Architecture des Ordinateurs Avancée (L3)

Cours 6 - Mieux exploiter le parallélisme : la vectorisation

Carine Pivoteau [1]

1. En collaboration avec X. Goaoc

Retour sur le TP 5 : prédiction de branchement

- Exo 2 - le seuil :

```
if(TAB[i] < threshold){
  a += TAB[i];
}
```

- Exo 3 - imbrications de probabilités :

```
if (v>24)
  if (v<51)
    j++;
```

vs.

```
if (v<51)
  if (v>24)
    j++;
```

Retour sur le TP 5 - exo 4 : ré-ordonnancement

```c
#include <stdio.h>
int main() {
  int i;
  int j=0,k=0;
  int l=0,res=0;

  for(i=1; i<10; i++){
    j+=i*i*i*i;
    k+=j*j*j*j;
    l+=j*j*k*k;
    res+=j/k;
    res+=l;
  }
  printf("%d\n",res);
  return 0;
}
```

```
.L2:
  mov esi, r8d
  imul  esi, r8d
  imul  esi, esi
  add ecx, esi
  mov esi, ecx
  imul  esi, ecx
  mov r9d, esi
  imul  r9d, ecx
  imul  r9d, ecx
  add edi, r9d
  imul  esi, edi
  imul  esi, edi
  add r10d, esi
  mov eax, ecx
  cdq
  idiv  edi
  add eax, r11d
  lea r11d, [rax+r10]
  add r8d, 1
  cmp r8d, 10
  jne .L2
```

```
.L2:
  mov esi, r8d
  imul  esi, r8d
  add r8d, 1
  imul  esi, esi
  add ecx, esi
  mov esi, ecx
  mov eax, ecx
  imul  esi, ecx
  cdq
  mov r9d, esi
  imul  r9d, ecx
  imul  r9d, ecx
  add edi, r9d
  imul  esi, edi
  idiv  edi
  imul  esi, edi
  add eax, r11d
  add r10d, esi
  cmp r8d, 10
  lea r11d, [rax+r10]
  jne .L2
```

Vectorisation

Exo : principe de la vectorisation

- Écrire une fonction qui augmente de 1 le code ascii de tous les caractères d'une chaîne de 800 caractères.
- Faire en sorte de pouvoir mesurer sont temps d'exécution.
- Lorsque l'on manipule un `char*`, on travaille octet par octet... et si on la manipulait comme un `long *` ?
- Écrire une deuxième fonction qui fait cela, mesurer son temps d'exécution et comparer à la première.

- Que ce serait-il passé si la longueur de la chaîne n'était pas un multiple de 8 ?
- Et si on avait ajouté `0xFF` au lieu de 1 à chaque caractère ?

(fichier **demo-C6-vect.c**)

Exo : principe de la vectorisation

- Écrire une fonction qui augmente de 1 le code ascii de tous les caractères d'une chaîne de 800 caractères.
- Faire en sorte de pouvoir mesurer sont temps d'exécution.
- Lorsque l'on manipule un `char*`, on travaille octet par octet... et si on la manipulait comme un `long *` ?
- Écrire une deuxième fonction qui fait cela, mesurer son temps d'exécution et comparer à la première.
- Que ce serait-il passé si la longueur de la chaîne n'était pas un multiple de 8 ?
- Et si on avait ajouté `0xFF` au lieu de 1 à chaque caractère ?

(fichier **demo-C6-vect.c**)

Exo : principe de la vectorisation

- Écrire une fonction qui augmente de 1 le code ascii de tous les caractères d'une chaîne de 800 caractères.
- Faire en sorte de pouvoir mesurer sont temps d'exécution.
- Lorsque l'on manipule un `char*`, on travaille octet par octet... et si on la manipulait comme un `long *` ?
- Écrire une deuxième fonction qui fait cela, mesurer son temps d'exécution et comparer à la première.

- Que ce serait-il passé si la longueur de la chaîne n'était pas un multiple de 8 ?
- Et si on avait ajouté `0xFF` au lieu de 1 à chaque caractère ?

(fichier **demo-C6-vect.c**)

Exo : principe de la vectorisation

- Écrire une fonction qui augmente de 1 le code ascii de tous les caractères d'une chaîne de 800 caractères.
- Faire en sorte de pouvoir mesurer sont temps d'exécution.
- Lorsque l'on manipule un `char*`, on travaille octet par octet... et si on la manipulait comme un `long *` ?
- Écrire une deuxième fonction qui fait cela, mesurer son temps d'exécution et comparer à la première.

