Tiling pour une séquence de nids de boucles

Youcef Bouchebaba — Fabien Coelho

CRI-ENSMP
35 Rue Saint Honoré
Fontainebleau 77305
France

{boucheba,coelho}@cri.ensmp.fr

RÉSUMÉ. *Le tiling est souvent appliqué à un seul nid de boucles. Dans cet article nous appliquons le tiling et la fusion simultanément sur une séquence de nids de boucles parallèles afin de pouvoir minimiser les mouvements de données et la consommation d'énergie et/ou de maximiser la vitesse d'exécution. Chacun de ces nids utilise un patron de données qui sont calculées dans le nid précédent. Après l'application du tiling, nous devons garantir que toutes les données qui sont nécessaires pour le calcul d'une itération ont déjà été calculées par les itérations précédentes. Pour cela nous ajoutons un délai pour les calculs de chaque nid. Nous proposons une méthode pour mettre en relation les différents patrons, les délais ajoutés et les paramètres du tiling et nous donnons une solution pour une classe de tiling. Pour maintenir en mémoire les données vivantes, nous remplaçons les différents tableaux par des tampons circulaires.*

ABSTRACT. *Usually tiling is applied to one loop nest at a time. In this paper we apply tiling and fusion simultaneously to a sequence of parallel nested loops in order to minimize data movements and energy consumption and/or to maximize the speed of execution. Each of these nests uses as input a stencil of data computed in a previous nest. After fusion and tiling, we guarantee that data necessary to the execution of an iteration has been already computed by the previous iterations by delaying the computation of consumer nest. We take into account the relation among the various stencils, the added delays and the tiling parameters and we give a solution for a class of tiling. To store the live data elements, we compute the surface of these data for every array and during the code generation we replace this array by a buffer whose size is equal to the surface of live data.*

MOTS-CLÉS : *tiling hiérarchique, fusion de boucles, transformation de programmes*

KEYWORDS: *hierarchical tiling, loop fusion, program transformations*

1re soumission à *Technique et science informatiques*, le 31st August 2001.

1. Introduction

L'objectif de ces travaux est de minimiser l'énergie électrique consommée au cours de l'exécution des programmes de traitement du signal qui sont constitués de séquences de nids de boucles. La plus grande partie de cette consommation serait due aux transferts des données entre les différents niveaux de la hiérarchie mémoire [CAT 98]. Pour cela nous transformons ces programmes en utilisant le tiling [IRI 88, WOL 92, XUE 97, XUE 98, WOL 96, KAN 99, WOL 98], la fusion de boucles [SAR 97, ZIM 90, MCK 96, PUG 99, WOL 96, MAN 97] et les transformations unimodulaires [BAN 93, XUE 98, WOL 91, CHE 94], afin de minimiser ces transferts.

Malheureusement, la plupart des travaux sur le tiling sont consacrés uniquement aux codes qui sont constitués d'un seul nid de boucles. Pour les codes qui sont constitués de plusieurs nids, certains travaux proposent d'appliquer la fusion de ces différents nids et ensuite appliquer le tiling au nid fusionné.

Irigoin et Triolet [IRI 88] ont modélisé le tiling par une matrice $H \in Q^{n*n}$ et ont donné une condition suffisante pour l'application de ce tiling. Par contre Xue [XUE 97] a proposé d'utiliser le tiling comme une transformation de boucles et a donné la condition nécessaire et suffisante à son application. Dans notre approche nous avons combiné la modélisation de Xue [XUE 97] avec les différentes techniques de fusion de boucles. Contrairement à Xue [XUE 97] qui s'est intéressé à un seul niveau de tiling et aux codes qui sont constitués d'un seul nid de boucles, nous traitons deux niveaux de tiling et les codes en entrée sont des séquences de nids de boucles.

