Examen Novembre 2011 - Architectures Avancées

3H – Tous documents autorisés

Parties indépendantes

OPTIMISATION DE BOUCLES

On utilise le processeur superscalaire défini dans l'annexe 1, ou sa version scalaire
Soit la boucle suivante écrite en assembleur, qui travaille sur des tableaux de « floats » X[256]et Y[256]. Au démarrage R1 contient X[0] et R2 contient l'adresse de Y[0]. R4 contient la valeur 256. F0 contient la valeur 0.5.

```
        ADDI R4,R4, -2
Boucle : LF F1, 0(R1)
        LF F2, 8(R1)
        FADD F1,F1,F2
        FMUL F1,F1,F0
        SF F1, 4(R2)
        ADDI R1,R1,4
        ADDI R2,R2,4
        ADDI R4,R4,-1
        BNE R4, Boucle
```

Question 1) Donner le programme C équivalent.

Question 2) Donner l'exécution cycle par cycle de la boucle optimisée avec un processeur scalaire. Quel est, en nombre de cycles, le temps d'exécution par itération de la boucle ?

Question 3) Donner l'exécution cycle par cycle de la boucle optimisée en plaçant les instructions dans les différents pipelines E0, E1, FA et FM. Quel est, en nombre de cycles, le temps d'exécution par itération de la boucle ?

Caches (1ère partie)

Soit le programme C suivant

```
float X[512], Y[512]
for (i=0 ; i<510 ; i++)
        Y[i] = X[i] + X[i+2];
```

On utilise un cache donnée de 8Ko, avec des lignes de 32 octets. On suppose que les deux tableaux sont rangés l'un derrière l'autre (&Y[0] = &X[511] +4) et que &X[0] = 0. Le cache est à allocation d'écriture (il y a des défauts de cache en écriture) et à réécriture (write back).

Question 4) Quel est le nombre total de défauts de cache dans les deux cas suivants

- cache à correspondance directe ?
- cache associatif deux voies ?

Caches (2ère partie)

On suppose des caches à écriture allouée. Les lignes de cache contiennent p « flottants ».
On ne considère que les défauts de démarrage

- il n'y a pas de réutilisation des données, sauf pour la boucle interne de ikj
- il n'y a pas de défauts de conflit

On considère le produit de matrices Z[N][N] = X N][N] * Y N][N] avec des données flottantes.

Question 5) Quel est le nombre total de défauts de cache en fonction de N dans les deux cas suivants (les algorithmes sont rappelés en annexe)

- produit ikj
- produit ijk après transposition

Instructions SIMD

Soit l'instruction SIMD SSE2 SHUFPD qui travaille sur des doubles (64 bits) dont l'intrinsics est __m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)

Soit le define
define SHUFPD (a,b) __m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)

Ah correspond aux 64 bits de poids fort de A, Ab correspond aux 64 bits de poids faible de A, Bh correspond aux 64 bits de poids fort de B, Bb correspond aux 64 bits de poids faible de B. Alors

SHUFPD(A,B,0) = (Bb|Ab)
SHUFPD(A,B,1) = (Bb|Ah)
SHUFPD(A,B,2) = (Bh|Ab)
SHUFPD(A,B,3) = (Bh|Ah)

Soient les déclarations suivantes :
_m128d **XS ; //mots de 128 bits contenant des doubles
double **X ;
XS=X ;

Question 6) Donner le résultat des intrinsics suivants

- `SHUFPD (ld2d(&XS [i][j]), ld2d(&XS [i][j-1]),1)`
- `SHUFPD (ld2d(&XS [i][j+1]), ld2d(&XS [i][j]),1)`

Question 7) Donner le programme C scalaire correspondant au programme C SIMD suivant :

```
_m128d **XS ,**YS, D, tmp, tmp2,tmp3, tmp4 ;//mots de 128 bits contenant
                                              des doubles
double **X, **Y ;
XS=X ; YS=Y ;

D= _mm_set1_pd (0.25) ; // crée un mot de deux doubles égaux à 0.25
for (i=1 ; i<255 ; i++)
for (j=1 ; j<127 ; j++)
{
tmp = addpd (ld2d (&XS[i-1][j], ld2d(&XS[i+1][j]) ;
tmp2 = SHUFPD (ld2d(&XS [i][j]), ld2d(&XS [i][j-1]),1);
tmp3 = SHUFPD (ld2d(&XS [i][j+1]),ld2d(&XS [i][j]),1);
tmp2 = addpd (tp2,tmp3;
tmp = addpd (tmp;tmp2);
tmp=mulpd (tmp, D);
st2d (&YS[i][j], tmp)
}
```

PROGRAMMATION OPENMP

Soit le programme C ci-dessous (Jacobi).

```
#include <stdio.h>
double X[128][128], Xold[128][128];
```

```
/* version séquentielle */

int main(void)
{
      int i,j,k, maxit, size;
      double tol, error, resid;
tol = 0.0001;
error = 10*tol;
maxit=50000;
size=128*128;

for (i=0; i<128;i++)
      for (j=0; j<128; j++)
      X[i][j] =(double) (i*i+j*j+i+j);
k=0;


while (k<maxit & error>tol)
{
      error = 0.0;
      {
            for (i = 0; i<128;i++)
                  for (j=0; j<128 ; j++)
                  Xold[i][j]=X[i][j];
      }
for (i=1; i<128-1; i++)
      for (j=1 ; j<128-1; j++)      {
      X[i][j] = 0.25*(Xold[i-1][j]+Xold[i+1][j]+Xold[i][j-1]
            +Xold[i][j+1]);
      resid=X[i][j] - Xold[i][j];
      error+=resid*resid;}
      k++;
      error=sqrt(error)/(double)size;}
}
}
```

Question 8) Donner une version OpenMP du programme.

