Architecture des ordinateurs
L3 Informatique 2020-2021 Binôme :
TP 6 - vectorisation et autres optimisations

- Chaque binôme rendra une feuille d'énoncé complétée et la déposera sur *e-learning* avec les programmes écrits pour répondre aux questions.
- Toutes les mesures de temps doivent être faites au moyen de la fonction `print_timing` écrite au TP 2.
- Ce TP a pour objectif de comprendre quand et comment on peut utiliser la véctorisation.

Exercice 1. (★ Vectorisation, cas d'école) On souhaite vectoriser le code suivant :

```
1 int sum0 (int n){
2     int i,s = 0;
3     for(i = 0; i < n; i++)
4       s += i;
5     return s;
6 }
```

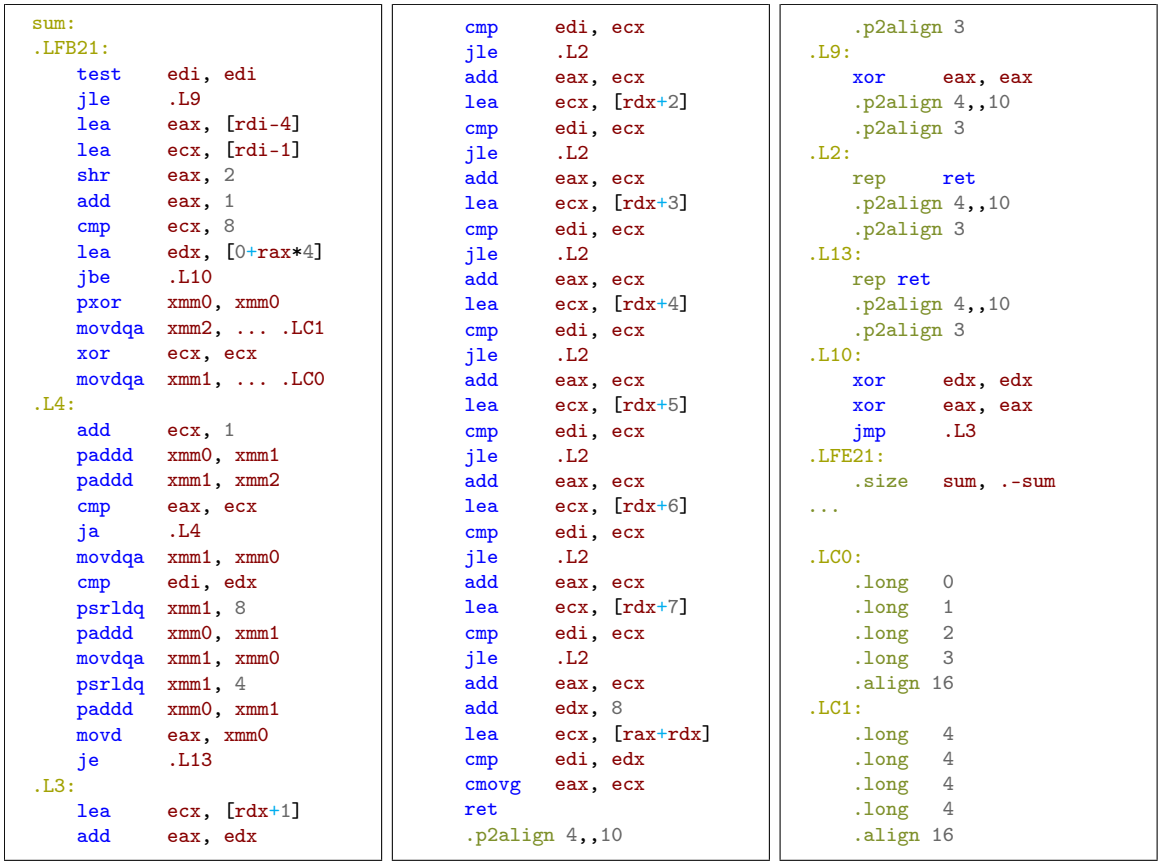
```
1 int sum1 (int n){
2     int i,s = 0;
3     for(i = 0; i < n; i++)
4       if (i < 10) s += i;
5     return s;
6 }
```

a. Pour vectoriser ce code on utilise des vecteurs pouvant contenir 4 entiers et on s'autorise les opérations listées dans l'annexe donnée page 6

Supposons que `n` est un multiple de 4. Écrire l'algorithme vectorisé correspondant à `sum0`.

