Optimisations

Denis Bouhineau Fabienne Carrier Stéphane Devismes

Université Grenoble Alpes

20 avril 2020

Plan

Plan

Résultats connus

- Calcul avec des constantes (version attendue)

Résultats connus

- Calcul avec des constantes (version attendue)

```
#include <stdio.h>

int main () {
  int i=4, j=5;
  printf("4+5=%d\n",i+j);
  return;}
```

Résultats connus

- Calcul avec des constantes (version attendue)

```
#include <stdio.h>

int main () {
  int i=4, j=5;
  printf("4+5=%d\n",i+j);
  return;}
```

```
        .file   "progConstantes.c"
        .section .rodata .align 2
.LC0:   .ascii  "4+5=%d\012\000"
        .text .align 2
        .global main .type main, %function
main:   stmfd   sp!, {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #8
        mov     r3, #4
        str     r3, [fp, #-8]
        mov     r3, #5
        str     r3, [fp, #-12]
        ldr     r2, [fp, #-8]
        ldr     r3, [fp, #-12]
        add     r3, r2, r3
        ldr     r0, .L2
        mov     r1, r3
        bl      printf
        mov     r0, r3
        sub     sp, fp, #4
        ldmfd   sp!, {fp, lr}
        bx      lr
.L3:    .align  2
.L2:    .word   .LC0
        .size main, .-main
        .ident "GCC: (GNU) 4.5.3"
```

Résultats connus

- Calcul avec des constantes (version optimisée)

```
#include <stdio.h>

int main () {
  int i=4, j=5;
  printf("4+5=%d\n",i+j);
  return;}
```

```
        .file   "progConstantes.c"
        .text
        .align  2
        .global main
        .type   main, %function
main:
        stmfd   sp!, {r3, lr}
        ldr     r0, .L2
        mov     r1, #9
        bl      printf
        ldmfd   sp!, {r3, lr}
        bx      lr
.L3:
        .align  2
.L2:
        .word   .LC0
        .size   main, .-main
        .section .rodata.str1.4,"aMS",%progbits,1
        .align  2
.LC0:
        .ascii  "4+5=%d\012\000"
        .ident  "GCC: (GNU) 4.5.3"
```

Opérations équivalentes

- Divisions équivalentes

Division arbitraire

- Division arbitraire (version attendue)

Division arbitraire

- Division arbitraire (version attendue)

```
#include <stdio.h>

int main () {
  int i,j;
  scanf("%d",&i);
  scanf("%d",&j);
  printf("%d / %d=%d\n",i,j,i/j);
  return;}
```

Division arbitraire

- Division arbitraire (version attendue)

```
#include <stdio.h>

int main () {
  int i,j;
  scanf("%d",&i);
  scanf("%d",&j);
  printf("%d / %d=%d\n",i,j,i/j);
  return;}
```

```
main:   stmfd   sp!, {r4, r5, fp, lr}
        add     fp, sp, #12
        sub     sp, sp, #8
        sub     r3, fp, #16
        ldr     r0, .L2
        mov     r1, r3
        bl      scanf
        sub     r3, fp, #20
        ldr     r0, .L2
        mov     r1, r3
        bl      scanf
        ldr     r5, [fp, #-16]
        ldr     r4, [fp, #-20]
        ldr     r2, [fp, #-16]
        ldr     r3, [fp, #-20]
        mov     r0, r2
        mov     r1, r3
        bl      __aeabi_idiv
        mov     r3, r0
        ldr     r0, .L2+4
        mov     r1, r5
        mov     r2, r4
        bl      printf
        mov     r0, r3
        sub     sp, fp, #12
        ldmfd   sp!, {r4, r5, fp, lr}
        bx      lr
.L3:    .align  2
.L2:    .word   .LC0 .word      .LC1
        .size   main, .-main .ident "GCC: (GNU) 4.5.3"
```

Division par 2

- Division par 2 (version optimisée)

```
#include <stdio.h>

int main () {
  int i;
  scanf("%d",&i);
  printf("%d / 2 =%d\n",i,i/2);
  return;}
```

Division par 2

- Division par 2 (version optimisée)

```
#include <stdio.h>

int main () {
  int i;
  scanf("%d",&i);
  printf("%d / 2 =%d\n",i,i/2);
  return;}
```

```
main:
        stmfd   sp!, {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #8
        sub     r3, fp, #8
        ldr     r0, .L2
        mov     r1, r3
        bl      scanf
        ldr     r2, [fp, #-8]
        ldr     r3, [fp, #-8]
        mov     r1, r3, lsr #31
        add     r3, r1, r3
        mov     r3, r3, asr #1
        ldr     r0, .L2+4
        mov     r1, r2
        mov     r2, r3
        bl      printf
        mov     r0, r3
        sub     sp, fp, #4
        ldmfd   sp!, {fp, lr}
        bx      lr
.L3:
        .align  2
.L2:
        .word   .LC0
        .word   .LC1
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.3"
```

Division par 10

- Division par 10 (version optimisée)

```
#include <stdio.h>

int main () {
  int i;
  scanf("%d",&i);
  printf("%d / 10=%d\n",i,i/10);
  return 0;}
```

Division par 10

- Division par 10 (version optimisée)

```
#include <stdio.h>

int main () {
  int i;
  scanf("%d",&i);
  printf("%d / 10=%d\n",i,i/10);
  return 0;}
```

```
main:
   [...]
        bl      scanf
        ldr     r2, [fp, #-8]
        ldr     r3, [fp, #-8]
        ldr     r1, .L2+4
        smull   r0, r1, r3, r1
        mov     r1, r1, asr #2
        mov     r3, r3, asr #31
        rsb     r3, r3, r1
        ldr     r0, .L2+8
        mov     r1, r2
        mov     r2, r3
        bl      printf
        mov     r0, r3
        sub     sp, fp, #4
        ldmfd   sp!, {fp, lr}
        bx      lr
.L3:
        .align  2
.L2:
        .word   .LC0
        .word   1717986919
        .word   .LC1
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.3"
```

Division par 10

- Division par 10 (version optimisée)

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?
- 1717986919

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?