- Que ce serait-il passé si la longueur de la chaîne n'était pas un multiple de 8 ?
- Et si on avait ajouté `0xFF` au lieu de 1 à chaque caractère ?

(fichier **demo-C6-vect.c**)

Exo : principe de la vectorisation

- Écrire une fonction qui augmente de 1 le code ascii de tous les caractères d'une chaîne de 800 caractères.
- Faire en sorte de pouvoir mesurer sont temps d'exécution.
- Lorsque l'on manipule un `char*`, on travaille octet par octet... et si on la manipulait comme un `long *` ?
- Écrire une deuxième fonction qui fait cela, mesurer son temps d'exécution et comparer à la première.

- Que ce serait-il passé si la longueur de la chaîne n'était pas un multiple de 8 ?
- Et si on avait ajouté `0xFF` au lieu de 1 à chaque caractère ?

(fichier **demo-C6-vect.c**)

La vectorisation, une autre forme de parallélisme

- Les processeurs ont d'abord été conçus pour traiter de façon séquentielle des instructions ne manipulant qu'une seule donnée. On parle de processeurs SISD (*single instruction single data*).
- Le pipeline classique permet d'utiliser plus efficacement chaque partie du processeur... mais chaque partie continue à traiter les instructions séquentiellement, donc ça reste SISD.
- Avec la vectorisation, **chaque instruction opère sur plusieurs données à la fois** : on travaille sur des **vecteurs**. On parle de processeurs SIMD (*single instruction multiple data*)
- Forme de parallélisme très présente (entre autres) dans les processeurs graphiques (*GPU*).

Processeur vectoriel

- Registres vectoriels : ils peuvent se fractionner en blocs, de sorte que les calculs ne débordent pas d'un bloc à l'autre. Les registres vectoriels ont été créés initialement en 128 bits (`xmm`, 1999), puis ont été étendus à 256 (`ymm`, 2011) puis à 512 bits (`zmm`, 2013). Le plus souvent, ils peuvent stocker indifféremment des entiers et des flottants. Leur contenu s'adapte suivant leur taille : un registre de 128 bits peut stocker 8 entiers/flottants de 16 bits ou 4 entiers/flottants de 32 bits, ...
- On utilise une unité de calcul vectorielle spécifique, souvent elle-même dotée d'un pipeline...
- ... et un jeu d'instructions dédiées : mélange d'instructions scalaires et vectorielles (fractionnées en blocs).
- Cela peut nécessiter une adaptation du modèle de cache.

Jeu d'instructions pour les registres vectoriels

- instructions logiques bit à bit classiques
- décalages/ rotation logiques et arithmétiques
- instructions vectorielles arithmétiques pour les entiers et les flottants (dont certaines un peu plus sophistiquées : max/min, moyenne, racine carrée...)
- déplacements
- opérations avec masques
- comparaison vectorielles mais pas de branchements
- et aussi : *shuffle*, *interleave*, ...

Exo : vectoriser “à la main”

On souhaite vectoriser le code suivant (pour des vecteurs de 4 entiers) :

```
int main() {
  int i, n = 400, count = 0;
  // remplissage du tableau

  for(i = 0; i < n; i++){
    count += tab[i];
  }
}
```

- De quelles instructions (vectorielles ou non) avons-nous besoin ?
- Écrire l’algorithme vectorisé qui fait la somme des cases.

Exo : vectoriser “à la main”

- La somme des cases du tableau (fichier **demo-C6-facile-0.c**) :

```c
int main() {
  int i, n = 400;
  int count = 0;

  // remplissage du tableau

  for(i = 0; i < n; i++){
     count += tab[i];
  }
}
```

```
  r0 <- 0
  x1 <- 0,0,0,0
.L
  x3 <- tab[r0:r0+4]
  x1 <- x1 + x3
  r0 <- r0 + 4
  cmp r0 400
  jne .L
  r1 <- x1[0]+x1[1]+x1[2]+x1[3]
```

- Comparer au code produit par gcc -O2

Exo : vectoriser “à la main”

