Analyse et optimisation en C++

Denis Steckelmacher
dsteckel@ulb.ac.be
http://public.steckdenis.be

Université Libre de Bruxelles

30 mars 2015

Table des matières

Table des matires

Introduction

1. Les langages compilés sont difficiles à déboguer
2. En C/C++, aucun information donnée en cas de crash
3. Des outils existent pour en avoir !
4. Et ils donnent encore bien plus d'informations que ça !

Analyse

1. Trouver quelle ligne cause un crash
2. Inspecter et exécuter le code pas-à-pas
3. Détecter les problèmes de mémoire (new/delete, valeurs non initialisées)
4. Analyser les performances d'un programme

Optimisation

1. L'analyse donne plein d'informations
2. Optimisations faites par le compilateur
3. Optimisation de la mémoire
4. Optimisation des calculs

Cas d'étude

1. Population de N joueurs
2. Répartis dans deux classes : A et B
3. Joueurs tolérants et intolérants
4. À chaque tour, deux joueurs se rencontrent et peuvent (ou non) s'aider
5. Tous les autres joueurs observent et jugent le donneur
6. Les joueurs tolérants et intolérants jugent la transaction de manière différente

Ce programme sera étudié et débogué pendant la conférence.

Chaine de compilation

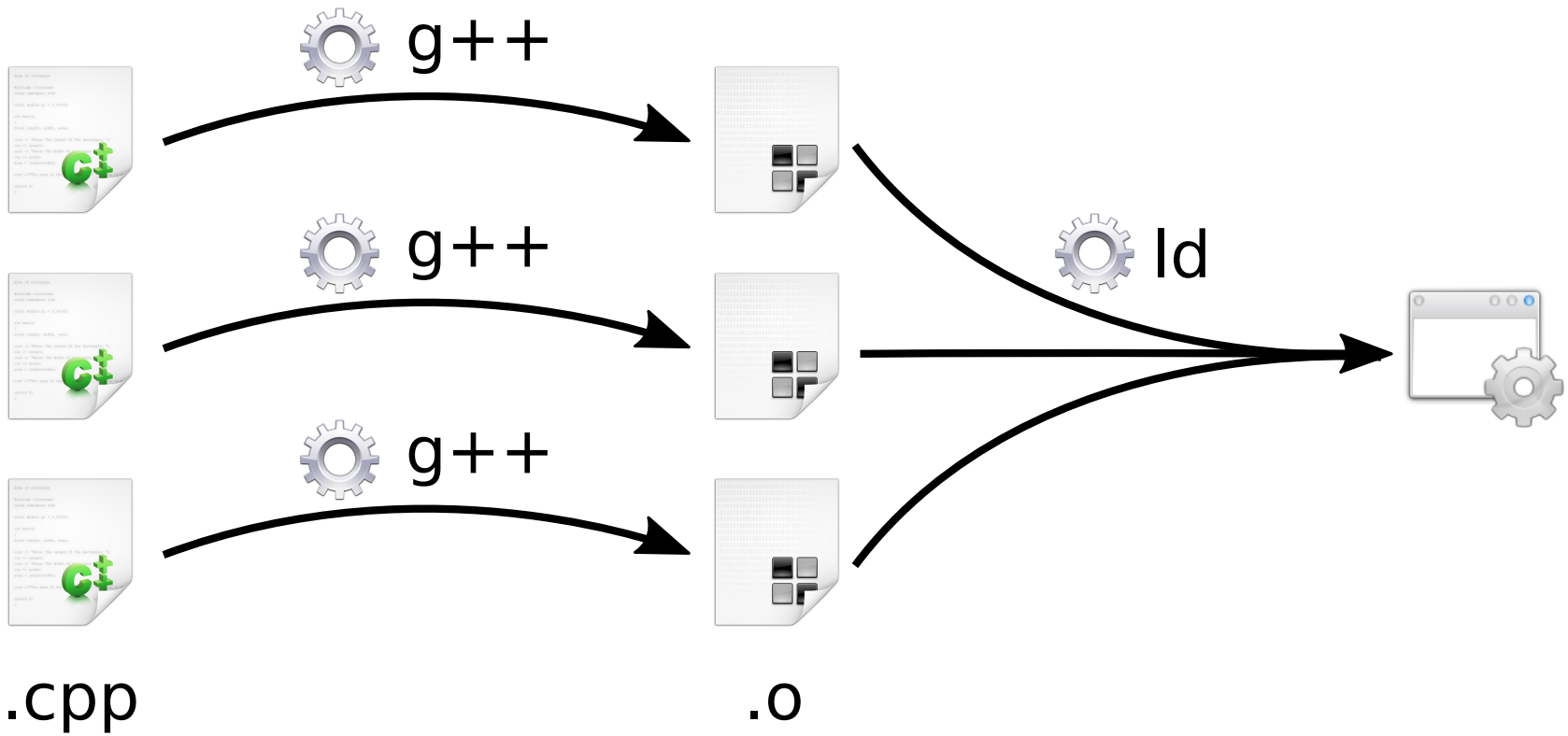


Table des matires

Les joueurs

```
1 struct Player
2 {
3     bool isA;
4     bool isTolerant;
5     int payoff;
6     bool *judgesGood;
7 };
```

Initialisation

```
1  for (int i=0; i<N-1; ++i) {
2      players[i].isA = rand() & 1;
3      players[i].isTolerant = rand() & 1;
4      players[i].payoff = 0;
5      players[i].judgesGood = new bool[N];
6
7      // Initialiser les jugements
8      for (int j=0; j<N; ++j) {
9          players[i].judgesGood[j] = rand()&3;
10     }
11 }
```

Compilation

- Par défaut, G++ produit des exécutables ne contenant que du code machine
- L'option `-g` lui demande de rajouter des *informations de débogage*
- Ces informations sont requises pour que les outils présentés aujourd'hui fonctionnent

```
$ g++ -g -o try1_buggy try1_buggy.cpp
```

Crash

Le dernier joueur n'a pas été initialisé, essayer d'y accéder ne fonctionnera pas !

```
$ ./try1_buggy
Segmentation fault
```

Pas beaucoup d'informations sur la cause du crash...

Table des matires

GDB

- Débogueur : exécute et observe un programme
- Interrompt le programme s'il crash ou si on le demande explicitement
- Permet de voir où le programme a été interrompu (y compris en cas de crash)
- Permet de consulter la valeur des variables
- (et bien plus encore)

Lancer GDB

```
$ gdb --args le_programme argument1 argument2 ...
```

Lancer GDB

```
$ gdb --args ./try1_buggy
GNU gdb (GDB; openSUSE 13.2) 7.8
Copyright (C) 2014 Free Software Foundation, Inc.
[...]

(gdb)
```

Lancer le programme dans GDB

```
(gdb) run
Starting program: /.../try1_buggy
[Thread debugging using libthread_db enabled]

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400a54 in main (argc=<optimized out>,
                            argv=<optimized out>)
                    at try1_buggy.cpp:87

87    other->judgesGood[player_a_index] = true;
```

Inspecter les variables

```
87    other->judgesGood[player_a_index] = true;

(gdb) print other
$1 = (Player *) 0x405e80
(gdb) print *other
$2 = {isA = false, isTolerant = false,
      payoff = 0, judgesGood = 0x0}
```

judgesGood vaut 0 alors que cet attribut est censé être un pointeur sur N jugements.

S'arrêter sur une ligne spécifique

```
24 for (int i=0; i<N-1; ++i) {
25     players[i].isA = rand() & 1;
26     players[i].isTolerant = rand() & 1;
27     players[i].payoff = 0;
28     players[i].judgesGood = new bool[N];
29
30     ...
31 }
```

```
(gdb) break try1_buggy.cpp:24
Breakpoint 1 at 0x4008ac: file try1_buggy.cpp, line 24.
```

S'arrêter sur une ligne spécifique

```
(gdb) run
Starting program: /.../try1_buggy
[Thread debugging using libthread_db enabled]

Breakpoint 1, main (argc=<optimized out>,
                    argv=<optimized out>)
              at try1_buggy.cpp:24

24      for (int i=0; i<N-1; ++i) {

(gdb)
```

Se déplacer de ligne en ligne

```
(gdb) next
25      players[i].isA = rand() & 1;
(gdb) next
26      players[i].isTolerant = rand() & 1;
(gdb) next
27      players[i].payoff = 0;
(gdb) next
28      players[i].judgesGood = new bool[N];
(gdb) print players[i]
$3 = {isA = true, isTolerant = false,
      payoff = 0, judgesGood = 0x406290}
```

Next vs Step

- `next` avance jusqu'à la ligne suivante sans rentrer dans les fonctions
- `step` rentre dans les fonctions

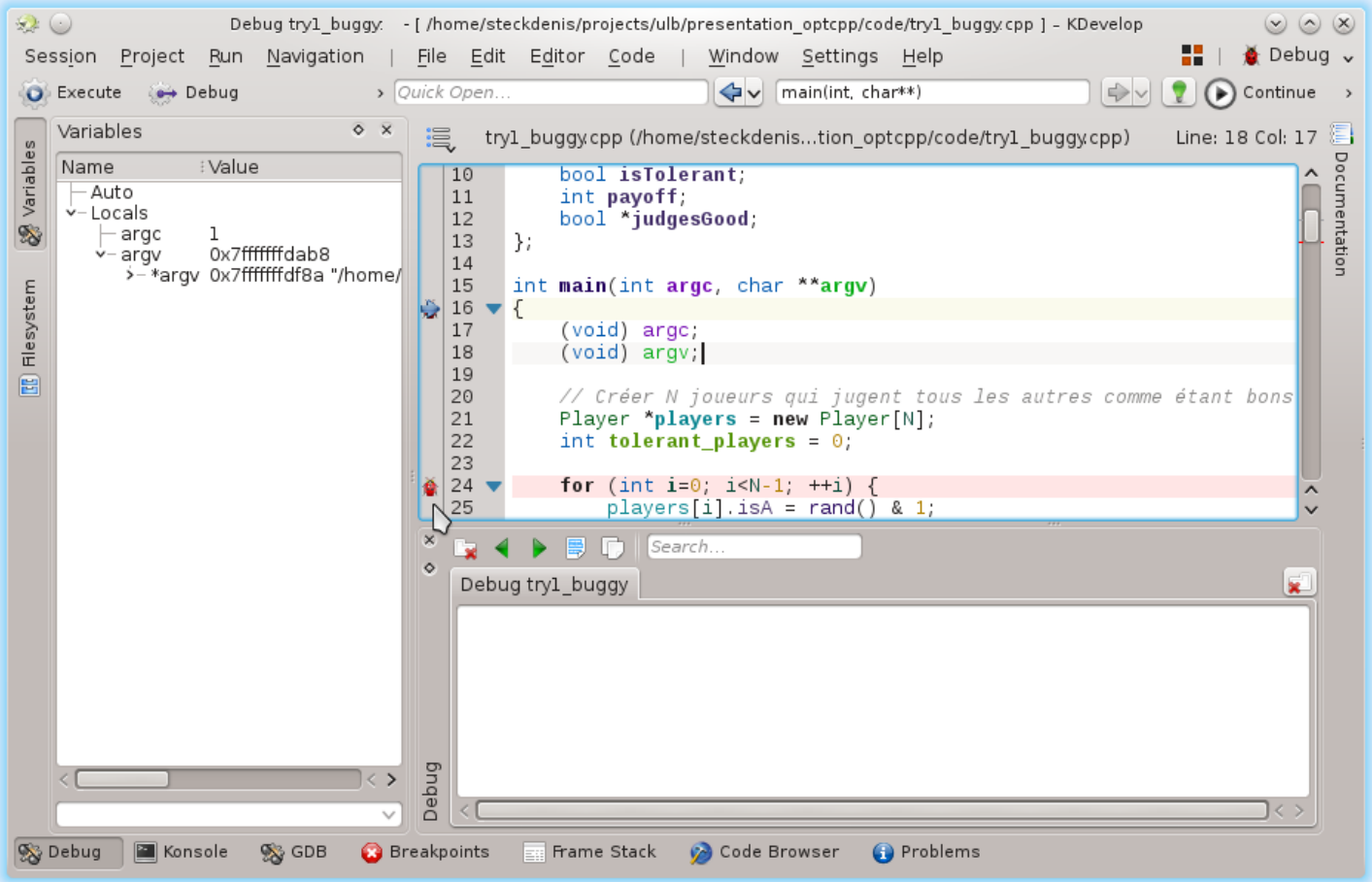
```
1 int a = func(b, c);
2 int b = inv(a);
```