- 1717986919 = 1100110011001100110011001100111 en binaire

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?

- 1717986919 = 1100110011001100110011001100111 en binaire
- 1717986919 * 10 = 17179869190

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?

- 1717986919 = 1100110011001100110011001100111 en binaire

- 1717986919 * 10 = 17179869190 = 10000000000000000000000000000110 en binaire !

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?

- 1717986919 = 1100110011001100110011001100111 en binaire

- 1717986919 * 10 = 17179869190 = 10000000000000000000000000000110 en binaire !
- 1717986919 * 10 = 17179869190 égal à peu près à 2 puissance 34 !

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?

- 1717986919 = 1100110011001100110011001100111 en binaire

- 1717986919 * 10 = 17179869190 = 10000000000000000000000000000110 en binaire !
- 1717986919 * 10 = 17179869190 égal à peu près à 2 puissance 34 !
- 2 puissance 34 = 17179869184

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?

- 1717986919 = 1100110011001100110011001100111 en binaire

- 1717986919 * 10 = 17179869190 = 10000000000000000000000000000110 en binaire !
- 1717986919 * 10 = 17179869190 égal à peu près à 2 puissance 34 !
- 2 puissance 34 = 17179869184
- 2 puissance 34 / 10 = 1717986918

Division par 10

- Division par 10 (version optimisée)
- Pourquoi 1717986919 ?
- Pourquoi ASR ?

- 1717986919 = 1100110011001100110011001100111 en binaire

- 1717986919 * 10 = 17179869190 = 1000000000000000000000000000110 en binaire !
- 1717986919 * 10 = 17179869190 égal à peu près à 2 puissance 34 !
- 2 puissance 34 = 17179869184
- 2 puissance 34 / 10 = 1717986918
- Conclusion : $i/10 = i * (2puissance34/10) >> 34$

Arithmétique (conclusions)

- Les expressions calculées ne sont pas nécessairement celles programmées

Arithmétique (conclusions)

- Les expressions calculées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les expressions sous forme logiques / littérales

Arithmétique (conclusions)

- Les expressions calculées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les expressions sous forme logiques / littérales
- Le compilateur optimise !

Arithmétique (conclusions)

- Les expressions calculées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les expressions sous forme logiques / littérales
- Le compilateur optimise !
- Gain espéré : temps de calcul

Plan

Conditionnelles

- Conditionnelle (version attendue)

Conditionnelles

- Conditionnelle (version attendue)

```
#include <stdio.h>

int main () {
  int i=4;
  if (i&1) {
    printf("4 est impair\n");}
  else {
          printf("4 est pair\n");}
  return 0;}
```

Conditionnelles

- Conditionnelle (version attendue)
- (version assembleur X86)

```
#include <stdio.h>

int main () {
  int i=4;
  if (i&1) {
    printf("4 est impair\n");}
  else {
          printf("4 est pair\n");}
  return 0;}
```

```
main:
.LFB0:  .cfi_startproc
        pushq       %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        subq        $16, %rsp
        movl        $4, -4(%rbp)
        movl        -4(%rbp), %eax
        andl        $1, %eax
        testl       %eax, %eax
        je          .L2
        movl        $.LC0, %edi
        call        puts
        jmp         .L3
.L2:    movl        $.LC1, %edi
        call        puts
.L3:    movl        $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Conditionnelle

- Conditionnelle (version optimisée)
- (version assembleur X86)

```
#include <stdio.h>

int main () {
  int i=4;
  if (i&1) {
    printf("4 est impair\n");}
  else {
          printf("4 est pair\n");}
  return 0;}
```

Conditionnelle

- Conditionnelle (version optimisée)
- (version assembleur X86)

```
#include <stdio.h>

int main () {
  int i=4;
  if (i&1) {
    printf("4 est impair\n");}
  else {
          printf("4 est pair\n");}
  return 0;}
```

```
main:
.LFB24:
        .cfi_startproc
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
        movl    $.LC0, %edi
        call    puts
        addq    $8, %rsp
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

Boucles

- Boucle (version attendue)

```
#include <stdio.h>

int main () {
  int i, s=0, a;
  for(i=0;i<10;i++) {
    scanf("%d",&a);
    s=s+a;}
  printf("Somme : %d\n",s);
  return 0;}
```

Boucles

- Boucle (version attendue)
- (version clang)

```
#include <stdio.h>

int main () {
  int i, s=0, a;
  for(i=0;i<10;i++) {
    scanf("%d",&a);
    s=s+a;}
  printf("Somme : %d\n",s);
  return 0;}
```

```
main:    [...]    movl    $0, -8(%rbp)
.LBB0_1:           # =>This Inner Loop Header: Depth=
         cmpl      $10, -8(%rbp)
         jge       .LBB0_4
# BB#2:            #   in Loop: Header=BB0_1 Depth=1
         movabsq   $.L.str, %rdi
         leaq      -16(%rbp), %rsi
         movb      $0, %al
         callq     __isoc99_scanf
         movl      -12(%rbp), %ecx
         addl      -16(%rbp), %ecx
         movl      %ecx, -12(%rbp)
         movl      %eax, -20(%rbp) # 4-byte Spill
# BB#3:            #   in Loop: Header=BB0_1 Depth=1
         movl      -8(%rbp), %eax
         addl      $1, %eax
         movl      %eax, -8(%rbp)
         jmp       .LBB0_1
.LBB0_4: movabsq   $.L.str1, %rdi
         movl      -12(%rbp), %esi
         movb      $0, %al
         callq     printf
         movl      $0, %esi
         movl      %eax, -24(%rbp) # 4-byte Spill
         movl      %esi, %eax
         addq      $32, %rsp
         popq      %rbp
         retq
.Ltmp3:  .size      main, .Ltmp3-main
         .cfi_endproc
```