- La somme des cases du tableau (fichier **demo-C6-facile-0.c**) :

```
int main() {
  int i, n = 400;
  int count = 0;

  // remplissage du tableau

  for(i = 0; i < n; i++){
     count += tab[i];
  }
}
```

```
  r0 <- 0
  x1 <- 0,0,0,0
.L
  x3 <- tab[r0:r0+4]
  x1 <- x1 + x3
  r0 <- r0 + 4
  cmp r0 400
  jne .L
  r1 <- x1[0]+x1[1]+x1[2]+x1[3]
```

- Comparer au code produit par gcc -O2

```
  ...
  pxor  xmm0, xmm0
  lea rdx, [rax+1600]
  .p2align 4,,10
  .p2align 3
.L5:
  movdqu  xmm1, XMMWORD PTR [rax]
  add rax, 16
  cmp rax, rdx
  paddd xmm0, xmm1
  jne .L5
  movdqa  xmm1, xmm0
  mov esi, OFFSET FLAT:.LC0
  mov edi, 1
  xor eax, eax
  psrldq  xmm1, 8
  paddd xmm0, xmm1
  movdqa  xmm1, xmm0
  psrldq  xmm1, 4
  paddd xmm0, xmm1
  movd  DWORD PTR [rsp+12], xmm0
  ...
```

```
  ...
  pxor  xmm0, xmm0                  # x1 <- 0,0,0,0
  lea rdx, [rax+1600]
  .p2align 4,,10
  .p2align 3
.L5:
  movdqu  xmm1, XMMWORD PTR [rax] # x3 <- tab[r0:r0+4]
  add rax, 16                       # r0 <- r0 + 4
  cmp rax, rdx
  paddd xmm0, xmm1                  # x1 <- x1 + x3
  jne .L5
  movdqa  xmm1, xmm0                # on met xmm0 dans xmm1
  mov esi, OFFSET FLAT:.LC0
  mov edi, 1
  xor eax, eax
  psrldq  xmm1, 8                   # décal. droite de 8 octets : xmm1[?,?,0,1]
  paddd xmm0, xmm1                  # xmm0 = xmm0[?,?,2+0,1+3]
  movdqa  xmm1, xmm0                # on remet dans xmm1
  psrldq  xmm1, 4                   # décal. droite de 4 octets : xmm0[?,?,?,2+0]
  paddd xmm0, xmm1                  # xmm0 = xmm0[?,?,?,2+0+1+3]
  movd  DWORD PTR [rsp+12], xmm0  # on récupère la dernière portion
  ...
```

Exo : vectoriser “à la main”

```c
int main() {
  int i, n = 400;
  int count = 0;

  // remplissage du tableau

  for(i = 0; i < n; i++){
      if (tab[i] == 42)
         count++;
  }
}
```

(fichier **demo-C6-facile.c**)

```
  r0 <- 0
  x1 <- 0,0,0,0
  x2 <- 42,42,42,42
.L
  x3 <- tab[r0:r0+4]
  // 1 dans les blocs où le test est vrai
  x3 <- x3 == x2
  x1 <- x1 + x3
  r0 <- r0 + 4
  cmp r0 400
  jne .L
  r1 <- x1[0]+x1[1]+x1[2]+x1[3]
```

- Comparer au code produit par gcc -O2

Exo : vectoriser “à la main”

```c
int main() {
  int i, n = 400;
  int count = 0;

  // remplissage du tableau

  for(i = 0; i < n; i++){
      if (tab[i] == 42)
         count++;
  }
}
```

(fichier **demo-C6-facile.c**)

```
  r0 <- 0
  x1 <- 0,0,0,0
  x2 <- 42,42,42,42
.L
  x3 <- tab[r0:r0+4]
  // 1 dans les blocs où le test est vrai
  x3 <- x3 == x2
  x1 <- x1 + x3
  r0 <- r0 + 4
  cmp r0 400
  jne .L
  r1 <- x1[0]+x1[1]+x1[2]+x1[3]
```

