Optimisation de code : exemple sur le processeur Cell

P. Fortin
pierre.fortin @ upmc.fr

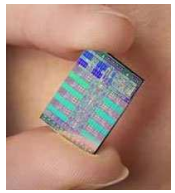
Programmation parallèle avancée (PPA)
Master 2 Informatique SAR-STL, UPMC

Remerciements et bibliographie

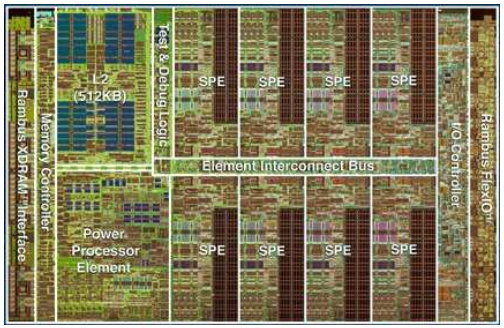
- Documentation sur le site IBM du processeur Cell : `http://www.ibm.com/developerworks/power/cell/documents.html`
- A. Arevalo, R.M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, C. Almond, *Programming the Cell Broadband Engine Architecture, Examples and Best Practices*, IBM Redbook SG24-7575, 2008.
- Cours de J.-L. Lamotte *Programmation parallèle avancée*, Master 2 Informatique SAR-STL, UPMC.

Le processeur Cell

- 9 coeurs :
  - 1 PPE
  - 8 SPE avec unités vectorielles
- 3.2 Ghz
- PS3 : 6 SPE exploitables (7 SPE -1 pour « l'hyperviseur »)



(Image IBM)



(Image IBM)

PPE et SPE

- PPE (Power Processor Element, basé sur 64-bit PowerPC) :
  - *General-purpose computing* avec SMT 2 voies ($\rightarrow$ 2 threads)
- SPE (Synergistic Processor Element) :
  - Spécialisés dans le calcul haute performance
  - Unités vectorielles (SIMD)
  - Mémoire locale :
    - Local Store (LS) de 256 Ko
    - Gestion explicite des transferts : mémoire principale $\leftrightarrow$ LS avec DMA (Direct Memory Access) à travers EIB (Element Interconnect Bus, bande passante max de 200 Go/s) avec recouvrement matériel des communications/calcul
    - But : s'attaquer au « memory wall »
  - Pas de caches
  - Pas de réordonnancement dynamique des instructions (exécution *in-order*)
  - Utilisation optimale : 1 thread / SPE (LS $\rightarrow$ coût du changement de contexte)

Performance crête

- Simple précision
  - 1 SPE : 3.2 Ghz * 4 (SIMD) * 2 (FMA)= **25.6 Gflop/s**
  - 8 SPE : $>$ **200 Gflop/s**
  - 25.6 Gflop/s pour PPE
  - $\ll$ Graphics SP-Float $\gg$ (pas IEEE 754)
- Double précision
  - Cell : $>$20 Gflop/s (1,8 Gflop/s par SPE, 6,4 Gflop/s pour le PPE)
  - 8 SPE : $>$**100 Gflop/s** pour PowerXCell 8i (IBM QS22 blade, Roadrunner)
  - Respect de la norme IEEE 754

Points forts et points faibles

- Maîtrise totale de l'architecture par le programmeur
- DMA performants : HWAs (SPE) $\ll$ proches $\gg$ du PPE
- Optimisation progressive du code $\rightarrow$ bonnes performances au final
- Pas besoin de parallélisme massif
- Programmation hardue
- PPE pas assez puissant
- Synchronisation PPE-SPE : pas de méchanisme d'attente passive du SPE au niveau du PPE
- IBM : abandon du Cell 2, $\ll$ intégration $\gg$ du Cell dans les nouveaux processeurs Power[8,9...]