Chacun de ces nids accède à des données suivant un *patron*. Ces données sont produites dans le nid précédent. Après l'application du tiling, nous devons garantir que toutes les données qui sont nécessaires pour le calcul d'une itération ont déjà été calculées par les itérations précédentes. Pour cela, nous ajoutons un délai [PUG 99] à chaque nid qui sera calculé en fonction du tiling et du *patron*. Contrairement aux autres travaux qui déterminent quel tiling est valide en fonction des dépendances du code initial, nous déterminons quels sont les délais que nous devons ajouter aux nids pour qu'un tiling donné soit valide. Notre tiling est représenté par deux matrices : (1) une matrice de tiling hiérarchique à deux niveaux qui donne les différents coefficients des tuiles et (2) une matrice de permutation qui permet d'échanger plusieurs boucles et donc de spécifier l'ordonnancement des tuiles.

Pour maintenir en mémoire les données vivantes et pour éviter de charger plusieurs fois les mêmes données, nous remplaçons les tableaux associés aux différents nids par des tampons circulaires, dont la taille est égale au nombre de données vivantes du tableau correspondant.

2. Exemple

Afin de mieux comprendre le problème traité, voici un exemple de code constitué de deux nids de boucles (*Figure 1*), auquel nous appliquons le tiling qui est défini par la matrice de tiling hiérarchique $\mathcal{A} = \begin{pmatrix} 4 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 2 & 1 \end{pmatrix}$ et la matrice de permutation $\mathcal{P} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$.

```
do (i = 1, N_i)
    do (j = 1, N_j)
       A1(i, j) = ....
    enddo
enddo
do (i = 2, N_i - 1)
    do (j = 2, N_j - 1)
       A2(i, j) = A1(i - 1, j) + A1(i, j - 1)
                  +A1(i, j) + A1(i, j + 1) + A1(i + 1, j)
    enddo
enddo
```

FIG. 1. *Exemple de code.*

2.1. *Transformation par* $\mathcal{A}$

La matrice $\mathcal{A}$ permet de transformer chaque point $(i,\ j)^t \in Z^2$ du domaine initial en un point $(i_1,\ i_2,\ i_3,\ j_1,\ j_2,\ j_3)^t \in Z^6$ avec $(i,\ j)^t = \mathcal{A} \ . \ (i_1,\ i_2,\ i_3,\ j_1,\ j_2,\ j_3)^t$ (voir *Figure 2*). Les coefficients de cette matrice nous donnent les tailles des différentes tuiles :

– $a_{1,1} \times a_{2,4}$ est le tuilage le plus externe (premier niveau du tiling) ;

– $a_{1,2} \times a_{2,5}$ est le tuilage le plus interne (deuxième niveau du tiling).

2.2. *Transformation par* $\mathcal{P}$

La matrice $\mathcal{P}$ transforme chaque point $(i_1,\ i_2,\ i_3,\ j_1,\ j_2,\ j_3)^t \in Z^6$ en un point $(l_1,\ l_2,\ l_3,\ l_4,\ l_5,\ l_6)^t$ avec $(l_1,\ l_2,\ l_3,\ l_4,\ l_5,\ l_6)^t = \mathcal{P} \ . \ (i_1,\ i_2,\ i_3,\ j_1,\ j_2,\ j_3)^t \in Z^6$ (voir *Figure 3*).

À chaque itération $(i,\ j)^t$ du deuxième nid du code de la *Figure 1* nous calculons l'élément $A_2(i,\ j)$ en fonction d'un *patron* de données du tableau A_1 qui sont produites dans le nid précédent. Ce *patron* est représenté par les vecteurs $\vec{v}_1 = (-1,\ 0)^t$, $\vec{v}_2 = (0,\ -1)^t$, $\vec{v}_3 = (0,\ 0)^t$, $\vec{v}_4 = (0,\ 1)^t$ et $\vec{v}_5 = (1,\ 0)^t$.

```
do (i_1 = 0, N_i/4 - 1)
 do (i_2 = 0, 3)
  do (i_3 = 0, 1)
   do (j_1 = 0, N_j/4 - 1)
    do (j_2 = 0, 3)
     do (j_3 = 0, 1)
      A_1(4i_1 + 2i_2 + i_3 + 1, 4j_1 + 2j_2 + j_3 + 1) = ....
      if (4i_1 + 2i_2 + i_3 >= 2) and (4j_1 + 2j_2 + j_3 >= 2)
        A_2(4i_1 + 2i_2 + i_3, 4j_1 + 2j_2 + j_3) = A_1(4i_1 + 2i_2 + i_3 - 1, 4j_1 + 2j_2 + j_3)
                                                  + A_1(4i_1 + 2i_2 + i_3, 4j_1 + 2j_2 + j_3 - 1)
                                                  + A_1(4i_1 + 2i_2 + i_3, 4j_1 + 2j_2 + j_3)
                                                  + A_1(4i_1 + 2i_2 + i_3, 4j_1 + 2j_2 + j_3 + 1)
                                                  + A_1(4i_1 + 2i_2 + i_3 + 1, 4j_1 + 2j_2 + j_3)
     enddo
    enddo
   enddo
  enddo
 enddo
enddo
```