LOI D'AMDAHL

Question 9) Quel est la fraction maximale de la partie séquentielle d'un programme pour obtenir une efficacité parallèle de 50% avec 16 processeurs ?

PREDICTION DE BRANCHEMENT

Un programme a cinq branchements conditionnels notés Branch 1 à Branch 5. Le comportement des branchements est le suivant pour une exécution du programme (P pour branchement pris et N pour branchement non pris).
Branch 1: P-P-P-P-P
Branch 2: N-N-N
Branch 3: P-N-P-N-P-N-P-N
Branch 4: P-P-P-N-N-N
Branch 5: P-P-P-N-P-P-P-N-P
On suppose que le comportement de chaque branchement reste le même pour chaque exécution du programme. Les prédicteurs 1 bit et 2 bits sont toujours initialisés dans le même état avant chaque séquence d'exécution d'un des branchements.

Question 10) Donner le nombre de bonnes prédictions pour chaque branchement avec les prédicteurs suivants :

a) **Toujours pris**
b) **Toujours non pris**
c) **Prédicteur 1 bit, initialisé à prédit pris.**
d) **Prédicteur 2 bits, initialisé à prédit faiblement pris.**

Annexe 1

Soit un processeur superscalaire à ordonnancement statique qui a les caractéristiques suivantes :
- les instructions sont de longueur fixe (32 bits)
- Il a 32 registres entiers (dont R0=0) de 32 bits et 32 registres flottants (de F0 à F31) de 32 bits.
- Il peut lire et exécuter 4 instructions par cycle.
- L'unité entière contient deux pipelines d'exécution entière sur 32 bits, soit deux additionneurs, deux décaleurs. Tous les bypass possibles sont implantés.
- L'unité flottante contient un pipeline flottant pour l'addition et un pipeline flottant pour la multiplication. - L'unité Load/Store peut exécuter jusqu'à deux chargements par cycle, mais ne peut effectuer qu'un load et un store simultanément. Elle ne peut effectuer qu'un seul store par cycle.
- Il dispose d'un mécanisme de prédiction de branchement qui permet de "brancher" en 1 cycle si la prédiction est correcte. Les sauts et branchements ne sont pas retardés.

La Table 1 donne les instructions disponibles et le pipeline qu'elles utilisent : E0 et E1 sont les deux pipelines entiers, FA est le pipeline flottant de l'addition et FM le pipeline flottant de la multiplication. Les instructions peuvent être exécutées simultanément si elles utilisent chacune un pipeline séparé.L'addition et la multiplication flottante sont pipelinées. La division flottante n'est pas pipelinée (une division ne peut commencer que lorsque la division précédente est terminée).L'ordonnancement est statique.

JEU D'INSTRUCTIONS (extrait)

LF	LF Fi, dép.(Ra)	2	E0 ou E1	Fi ← M (Ra + dépl.16 bits avec ES)
SF	SF Fi, dép.(Ra)	0	E0	Fi -> M (Ra + dépl.16 bits avec ES)
ADD	ADD Rd,Ra, Rb	1	E0 ou E1	Rd ←Ra + Rb
ADDI	ADDI Rd, Ra, IMM	1	E0 ou E1	Rd ←Ra + IMM-16 bits avec ES
SUB	SUB Rd,Ra, Rb	1	E0 ou E1	Rd ←Ra - Rb
FADD	FADD Fd, Fa, Fb	3	FA	Fd ←Fa + Fb
FMUL	FMUL Fd, Fa, Fb	3	FM	Fd ←Fa x Fb
BEQ	BEQ Ri, dépl	1	E1	si Ri=0 alors CP ← NCP + depl
BNE	BNE Ri, dépl	1	E1	si Ri≠0 alors CP ← NCP + depl

Table 1 : instructions disponibles

ANNEXE 2 : Instructions SIMD IA-32 utilisables

#define ld2d(a)	_mm_load_pd(&a)	chargement aligné de deux doubles
#define st2d(a, b)	_mm_store_pd(&a, b)	rangement aligné de deux doubles
#define addpd(a,b)	_mm_add_pd(a,b)	Addition SIMD de 2 doubles
#define mulpd(a,b)	_mm_mul_pd(a,b)	Multiplication SIMD de 2 doubles
#define setd (a)	_mm_set1_pd (a)	Mot constitué de deux doubles a

ANNEXE 3 : Version des produits de matrice

Produit ikj

```
for (i=0;i<N;i++)
      for (k=0;k<N;k++)
{
            S=Y[i][k];
            Z [i][0]=0.0;
                  for (j=0; j<N; j++)
                        Z[i][j]+=S*X[k][j];
}
```

Produit ijk après transformation

```
for (i=0;i<N;i++)
      for (j=0;j<N;j++)
      XT[j][i]=X[i][j]; //transposition
for (i=0;i<N;i++)
      for (j=0;j<N;j++)
      {
      S=0.0;
            for (k=0; k<N; k++)
                S+=Y[i][k]*XT[j][k];
            Z[i][j]=S;
        }
```