Architecture (avancée) des ordinateurs
L3 Informatique 2017-2018
Examen 1ère session – lundi 9 avril

Nom :
Prénom :
Numéro :

- Cet examen dure 2h. Les réponses sont à écrire directement sur le sujet. Le sujet comporte 8 pages et les deux dernières sont détachables. Il y a 5 exercices + 1 exercice bonus.
- Le barême est donné à titre indicatif.
- Le seul document autorisé est une feuille A4 recto-verso manuscrite et personnelle. Les systèmes électroniques (calculatrice, téléphone portable, etc.) sont interdits.

Exercice 1. (4 points) Cet exercice teste votre compréhension générale des thèmes abordés dans le cours.

1. Pourquoi certains nombres réels sont-ils représentables exactement en flottant double précision (norme IEEE-2008) et d'autres pas ?

2. Donner un exemple de nombre $\frac{1}{4} < x < \frac{1}{2}$ non représentable en flottant double précision.

3. Qu'est-ce qu'une ligne de cache ?

4. On considère un tableau 2D. Pourquoi est-ce qu'accéder à 10 éléments consécutifs en ligne prend moins de temps qu'accéder à 10 éléments consécutifs en colonne ?

5. Un pipeline à 5 niveaux permet en théorie de traiter 5 instructions en parallèle, et donc d'aller 5 fois plus vite. En pratique, le gain est bien inférieur. Comment cela s'explique-t-il ?

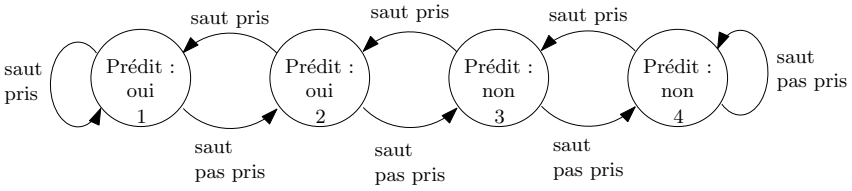
Exercice 2. (4 points) On s'intéresse dans cet exercice au comportement de prédicteurs de branchements lors de l'exécution de la fonction `compte_pairs` sur les tableaux `A` et `B` :

```
1  int compte_pairs(int n, int* tab){
2    int s;
3    for (int i = 0; i < n; ++i){
4        if (tab[i] % 2  == 0){
5            s += 1;
6        }
7    }
8    return s;
9  }
```

i	0	1	2	3	4	5	6	7	8	9
A[i]	0	1	2	3	4	5	6	7	8	9
B[i]	1	0	0	0	0	1	1	1	1	1

i	10	11	12	13	14	15	16	17	18	19
A[i]	10	11	12	13	14	15	16	17	18	19
B[i]	0	1	0	1	1	1	1	1	0	1

1. On suppose que le branchement correspondant à la ligne 5 est prédit par l'automate suivant (un saut est pris si la condition est évaluée à VRAI) :



   a. Supposons que le prédicteur est initialement dans l'état 1. Indiquer, **pour chaque tableau**, les indices pour lesquels le prédicteur de branchement fait une erreur de prédiction.

   b. Même question en supposant que le prédicteur est initialement dans **l'état 3**.

2. Dans le cas du tableau `A`, pourquoi le nombre d'erreurs de prédiction varie-t-il autant suivant l'état initial ?

3. Modifier le prédicteur de la question 1 (en gardant 4 états) pour limiter le nombre d'erreurs de prédictions dans le cas du tableau `A`, quel que soit l'état initial.

Exercice 3. (3 points) On considère la fonction suivante :

```
1  int f0(int n, int *TAB){
2    int a = 0, sum = 0;
3    for (int i = 0; i < n ; i++) {
4      a = TAB[i];
5      sum += TAB[a];
6    }
7    return sum;
8  }
```