FIG. 2. *Code après l'application de la matrice $\mathcal{A}$.*

```
do (l_1 = 0, N_i/4 - 1)
 do (l_2 = 0, N_j/4 - 1)
 do (l_3 = 0, 3)
  do (l_4 = 0, 3)
   do (l_5 = 0, 1)
     do (l_6 = 0, 1)
      A_1(4l_1 + 2l_3 + l_5 + 1, 4l_2 + 2l_4 + l_6 + 1) = ....
      if (4l_1 + 2l_3 + l_5 >= 2) and (4l_2 + 2l_4 + l_6 >= 2)
        A_2(4l_1 + 2l_3 + l_5, 4l_2 + 2l_4 + l_6) = A_1(4l_1 + 2l_3 + l_5 - 1, 4l_2 + 2l_4 + l_6)
                                                  + A_1(4l_1 + 2l_3 + l_5, 4l_2 + 2l_4 + l_6 - 1)
                                                  + A_1(4l_1 + 2l_3 + l_5, 4l_2 + 2l_4 + l_6)
                                                  + A_1(4l_1 + 2l_3 + l_5, 4l_2 + 2l_4 + l_6 + 1)
                                                  + A_1(4l_1 + 2l_3 + l_5 + 1, 4l_2 + 2l_4 + l_6)
     enddo
    enddo
   enddo
  enddo
 enddo
enddo
```

FIG. 3. *Code après la permutation $\mathcal{P}$.*

Après l'application du tiling défini par les matrices $\mathcal{A}$ et $\mathcal{P}$ sur ce code nous obtenons un seul nid de boucles (*Figure 3*). Le corps de ce nid est constitué de deux instructions ($A_1() = ...$ et $A_2() = ...$) et son vecteur d'itération sera représenté par le vecteur $(l_1,\ l_2,\ l_3,\ l_4,\ l_5,\ l_6)^t$. Pour que ce tiling soit valide nous devons assurer que tous les éléments du tableau A_1 qui sont nécessaires pour le calcul d'un élément du tableau A_2 à une itération $(l_1,\ l_2,\ l_3,\ l_4,\ l_5,\ l_6)^t$ ont déjà été calculés par les itérations précédentes. Pour satisfaire cette condition nous translatons le calcul de l'instruction $(A_1(i,j)\ =\ ...)$ du code avant le tiling avec un délai $\vec{d} = (d_1,\ d_2)^t$, l'instruction $(A_1(i,j) = ...)$ est alors remplacée par $(A_1(i+d_1,\ j+d_2) = ...)$.

Si on prend $\vec{d} = (1,\ 1)^t$, à une itération $(i_0,\ j_0)^t$:

1) dans le premier nid, nous calculons l'élément $A_1(i_0+1,\ j_0+1)$;

2) dans le deuxième nid, nous utilisons $A_1(i_0-1,\ j_0)$, $A_1(i_0,\ j_0-1)$, $A_1(i_0,\ j_0)$, $A_1(i_0,\ j_0+1)$ et $A_1(i_0+1,\ j_0)$.