Interfaces graphiques pour GDB

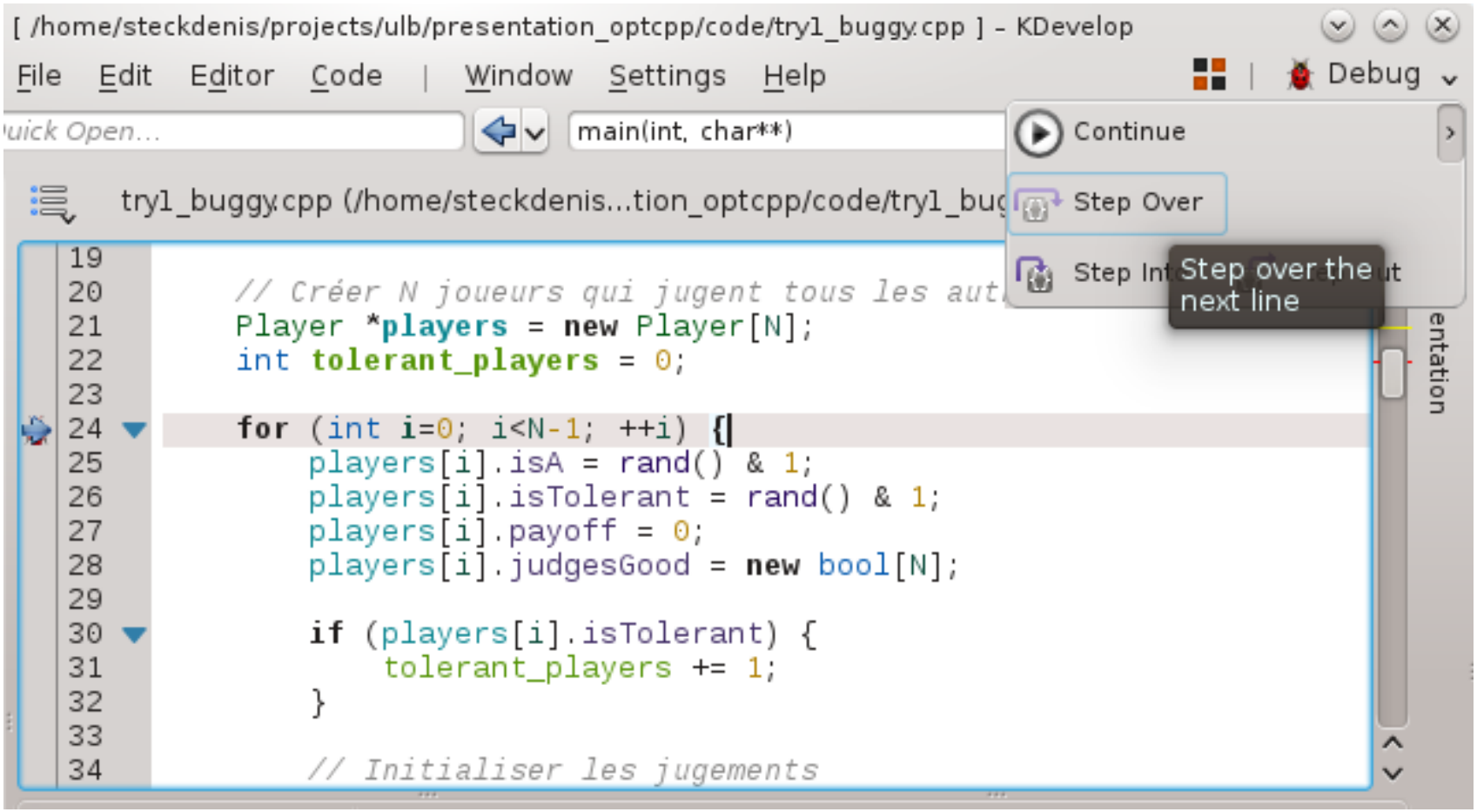
- Déboguer en ligne de commande est rapide mais pas facile
- Il existe de nombreuses GUIs pour GDB !
- DDD (basique et *old-school*), Eclipse, Qt Creator, KDevelop, etc

```
$ kdevelop -d gdb ./try1_buggy
```

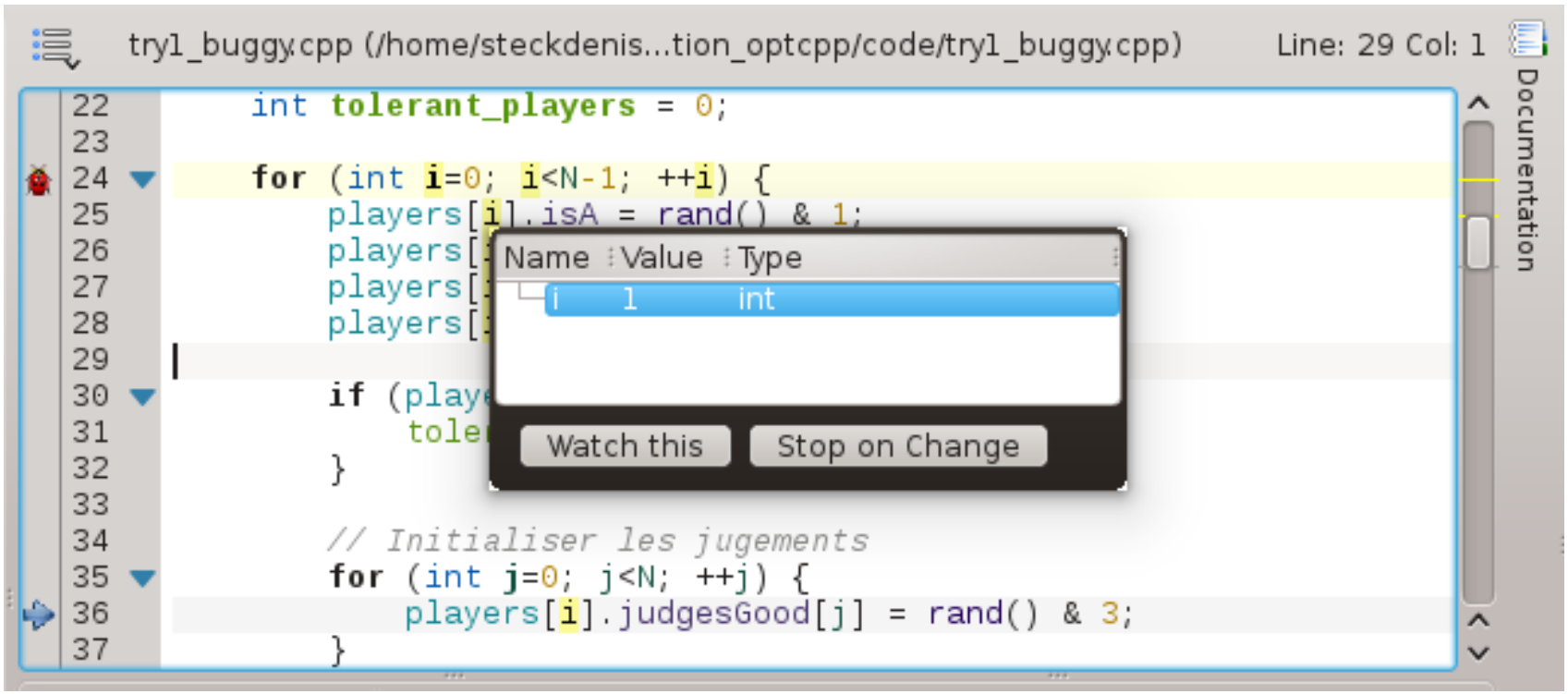
Déboguer avec KDevelop (Breakpoints)



Déboguer avec KDevelop (Step)



Déboguer avec KDevelop (Inspection)



Déboguer avec KDevelop (Inspection)

Déboguer avec KDevelop (Crash)

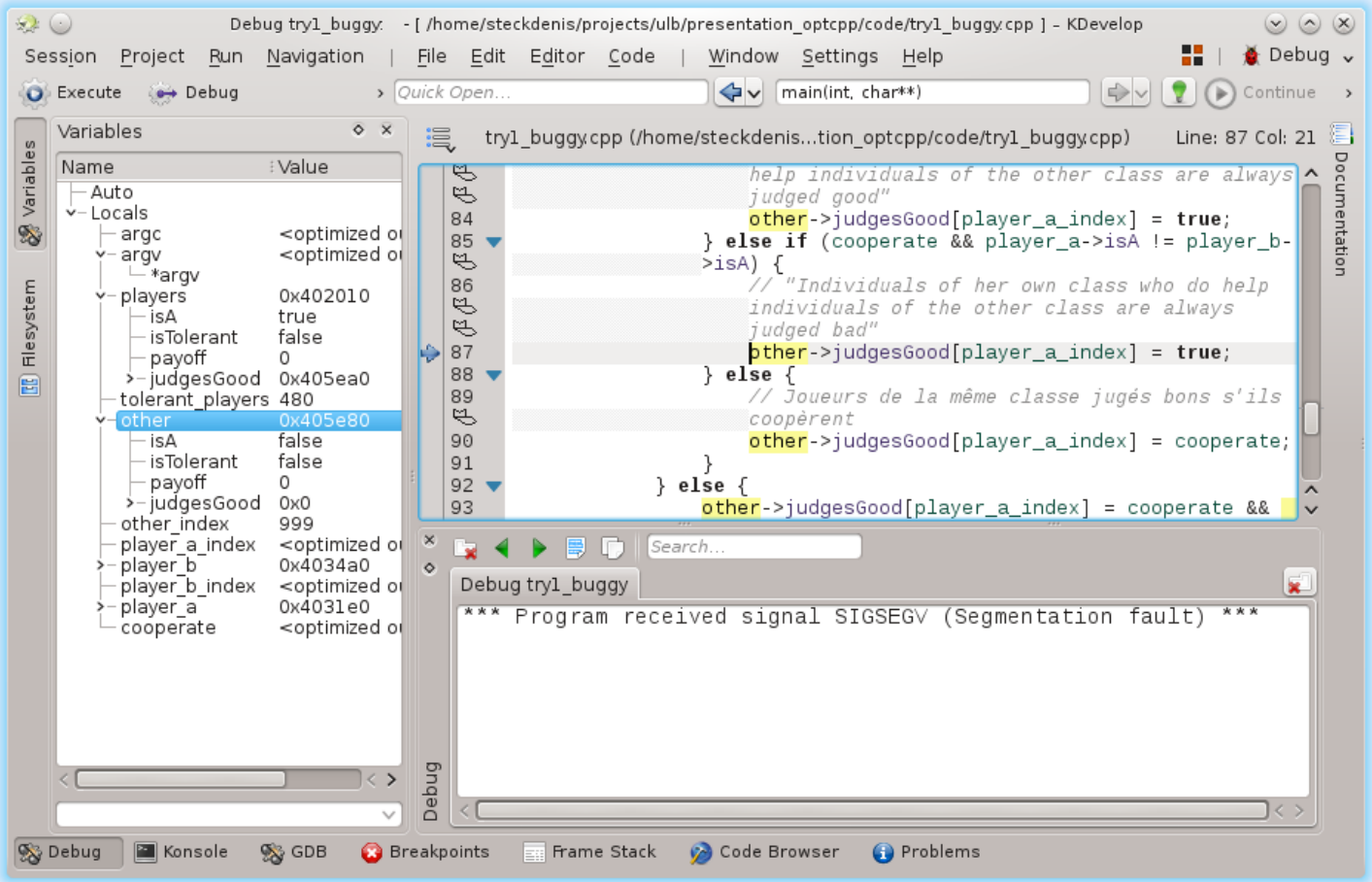


Table des matires

Valgrind

1. Lance un programme dans une machine virtuelle et l'observe *très* attentivement
2. Détecte les variables non-initialisées, les mauvaises allocations, les lectures/écritures invalides, les *memory leaks*, etc
3. Analyse les performances d'un programme (abordé plus loin)
4. Est également capable de détecter les complexes erreurs liées au *multi-threading*

Lancer Valgrind (Memcheck)

```
$ valgrind --tool=memcheck
           --track-origins=yes
           --leak-check=full
```