Détails de l'architecture d'un SPE

- Unités vectorielles (SIMD) :
  - 128 registres de 128 bits : 4 floats (vector **float**) ou 2 double (vector **double**)
  - exemples d'instructions vectorielles :

```
v0 = spu_add(v1, v2),
v0 = spu_mul(v1, v2),
v0 = spu_madd(v1, v2, v3),
v = spu_splats(s),
s = spu_extract(v, i) ...
```

- Accès mémoire au LS : lecture (*load*) d'un registre vectoriel (128 bits) en 6 cycles depuis le LS ; idem pour l'écriture (*store*)
- 2 pipelines indépendants
  - pipeline 0 (*even*) : *Single-precision floating-point operations* (*add, mul, madd...*) en 6 cycles, ...
  - pipeline 1 (*odd*) : *loads* / *stores* en 6 cycles, ...

Contexte et mesure de temps

- Optimisation sur 1 SPE du calcul du produit scalaire de 2 vecteurs de 16384 éléments chacun (BLAS de niveau 1)
- Pas de DMA nécessaire entre la mémoire principale et le LS du SPE (tous les éléments des 2 vecteurs sont initialisés à 1.0).
- Grâce au SPU *decrementer* (compteur matériel) : mesures de temps très précises sur le SPE.
- Options d'optimisation à la compilation : `-O3`
- Quelle performance maximale peut-on obtenir ?

Les différentes versions

Voir annexes pour les codes et les mesures de performance.

Annexes : les différentes versions

Code scalaire :

```
#define MAX 16384
float a[MAX],b[MAX] __attribute__ (( aligned (128) ));

  int i;
  float res = 0.0;
  double nb_flops = 0.0;

  /* SPE timing: */
  SPU_INIT_TIMERS();

  /* Initialisation des vecteurs : */
  for (i=0;i<MAX;i++){
    a[i]=1.f;
    b[i]=1.f;
  }

  /* SPE timing: */
  SPU_START_TIMER(timer);


  for (i=0; i<MAX; i++){
    res += a[i] * b[i];
  }

  /* SPE timing: */
  SPU_STOPnADD_TIMER(timer);
  SPU_TERMINATE_TIMERS();

  printf("SPE %llx : res = %.7e \n", spe, res);

  /* Calcul des Gflop/s : */
  /* 2 flop (madd) par "MAX float elements" */
  nb_flops = ((double) MAX)* 2 ;
  printf("Nb of flops : %g \n", nb_flops);
  printf("Tics : %u \t Time : %g seconds\n", SPU_TIMER_IN_TICS(timer), SPU_TIMER_IN_SECONDS(timer));
  printf("Performance : %g Gflop/s \n", nb_flops / (1000000000.0 * SPU_TIMER_IN_SECONDS(timer)));
```

Performance : 0,53 Gflop/s

Code SIMD

Vectorisation du code, et réduction finale (surcoût)

```
#define MAX 4096
vector float a[MAX],b[MAX] __attribute__ (( aligned (128) ));

  int i;
  float res = 0.0;
  double nb_flops = 0.0;
  register vector float s;

  /* SPE timing: */
  SPU_INIT_TIMERS();

  /* Initialisation des vecteurs : */
  for (i=0;i<MAX;i++){
    a[i]=spu_splats(1.f);
    b[i]=spu_splats(1.f);
  }
  s=spu_splats(0.f);

  /* SPE timing: */
  SPU_START_TIMER(timer);

  for (i=0; i<MAX; i++){
    s=spu_madd(a[i],b[i],s);
  }

  /* Calculs restants: */
  res = spu_extract(s,0) + spu_extract(s,1) + spu_extract(s,2) + spu_extract(s,3);

  /* SPE timing: */
  SPU_STOPnADD_TIMER(timer);
  SPU_TERMINATE_TIMERS();

  printf("SPE %llx : res = %.7e \n", spe, res);

  /* Calcul des Gflop/s : */
  /* 2 flop (madd) par "MAX vector float elements" */
  nb_flops = ((double) MAX)* 2 * 4;
  printf("Nb of flops : %g \n", nb_flops);
  printf("Tics : %u \t Time : %g seconds\n", SPU_TIMER_IN_TICS(timer), SPU_TIMER_IN_SECONDS(timer));
  printf("Performance : %g Gflop/s \n", nb_flops / (1000000000.0 * SPU_TIMER_IN_SECONDS(timer)));
```

