
12     b[i+1]=res2;
13   }
14   // code pour la dernière ligne si n est impair
15   res1=0;
16   for(;i<n;i++)
17     res1 += a[i][j];
18   b[i]=res;
19 }
```

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

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

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

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

```
1  #define OP *
2  #define START 1
3  template< typename T>
4  void reduce(const vector<T> &V, T &res){
5    res=START;
6    for(size_t i=0;i< V.size(); i++)
7      res= res OP V[i];
8  }
```

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

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

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

```
1  #define OP *
2  #define START 1
3  template< typename T>
4  void reduce(const vector<T> &V, T &res){
5    res=START;
6    for(size_t i=0;i< V.size(); i++)
7      res= res OP V[i];
8  }
```

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

Modèle de calcul SIMD: vectorisation sur les processeurs