1. *Memcheck* est l'outil Valgrind détectant les problèmes de mémoire
2. Dire d'où proviennent les erreurs mémoires
3. Détecter les *memory leaks*

Détection du joueur non-initialisé

```
$ valgrind --tool=memcheck
           --track-origins=yes
           ./try1_buggy
Memcheck, a memory error detector
[...]

Conditional jump or move depends on uninitialised values
   at 0x400A3F: main (try1_buggy.cpp:78)
Uninitialised value was created by a heap allocation
   at 0x4C29D90: operator new[](unsigned long)
   by 0x40082F: main (try1_buggy.cpp:21)
```

Détection du joueur non-initialisé

```
19 // Créer N joueurs qui jugent tous
20 // les autres comme étant bons
21 Player *players = new Player[N];
22 int tolerant_players = 0;
23
24 for (int i=0; i<N; ++i) {
25     [...]
26 }
```

- On sait où de la mémoire non-initialisée a été allouée
- On ne sait pas pourquoi (Valgrind n'est pas humain, si ?)
- On peut corriger le `for` auquel il manquait une itération

Détection de la fuite de mémoire

```
$ valgrind --tool=memcheck
           --track-origins=yes
           --leak-check=full
           ./try2_buggy
Memcheck, a memory error detector
[...]

1,016,000 (16,000 direct, 1,000,000 indirect)
bytes in 1 blocks are definitely lost
in loss record 2 of 2
   at 0x4C29D90: operator new[](unsigned long)
   by 0x40082F: main (try2_memleak.cpp:21)
```

Détection de la fuite de mémoire

- On a alloué de la mémoire, mais on a oublié de la libérer quand on n'en avait plus besoin !
- À la fin de `main()`, la mémoire peut être libérée.

```
100 // Supprimer les données
101 for (int i=0; i<N; ++i) {
102     delete[] players[i].judgesGood;
103 }
104
105 delete[] players;
```

Table des matires

Analyse de performance

- Valgrind va compter le nombre de fois que chaque instruction de chaque ligne de code s'exécute
- Ces informations sont ensuite affichées pour chaque fonction
- Permet de voir quelle fonction occupe le plus de ressources
- Dans une fonction, permet de voir quelle ligne met le plus de temps à s'exécuter

Lancer Valgrind (Callgrind)

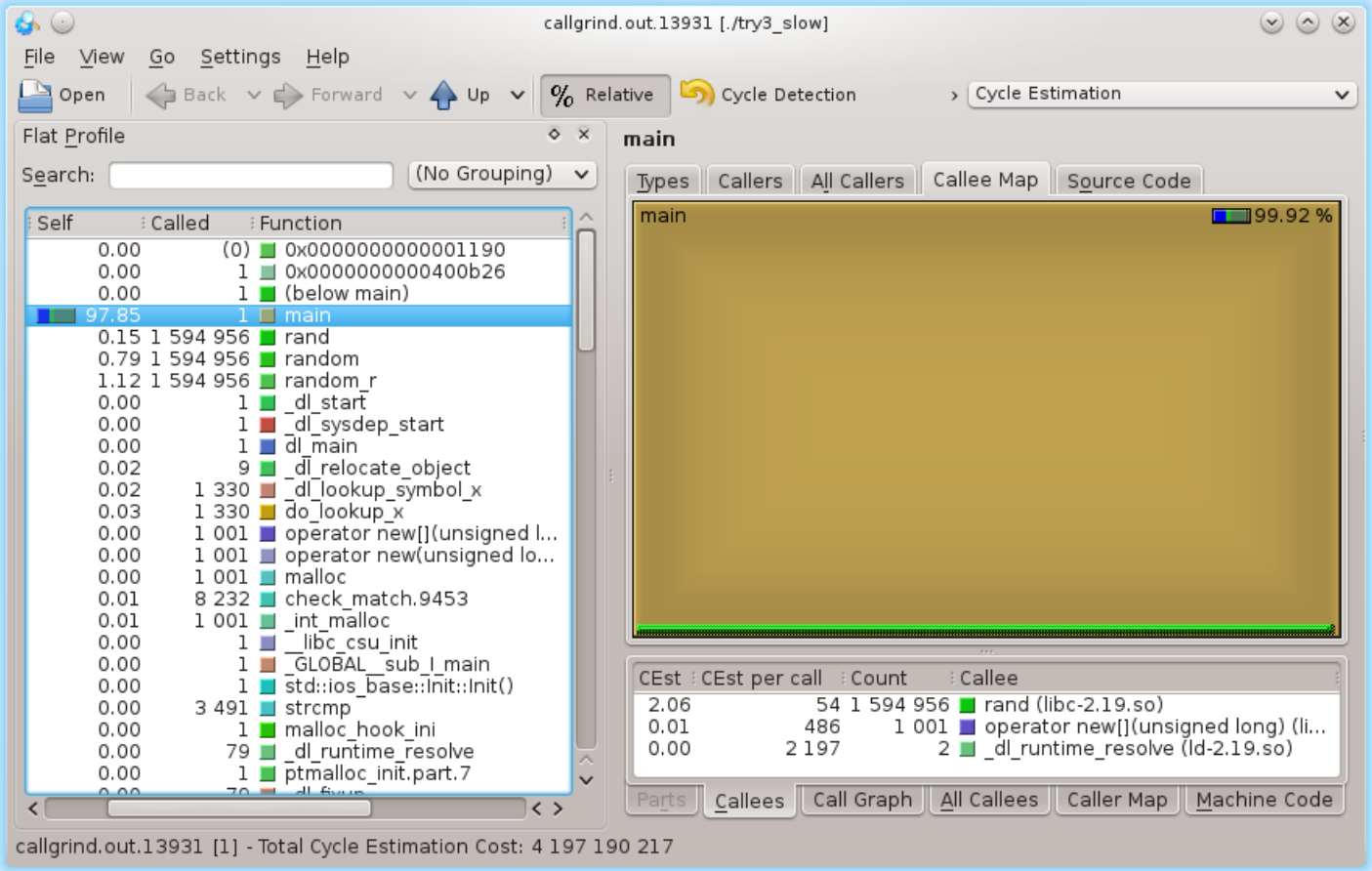
```
$ valgrind --tool=callgrind
           --dump-instr=yes
           --simulate-cache=yes
           --collect-jumps=yes
           --branch-sim=yes
```

1. *Callgrind* est l'outil Valgrind comptant les instructions
2. Enregistrer le nombre de fois que chaque instruction s'exécute
3. Différentes options qui seront plus claire plus tard

Sortie de Callgrind

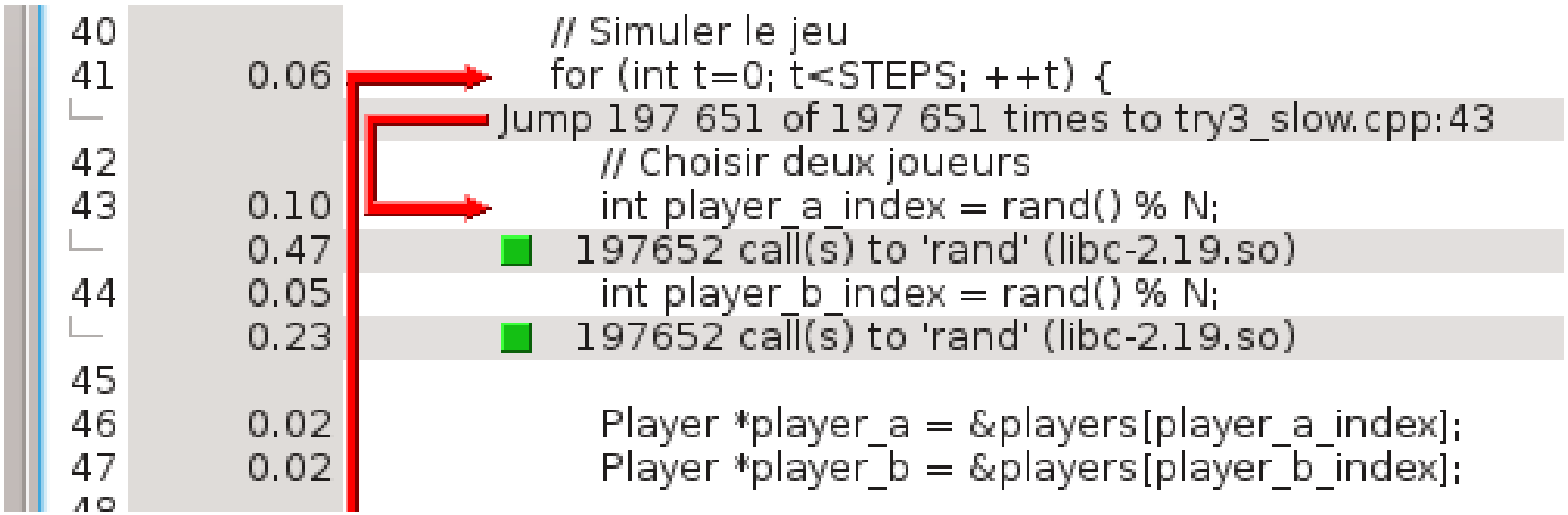
- Callgrind affiche des statistiques dans la console, elles prendront leur sens plus tard
- Il produit également un fichier `callgrind.out.NNNN`
- Ce fichier s'ouvre à l'aide de **KCacheGrind**

Aperçu de KCacheGrind



Carte des appels

Comptage des appels de fonction



Découverte de ce qui est lent

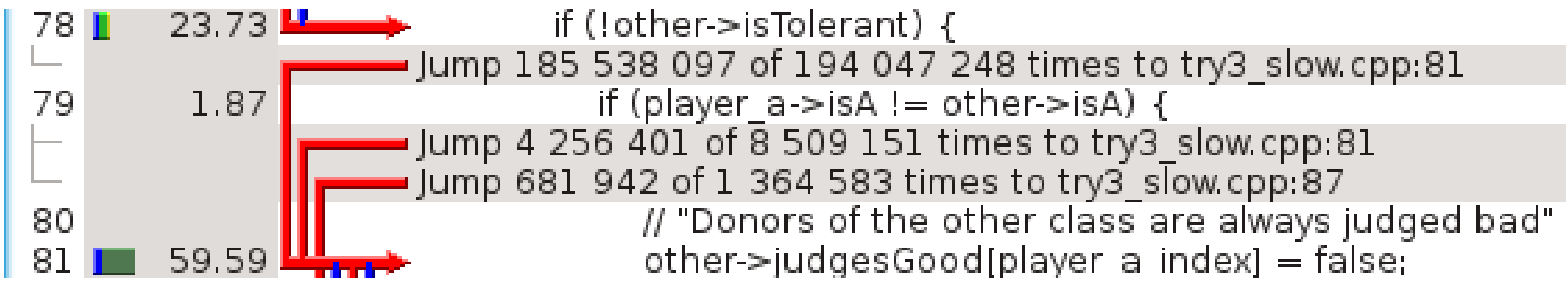


Table des matires

Perf

Valgrind émule le programme :

1. C'est **lent**, de 20 à 100 fois plus lent que la vitesse native
2. Émuler le processeur masque les complexes interactions qu'il a avec le reste de l'ordinateur
3. Mais c'est précis (chaque instruction est comptée et analysée)

Perf observe le programme :

1. Le programme n'est pas ralenti
2. Quelques milliers de fois par seconde, Perf regarde où est le programme et quel est son état
3. Correspond exactement à l'état du processeur
4. Mais la plupart des instructions sont manquées

Lancer Perf

```
$ perf record ./try3_slow
```

1. N'écrit rien dans la console
2. Crée un fichier `perf.data`

Analyser les données

```
$ perf annotate
```

1. Ouvre le dernier `perf.data` créé
2. Affiche les données à la manière de KCacheGrind, mais dans la console

Analyser les données

```
main  /home/steckdenis/projects/ulb/presentation_optcpp/code/try3_slow
                  // A va maintenant être jugé par tous les autres joueurs
                  for (int other_index=0; other_index<N; ++other_index) {
 40.13       ↓ je     1c8
                     Player *other = &players[other_index];

                     if (!other->isTolerant) {
  0.03  1a2:   cmpb   $0x0,(%rdx)
 36.76       ↑ jne    190
                        if (player_a->isA != other->isA) {
  0.28         movzbl (%r14),%eax
  0.54         cmp    -0x1(%rdx),%al
             ↑ jne    190
Press 'h' for help on key bindings
```

Table des matires

Pourquoi optimiser ?