On mesure (correctement) le temps d'exécution de `f0` pour deux tableaux `A` et `B` de taille n, tous deux remplis avec tous les entiers de 0 à $n-1$. Le tableau `A` contient ces entiers dans l'ordre croissant et `B` contient les mêmes entiers mélangés aléatoirement.

```
A: n=1000 (~4Ko): 0.0003930 msec        A: n=32000 (~128Ko): 0.0127410 msec
B: n=1000 (~4Ko): 0.0003880 msec        B: n=32000 (~128Ko): 0.0219070 msec

A: n=2000 (~8Ko): 0.0007840 msec        A: n=64000 (~256Ko): 0.0246500 msec
B: n=2000 (~8Ko): 0.0008760 msec        B: n=64000 (~256Ko): 0.0593110 msec

A: n=4000 (~16Ko): 0.0016580 msec       A: n=128000 (~512Ko): 0.0484480 msec
B: n=4000 (~16Ko): 0.0017330 msec       B: n=128000 (~512Ko): 0.1612830 msec

A: n=8000 (~32Ko): 0.0030260 msec       A: n=256000 (~1024Ko): 0.1000610 msec
B: n=8000 (~32Ko): 0.0032370 msec       B: n=256000 (~1024Ko): 0.3746910 msec

A: n=16000 (~64Ko): 0.0064230 msec      A: n=512000 (~2048Ko): 0.2019430 msec
B: n=16000 (~64Ko): 0.0080750 msec      B: n=512000 (~2048Ko): 1.0338629 msec
```

1. Analyser, en justifiant, les temps d'exécution observés.

2. Quelle est la principale difficulté si on veut vectoriser cette fonction efficacement ?

Exercice 4. (3 points) On considère la fonction suivante :

```
1  int f1(int n, int *TAB){
2    int a = 0, sum1 = 0, sum2 = 0;
3
4    int threshold = n/2;
5
6    for (int i = 0; i < n ; i++) {
7      a = TAB[i];
8      if(a < threshold)
9        sum1 += a;
10     if(a >= threshold)
11       sum2 += a;
12   }
13   return sum1*sum2;
14 }
```

On mesure (correctement) le temps d'exécution de `f1` pour deux tableaux `A` et `B` de taille n, tous deux remplis avec tous les entiers de 0 à $n-1$. Le tableau `A` contient ces entiers dans l'ordre croissant et `B` contient les mêmes entiers mélangés aléatoirement.

```
A: n=4000 (~16Ko): 0.0155600002 msec     A: n=64000 (~256Ko): 0.2121500075 msec
B: n=4000 (~16Ko): 0.0281100012 msec     B: n=64000 (~256Ko): 0.5542899966 msec

A: n=8000 (~32Ko): 0.0284699984 msec     A: n=128000 (~512Ko): 0.4246900082 msec
B: n=8000 (~32Ko): 0.0623800009 msec     B: n=128000 (~512Ko): 1.1369999647 msec

A: n=16000 (~64Ko): 0.0581500009 msec    A: n=256000 (~1024Ko): 0.8510099649 msec
B: n=16000 (~64Ko): 0.1341499984 msec    B: n=256000 (~1024Ko): 2.2027199268 msec

A: n=32000 (~128Ko): 0.1098299995 msec   A: n=512000 (~2048Ko): 1.7076700926 msec
B: n=32000 (~128Ko): 0.2821199894 msec   B: n=512000 (~2048Ko): 4.4484000206 msec
```

1. Analyser, en justifiant, les temps d'exécution observés.

2. Quelle est la principale difficulté si on veut vectoriser cette fonction efficacement ?

Exercice 5. (Problème, 7 points) On s'intéresse dans cet exercice à la conversion d'une image couleur en niveaux de gris selon la méthode décrite à la page 7.

1. On exécute la conversion sur un système doté de deux niveaux de mémoire : la RAM et une mémoire cache de 64 Ko constituée de 1024 lignes de 64 octets chacune. La pagination fonctionne en *direct mapping*, c'est à dire que le bloc contenant l'octet `x` est chargé en cache dans le bloc d'adresse `(x/64) % 1024`.

   a. On suppose que `LARGEUR` = 1600 et `HAUTEUR` = 1000. Combien de cache-miss est-ce que la conversion en niveau de gris déclenche (donner le calcul sans l'effectuer) ?

   Solution A

   Solution B

   b. On suppose que `LARGEUR` = 1024 et `HAUTEUR` = 1024. Combien de cache-miss est-ce que la conversion en niveau de gris déclenche (donner le calcul sans l'effectuer) ?

   Solution A

   Solution B

2. On modifie un seul paramètre de la mémoire cache : on suppose qu'elle est 8-associative. Le bloc contenant l'octet x est donc chargé en cache dans le sous-cache d'adresse `(x/64) % 128`, et à l'intérieur de chaque sous-cache la stratégie de remplacement suit la règle LRU (*least recently used*). On suppose que `LARGEUR` = 1024 et `HAUTEUR` = 1024. Combien de cache-miss est-ce que la conversion en niveau de gris déclenche (donner le calcul sans l'effectuer) ?