Performance : 3,20 Gflop/s

Déroulage de boucle

Déroulage de boucle (*loop unrolling*) d'un facteur 4.

```
for(i=0;i<MAX;i+=4){
  s=spu_madd(a[i],b[i],s);
  s=spu_madd(a[i+1],b[i+1],s);
  s=spu_madd(a[i+2],b[i+2],s);
  s=spu_madd(a[i+3],b[i+3],s);
}
```

Performance : 3,94 Gflop/s

Dépendances

Les FMA ne sont pas indépendants ce qui empêche le remplissage des pipelines comme le montre `spu-timing` :

```
                                                                .L4:
000018 0D                  -89                                    ai      $12,$12,-1
000018 1D                   890123                                lqd     $18,0($11)
000019 1                     901234                              lqd     $5,0($10)
000020 1                      012345                             lqd     $16,16($11)
000021 1                       123456                            lqd     $9,16($10)
000022 1                        234567                           lqd     $15,32($11)
000023 1                         345678                          lqd     $7,32($10)
000024 1                          456789                         lqd     $8,48($11)
000025 0D                          567890                        fma     $17,$18,$5,$13
000025 1D                          567890                        lqd     $6,48($10)
000026 0                            67                           ai      $11,$11,64
000027 0                             78                          ai      $10,$10,64
000031 0                              ---123456                  fma     $3,$16,$9,$17
000037 0                                   -----789012           fma     $4,$15,$7,$3
000043 0D                                        -----345678      fma     $13,$8,$6,$4
                                                                .L15:
000043 1D                                             3456      brnz    $12,.L4
```

→ Utilisation de variables supplémentaires (et besoin de calculs supplémentaires).

```
int i;
register vector float s0,s1,s2,s3;
float res = 0.0;
double nb_flops = 0.0;

/* SPE timing: */
SPU_INIT_TIMERS();

/* Initialisation des vecteurs : */
for (i=0;i<MAX;i++){
  a[i]=spu_splats(1.f);
  b[i]=spu_splats(1.f);
}
s0=spu_splats(0.f);
s1=spu_splats(0.f);
s2=spu_splats(0.f);
s3=spu_splats(0.f);

/* SPE timing: */
SPU_START_TIMER(timer);

for(i=0;i<MAX;i+=4){
  s0=spu_madd(a[i],b[i],s0);
  s1=spu_madd(a[i+1],b[i+1],s1);
  s2=spu_madd(a[i+2],b[i+2],s2);
  s3=spu_madd(a[i+3],b[i+3],s3);
}

/* Calculs restants : */
s0=spu_add(s0,s1);
s2=spu_add(s2,s3);
s0=spu_add(s0,s2);
res = spu_extract(s0,0) + spu_extract(s0,1) + spu_extract(s0,2) + spu_extract(s0,3);

/* SPE timing: */
SPU_STOPnADD_TIMER(timer);
```

Performance : 7,28 Gflop/s

Software pipelining

Analyse du code actuel généré :

```
                                                        .L4:
000019 0D                    -90                          ai      $12,$12,-1
000019 1D                     901234                      lqd     $9,0($11)
000020 1                       012345                    lqd     $5,0($10)
000021 1                        123456                   lqd     $6,16($11)
000022 1                         234567                  lqd     $18,16($10)
000023 1                          345678                 lqd     $7,32($11)
000024 1                           456789                lqd     $3,32($10)
000025 1                            567890               lqd     $8,48($11)
000026 0D                            678901              fma     $16,$9,$5,$16
000026 1D                            678901              lqd     $4,48($10)
000027 0                              78                 ai      $11,$11,64
000028 0                               890123            fma     $14,$6,$18,$14
000029 0                                90               ai      $10,$10,64
000030 0                                 012345          fma     $15,$7,$3,$15
000032 0D                                 -234567        fma     $13,$8,$4,$13
                                                        .L15:
000032 1D                                  2345          brnz    $12,.L4
```

Les FMA ne sont pas entièrement pipelinés, et peu de paires d'opérations effectuées en même temps sur les 2 pipelines (*dual-issue rate*) : pas de recouvrement des accès mémoires par du calcul ...