- Un programme lent est désagréable voire impossible à utiliser
- Un programme pas trop lent mais inefficace consommera trop d'énergie
- Le temps, c'est de l'argent, l'énergie aussi

Performances et prix des derniers processeurs Intel :

Processeur	Performances	Prix
Core i7-5960X	15,967	1049,99 $
Core i7-5930K	13,650	577,99 $
Core i7-5820K	12,901	373,32 $

Cas d'étude

- Le cas d'étude est maintenant fonctionnel
- Tests réalisés sur un Intel Core i5-3230M (2×3.2 Ghz, 32 KB L1, 512 KB L2, 3 MB L3) avec 6 GB de mémoire DDR3 1600 Mhz.
- Temps d'exécution du cas d'étude : 2,60 secondes
- Le code n'a pas l'air mauvais, comment l'optimiser ?

Table des matires

Optimisations automatiques

- Les compilateurs modernes (GCC, Clang, ICC, Visual Studio) sont capables d'optimiser votre code eux-mêmes
- Laissez-les faire ! Ne compliquez pas votre code inutilement si le compilateur sait faire la même chose
- Écrivez du code clair en sachant que votre compilateur l'optimisera pour vous

Avec GCC, Clang et ICC, les optimisations automatiques sont activées dès que l'option `-O1`, `-O2` ou `-O3` est présente sur la ligne de commande. Sous Visual Studio, il faut activer le mode *Release*.

Factorisation

Ne calcule une valeur qu'une seule fois, même si elle est utilisée plusieurs fois.

```
1 players[i + 1].isA = rand() & 1;
2 players[i + 1].isTolerant = rand() & 1;
```

Devient :

```
1 Player *tmp = &players[i + 1];
2
3 tmp->isA = rand() & 1;
4 tmp->isTolerant = rand() & 1;
```

Propagation des constantes

Si un calcul est fait sur des données connues à la compilation, le compilateur fait les calculs

```
1 int a = 2 + 3 * 5;
2 float f = exp(2);
```

Devient :

```
1 int a = 17;
2 float f = 7.3890561;
```

Réordonnancement

Parfois, faire une opération avant une autre la rend plus rapide.

```
1 c = a + b;
2 d = exp(e);
```

Devient :

```
1 d = exp(e);
2 c = a + b;
```

Inlining

Recopie une fonction au lieu de l'appeler. Cette optimisation est l'une des plus importante et des plus efficaces.

```
1 int add(int a, int b) {
2     return a + b;
3 }
4
5 int s = add(3, 5)
```

Devient :

```
1 int s = 8; // Constantes propagées
```

Suppression du code mort

Du code jamais appelé peut être supprimé. Généralement, l'inlining produit beaucoup de code mort.

```
1 int add(int a, bool negate) {
2     if (negate) { return -a; }
3     else { return a; }
4 }
5
6 int t = negate(var, true);
```

Devient :

```
1 int t = -var;
```

Factorisation des invariants de boucle

Ce qui ne dépend pas d'une variable qui change dans une boucle peut en être sorti.

```
1 for (int i=0; i<N; ++i) {
2     tab[i] = exp(a);
3 }
```

Devient :

```
1 float tmp = exp(a);
2
3 for (int i=0; i<N; ++i) {
4     tab[i] = tmp;
5 }
```

Duplication de boucles

Si une boucle contient un gros if, elle sera remplacée par un gros if qui contient des boucles

```
1 for (int i=0; i<N; ++i) {
2     if (a == b) {
3         tab[i] += 1;
4     } else {
5         tab[i] -= 1;
6     }
7 }
```

Duplication de boucles

Si une boucle contient un gros if, elle sera remplacée par un gros if qui contient des boucles

```
1 if (a == b) {
2     for (int i=0; i<N; ++i) {
3         tab[i] += 1;
4     }
5 } else {
6     for (int i=0; i<N; ++i) {
7         tab[i] -= 1;
8     }
9 }
```

Échange de boucles

Deux boucles peuvent être échangées pour optimiser les calculs

```
1 for (int x=0; x<N; ++x) {
2     for (int y=0; y<N; ++y) {
3         tab[y * N + x] += 1;
4     }
5 }
```

Échange de boucles

Deux boucles peuvent être échangées pour optimiser les calculs

```
1 for (int y=0; y<N; ++y) {
2     int tmp = y * N;
3
4     for (int x=0; x<N; ++x) {
5         tab[tmp + x] += 1;
6     }
7 }
```

Déroulage de boucles

Les boucles dont le nombre de tours est connu peuvent être déroulées

```
1 for (int i=0; i<30; ++i) {
2     tab[i] *= 2;
3 }
```

Devient :

```
1 for (int i=0; i<28; i += 3) {
2     tab[i] *= 2;
3     tab[i+1] *= 2;
4     tab[i+2] *= 2;
5 }
```

Déroulage de boucles

De même quand le nombre de tours est inconnu (mais le code généré est alors plus compliqué)

```
1 for (int i=0; i<N; ++i) {
2     tab[i] *= 2;
3 }
```

Déroulage de boucles

De même quand le nombre de tours est inconnu (mais le code généré est alors plus compliqué)

```
1 for (int i=0; i<N-2; i += 3) {
2     tab[i] *= 2;
3     tab[i+1] *= 2;
4     tab[i+2] *= 2;
5 }
6
7 for (int i=N/3*3; i<N; ++i) {
8     tab[i] *= 2;
9 }
```

Dévirtualisation

Très importante en C++, cette optimisation remplace des appels de méthode indirects par des appels directs

```
1 void test(ClasseMere *o) {
2     o->faireQuelqueChose();
3 }
4
5 ClasseFille *f = new ClasseFille;
6
7 test(f);
```

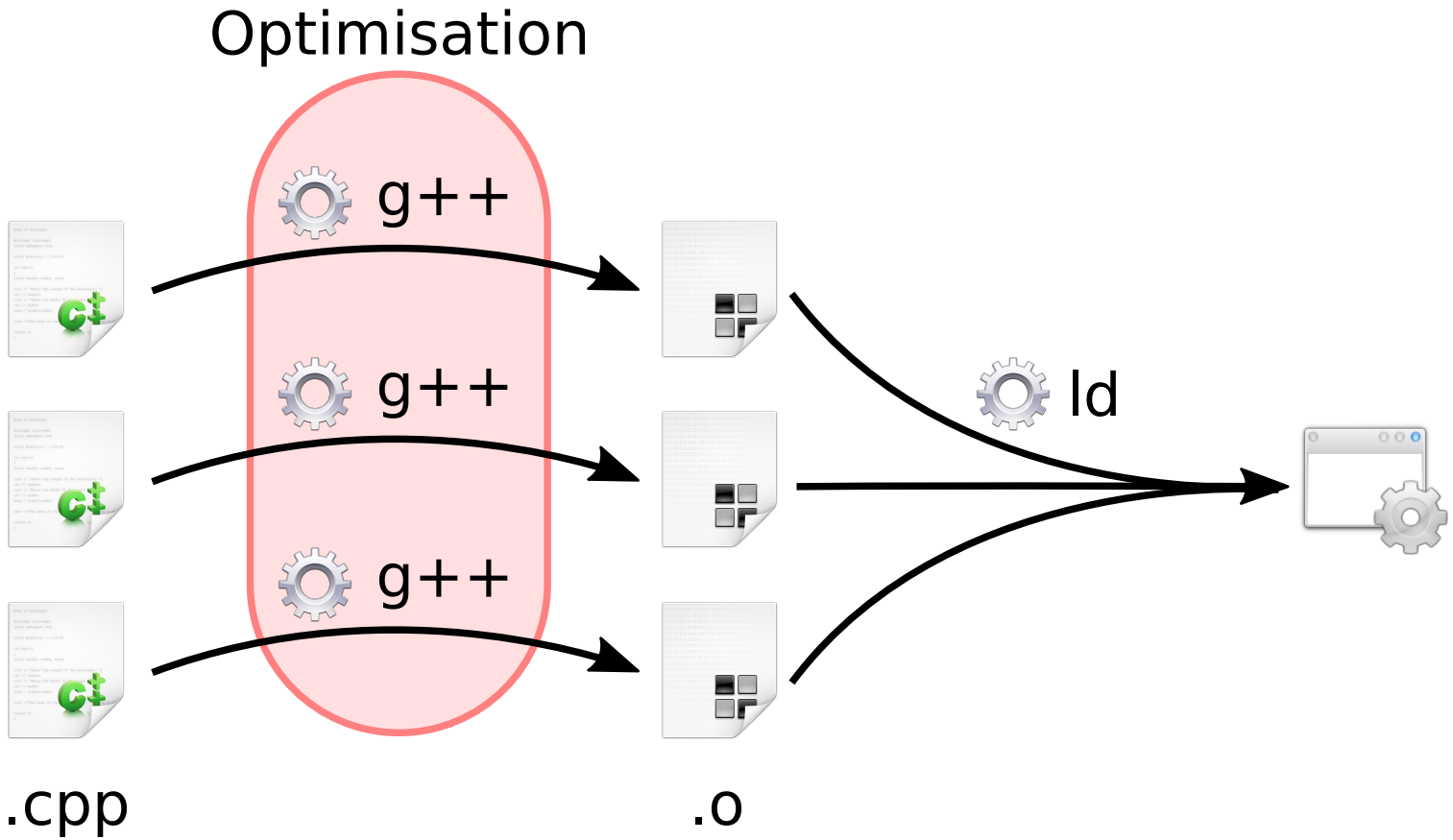
Dévirtualisation

Très importante en C++, cette optimisation remplace des appels de méthode indirects par des appels directs

```
1 ClasseFille *f = new ClasseFille;
2
3 f->ClasseFille::faireQuelqueChose();
```

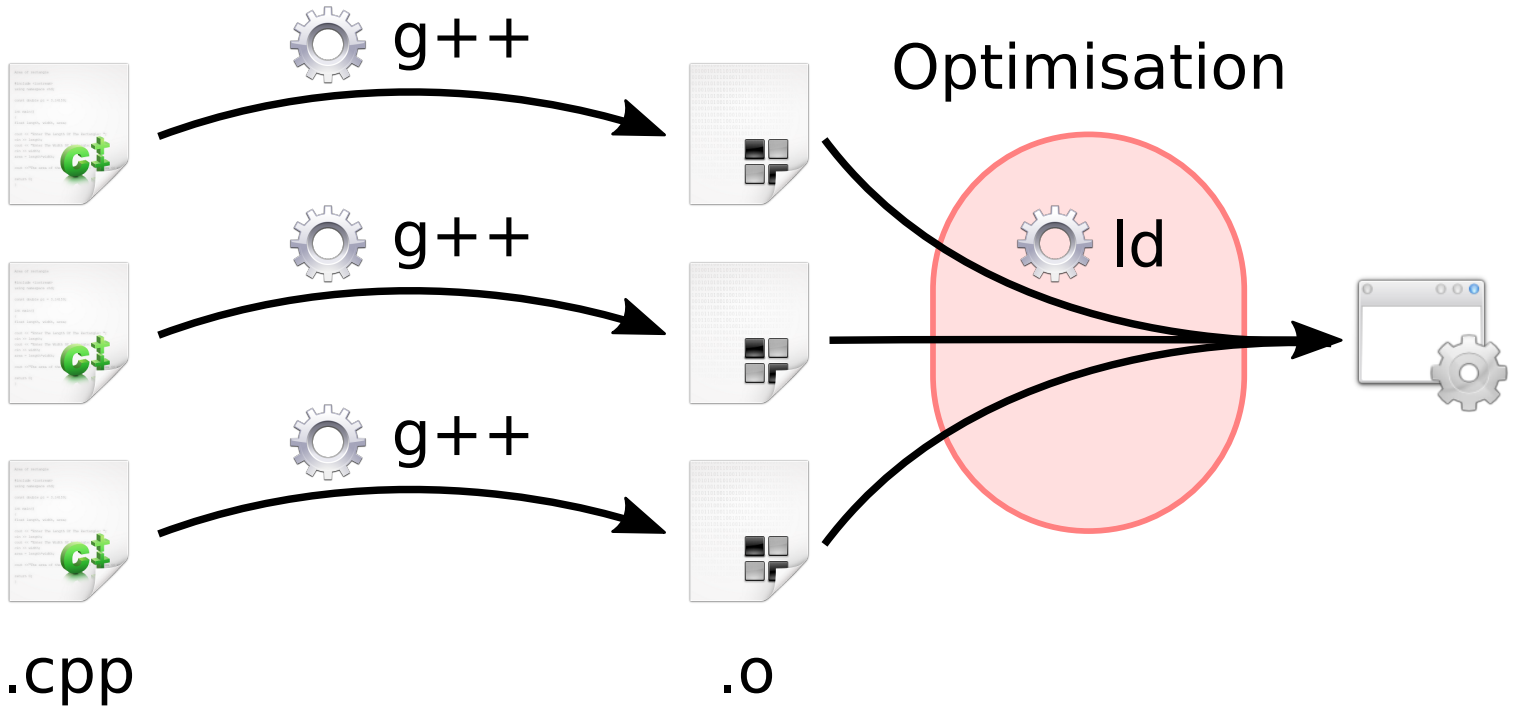
Link-Time Optimization

L'optimisation se fait au moment de produire chaque fichier .o, qui sont ensuite fusionnés