- Comparer au code produit par gcc -O2

```
LCPI0_0:
  .long 42                              ## 0x2a
  .long 42                              ## 0x2a
  .long 42                              ## 0x2a
  .long 42                              ## 0x2a
_main:
## %bb.0:
  push  rbp
  mov rbp, rsp
  push  r14
  push  rbx
  mov edi, 400
  mov esi, 4
  call  _calloc
  mov r14, qword ptr [rip + _A@GOTPCREL]
  mov qword ptr [r14], rax
  test  rax, rax
  je  LBB0_6
## %bb.1:
  xor ebx, ebx
LBB0_2:
  call  _rand
  mov rcx, qword ptr [r14]
  mov dword ptr [rcx + 4*rbx], eax
  inc rbx
  cmp rbx, 400
  jne LBB0_2
## %bb.3:
  pxor  xmm0, xmm0
  mov   eax, 12
  movdqa  xmm2, xmmword ptr [rip + LCPI0_0]
  pxor  xmm1, xmm1
```

```
LCPI0_0:
  .long 42                              ## 0x2a
  .long 42                              ## 0x2a
  .long 42                              ## 0x2a
  .long 42                              ## 0x2a
_main:
## %bb.0:
  push  rbp
  mov rbp, rsp
  push  r14
  push  rbx
  mov edi, 400
  mov esi, 4
  call  _calloc
  mov r14, qword ptr [rip + _A@GOTPCREL]
  mov qword ptr [r14], rax
  test  rax, rax
  je  LBB0_6
## %bb.1:
  xor ebx, ebx
LBB0_2:
  call  _rand
  mov rcx, qword ptr [r14]
  mov dword ptr [rcx + 4*rbx], eax
  inc rbx
  cmp rbx, 400
  jne LBB0_2
## %bb.3:
  pxor  xmm0, xmm0                                  # xmm0 = [0, 0, 0, 0]
  mov   eax, 12                                     # ??
  movdqa  xmm2, xmmword ptr [rip + LCPI0_0]         # xmm2 = [42,42,42,42]
  pxor  xmm1, xmm1
```

```
LBB0_4:
  movdqu  xmm3, xmmword ptr [rcx + 4*rax - 48]
  movdqu  xmm4, xmmword ptr [rcx + 4*rax - 32]
  movdqu  xmm5, xmmword ptr [rcx + 4*rax - 16]
  movdqu  xmm6, xmmword ptr [rcx + 4*rax]

  pcmpeqd xmm3, xmm2
  psubd   xmm0, xmm3
  pcmpeqd xmm4, xmm2
  psubd   xmm1, xmm4
  pcmpeqd xmm5, xmm2
  psubd   xmm0, xmm5
  pcmpeqd xmm6, xmm2
  psubd   xmm1, xmm6
  add rax, 16
  cmp rax, 412
  jne LBB0_4
## %bb.5:
  paddd   xmm1, xmm0
  pshufd  xmm0, xmm1, 78

  paddd   xmm0, xmm1
  phaddd  xmm0, xmm0
  movd  esi, xmm0
  lea rdi, [rip + L_.str]
  xor eax, eax
  call  _printf
  xor eax, eax
  pop rbx
  pop r14
  pop rbp
  ret
```

```
LBB0_4:
  movdqu  xmm3, xmmword ptr [rcx + 4*rax - 48]
  movdqu  xmm4, xmmword ptr [rcx + 4*rax - 32]
  movdqu  xmm5, xmmword ptr [rcx + 4*rax - 16]
  movdqu  xmm6, xmmword ptr [rcx + 4*rax]

  pcmpeqd xmm3, xmm2 # quand le test est vrai, pcmpeqd remplit avec des 11111...
  psubd   xmm0, xmm3 # ... ce qui vaut -1; donc on soustrait pour incrémenter :)
  pcmpeqd xmm4, xmm2
  psubd   xmm1, xmm4 # idem avec xmm4 et résultat dans xmm1
  pcmpeqd xmm5, xmm2
  psubd   xmm0, xmm5 # idem avec xmm5 et résultat dans xmm0
  pcmpeqd xmm6, xmm2
  psubd   xmm1, xmm6 # idem avec xmm6 et résultat dans xmm1
  add rax, 16
  cmp rax, 412
  jne LBB0_4
## %bb.5:
  paddd   xmm1, xmm0      # les 4 sommes sont dans xmm1
  pshufd  xmm0, xmm1, 78 # xmm0 = xmm1[2,3,0,1]

  paddd   xmm0, xmm1   # xmm0=xmm1[2,3,0,1]+xmm1[0,1,2,3]=xmm1[0+2,1+3,2+0,3+1]
  phaddd  xmm0, xmm0   # xmm0=xmm1[?,?,0+2+1+3,2+0+3+1]
  movd  esi, xmm0      # met le 4e entier de xmm0 dans esi
  lea rdi, [rip + L_.str]
  xor eax, eax
  call  _printf
  xor eax, eax
  pop rbx
  pop r14
  pop rbp
  ret
```