Boucle

- Boucle (version optimisée)
- (version clang)

```
#include <stdio.h>

int main () {
  int i, s=0, a;
  for(i=0;i<10;i++) {
    scanf("%d",&a);
    s=s+a;}
  printf("Somme : %d\n",s);
  return 0;}
```

Boucle

- Boucle (version optimisée)
- (version clang)

```
#include <stdio.h>

int main () {
  int i, s=0, a;
  for(i=0;i<10;i++) {
    scanf("%d",&a);
    s=s+a;}
  printf("Somme : %d\n",s);
  return 0;}
```

```
main:   [...]
        leaq    4(%rsp), %r14
        movl    $.L.str, %edi
        xorl    %eax, %eax
        movq    %r14, %rsi
        callq    __isoc99_scanf
        movl    4(%rsp), %ebx
        movl    $.L.str, %edi
        xorl    %eax, %eax
        movq    %r14, %rsi
        callq    __isoc99_scanf
        addl    4(%rsp), %ebx
        movl    $.L.str, %edi
        xorl    %eax, %eax
        movq    %r14, %rsi
        callq    __isoc99_scanf
        addl    4(%rsp), %ebx
        movl    $.L.str, %edi
        xorl    %eax, %eax
        movq    %r14, %rsi
        callq    __isoc99_scanf
        addl    4(%rsp), %ebx
        movl    $.L.str, %edi
        xorl    %eax, %eax
        movq    %r14, %rsi
        [...]
```

Structures de contrôle (conclusions)

- Les structures de contrôle utilisées ne sont pas nécessairement celles programmées

Structures de contrôle (conclusions)

- Les structures de contrôle utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les structures de contrôle les plus claires

Structures de contrôle (conclusions)

- Les structures de contrôle utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les structures de contrôle les plus claires
- Le compilateur optimise ! (tous les compilateurs, pour tous les asm)

Structures de contrôle (conclusions)

- Les structures de contrôle utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les structures de contrôle les plus claires
- Le compilateur optimise ! (tous les compilateurs, pour tous les asm)
- Mais attention : dérouler une boucle peut prendre de la place dans le fichier exécutable et du temps à la compilation ...

Structures de contrôle (conclusions)

- Les structures de contrôle utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les structures de contrôle les plus claires
- Le compilateur optimise ! (tous les compilateurs, pour tous les asm)
- Mais attention : dérouler une boucle peut prendre de la place dans le fichier exécutable et du temps à la compilation ...
- Gain espéré : temps d'exécution des branchements et de calcul des compteurs d'itération

Plan

Fonctions en ligne (inline)

- Fonction (version attendue)

```
#include <stdio.h>

int somme(int a,int b) {
        return a+b;}

int main () {
  int i, j;
  scanf("%d",&i);
  scanf("%d",&j);
  printf("4+5=%d\n",somme(i,j));
  return;}
```

Fonctions en ligne (inline)

- Fonction (version attendue)

```
#include <stdio.h>

int somme(int a,int b) {
        return a+b;}

int main () {
  int i, j;
  scanf("%d",&i);
  scanf("%d",&j);
  printf("4+5=%d\n",somme(i,j));
  return;}
```

```
somme:
[...]   movl    -8(%rbp), %eax
        movl    -4(%rbp), %edx
        addl    %edx, %eax
        popq    %rbp
        ret
[...]   main:
[...]   movl    $0, %eax
        call    __isoc99_scanf
        leaq    -4(%rbp), %rax
        movq    %rax, %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    __isoc99_scanf
        movl    -4(%rbp), %edx
        movl    -8(%rbp), %eax
        movl    %edx, %esi
        movl    %eax, %edi
        call    somme
        movl    %eax, %esi
        movl    $.LC1, %edi
        movl    $0, %eax
        call    printf
        nop
        leave
        ret
```

Fonctions en ligne (inline)

- Fonction (version optimisée)

```
#include <stdio.h>

int somme(int a,int b) {
        return a+b;}

int main () {
  int i, j;
  scanf("%d",&i);
  scanf("%d",&j);
  printf("4+5=%d\n",somme(i,j));
  return;}
```

Fonctions en ligne (inline)

- Fonction (version optimisée)

```
#include <stdio.h>

int somme(int a,int b) {
        return a+b;}

int main () {
  int i, j;
  scanf("%d",&i);
  scanf("%d",&j);
  printf("4+5=%d\n",somme(i,j));
  return;}
```

```
somme:
[...]
        leal    (%rdi,%rsi), %eax
        ret
main:
[...]
        movl    $.LC0, %edi
        movl    $0, %eax
        call    __isoc99_scanf
        leaq    12(%rsp), %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    __isoc99_scanf
        movl    12(%rsp), %edx
        addl    8(%rsp), %edx
        movl    $.LC1, %esi
        movl    $1, %edi
        movl    $0, %eax
        call    __printf_chk
        addq    $24, %rsp
        ret
```

Fonctions récursives

- Fonction récursive (version attendue)

```
int somme(int a,int s) {
  if (!a) {
        return s;}
  else {
    return somme(a-1,a+s);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i,0));
  return 0;}
```

Fonctions récursives

- Fonction récursive (version attendue)
- Somme (version attendue)

```
int somme(int a,int s) {
  if (!a) {
        return s;}
  else {
    return somme(a-1,a+s);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i,0));
  return 0;}
```

```
somme:
        stmfd   sp!, {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #8
        str     r0, [fp, #-8]
        str     r1, [fp, #-12]
        ldr     r3, [fp, #-8]
        cmp     r3, #0
        bne     .L2
        ldr     r3, [fp, #-12]
        b       .L3
.L2:
        ldr     r3, [fp, #-8]
        sub     r2, r3, #1
        ldr     r1, [fp, #-8]
        ldr     r3, [fp, #-12]
        add     r3, r1, r3
        mov     r0, r2
        mov     r1, r3
        bl      somme
        mov     r3, r0
.L3:
        mov     r0, r3
        sub     sp, fp, #4
        ldmfd   sp!, {fp, lr}
        bx      lr
```