Accès mémoire explicites

Mise en évidence des *loads* effectués depuis le LS vers les registres vectoriels.

```
int i;
register vector float x0,x1,x2,x3;
register vector float y0,y1,y2,y3;
register vector float s0,s1,s2,s3;
float res = 0.0;
double nb_flops = 0.0;

/* SPE timing: */
SPU_INIT_TIMERS();

/* Initialisation des vecteurs : */
for (i=0;i<MAX;i++){
  a[i]=spu_splats(1.f);
  b[i]=spu_splats(1.f);
}
s0=spu_splats(0.f);
s1=spu_splats(0.f);
s2=spu_splats(0.f);
s3=spu_splats(0.f);

/* SPE timing: */
SPU_START_TIMER(timer);

for(i=0;i<MAX;i+=4){
  x0=a[i];
  y0=b[i];
  s0=spu_madd(x0,y0,s0);

  x1=a[i+1];
  y1=b[i+1];
  s1=spu_madd(x1,y1,s1);

  x2=a[i+2];
  y2=b[i+2];
  s2=spu_madd(x2,y2,s2);
```

```
    x3=a[i+3];
    y3=b[i+3];
    s3=spu_madd(x3,y3,s3);
}

/* Calculs restants : */
s0=spu_add(s0,s1);
s2=spu_add(s2,s3);
s0=spu_add(s0,s2);
res = spu_extract(s0,0) + spu_extract(s0,1) + spu_extract(s0,2) + spu_extract(s0,3);

/* SPE timing: */
SPU_STOPnADD_TIMER(timer);
```

Mise en place du *Software pipelining*

Recouvrir les accès mémoire par du calcul (maximisation du *dual-issue rate*).
Et aussi : entrelacement des instructions à la main (*manual instruction interleaving*).

```
/* SPE timing: */
SPU_START_TIMER(timer);

x0=a[0];
y0=b[0];
x1=a[1];
y1=b[1];
for(i=2;i<MAX;i+=4){
    x2=a[i];
    s0=spu_madd(x0,y0,s0);
    y2=b[i];

    x3=a[i+1];
    s1=spu_madd(x1,y1,s1);
    y3=b[i+1];

    x0=a[i+2];
    s2=spu_madd(x2,y2,s2);
    y0=b[i+2];

    x1=a[i+3];
    s3=spu_madd(x3,y3,s3);
    y1=b[i+3];
}

/* Calculs restants: */
s0=spu_madd(x0,y0,s0);
s1=spu_madd(x1,y1,s1);

s0=spu_add(s0,s1);
s2=spu_add(s2,s3);
s0=spu_add(s0,s2);
res = spu_extract(s0,0) + spu_extract(s0,1) + spu_extract(s0,2) + spu_extract(s0,3);

/* SPE timing: */
SPU_STOPnADD_TIMER(timer);
```