Et si `n` n'est pas un multiple de 4, comme fait-on ?

b. Le code assembleur correspondant, produit en compilant avec `gcc -O3` est donné par la Figure 1. Commenter le code pour expliquer comment le compilateur `gcc` vectorise cette somme et comparer à ce que vous avez proposé à la question précédente (en essayant de trouver les portions de code correspondant à votre algorithme).



```
sum:
.LFB21:
    test    edi, edi
    jle     .L9
    lea     eax, [rdi-4]
    lea     ecx, [rdi-1]
    shr     eax, 2
    add     eax, 1
    cmp     ecx, 8
    lea     edx, [0+rax*4]
    jbe     .L10
    pxor    xmm0, xmm0
    movdqa  xmm2, ... .LC1
    xor     ecx, ecx
    movdqa  xmm1, ... .LC0
.L4:
    add     ecx, 1
    paddd   xmm0, xmm1
    paddd   xmm1, xmm2
    cmp     eax, ecx
    ja      .L4
    movdqa  xmm1, xmm0
    cmp     edi, edx
    psrldq  xmm1, 8
    paddd   xmm0, xmm1
    movdqa  xmm1, xmm0
    psrldq  xmm1, 4
    paddd   xmm0, xmm1
    movd    eax, xmm0
    je      .L13
.L3:
    lea     ecx, [rdx+1]
    add     eax, edx
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+2]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+3]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+4]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+5]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+6]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    lea     ecx, [rdx+7]
    cmp     edi, ecx
    jle     .L2
    add     eax, ecx
    add     edx, 8
    lea     ecx, [rax+rdx]
    cmp     edi, edx
    cmovg   eax, ecx
    ret
    .p2align 4,,10
    .p2align 3
.L9:
    xor     eax, eax
    .p2align 4,,10
    .p2align 3
.L2:
    rep     ret
    .p2align 4,,10
    .p2align 3
.L13:
    rep ret
    .p2align 4,,10
    .p2align 3
.L10:
    xor     edx, edx
    xor     eax, eax
    jmp     .L3
.LFE21:
    .size   sum, .-sum
...

.LC0:
    .long   0
    .long   1
    .long   2
    .long   3
    .align 16
.LC1:
    .long   4
    .long   4
    .long   4
    .long   4
    .align 16
```

FIGURE 1 – Code assembleur optimisé de l'exercice 1.

Note : un descriptif complet des instructions x86 (instructions vectorielles comprises) est donné là : `https://kernfunny.org/x86/`.

c. Supposons que `n` est un multiple de 4. Écrire l'algorithme vectorisé correspondant à `sum1`.

Exercice 2. (★ Vectorisation et ordre des instructions) On se demande si les boucles suivantes sont vectorisables par gcc. Pour chacune d'entre elles, proposer une réponse argumentée **avant** de vérifier en compilant le code. Utilisez `-O3`, prenez soin d'écrire les boucles dans des fonctions et de déclarer les pointeurs de tableaux en variables globales (`int *A`, etc.) et d'allouer la mémoire dans le main par `calloc`.

```
for(i = 0; i < n; i++){
  A[i]=A[i+1];
}

for(i = 0; i < n; i++){
  A[i]=B[i];
}

for(i = 0; i < n-1; i++){
  A[i]=B[i+1];
}

for(i = 1; i < n-4; i++){
  A[i]=B[i];
  A[i+1]=B[i+1];
}

for(i = 1; i < n-4; i+=2){
  A[i]=B[i];
  A[i+2]=B[i+2];
}

for(i = 1; i < n-4; i++){
  A[i]=B[i];
  A[i+4]=B[i+4];
}

for(i = 1; i < n-4; i++){
  A[i]=B[i];
  B[i]=C[i];
}

for(i = 1; i < n-4; i++){
  A[i]=B[i+1];
  B[i]=C[i];
}

for(i = 1; i < n-4; i++){
  A[i]=B[i-1];
  B[i]=C[i];
}

for(i = 4; i < n-4; i++){
  A[i]=B[i+4];
  B[i]=C[i];
}
```

Exercice 3. (★★ Peut-on favoriser la vectorisation par `gcc` ?) On cherche à comprendre comment donner un coup de pouce au compilateur pour l'aider à vectoriser le code C.