Fonctions récursives

- Fonction récursive (version attendue)
- Main (version attendue)

```
int somme(int a,int s) {
  if (!a) {
        return s;}
  else {
    return somme(a-1,a+s);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i,0));
  return 0;}
```

```
main:
        stmfd   sp!, {r4, fp, lr}
        add     fp, sp, #8
        sub     sp, sp, #12
        sub     r3, fp, #16
        ldr     r0, .L5
        mov     r1, r3
        bl      scanf
        ldr     r4, [fp, #-16]
        ldr     r3, [fp, #-16]
        mov     r0, r3
        mov     r1, #0
        bl      somme
        mov     r3, r0
        ldr     r0, .L5+4
        mov     r1, r4
        mov     r2, r3
        bl      printf
        mov     r3, #0
        mov     r0, r3
        sub     sp, fp, #8
        ldmfd   sp!, {r4, fp, lr}
        bx      lr
```

Fonctions récursives

- Fonction récursive (version optimisée)
- Somme (version optimisée)

```
int somme(int a,int s) {
  if (!a) {
        return s;}
  else {
    return somme(a-1,a+s);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i,0));
  return 0;}
```

```
somme:
        stmfd   sp!, {r3, lr}
        subs    r3, r0, #0
        beq     .L2
        sub     r0, r3, #1
        add     r1, r1, r3
        bl      somme
        mov     r1, r0
.L2:
        mov     r0, r1
        ldmfd   sp!, {r3, lr}
        bx      lr
```

Fonctions récursives

- Fonction récursive (version optimisée)
- Main (version optimisée)

```
int somme(int a,int s) {
  if (!a) {
        return s;}
  else {
    return somme(a-1,a+s);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i,0));
  return 0;}
```

```
main:
        stmfd   sp!, {r4, lr}
        sub     sp, sp, #8
        ldr     r0, .L4
        add     r1, sp, #4
        bl      scanf
        ldr     r4, [sp, #4]
        mov     r0, r4
        mov     r1, #0
        bl      somme
        mov     r2, r0
        ldr     r0, .L4+4
        mov     r1, r4
        bl      printf
        mov     r0, #0
        add     sp, sp, #8
        ldmfd   sp!, {r4, lr}
        bx      lr
```

Fonctions récursives

- Fonction récursive (version + optimisée)
- Somme (version + optimisée)

```
int somme(int a,int s) {
  if (!a) {
        return s;}
  else {
    return somme(a-1,a+s);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i,0));
  return 0;}
```

```
somme:
        subs    r3, r0, #0
        bne     .L5
        b       .L2
.L4:
        mov     r3, r2
.L5:
        subs    r2, r3, #1
        add     r1, r1, r3
        bne     .L4
.L2:
        mov     r0, r1
        bx      lr
```

Fonctions récursives

- Fonction récursive (version + optimisée)
- Main (version + optimisée)

```
int somme(int a,int s) {
  if (!a) {
        return s;}
  else {
    return somme(a-1,a+s);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i,0));
  return 0;}
```

```
main:
        str     lr, [sp, #-4]!
        sub     sp, sp, #12
        add     r1, sp, #4
        ldr     r0, .L14
        bl      scanf
        ldr     r1, [sp, #4]
        cmp     r1, #0
        movne   r3, r1
        movne   r2, #0
        bne     .L9
        b       .L13
.L11:
        mov     r3, r0
.L9:
        subs    r0, r3, #1
        add     r2, r2, r3
        bne     .L11
.L8:
        ldr     r0, .L14+4
        bl      printf
        mov     r0, #0
        add     sp, sp, #12
        ldr     lr, [sp], #4
        bx      lr
.L13:
        mov     r2, r1
        b       .L8
```

Fonctions récursives, Exemple (exécution)

- Somme récursive non terminale
- Somme récursive terminale

Fonctions récursives, Exemple (exécution)

- Somme récursive non terminale

```
int somme(int a) {
  if (!a) {
        return 0;}
  else {
    return a+somme(a-1);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i));
  return 0;}
```

- Somme récursive terminale

```
int somme(int a,int s) {
  if (!a) {
        return s;}
  else {
    return somme(a-1,a+s);}}

int main () {
  int i;
  scanf("%d",&i);
  printf("1+2+...+%d=%d\n",i,somme(i,0));
  return 0;}
```

Fonctions récursives, Exemple (exécution)

- Version récursive non terminale : plante à 300 000 (segmentation fault)

Fonctions récursives, Exemple (exécution)

- Version récursive non terminale : plante à 300 000 (segmentation fault)
- Version récursive non terminale optimisée par le compilateur : plante à 600 000 (segmentation fault)

Fonctions récursives, Exemple (exécution)

- Version récursive non terminale : plante à 300 000 (segmentation fault)
- Version récursive non terminale optimisée par le compilateur : plante à 600 000 (segmentation fault)
- Version récursive terminale : plante à 300 000 (segmentation fault)

Fonctions récursives, Exemple (exécution)

- Version récursive non terminale : plante à 300 000 (segmentation fault)
- Version récursive non terminale optimisée par le compilateur : plante à 600 000 (segmentation fault)
- Version récursive terminale : plante à 300 000 (segmentation fault)
- Version récursive terminale optimisée par le compilateur : plante à 600 000 (segmentation fault)

Fonctions récursives, Exemple (exécution)

- Version récursive non terminale : plante à 300 000 (segmentation fault)
- Version récursive non terminale optimisée par le compilateur : plante à 600 000 (segmentation fault)
- Version récursive terminale : plante à 300 000 (segmentation fault)
- Version récursive terminale optimisée par le compilateur : plante à 600 000 (segmentation fault)
- Version + optimisée par le compilateur : 3 000 000 000 termes calculés en 1s

Fonctions (conclusions)

- Les fonctions utilisées ne sont pas nécessairement celles programmées

Fonctions (conclusions)

- Les fonctions utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les fonctions les plus claires

Fonctions (conclusions)

- Les fonctions utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les fonctions les plus claires
- Le compilateur optimise !