Link-Time Optimization

Le compilateur n'optimise plus rien, mais laisse le lieur faire le travail. Il a alors connaissance de tous les fichiers .o et l'inlining devient bien plus performant (30% de gains sur Firefox)

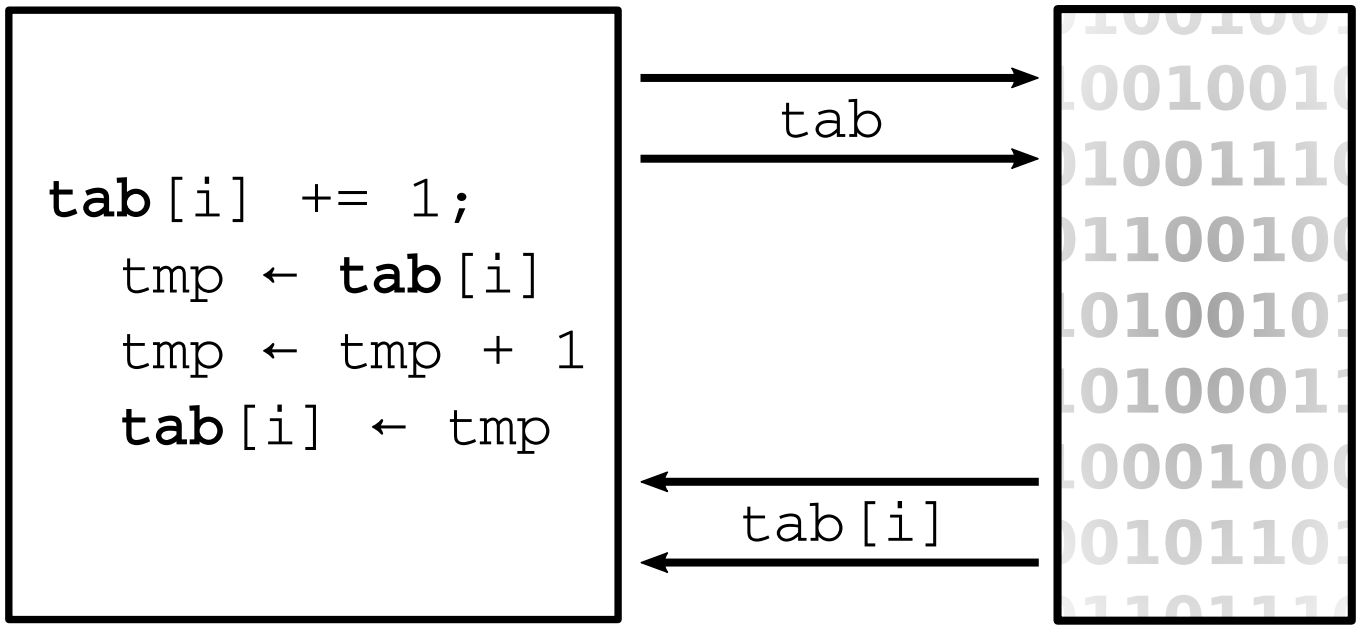


Link-Time Optimization

- Toutes les optimisations présentées ici sont activées par `-O2`
- Sauf LTO, qui s'active à l'aide de `-flto`

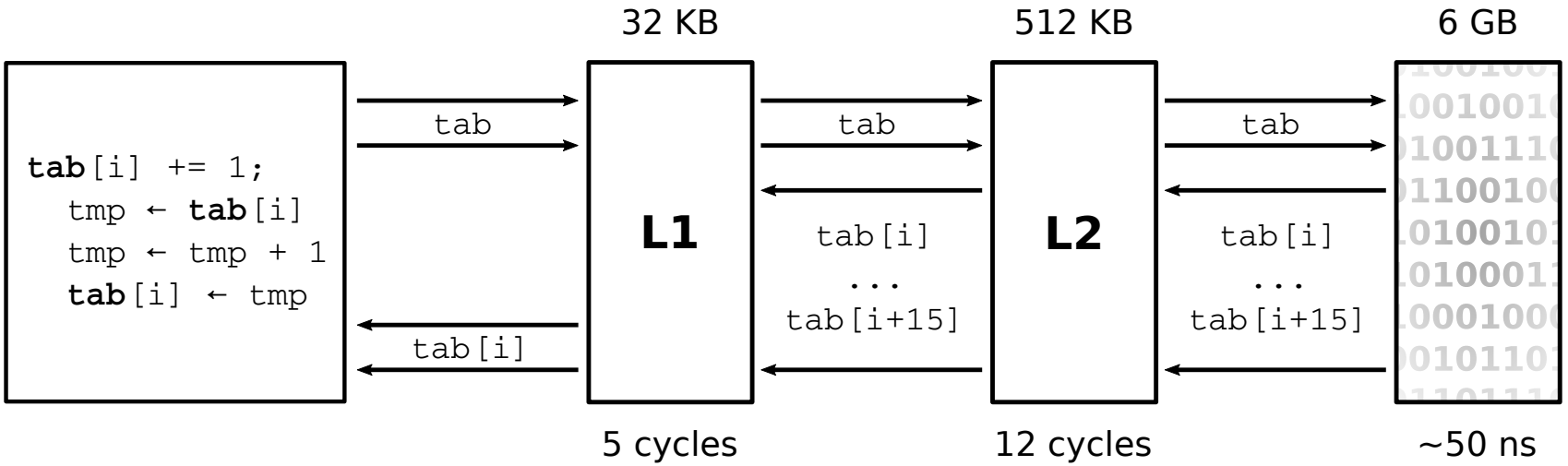
Table des matires

Accès mémoire direct



La mémoire centrale est beaucoup plus lente que le processeur !

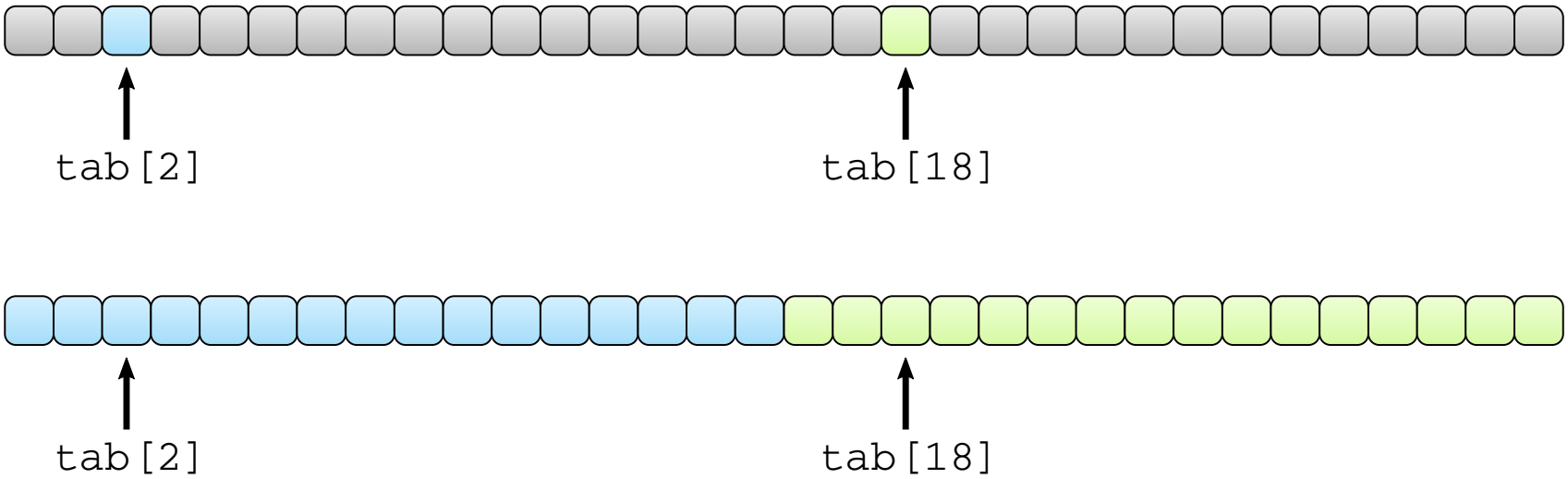
Hiérarchie des caches



Les caches réduisent la latence à condition de respecter certaines conditions

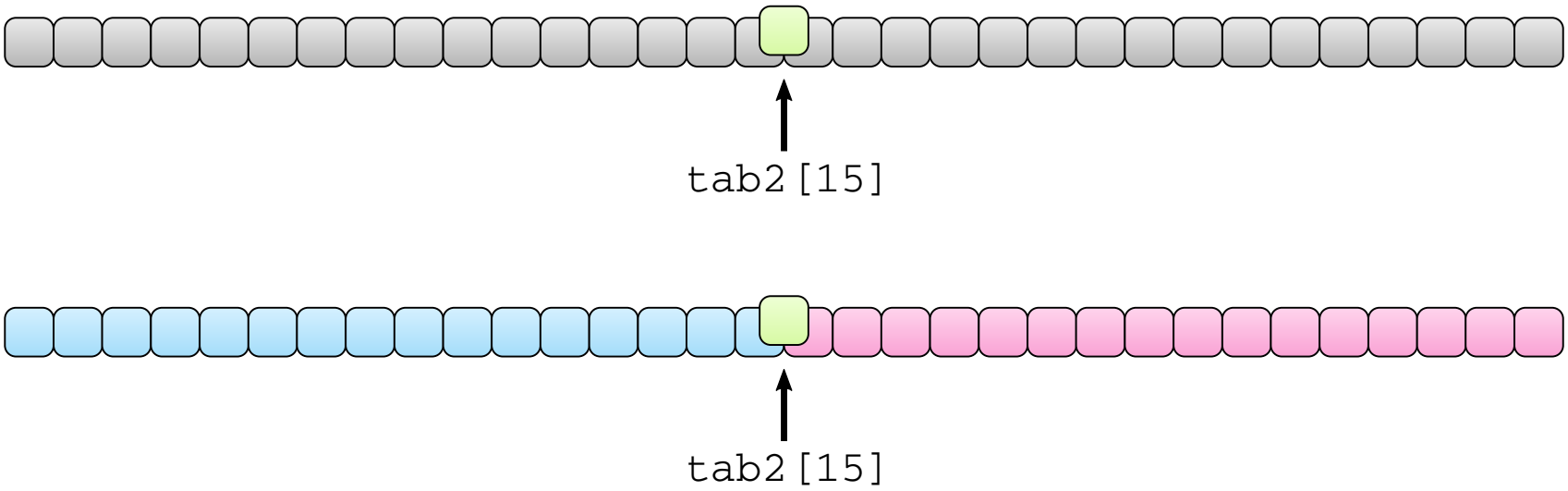
Accès fragmentés

Temps d'accès ralenti et des données inutiles sont lues



Accès non-alignés

Entre $2 \times S$ et 64 octets doivent être lus depuis la mémoire



Registres

- Temps d'accès très rapide
- Très peu nombreux (16 sur les derniers Intel et AMD 64-bit)
- Le compilateur met les variables les plus utilisées dans les registres, les autres en mémoire
- Évitez d'utiliser plus de 16 variables à la fois !