Les éléments $A_1(i_0-1,\ j_0)$, $A_1(i_0,\ j_0-1)$, $A_1(i_0,\ j_0)$, $A_1(i_0,\ j_0+1)$ et $A_1(i_0+1,\ j_0)$ sont calculés dans le premier nid respectivement aux itérations $(i_0-2,\ j_0-1)^t$, $(i_0-1,\ j_0-2)^t$, $(i_0-1,\ j_0-1)^t$, $(i_0-1,\ j_0)^t$ et $(i_0,\ j_0-1)^t$. Nous pouvons vérifier qu'après le tiling défini par les matrices $\mathcal{A}$ et $\mathcal{P}$, $\forall (i_0,\ j_0)^t$: les itérations $(i_0-2,\ j_0-1)^t$, $(i_0-1,\ j_0-2)^t$, $(i_0-1,\ j_0-1)^t$, $(i_0-1,\ j_0)^t$ et $(i_0,\ j_0-1)^t$ sont exécutées avant l'itération $(i_0,\ j_0)^t$, par conséquent le tiling défini par $\mathcal{A}$ et $\mathcal{P}$ sur le code de la $Figure$ 1 est valide. En réalité le choix de $\vec{d} = (1,\ 1)^t$ rend les boucles i et j complètement permutables après la fusion des deux nids [WOL 92]. Quel que soit le tiling orthogonal appliqué, il sera valide ($Figure$ 4).

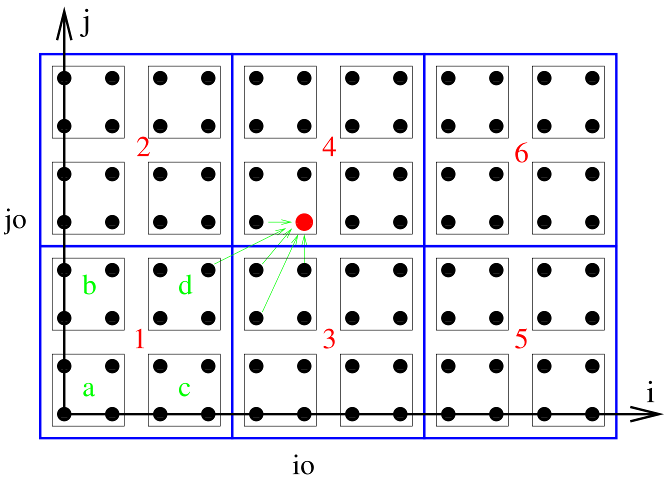


FIG. 4. : $\vec{d} = (1,\ 1)$.

De même si on choisit $\vec{d} = (4,\ 0)^t$, le tiling défini par les matrices $\mathcal{A}$ et $\mathcal{P}$ est valide. Par contre les boucles i et j du code fusionné ne sont pas complètement permutables, par conséquent le choix d'une autre taille de tuile, par exemple 5×5, n'est pas nécessairement valide (*Figure 5*).

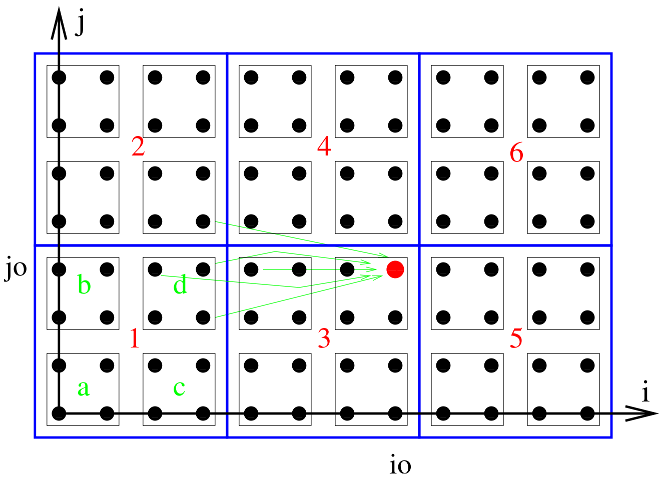


FIG. 5. $\vec{d} = (4,\ 0)$.