Fonctions (conclusions)

- Les fonctions utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les fonctions les plus claires
- Le compilateur optimise !
- Gain espéré : temps d'exécution des appels (branchement+gestion paramètres)

Fonctions (conclusions)

- Les fonctions utilisées ne sont pas nécessairement celles programmées
- Le programmeur peut conserver les fonctions les plus claires
- Le compilateur optimise !
- Gain espéré : temps d'exécution des appels (branchement+gestion paramètres)
- Gain espéré : place dans la pile (et cela peut fonctionner !)

Plan

Pipeline (rappel)

Hypothèses,

Pipeline (rappel)

Hypothèses,
La chaîne de traitement d'une instruction comporte 5 étapes :

Pipeline (rappel)

Hypothèses,
La chaîne de traitement d'une instruction comporte 5 étapes :

- IF : récupération de l'instruction

Pipeline (rappel)

Hypothèses,
La chaîne de traitement d'une instruction comporte 5 étapes :

- IF : récupération de l'instruction
- ID : récupération des opérandes

Pipeline (rappel)

Hypothèses,
La chaîne de traitement d'une instruction comporte 5 étapes :

- IF : récupération de l'instruction
- ID : récupération des opérandes
- EX : exécution de l'opération

Pipeline (rappel)

Hypothèses,
La chaîne de traitement d'une instruction comporte 5 étapes :

- IF : récupération de l'instruction
- ID : récupération des opérandes
- EX : exécution de l'opération
- MEM : écriture ou lecture mémoire

Pipeline (rappel)

Hypothèses,
La chaîne de traitement d'une instruction comporte 5 étapes :

- IF : récupération de l'instruction
- ID : récupération des opérandes
- EX : exécution de l'opération
- MEM : écriture ou lecture mémoire
- WB : écriture registre

Exécution de trois instructions

Exécution ”simple” de trois instructions :

(source wikipedia)

Exécution de trois instructions

Exécution ”simple” de trois instructions :

(source wikipedia)

- temps d’exécution : 15 Δ

Exécution de cinq instructions

Avec pipeline

Exécution de cinq instructions

Avec pipeline

Exécution pipelinée de cinq instructions, les unes à la suite des autres :

(source wikipedia)

Exécution de cinq instructions

Avec pipeline

Exécution pipelinée de cinq instructions, les unes à la suite des autres :

(source wikipedia)

- temps d'exécution : 9 Δ

Rupture de pipeline

Mais !

Rupture de pipeline

Mais !

les pipelines peuvent être rompus :

Rupture de pipeline

Mais !
les pipelines peuvent être rompus :

- Branchement

Rupture de pipeline

Mais !
les pipelines peuvent être rompus :

- Branchement
- Dépendance entre données

Rupture de pipeline

Mais !
les pipelines peuvent être rompus :

- Branchement
- Dépendance entre données
- ...

Pipeline et branchement

Ruptures de pipeline et branchement, cela concerne :

- Conditionnelles

Pipeline et branchement

Ruptures de pipeline et branchement, cela concerne :

- Conditionnelles
- Boucles

Pipeline et branchement

Ruptures de pipeline et branchement, cela concerne :

- Conditionnelles
- Boucles
- Appels de fonction

Pipeline et branchement

Ruptures de pipeline et branchement, cela concerne :

- Conditionnelles
- Boucles
- Appels de fonction
- Le compilateur peut s'en occuper

Pipeline et branchement

Ruptures de pipeline et branchement, cela concerne :

- Conditionnelles
- Boucles
- Appels de fonction
- Le compilateur peut s'en occuper
- Le programmeur peut aider

Exemple Syracuse : arithmétique, boucle, conditionnelle, fonction

- Syracuse (version originale)

Exemple Syracuse : arithmétique, boucle, conditionnelle, fonction

- Syracuse (version originale)

```
int syracuse(int a) {
        for(i=0;a>1;i++) {
          if (a&1) {a=3*a+1;}
          else {a=a/2;}}
        return i;}

int main () {
  scanf("%d",&i);
  for(m=0;i>0;i--) {
        n=syracuse(i);
    if (n>m) {m=n;im=i;}}
  printf("i=%d, m=%d\n",im,m);
  return 0;}
```

Exemple Syracuse : arithmétique, boucle, conditionnelle, fonction

- Syracuse (version originale)

```
int syracuse(int a) {
        for(i=0;a>1;i++) {
          if (a&1) {a=3*a+1;}
          else {a=a/2;}}
        return i;}

int main () {
  scanf("%d",&i);
  for(m=0;i>0;i--) {
        n=syracuse(i);
    if (n>m) {m=n;im=i;}}
  printf("i=%d, m=%d\n",im,m);
  return 0;}
```

- Syracuse (version un peu optimisée par le programmeur)

```
int syracuse(int a) {
        for(i=0;a>1;i++) {
          if (a&1) {a=(3*a+1)/2;}
          else {a=a/2;}}
        return i;}

int main () {
  scanf("%d",&i);
  for(m=0;i>0;i--) {
        n=syracuse(i);
    if (n>m) {m=n;im=i;}}
  printf("i=%d, m=%d\n",im,m);
  return 0;}
```

Pipeline et branchement, Exemple Syracuse

- Version originale : 1 500 000 suites en 1s

Pipeline et branchement, Exemple Syracuse

- Version originale : 1 500 000 suites en 1s
- Version optimisée par le programmeur : les mêmes suites en 0.7s

Pipeline et branchement, Exemple Syracuse

- Version originale : 1 500 000 suites en 1s
- Version optimisée par le programmeur : les mêmes suites en 0.7s
- Version originale optimisée par le compilateur : les mêmes suites en 0.7s

Pipeline et branchement, Exemple Syracuse

- Version originale : 1 500 000 suites en 1s
- Version optimisée par le programmeur : les mêmes suites en 0.7s
- Version originale optimisée par le compilateur : les mêmes suites en 0.7s
- Version optimisée par le programmeur et par le compilateur : les mêmes suites en 0.5s

Pipeline et conflit de variables, Exemple FiboLike

- FiboLike linéaire

Pipeline et conflit de variables, Exemple FiboLike

- FiboLike linéaire

```
main() {int i,x,y;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

Pipeline et conflit de variables, Exemple FiboLike

- FiboLike linéaire
- 8 add., 28 accès mém.