   Solution A

   Solution B

3. Pour les 2 solutions (A et B), proposez soit une version vectorisée en pseudo-code de la fonction de conversion en niveaux de gris, soit une explication de ce qui rend cela difficile. Le pseudo-code utilisera les conventions de la page 8 pour les instructions vectorisées.

   Solution A

   Solution B

Exercice 6 (Bonus : vectorisation bis). On souhaite vectoriser la fonction de l'exercice 4 :

```
1  int f1(int n, int *TAB){
2    int a = 0, sum1 = 0, sum2 = 0;
3
4    int threshold = n/2;
5
6    for (int i = 0; i < n ; i++) {
7      a = TAB[i];
8      if(a < threshold)
9        sum1 += a;
10     if(a >= threshold)
11       sum2 += a;
12   }
13   return sum1*sum2;
14 }
```

1. On suppose que `n` est un multiple de 4. Écrire l'algorithme vectorisé correspondant à `f1`. Le pseudo-code utilisera les conventions de la page 8 pour les instructions vectorisées.

2. Lorsque l'on mesure les temps d'exécution de la version vectorisée comme à l'exercice 4, les temps d'exécution sont sensiblement les mêmes pour les deux tableaux. Comment cela peut-il s'expliquer ?

Codage des images. Il est courant de décrire la couleur d'un pixel[1] par trois valeurs : une composante rouge, une composante verte et une composante bleue. La norme la plus utilisée actuellement, le *truecolor*, décrit chacune de ces trois composantes par un entier 8 bits (`char`). Ainsi, le triplet $(0,0,0)$ désigne le noir, le triplet $(255,255,255)$ désigne le blanc, et $(x,0,0)$ désigne un rouge qui est vif si x est proche de 255 et sombre si x est proche de 0.

Niveaux de gris. En *truecolor*, une couleur est un niveau de gris si les trois composantes sont égales. Pour convertir une image en niveau de gris, une méthode consiste à remplacer, pour chaque pixel, les trois composantes par leur moyenne : ainsi, on remplace le triplet $(50,70,180)$ par $(100,100,100)$ puisque $\frac{50+70+180}{3} = 100$.

Structure de données. On numérote les pixels de l'image ligne par ligne, en commençant en haut à gauche. On veut stocker une image de taille `LARGEUR`×`HAUTEUR`, donc on réserve le tableau suivant :

```
char* image = malloc(LARGEUR*HAUTEUR*3);
```

On envisage deux solutions pour organiser les informations dans le tableau :

▷ **La solution A** écrit les trois composantes de chaque pixel à la suite. Les composantes du $i^{\text{ème}}$ pixel sont donc `(image[3*i],image[3*i+1],image[3*i+2])`.

▷ **La solution B** divise `image` en trois sous-tableaux

```
char* rouge=image, vert=rouge+LARGEUR*HAUTEUR, bleu=vert+LARGEUR*HAUTEUR;
```

Les composantes du $i^{\text{ème}}$ pixel sont donc `(rouge[i],vert[i],bleu[i])`, c'est à dire `(image[i],image[i+LARGEUR*HAUTEUR],image[i+2*LARGEUR*HAUTEUR])`.

Méthode de conversion. Le code de conversion en niveaux de gris est :

```
char* dest = malloc(LARGEUR*HAUTEUR*3);
int i,moy;
char gris;
// U et V sont deux constantes définies ci-dessous

for (i=0; i<LARGEUR*HAUTEUR; i++){
  moy = (int)image[V*i] + (int)image[V*i+U] + (int)image[V*i+2*U]; // on cast en int
  moy = moy/3;                                                     // pour éviter le % 256
  gris = (char)moy;                                                // du calcul en char
  dest[V*i] = gris;
  dest[V*i+U]= gris;
  dest[V*i+2*U] = gris;
}
```

Les constantes `U` et `V` sont définies ainsi :

Solution A

```
#define U 1
#define V 3
```

Solution B

```
#define U LARGEUR*HAUTEUR
#define V 1
```

1. Un pixel est un point de l'image. Une image en résolution 2000×1000 est composée de 2 millions de pixels.

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 tranfert :

- 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 `0,1...1,1...1,1...1` (ici aussi en binaire).

Pour la converstion 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`)