Conditions

- La condition de chaque `if` doit être évaluée
- Le processeur ne peut que difficilement prédire si un if sera vrai ou faux
- Les while et les for cachent des sauts conditionnels

Table des matires

Cas d'étude (rappel)

- Temps d'exécution : 2,60 secondes

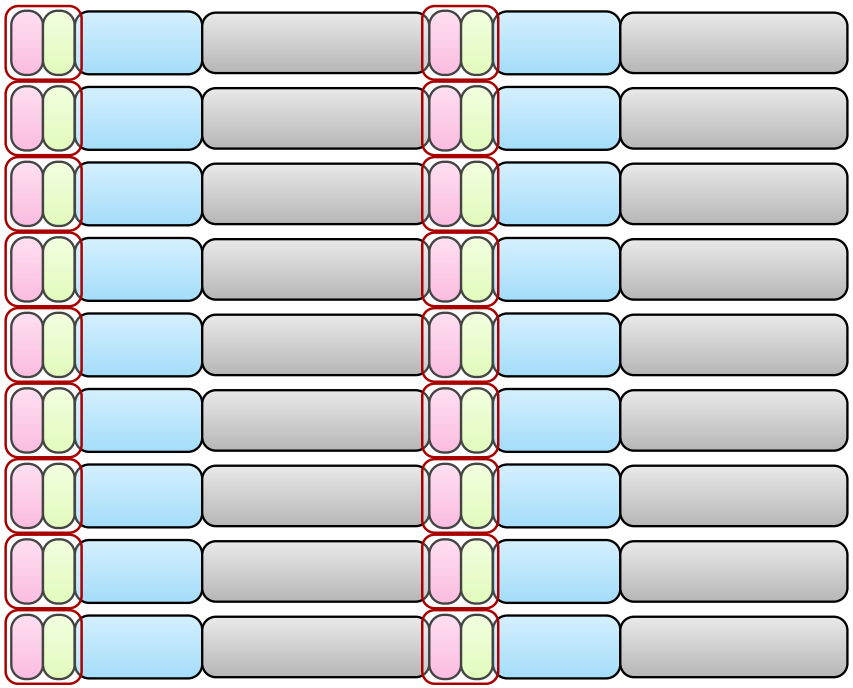
Structure Player

```
7  struct Player
8  {
9      bool isA;
10     bool isTolerant;
11     int payoff;
12     bool *judgesGood;
13 };
```

Utilisation de Player

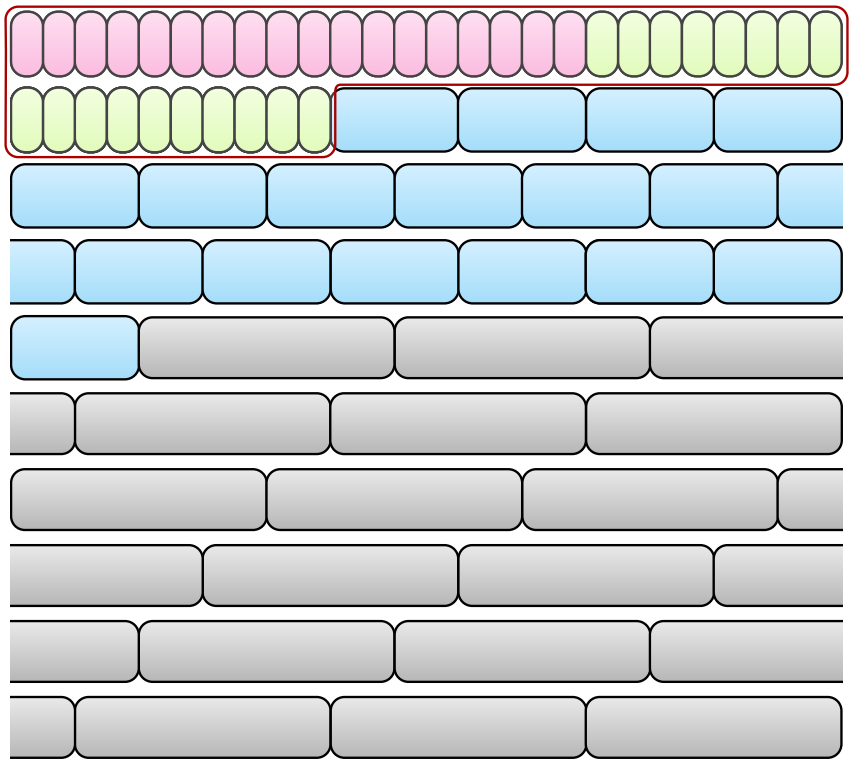
```
27 players[i].payoff = 0;
28 players[i].judgesGood = new bool[N];
```

Liste de structures (AOS)



Petits accès fragmentés

Structure de listes (SOA)



Un seul accès contigu

Structure Player (AOS)

```
7  struct Player
8  {
9      bool isA;
10     bool isTolerant;
11     int payoff;
12     bool *judgesGood;
13 };
14
15 Player *players = new Player[N];
```

Structure Players (SOA)

```
7  struct Players
8  {
9      bool *isA;
10     bool *isTolerant;
11     int *payoff;
12     bool **judgesGood;
13 };
14
15 players.isA = new bool[N];
16 players.isTolerant = new bool[N];
17 players.payoff = new int[N];
18 players.judgesGood = new bool*[N];
```

AOS vers SOA

```
1 players[i].payoff = 0;
2 players[i].judgesGood[j] = rand() & 3;
```

Le [i] change de place :

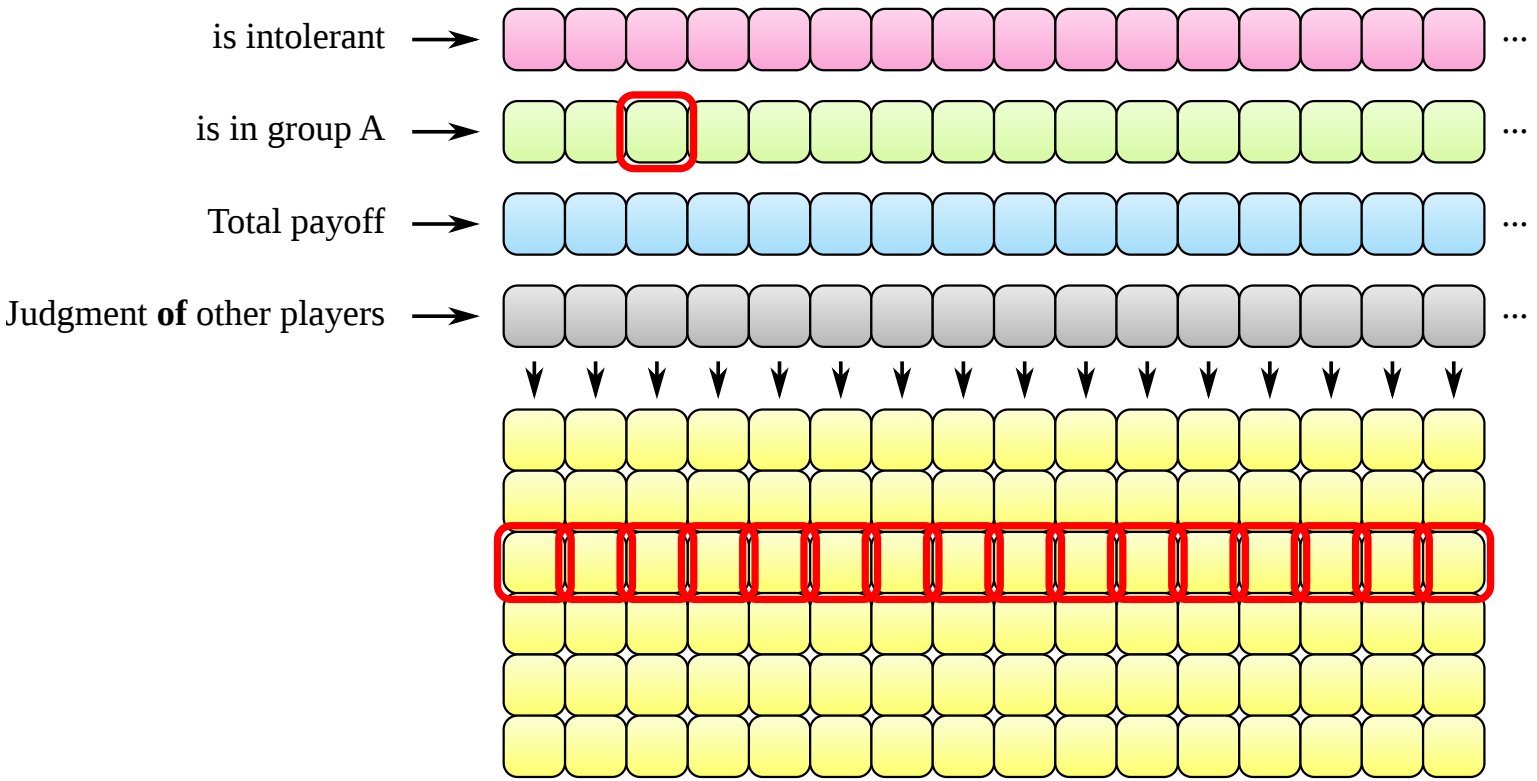
```
1 players.payoff[i] = 0;
2 players.judgesGood[i][j] = rand() & 3;
```

Performance

1. Avant : 2,60 secondes
2. Après : 2,56 secondes ($\times 1.01$)

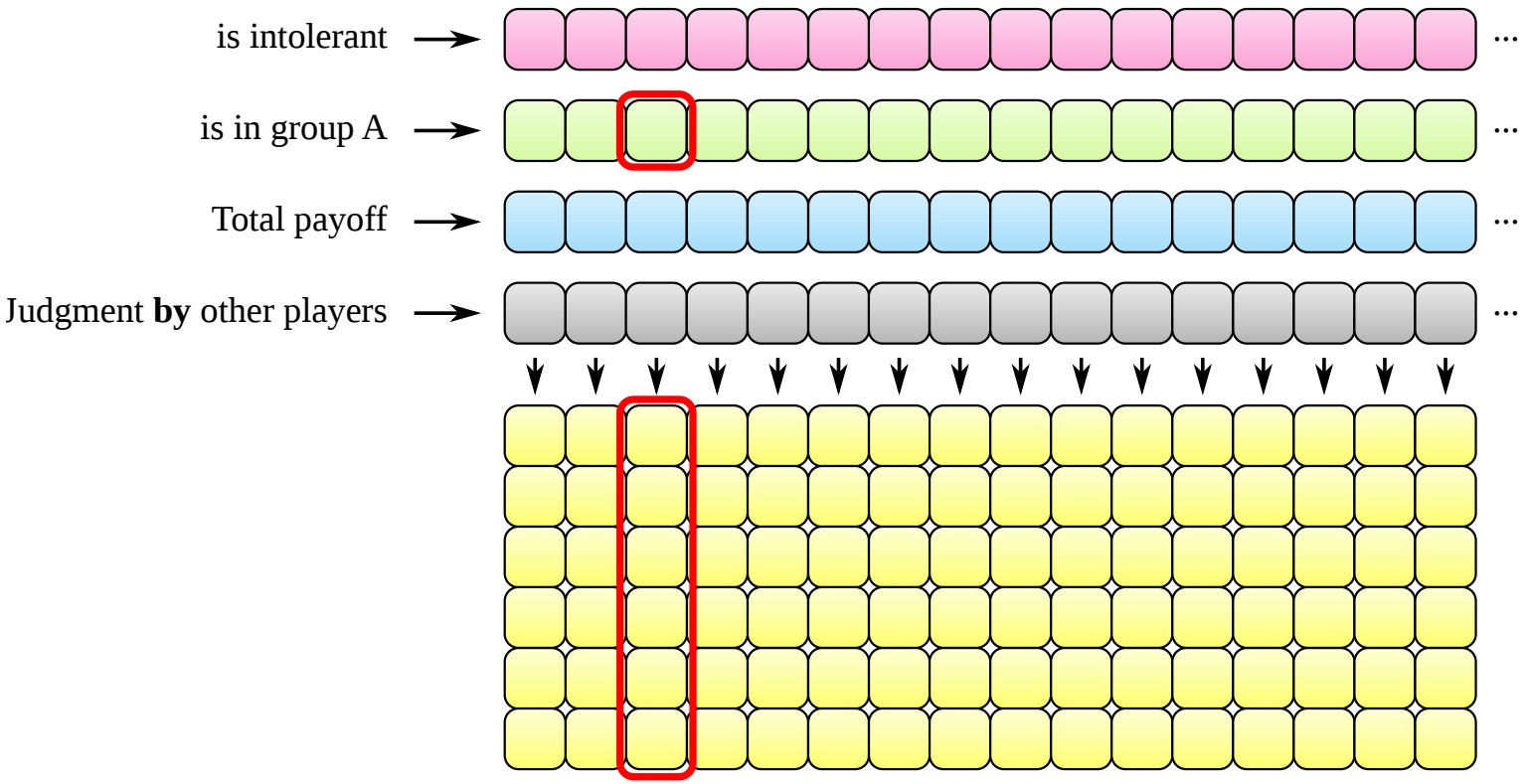
```
1 for (t=0; t<STEPS; ++t) {
2     ...
3     for (i=0; i<N; ++i) {
4         ...
5     }
6 }
```