```
main() {int i,x,y;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

Pipeline et conflit de variables, Exemple FiboLike

- FiboLike linéaire
- 8 add., 28 accès mém.

```
main() {int i,x,y;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

- FiboLike binaire

Pipeline et conflit de variables, Exemple FiboLike

- FiboLike linéaire
- 8 add., 28 accès mém.

```
main() {int i,x,y;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

- FiboLike binaire

```
main() {
  int i,x,y,xx,xxxx,
      xxxxxxxx,t,tt,ttt,tttt;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    xx=x+x;
    t=x+x;
    xxxx=xx+xx;
    tt=x+x;
    xxxxxxxx=xxxx+xxxx;
    ttt=x+x;
    y=y+xxxxxxxx;
    tttt=x+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

Pipeline et conflit de variables, Exemple FiboLike

- FiboLike linéaire
- 8 add., 28 accès mém.

```
main() {int i,x,y;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

- FiboLike binaire
- 8 add., 28 accès mém.

```
main() {
  int i,x,y,xx,xxxx,
      xxxxxxxx,t,tt,ttt,tttt;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    xx=x+x;
    t=x+x;
    xxxx=xx+xx;
    tt=x+x;
    xxxxxxxx=xxxx+xxxx;
    ttt=x+x;
    y=y+xxxxxxxx;
    tttt=x+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

Pipeline et conflit de variables, Exemple FiboLike

- FiboLike linéaire
- 8 add., 28 accès mém.

```
main() {int i,x,y;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    y=y+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

- FiboLike binaire
- 8 add., 28 accès mém.
- moins de conflits

```
main() {
  int i,x,y,xx,xxxx,
      xxxxxxxx,t,tt,ttt,tttt;
  scanf("%d",&i);
  for(x=0,y=1;i;i--) {
    xx=x+x;
    t=x+x;
    xxxx=xx+xx;
    tt=x+x;
    xxxxxxxx=xxxx+xxxx;
    ttt=x+x;
    y=y+xxxxxxxx;
    tttt=x+x;
    x=y;}
  printf("x=%d, y=%d \n",x,y);
  return 0;}
```

Pipeline et conflit de variables, Exemple FiboLike

- Version linéaire : 70 000 000 termes en 1s

Pipeline et conflit de variables, Exemple FiboLike

- Version linéaire : 70 000 000 termes en 1s
- Version binaire : 70 000 000 termes en 0.6s

Pipeline et conflit de variables, Exemple FiboLike

- Version linéaire : 70 000 000 termes en 1s
- Version binaire : 70 000 000 termes en 0.6s
- (et le programmeur peut enlever les opérations qui ne servent à rien)

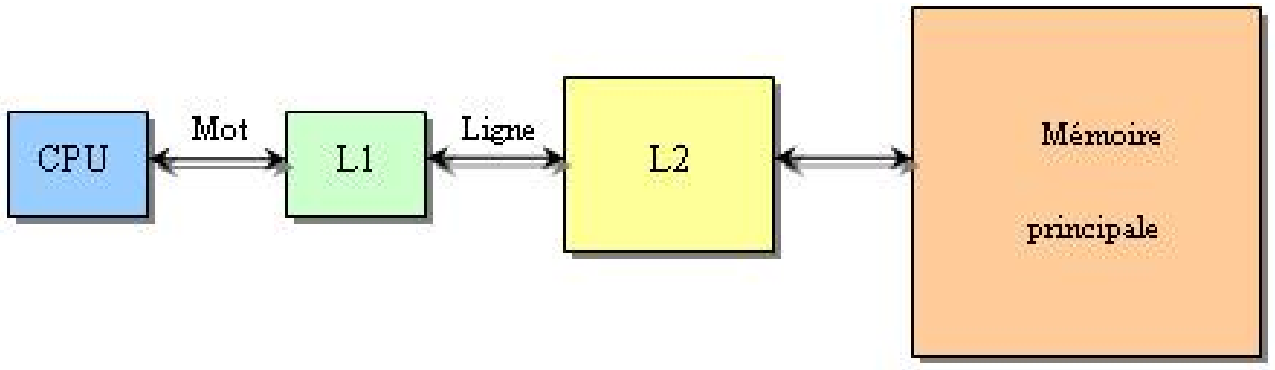
Organisation mémoire avec cache (rappels)

Organisation mémoire avec cache (rappels)

Structuration en caches de plusieurs niveaux

Organisation mémoire avec cache (rappels)

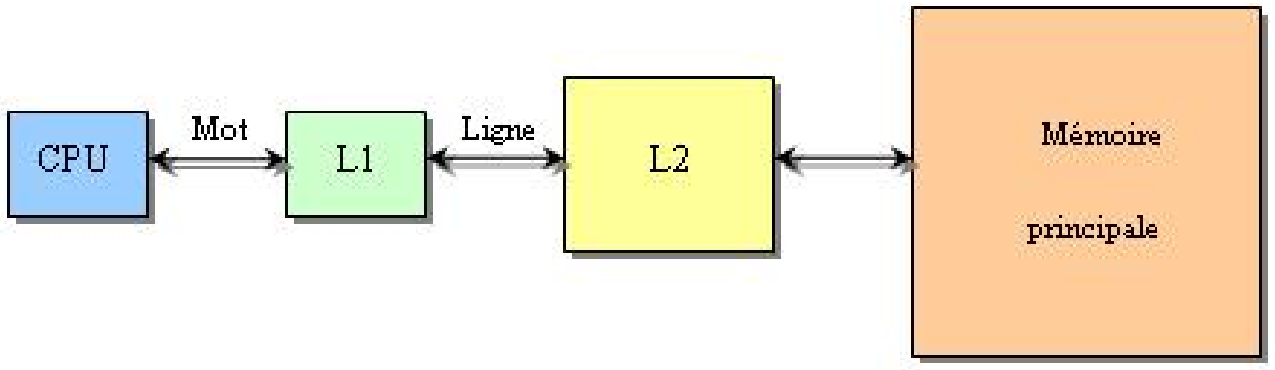
Structuration en caches de plusieurs niveaux



source wikipedia.