a. Pour cela, on repart de l'exemple de la somme calculée à l'exercice 1 et on le ré-écrit de la façon suivante :

```
1  int sum2 (int count){
2      int s = 0, i;
3      for(i = 0; i < count; i+=4){
4          s+=i;
5          s+=i+1;
6          s+=i+2;
7          s+=i+3;
8      }
9      return s;
10 }
```

Pensez-vous que cela va l'aider ? Vérifier en produisant le code assembleur avec `gcc -O3` et expliquer ce qui s'est passé.

b. On fait une deuxième tentative :

```
1  int sum3 (int count){
2      int s = 0, a, b, c, i;
3      for(i = 0; i < count; i+=4){
4        a=i+1;
5        b=i+2;
6        c=i+3;
7        s+=i+a+b+c;
8      }
9      return s;
10 }
```

Qu'en pensez-vous ? Produisez le code assembleur avec `gcc -O3` et expliquer ce qui s'est passé.

Exercice 4. (★★★ Vectoriser un calcul de minimum)

a. Examinez ce que fait l'instruction vectorisée `pcmpgtd` et proposez un code assembleur qui calcule le minimum composante par composante des registres `xmm0` et `xmm1`

b. Voici un programme qui calcule et affiche le minimum d'un tableau :

```
int count = 400; if((A=(int*)calloc(count,sizeof(int)))==NULL) exit(1);
int lo = A[0], v, i;

for(i=1; i<count; i++){
  v = A[i];
  if (v < lo)
    lo = v;
}
printf("%d\n", lo);
```

Comment peut-on vectoriser ce calcul ? Donner un algorithme pour des vecteurs de taille 4. Si vous ne trouvez pas (ou sinon, pour confirmer votre intuition), produisez le code assembleur avec `gcc -O3` et utilisez-le pour répondre à la question précédente.

Et si vous avez du mal à lire le code, essayer de suivre ce qui se passe sur un exemple.

c. Pensez-vous qu'il s'agît de code produit automatiquement par un programme ou que le calcul du minimum a été codé par un humain ?

Pseudo code vectorisé. On utilise des vecteurs pouvant contenir 4 entiers (qu'on appelle des *composantes*). Chaque composante est traitée comme un `int`. Il est inutile de déclarer les variables vecteurs. On autorise les opérations suivantes sur les vecteurs.

Pour le transfert :

- initialiser un vecteur (ex : `u = 1,1,1,1` et `v = 0,33,42,806`)
- copier un vecteur dans un autre (ex : `w = v`)
- charger un vecteur depuis 4 cases consécutives de la mémoire à une adresse donnée en précisant la taille des cases (ex : `w = (char) TAB[i:i+3]` indique que l'on charge 4 cases de la taille d'un `char` dans `w` depuis l'adresse `TAB[i]`)
- charger un vecteur dans 4 cases consécutives de la mémoire à une adresse donnée en précisant la taille des cases (ex : `TAB[0:3] = (char) w` indique que l'on place les 4 composantes de `w` dans 4 cases mémoire consécutives de la taille d'un `char` à l'adresse `TAB`)

Pour le calcul :

- additionner (ou soustraire, multiplier, diviser) deux vecteurs composante par composante (ex : `w = u + v` ; avec les valeurs précédentes, on obtient que `w` vaut désormais `1,34,43,807`)
- faire des opération logiques bit à bit sur les vecteurs (ex : `z = u AND v`, `z = u OR v` ou `z = NOT u`, ...)
- comparer deux vecteurs composante par composante (ex : `z = u < v` signifie que si la i^{eme} composante de `u` est strictement inférieure à la i^{eme} composante de `v`, alors la i^{eme} composante de `z` vaut `111...111` et sinon elle vaut `000...000` (en binaire). Avec les valeurs précédentes, on obtient que `z` vaut `0...0,1...1,1...1,1...1` (ici aussi en binaire).
- calculer le minimum de deux vecteurs composante par composante (ex : `z = min(u ,v)` signifie que si la i^{eme} composante de `u` est strictement inférieure à la i^{eme} composante de `v`, alors la i^{eme} composante de `z` vaut la i^{eme} composante de `u` et sinon elle vaut la i^{eme} composante de `v`. Par exemple, avec `u = 1,100,1,27` et `v = 0,33,42,806`, si on fait `z = min(u ,v)` on obtient que `z` vaut `0,33,1,27`.

Pour la conversion d'un vecteur en `int` :

- faire la somme des 4 composantes d'un vecteur (ex : `int i = sum_comp(w)` ; avec les valeurs précédentes, on obtient que `i` vaut `885`)