3. Code en entrée

Au vu des applications concernées, nous supposons que le code en entrée est une séquence de nids de boucles à deux dimensions. Chacun de ces nids accède à des données suivant un *patron* qui est représenté implicitement dans ce nid et donné explicitement par l'utilisateur sous forme d'un ensemble de vecteurs $R^k = \{\vec{v}_1^k,\ \vec{v}_2^k, \ldots.., \vec{v}_{m_k}^k\}$.

`do` $\vec{i}_1 \in D_1$
  $A_1(\vec{i}_1) = A_0(\vec{i}_1 + \vec{v}_1^1) \otimes \ldots. \otimes A_0(\vec{i}_1 + \vec{v}_{m_1}^1)$
`enddo`
.
.
`do` $\vec{i}_k \in D_k$
  $A_k(\vec{i}_k) = A_{k-1}(\vec{i}_k + \vec{v}_1^k) \otimes \ldots. \otimes A_{k-1}(\vec{i}_k + \vec{v}_{m_k}^k)$
`enddo`
.
.
`do` $\vec{i}_n \in D_n$
  $A_n(\vec{i}_n) = A_{n-1}(\vec{i}_n + \vec{v}_1^n) \otimes \ldots. \otimes A_{n-1}(\vec{i}_n + \vec{v}_{m_n}^n)$
`enddo`

FIG. 6. *Code en entrée.*

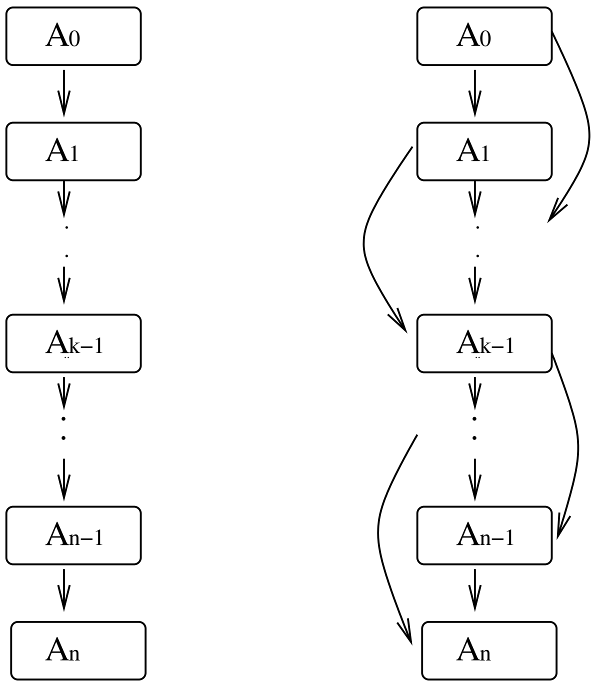
Le domaine D_0 associé au tableau A_0 est défini par l'utilisateur. Pour éviter des débordements au niveau des différents tableaux, les domaines $D_{k=1...n}$, sont dérivés de la manière suivante :

$$D_k = \{\vec{i}_k \mid \forall \vec{v} \in R^k : \vec{i}_k + \vec{v} \in D_{k-1}\}.$$

Nous supposons que les vecteurs des différents *patrons* sont ordonnés de la manière suivante :

$$\forall\, k : \vec{v}_1^k \preceq \vec{v}_2^k \preceq ..... \preceq \vec{v}_{m_k}^k$$

Dans cet article nous avons limité notre méthode aux codes qui sont sous forme d'une chaîne de tâches (*Figure 7(a)*), dont les nids de boucles sont de profondeur deux. Notre méthode est généralisable facilement au codes qui sont sous forme d'un graphe acyclique orienté (*Figure 7(b)*) et dont les nids sont de profondeur quelconque.