Organisation mémoire avec cache (rappels)

Structuration en caches de plusieurs niveaux



source wikipedia.

Pour le processeur, il n'y a qu'une mémoire : la Mémoire.

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
- Principe de localité temporelle :

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
- Principe de localité temporelle :
  - l'accès à une donnée X à un instant donné va probablement être suivi d'autres accès à cette même donnée

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
- Principe de localité temporelle :
  - l'accès à une donnée X à un instant donné va probablement être suivi d'autres accès à cette même donnée
- Le cache répond aux demandes du processeur :

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
- Principe de localité temporelle :
  - l'accès à une donnée X à un instant donné va probablement être suivi d'autres accès à cette même donnée
- Le cache répond aux demandes du processeur :
  - si la donnée est disponible dans le cache : ok

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
- Principe de localité temporelle :
  - l'accès à une donnée X à un instant donné va probablement être suivi d'autres accès à cette même donnée
- Le cache répond aux demandes du processeur :
  - si la donnée est disponible dans le cache : ok
  - si la donnée n'est pas disponible

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
- Principe de localité temporelle :
  - l'accès à une donnée X à un instant donné va probablement être suivi d'autres accès à cette même donnée
- Le cache répond aux demandes du processeur :
  - si la donnée est disponible dans le cache : ok
  - si la donnée n'est pas disponible
    - il faut aller la chercher en mémoire

Mémoire cache (rappels)

Principes de localité et utilisation :

- Principe de localité spatiale :
  - l'accès à une donnée X va probablement être suivi d'accès à d'autres données Y, Z proches de X.
- Principe de localité temporelle :
  - l'accès à une donnée X à un instant donné va probablement être suivi d'autres accès à cette même donnée
- Le cache répond aux demandes du processeur :
  - si la donnée est disponible dans le cache : ok
  - si la donnée n'est pas disponible
    - il faut aller la chercher en mémoire
    - et faire de la place dans le cache

Exemple Matrice lignes-colonnes vs colonnes-lignes

- Produit de Matrices version lignes-k-colonnes

Exemple Matrice lignes-colonnes vs colonnes-lignes

- Produit de Matrices version lignes-k-colonnes

```
int a[1000][1000], b[1000][1000],
    c[1000][1000], i, li, co, k, s ;
main() { //ikj
  scanf("%d",&i);
  for(;i;i--) {
    for(co=0;co<1000;co++) {
      for(li=0;li<1000;li++) {
        a[li][co]=1;b[li][co]=1;c[li][co]=0;}}
    for (li=0;li<1000;li++) {
      for (k=0;k<1000;k++) {
        s = a[li][k];
        for (co=0;co<1000;co++) {
          c[li][co] += s*b[k][co];}}}}
  printf("fin\n");
  return 0;}
```

Exemple Matrice lignes-colonnes vs colonnes-lignes

- Produit de Matrices version lignes-k-colonnes

```
int a[1000][1000], b[1000][1000],
    c[1000][1000], i, li, co, k, s ;
main() { //ikj
  scanf("%d",&i);
  for(;i;i--) {
    for(co=0;co<1000;co++) {
      for(li=0;li<1000;li++) {
        a[li][co]=1;b[li][co]=1;c[li][co]=0;}}
    for (li=0;li<1000;li++) {
      for (k=0;k<1000;k++) {
        s = a[li][k];
        for (co=0;co<1000;co++) {
          c[li][co] += s*b[k][co];}}}}
  printf("fin\n");
  return 0;}
```

- Produit de Matrices version colonnes-k-lignes

Exemple Matrice lignes-colonnes vs colonnes-lignes

- Produit de Matrices version lignes-k-colonnes

```
int a[1000][1000], b[1000][1000],
    c[1000][1000], i, li, co, k, s ;
main() { //ikj
  scanf("%d",&i);
  for(;i;i--) {
    for(co=0;co<1000;co++) {
      for(li=0;li<1000;li++) {
        a[li][co]=1;b[li][co]=1;c[li][co]=0;}}
    for (li=0;li<1000;li++) {
      for (k=0;k<1000;k++) {
        s = a[li][k];
        for (co=0;co<1000;co++) {
          c[li][co] += s*b[k][co];}}}}
  printf("fin\n");
  return 0;}
```

- Produit de Matrices version colonnes-k-lignes

```
int a[1000][1000], b[1000][1000],
    c[1000][1000], i, li, co, k, s ;
main() {  //jki
  scanf("%d",&i);
  for(;i;i--) {
    for(co=0;co<1000;co++) {
      for(li=0;li<1000;li++) {
        a[li][co]=1; b[li][co]=1;c[li][co]=0;}}
    for (co=0;co<1000;co++) {
      for (k=0;k<1000;k++) {
        s = b[k][co];
        for (li=0;li<1000;li++) {
          c[li][co] += a[li][k]*s;}}}}
  printf("fin\n");
  return 0;}
```

Exemple Matrice, exécution

A l'exécution

- Version lignes-k-colonnes : 1000*1000 en 2s9

Exemple Matrice, exécution

A l'exécution

- Version lignes-k-colonnes : 1000*1000 en 2s9
- Version colonnes-k-lignes : 1000*1000 en 9s2

Exemple Matrice, exécution

A l'exécution

- Version lignes-k-colonnes : 1000*1000 en 2s9
- Version colonnes-k-lignes : 1000*1000 en 9s2
- Remarque : le gain n'apparait que pour les grosses matrices

Plan

Multi- ?

Loi de Moore ...

Multi- ?