Juger



Petits accès fragmentés

Être jugé



Un gros accès

Juger

```
7  struct Players
8  {
9      bool *isA;
10     bool *isTolerant;
11     int *payoff;
12     bool **judgesGood;
13 };
14
15 for (int other=0; other<N; ++other) {
16     b = players.judgesGood[other][player_b];
17 }
```

Être jugé

```
7  struct Players
8  {
9      bool *isA;
10     bool *isTolerant;
11     int *payoff;
12     bool **judgedGood;
13 };
14
15 for (int other=0; other<N; ++other) {
16     b = players.judgesGood[player_b][other];
17 }
```

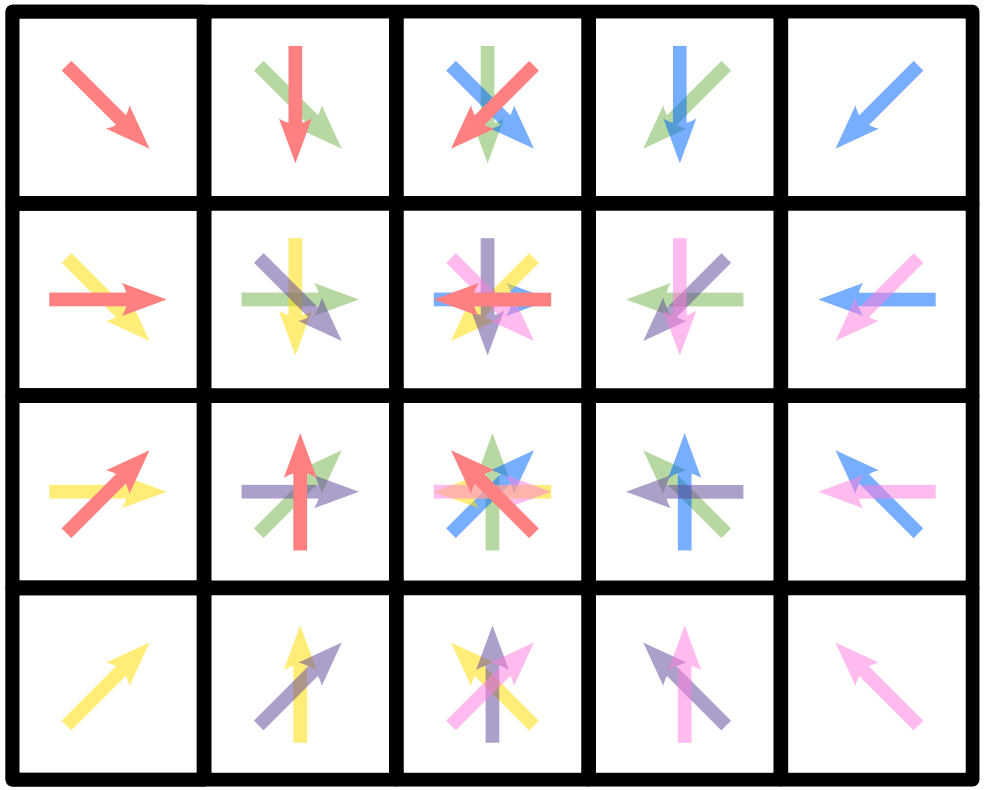
Performance

1. Avant : 2,56 secondes
2. Après : 0,72 secondes ($\times$3.55)

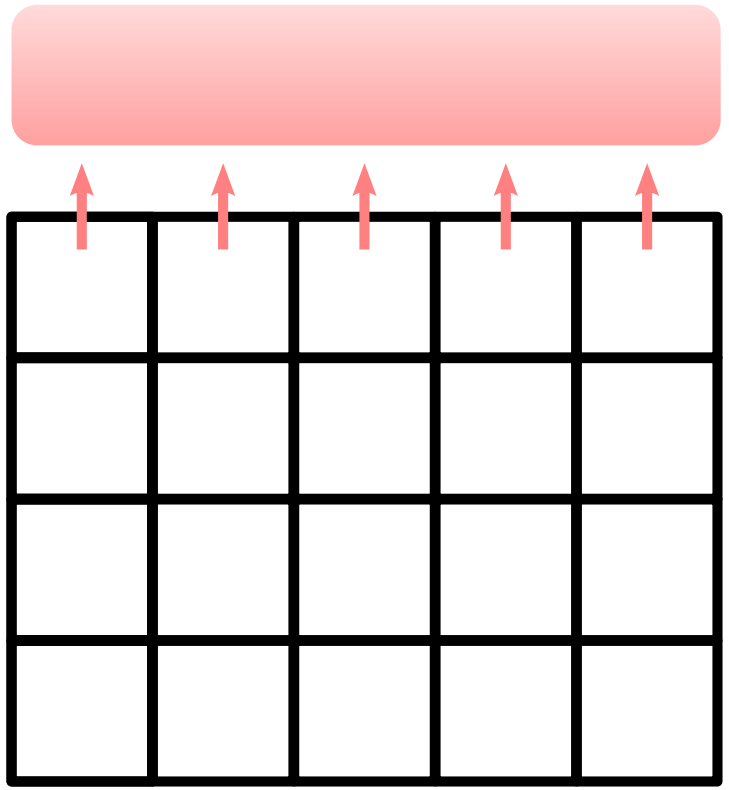
Aller plus loin

1. Utiliser plus de mémoire n'est pas toujours mauvais
2. Au lieu de lire 9 fois une grosse image, lisez-la 1 fois en stockant juste ce qu'il faut