(a) Chaîne de tâches (b) Graphe orienté acyclique de tâches

FIG. 7. *Programmes que nous pouvons traiter.*

4. Fusion sans tiling

Pour pouvoir fusionner tous les nids en un seul, nous devons nous assurer que tous les éléments du tableau A_k nécessaires pour le calcul d'un élément $A_{k+1}(i,\ j)$ à une itération $(i,\ j)^t$ dans le nid fusionné, ont déjà été calculés par les itérations précédentes. Pour satisfaire cette condition nous translatons le calcul de chaque élément $A_k(i,\ j)(1 \leq k \leq n)$ avec un délai $\vec{d_k} = (d_{k,1},\ d_{k,2})^t$. Soit Time$_k$ la translation associée au nid k et qui est définie de la manière suivante :

Time$_k$: $D^k \rightarrow Z^2$ tel que $\vec{i_k} \longmapsto \vec{i} = \vec{i_k} + \vec{d_k}$

La fusion de tous ces nids est valide, si et seulement si, chaque translation Time$_k$ satisfait cette condition :

$$\forall \vec{i_k}, \forall \vec{i}_{k+1}, \exists \vec{v} \in R^{k+1} \ : \vec{i_k} = \vec{i}_{k+1} + \vec{v} \Rightarrow \text{Time}_k(\vec{i_k}) \preceq \text{Time}_{k+1}(\vec{i}_{k+1}) \qquad [1]$$

La condition [1] signifie que si une itération $\vec{i_k}$ produit un élément qui sera consommé par une itération $\vec{i}_{k+1}$, alors la translation de l'itération $\vec{i_k}$ par Time$_k$ doit être lexicographiquement inférieure à la translation de l'itération $\vec{i}_{k+1}$ par Time$_{k+1}$.

$$[1] \Leftrightarrow (\forall \vec{i_k},\ \vec{i}_{k+1}\ , \exists \vec{v} \in R^{k+1} \ : \vec{i_k} = \vec{i}_{k+1} + \vec{v} \Rightarrow \vec{v} \preceq -\vec{d_k} + \vec{d}_{k+1}) \qquad [2]$$

Le code fusionné après la translation des différents domaines aura la forme suivante :

do $\vec{i} \in D_{iter}$
$S_1 : C_1(\vec{i})\ \ A_1(\vec{i} - \vec{d_1}) = A_0(\vec{i} - \vec{d_1} + \vec{v}_1^1) \otimes .... \otimes A_0(\vec{i} - \vec{d_1} + \vec{v}_{m_1}^1)$
.
.
$S_k : C_k(\vec{i})\ \ A_k(\vec{i} - \vec{d_k}) = A_{k-1}(\vec{i} - \vec{d_k} + \vec{v}_1^k) \otimes .... \otimes A_{k-1}(\vec{i} - \vec{d_k} + \vec{v}_{m_k}^k)$
.
.
$S_n : C_n(\vec{i})\ \ A_n(\vec{i} - \vec{d_n}) = A_{n-1}(\vec{i} - \vec{d_n} + \vec{v}_1^n) \otimes .. \otimes A_{n-1}(\vec{i} - \vec{d_n} + \vec{v}_{m_n}^n)$
enddo

FIG. 8. *Nid fusionné.*

Le domaine D_{iter} est l'union des translations des domaines $D_{k=1..n}$ par les vecteurs $\vec{d}_{k=1..n}$:

$D_{iter} = \cup_{k=1}^{n}(D'_k)$, avec $D'_k = \{\vec{i} = \vec{i_k} + \vec{d_k} \mid \vec{i_k} \in D_k\}$.

Ce domaine n'étant pas nécessairement convexe, nous utiliserons son enveloppe convexe pour générer le code.

Comme l'instruction S_k n'est pas exécutée pour toute itérations du domaine D_{iter}, nous la gardons par la condition suivante :

$C_k(\vec{i}) = \text{if } (\vec{i} \in D'_k)$.

Comme les vecteurs des différents patrons sont ordonnés de la manière suivante, $\forall\ k : \vec{v}_1^k \preceq \vec{v}_2^k \preceq ..... \preceq \vec{v}_{m_k}^k$, la condition de validité de la fusion sera équivalente à

$-\vec{d}_k \succeq -\vec{d}_{k+1} + \vec{v}^{k+1}_{m_{k+1}}$: $(1 \leq k \leq n-1)$

La condition de légalité de la fusion de boucles sans tiling a été donnée par beaucoup d'auteurs [ZIM 90, MCK 96, PUG 99, WOL 96, MAN 97]. Par contre, l'application simultanée de la fusion et du tiling n'a été traitée par personne.