Performance : 8,48 Gflop/s

Code optimal

Meilleur remplissage du pipeline 1 (*load/store*) avec un déroulage de boucle d'un facteur 8 (attention à la taille du code et au *register spilling*).

```
int i;
register vector float x0,x1,x2,x3,x4,x5,x6,x7;
register vector float y0,y1,y2,y3,y4,y5,y6,y7;
register vector float s0,s1, s2, s3, s4, s5, s6, s7;
float res = 0.0;
double nb_flops = 0.0;

/* SPE timing: */
SPU_INIT_TIMERS();

/* Initialisation des vecteurs : */
for (i=0;i<MAX;i++){
  a[i]=spu_splats(1.f);
  b[i]=spu_splats(1.f);
}
s0=spu_splats(0.f); s1=spu_splats(0.f); s2=spu_splats(0.f); s3=spu_splats(0.f);
s4=spu_splats(0.f); s5=spu_splats(0.f); s6=spu_splats(0.f); s7=spu_splats(0.f);

/* SPE timing: */
SPU_START_TIMER(timer);

x0=a[0]; y0=b[0]; x1=a[1]; y1=b[1];
x2=a[2]; y2=b[2]; x3=a[3]; y3=b[3];

for(i=4;i<(MAX);i+=8){
  x4=a[i];
  s0=spu_madd(x0,y0,s0);
  y4=b[i];

  x5=a[i+1];
  s1=spu_madd(x1,y1,s1);
  y5=b[i+1];

  x6=a[i+2];
  s2=spu_madd(x2,y2,s2);
  y6=b[i+2];

  x7=a[i+3];
  s3=spu_madd(x3,y3,s3);
  y7=b[i+3];

  x0=a[i+4];
  s4=spu_madd(x4,y4,s4);
  y0=b[i+4];

  x1=a[i+5];
  s5=spu_madd(x5,y5,s5);
  y1=b[i+5];

  x2=a[i+6];
  s6=spu_madd(x6,y6,s6);
  y2=b[i+6];

  x3=a[i+7];
  s7=spu_madd(x7,y7,s7);
  y3=b[i+7];

}

/* Calculs restants: */
```

```
s0=spu_madd(x0,y0,s0); s1=spu_madd(x1,y1,s1);
s2=spu_madd(x2,y2,s2); s3=spu_madd(x3,y3,s3);

s0=spu_add(s0,s1); s2=spu_add(s2,s3); s4=spu_add(s4,s5); s6=spu_add(s6,s7);
s0=spu_add(s0,s2); s4=spu_add(s4,s6);
s0=spu_add(s0,s4);
res = spu_extract(s0,0) + spu_extract(s0,1) + spu_extract(s0,2) + spu_extract(s0,3);

/* SPE timing: */
SPU_STOPnADD_TIMER(timer);
```

Analyse du code actuel généré :

```
                                                                .L4:
000025 0D                          567890                         fma     $24,$19,$16,$24
000025 1D                          567890                         lqd     $9,0($11)
000026 0D                           678901                        fma     $28,$18,$14,$28
000026 1D                           678901                        lqd     $7,0($10)
000028 0D                            −890123                      fma     $21,$17,$13,$21
000028 1D                            890123                       lqd     $8,16($11)
000030 0D                             −012345                     fma     $27,$15,$12,$27
000030 1D                             012345                      lqd     $5,16($10)
000031 0D                               12                        ai      $20,$20,−1
000031 1D                               123456                    lqd     $6,32($11)
000032 1                                 234567                   lqd     $3,32($10)
000033 1                                  345678                  lqd     $4,48($11)
000034 0D                                  456789                 fma     $26,$9,$7,$26
000034 1D                                  456789                 lqd     $30,48($10)
000035 0D                                   5                     nop     127
000035 1D                                   567890                lqd     $19,64($11)
000036 0D                                    678901               fma     $23,$8,$5,$23
000036 1D                                    678901               lqd     $16,64($10)
000037 0D                                     7                   nop     127
000037 1D                                     7                   hbrp    # 3
000038 0D                                      890123             fma     $25,$6,$3,$25
000038 1D                                      890123             lqd     $18,80($11)
000039 0D                                       9                 nop     127
000039 1D                                       901234            lqd     $14,80($10)
000040 0D                                        012345           fma     $22,$4,$30,$22
000040 1D                                        012345           lqd     $17,96($11)
000041 1                                           123456         lqd     $13,96($10)
000042 1                                            234567        lqd     $15,112($11)
000043 0D                                            34           ai      $11,$11,128
000043 1D                                            345678       lqd     $12,112($10)
000044 0D                                             45          ai      $10,$10,128
                                                                .L15:
000044 1D                                              4567        brnz    $20,.L4
```

Meilleur *dual-issue rate* (meilleur recouvrement des accès mémoire par du calcul).

Performance : 10,79 Gflop/s, soit 84,3 % de la performance maximale pour ce calcul.