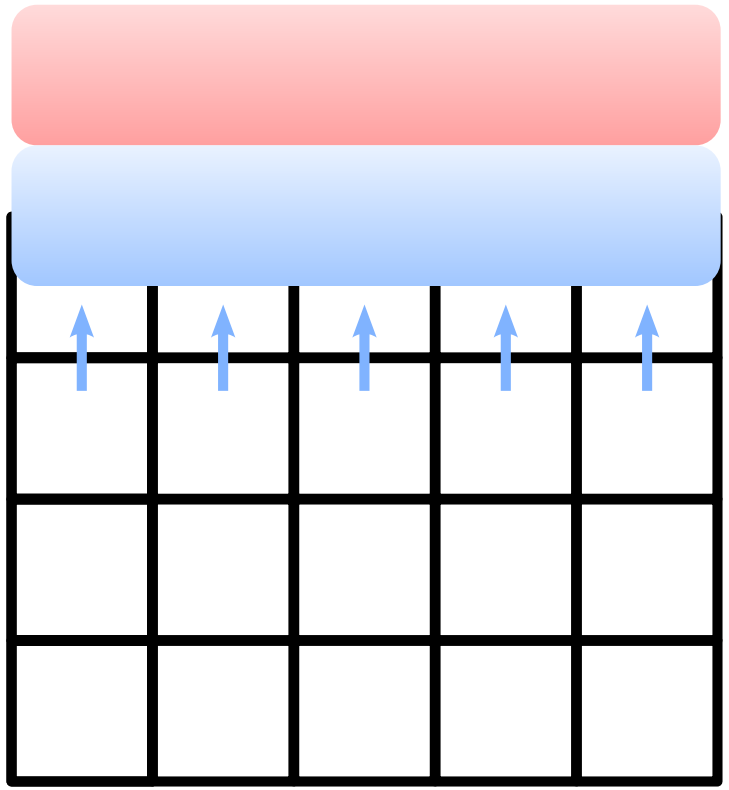
Exemple d'une convolution



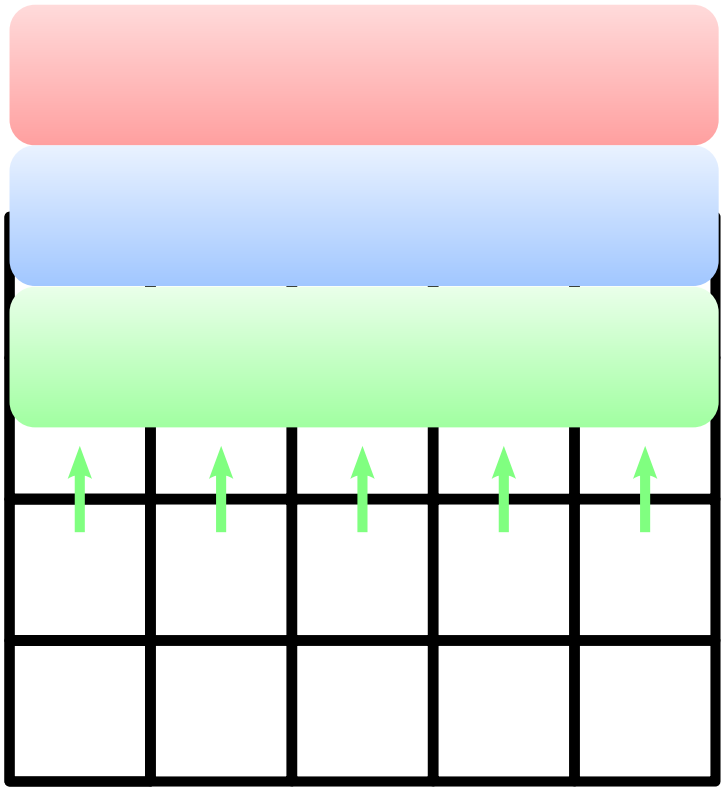
Exemple d'une convolution



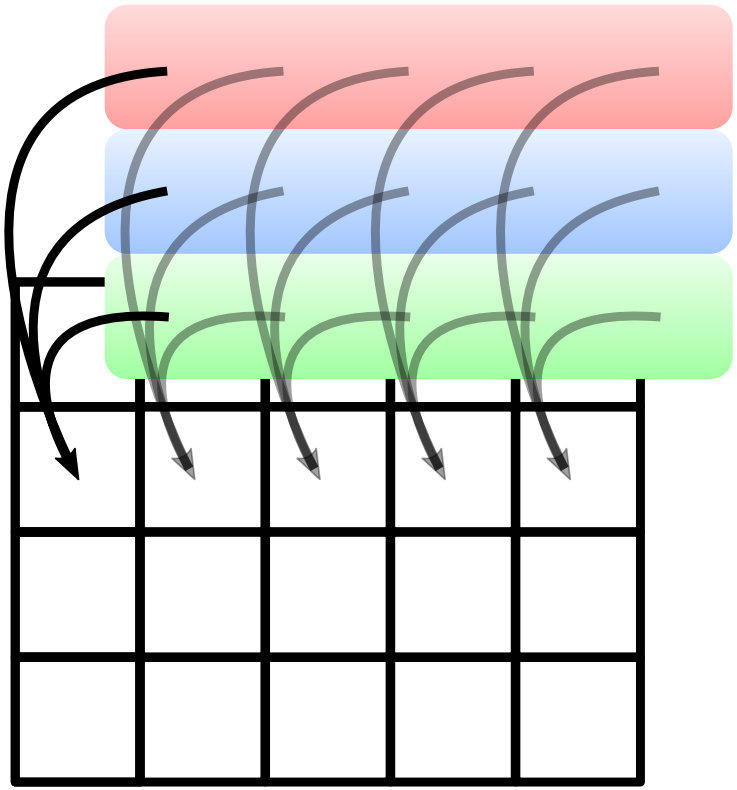
Exemple d'une convolution



Exemple d'une convolution



Exemple d'une convolution



Exemple d'une convolution

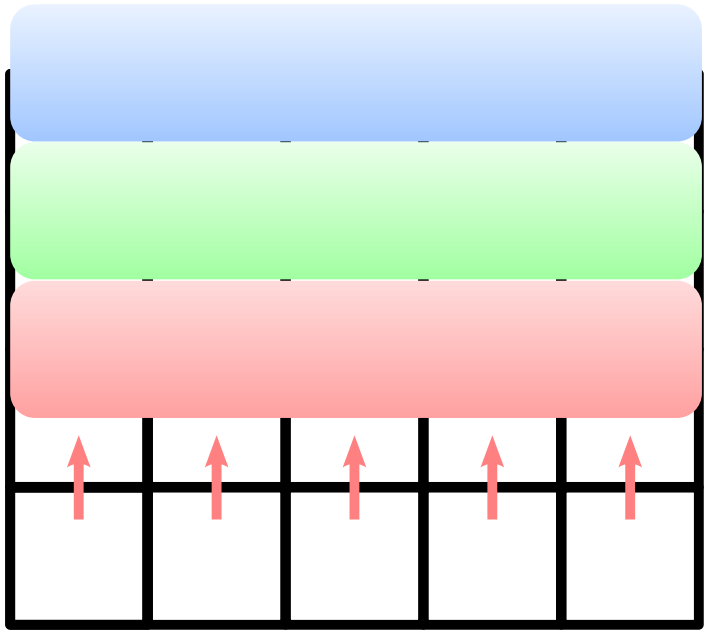


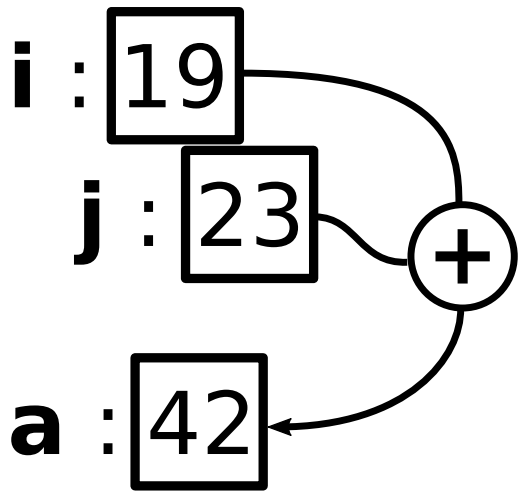
Table des matires

Opérations vectorielles

- Les caches, le décodage et l'ordonnancement des instructions, la gestion des sauts, tout ça prend beaucoup de place dans un CPU
- L'unité de calcul proprement dite est minuscule
- Alors autant en mettre beaucoup !
- Intel et AMD (SSE, AVX), ARM (NEON), IBM (AltiVec)
- SSE: 128 bits, $2 \times 8, 4 \times 4, 8 \times 2, 16 \times 1$, entiers, flottants et doubles
- AVX: SSE + 256 bits, $4 \times 8, 8 \times 4$, flottants et doubles
- AVX2: AVX + $8 \times 4, 16 \times 2, 32 \times 1$, entiers, flottants et doubles

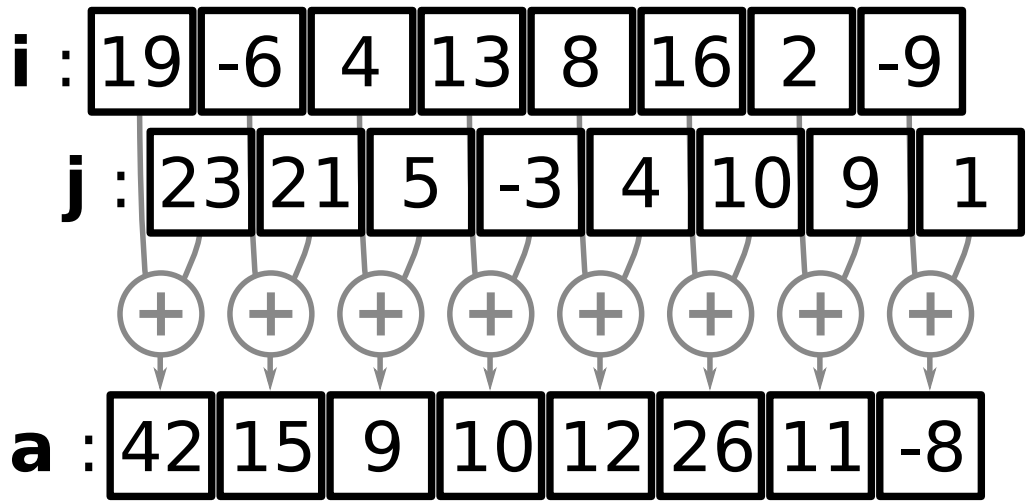
Addition scalaire

```
1 int a, i, j;
2 a = i + j;
```



Addition vectorielle (entiers)

```
1 #include <x86intrin.h>
2
3 __m128i a, i, j;
4 a = _mm_add_epi16(i, j);
```

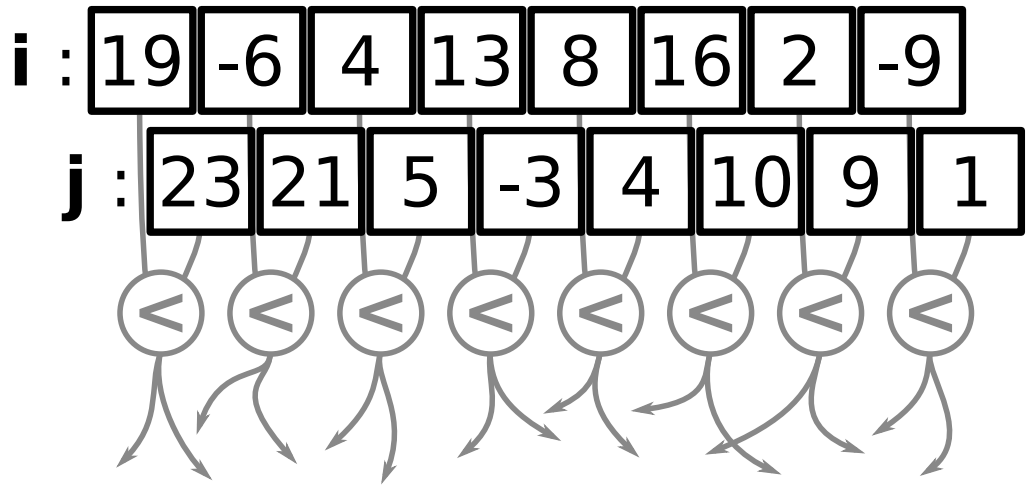


Addition vectorielle (flottants)

```
1 __m128 a, i, j;
2 a = _mm_add_ps(i, j);
```

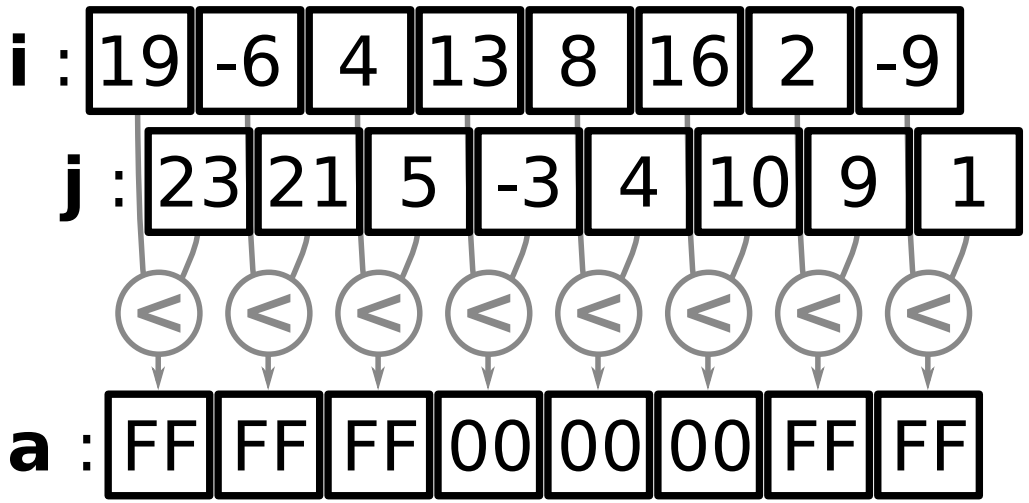
Comparaison de vecteurs

```
1 __m128i i, j;
2
3 if (i < j) { ??? }
```



Comparaison de vecteurs

```
1 __m128i a, i, j;
2 a = _mm_cmpgt_epi16(i, j);
```



Concevoir des algorithmes sans comparaison

```
1 int a;
2
3 if (cond) {
4     a = b + c;
5 } else {
6     a = -e;
7 }
```

Concevoir des algorithmes sans comparaison

```
1  int a;
2
3  // Deux branches
4  int a1 = b + c;
5  int a2 = -e;
6
7  // Fusion
8  int a1m = cond & a1;
9  int a2m = ~cond & a2;
10
11 a = a1m | a2m;
```

Concevoir des algorithmes sans comparaison

```
1  __m128i  a;
2
3  // Deux branches
4  __m128i a1 = _mm_add_epi8(b, c);
5  __m128i a2 = _mm_sub_epi8(zero, e);
6
7  // Fusion
8  __m128i a1m = _mm_and_si128(cond, a1);
9  __m128i a2m = _mm_andnot_si128(cond, a1);
10
11 a = _mm_or_si128(a1m, a2m);
```

Lire et stocker des vecteurs

```
1 char *pixels;
2 __m128i a;
3
4 a = _mm_loadu_si128((__m128i *)pixels);
5 _mm_storeu_si128((__m128i *)pixels, a);
```

Autovectorisation

- Utiliser des opérations vectorielles est difficile pour un humain, encore plus pour un compilateur
- Mais ils essaient !
- GCC, Clang, ICC et Visual C++ sont capables de quelques optimisations de base

Quelques exemples de code vectorisé par GCC ici : `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`

Performance

Le cas d'étude a été vectorisé en utilisant des opérations SSE.

1. Avant : 0,72 secondes
2. Après : 0,16 secondes ($\times$4.50)
3. 1000000 $\text{étapes} \times 1000\ \text{jugements} = 1000000000$
4. $1000000000/0.16 = 6.25$ milliards de jugements par seconde
5. Pas mal pour un processeur de 3,2 Ghz ! (près de deux jugements par cycle)

Performance

1. Chaque jugement nécessite de lire la tolérance du juge, le groupe de A et le groupe de B, et d'écrire le jugement
2. Chacune de ces informations fait un octet
3. $6.25 \times 3 = 18.75$ GB/s en lecture
4. $6.25 \times 1 = 6.25$ GB/s en écriture
5. $18.75 + 6.25 = 25$ GB/s d'échanges avec la mémoire
6. Un Intel Core i5-3230M a une bande passante mémoire de maximum 25.6 GB/s, mais ce n'est pas le facteur limitant ici !

Performance

Memory Specifications	
Max Memory Size (dependent on memory type)	32 GB
Memory Types	DDR3/L/-RS 1333/1600
Max # of Memory Channels	2
Max Memory Bandwidth	25.6 GB/s
ECC Memory Supported ‡	No

Analyse finale

La dernière version du cas d'étude a été profilée.

1. 45% : lire la mémoire
2. 14% : écrire dans la mémoire
3. 6% : décider si le joueur va coopérer ou non
4. 21% : `rand()`
5. 14% : autre (initialisation, boucles, etc)

Table des matires

Résumé

Des tas d'outils existent pour déboguer et analyser les performances de programmes.
L'optimisation n'est pas que réduire le nombre d'instructions d'un programme. Dans le cas d'étude, la vitesse a été multipliée par 16,25 sans retirer de ligne.
Avant optimisation, le processeur exécutait en moyenne 0.89 instructions par cycle (`perf stat`). Après optimisation, 2.35 instructions par cycle.

Références

- **GDB** : `http://sourceware.org/gdb/download/onlinedocs/gdb/index.html`
- **Valgrind** : `http://valgrind.org/`
- **Perf** : `https://perf.wiki.kernel.org/index.php/Main_Page`
- **Effets du cache** : `http://igoro.com/archive/gallery-of-processor-cache-effects/`
- Dragomir Milojevic, **Microprocessor Architectures** (ELEC-H473), 2015

(icônes Oxygen : `http://www.kde.org/`, LGPLv3)