Loi de Moore ...
quand les circuits peuvent être plus gros, faire plusieurs UAL ?
plusieurs cœurs ? les deux ?

Multi-UAL : Exemple de calculs de Fibonacci et de Sommes

- Fibonacci puis les Sommes

Multi-UAL : Exemple de calculs de Fibonacci et de Sommes

- Fibonacci puis les Sommes

```
main() {
int i,init, x=0,y=1,z,s0=0,s1=1,
  s2=2,s3=3,s4=4,s5=5,s6=6;
  scanf("%d",&i);
  init=i;
  for(;i;i--) {
    z=x+y;x=y;y=z;}
  i=init;
  for(;i;i--) {
    s0=s0+i; s1=s1+i; s2=s2+i;
    s3=s3+i; s4=s4+i; s5=s5+i;
    s6=s6+i;}
  printf("x=%d, y=%d, s=%d \n",x,y,s0);
  return 0;}
```

Multi-UAL : Exemple de calculs de Fibonacci et de Sommes

- Fibonacci puis les Sommes

```
main() {
int i,init, x=0,y=1,z,s0=0,s1=1,
  s2=2,s3=3,s4=4,s5=5,s6=6;
  scanf("%d",&i);
  init=i;
  for(;i;i--) {
    z=x+y;x=y;y=z;}
  i=init;
  for(;i;i--) {
    s0=s0+i; s1=s1+i; s2=s2+i;
    s3=s3+i; s4=s4+i; s5=s5+i;
    s6=s6+i;}
  printf("x=%d, y=%d, s=%d \n",x,y,s0);
  return 0;}
```

- Fibonacci en même temps que des sommes

Multi-UAL : Exemple de calculs de Fibonacci et de Sommes

- Fibonacci puis les Sommes

```
main() {
int i,init, x=0,y=1,z,s0=0,s1=1,
  s2=2,s3=3,s4=4,s5=5,s6=6;
  scanf("%d",&i);
  init=i;
  for(;i;i--) {
    z=x+y;x=y;y=z;}
  i=init;
  for(;i;i--) {
    s0=s0+i; s1=s1+i; s2=s2+i;
    s3=s3+i; s4=s4+i; s5=s5+i;
    s6=s6+i;}
  printf("x=%d, y=%d, s=%d \n",x,y,s0);
  return 0;}
```

- Fibonacci en même temps que des sommes

```
main() {
int i,init, x=0,y=1,z,s0=0,s1=1,
  s2=2,s3=3,s4=4,s5=5,s6=6;
  scanf("%d",&i);
  init=i;
  for(;i;i--) {
    z=x+y; x=y; y=z;
    s0=s0+i; s1=s1+i; s2=s2+i;}
  i=init;
  for(;i;i--) {
    s3=s3+i; s4=s4+i; s5=s5+i;
    s6=s6+i;          }
  printf("x=%d, y=%d, s=%d \n",x,y,s0);
  return 0;}
```

Exemple de calculs de Fibonacci et de Sommes (Multi-UAL)

- Fibonacci puis les Sommes : 125 000 000 termes en 1s

Exemple de calculs de Fibonacci et de Sommes (Multi-UAL)

- Fibonacci puis les Sommes : 125 000 000 termes en 1s
- Fibonacci en même temps que des sommes : 125 000 000 termes en 0.9s

Exemple de calculs de Fibonacci et de Sommes (Multi-UAL)

- Fibonacci puis les Sommes : 125 000 000 termes en 1s
- Fibonacci en même temps que des sommes : 125 000 000 termes en 0.9s
- Conclusion : ici, un gain faible ...

Multi-Cœurs

(rappel)

- Fibonacci puis les Sommes : 125 000 000 termes en 1s
- Fibonacci en même temps que des sommes : 125 000 000 termes en 0.9s

Multi-Cœurs

(rappel)

- Fibonacci puis les Sommes : 125 000 000 termes en 1s
- Fibonacci en même temps que des sommes : 125 000 000 termes en 0.9s
- Fibonacci sur un cœur, Sommes sur un autre cœur : 125 000 000 termes en 0.5s

Multi-Cœurs

(rappel)

- Fibonacci puis les Sommes : 125 000 000 termes en 1s
- Fibonacci en même temps que des sommes : 125 000 000 termes en 0.9s
- Fibonacci sur un cœur, Sommes sur un autre cœur : 125 000 000 termes en 0.5s
- Conclusion : ici un gain significatif ...

Plan

Optimisations (conclusions)

- Le compilateur peut optimiser

Optimisations (conclusions)

- Le compilateur peut optimiser
- Le programmeur peut aussi optimiser !

Optimisations (conclusions)

- Le compilateur peut optimiser
- Le programmeur peut aussi optimiser !
  - il faut travailler l'algorithmique et + tard le parallélisme

Optimisations (conclusions)

- Le compilateur peut optimiser
- Le programmeur peut aussi optimiser !
  - il faut travailler l'algorithmique et + tard le parallélisme
- Comprendre l'exécution en regardant le code machine

Optimisations (conclusions)

- Le compilateur peut optimiser
- Le programmeur peut aussi optimiser !
  - il faut travailler l'algorithmique et + tard le parallélisme
- Comprendre l'exécution en regardant le code machine
- Voire, contrôler le code machine ...

Optimisations (conclusions)

- Le compilateur peut optimiser
- Le programmeur peut aussi optimiser !
  - il faut travailler l'algorithmique et + tard le parallélisme
- Comprendre l'exécution en regardant le code machine
- Voire, contrôler le code machine ...
- Utiliser des profiler pour déterminer où se trouvent les coûts

Optimisations (conclusions)

- Le compilateur peut optimiser
- Le programmeur peut aussi optimiser !
  - il faut travailler l'algorithmique et + tard le parallélisme
- Comprendre l'exécution en regardant le code machine
- Voire, contrôler le code machine ...
- Utiliser des profiler pour déterminer où se trouvent les coûts
- Mais surtout, avant d'optimiser : programmer juste !