Exo : vectoriser “à la main” - fin

```c
int main() {
  int i, n = 400;
  int count = 0;

  // remplissage du tableau

  for(i = 0; i < n; i++){
      if (tab[i] == 42)
         count++;
  }
}
```

(fichier **demo-C6-facile.c**)

```
  r0 <- 0
  x1 <- 0,0,0,0
  x2 <- 42,42,42,42
.L
  x3 <- tab[r0:r0+4]
  // 1 dans les blocs où le test est vrai
  x3 <- x3 == x2
  x1 <- x1 + x3
  r0 <- r0 + 4
  cmp r0 400
  jne .L
  r1 <- x1[0]+x1[1]+x1[2]+x1[3]
```

- Comparer au code produit par gcc -O2
- Et si on remplace 400 par 403 ?

Difficultés rencontrées lors de la vectorisation

- Nombre d'itérations imprévisible à l'entrée d'une boucle (ça complique la gestion des itérations à effectuer en scalaire).
- Flot d'instructions non séquentiel (branchements, points d'entrée ou de sortie multiples, . . .).
- Accès en mémoire à des adresses non contiguës (par exemple accéder à `tab[index[i]]`) : problèmes d'efficacité de lecture/écriture en mémoire.
- **Dépendance de données** : sur une même variable, une lecture/écriture suivant une écriture posera problème, tandis qu'une lecture suivant une lecture ne posera jamais de problème.
- ...

Exemple

Comparer le code produit par gcc pour ces paires de fonctions.

```
void f1(char* t){
  while ((*t) != '\0'){
    (*t) += 1;
    ++t;
  }
}
```

```
void f2(char* t, int n){
  for (int i = 0; i < n; ++i){
    (*t) += 1;
    ++t;
  }
}
```

```
void f3(char* t, int n){
  for (int i = 0; i < n-1; ++i){
    (*t) += (*(t+1));
    ++t;
  }
}
```

```
void f4(char* t, int n){
  for (int i = 1; i < n; ++i){
    (*t) += (*(t-1));
    ++t;
  }
}
```

(fichier **demo-C5-boucles.c**)

Parallélisme : conclusion

- Parallélisme d'instructions grâce au pipeline (sur un seul cœur).
  - Efficace : améliore le taux d'utilisation d'un processeur.
  - Complexe à mettre en place, notamment à cause de la dépendance de données et des branchements.
- Parallélisme au niveau des données grâce à la vectorisation. Très efficace, mais avec des limites aussi (parfois difficile à exploiter).
- Autres formes de parallélisme non abordées dans ce cours : multi-cœurs, multi-threading, ...

C'est presque fini...

Conclusion

- L'idée que l'exécution d'un programme est purement séquentielle et que les instructions prennent toutes le même temps pour s'exécuter est très inexacte.
- Remarque : c'est la vue classique utilisée en complexité ; elle reste parfaitement valable mais ne donne qu'une appréciation "gros grain" du temps d'exécution.
- **Savoir quels sont les mécanismes haut niveau mis en œuvre dans le processeur, pour comprendre ce qui se passe au moment de l'exécution et pouvoir éventuellement l'exploiter.**
- Notions abordées en lien avec beaucoup d'autres cours :
  - Architecture de base, système, algo, ...
  - Concurrence, Java (Machine virtuelles), programmation distribuée, programmation GPU, ...

Devenir un hacker en 6 leçons... ?

Spectre et Meltdown (explication simplifiée)

- Les deux attaques utilisent une forme d'exécution anticipée :
  - l'exécution *Out-of-Order* (Meltdown) ;
  - l'exécution spéculative et la prédiction de branchement (Spectre).
- On écrit un programme qui essaie d'accéder à **une donnée x se trouvant à une adresse non autorisée**.
- Dans les deux cas, les vérifications de droits d'accès peuvent être faites **après** que x ait été chargée depuis la mémoire.
- Cette donnée x peut être utilisée comme une adresse pour charger une donnée de la mémoire ($mem[x]$), qui se retrouvera en cache.
- L'accès à x n'étant pas autorisé, le processeur va s'assurer qu'on ne voie plus la valeur de x... mais va laisser $mem[x]$ en cache.
- En mesurant les temps d'accès, il est possible de savoir quelles zones de cache sont occupées ou non et grâce au système d'adressage direct, on pourra alors deviner la valeur de x.

https://meltdownattack.com/

FIN