4.1. *Volume mémoire*

Pour maintenir en mémoire les données vivantes et pour éviter de charger plusieurs fois les mêmes données, nous remplaçons les tableaux A_1, $A_2 \ldots A_{n-1}$ par des tampons circulaires B_1, $B_2 \ldots B_{n-1}$.

4.1.1. *Données vivantes*

Soient $\vec{O}_k$ l'origine du domaine D'_k et $(N_{k,i} - 1,\ N_{k,j} - 1)^t + \vec{O}_k$ sa borne supérieure, c'est à dire que $D'_k = \{\vec{i}\ |\vec{O}_k \preceq \vec{i} \preceq (N_{k,i} - 1,\ N_{k,j} - 1)^t + \vec{O}_k\}$. Le volume mémoire $M_k(\vec{i})$ correspondant à une itération $\vec{i} = (i,\ j)^t \in D'_{k=2,n}$ est le nombre d'éléments du tableau A_{k-1} qui sont calculés avant $\vec{i}$ et qui ne sont pas complètement consommés :

$M_k(\vec{i}) = |E_k(\vec{i})|$ avec

$$E_k(\vec{i}) = \{\vec{i}_1\ \in D'_{k-1} |\ \exists \vec{v} \in R^k \text{ et } \exists\, \vec{i}_2 \in D'_k\ : \vec{i}_1 - \vec{d}_{k-1} = \vec{i}_2 - \vec{d}_k + \vec{v} \text{ et } \vec{i}_1 \preceq \vec{i} \preceq \vec{i}_2\}$$

À l'itération $\vec{i}$, pour calculer $A_k(\vec{i} - \vec{d}_k)$, nous utilisons m_k données du tableau A_{k-1} qui sont produites respectivement par $\vec{i}_1$, $\vec{i}_2 \ldots \vec{i}_{m_k}$ tels que :

$\vec{i}_q = \vec{i} - (\vec{d}_k - \vec{d}_{k-1} - \vec{v}^k_q)\ \ (q = 1, m_k)$.

La plus ancienne de ces productions est $\vec{i}_1$. Par conséquent, le volume $M_k(\vec{i})$ est borné par le nombre d'itérations dans D'_{k-1} entre $\vec{i}_1$ et $\vec{i}$. Cette borne supérieure est donnée par :

$\text{Sup}_k = (N_{k-1,j},\ 1)^t \cdot (\vec{d}_k - \vec{d}_{k-1} - \vec{v}^k_1) + 1.$

4.1.2. *Génération du code*

Soit B_k le tampon associé au tableau A_k. La borne supérieure Sup_{k+1} donnée précédemment, représente le nombre maximum de données vivantes du tableau A_k que nous pouvons avoir à une itération $\vec{i}$ du domaine D'_{k+1}. Par conséquent, la taille du tampon B_k sera égale à Sup_{k+1}. Afin de ranger et récupérer les éléments du tableau A_k dans le tampon B_k, nous définissons une fonction d'accès F_k :

$F_k : D'_k \rightarrow N$ tel que $\vec{i} \longmapsto F_k(\vec{i})$ avec

1) $F_k(\vec{O}_k) = 0$

2) $F_k(succ(\vec{i})) = \begin{cases} F_k(\vec{i}) + 1 \text{ if } (F_k(\vec{i}) \neq \text{Sup}_{k+1} - 1) \\ 0 \qquad\qquad \text{sinon} \end{cases}$

Pour satisfaire ces deux conditions, il suffit de choisir :

$$F_k(\vec{i}) = ((\vec{i} - \vec{O}_k).(N_{k,j},\ 1)^t) \mod \text{Sup}_{k+1}.$$

Nous considérons l'instruction S_k du code fusionné de la *Figure 8*. À une itération $\vec{i}$, nous calculons l'élément $A_k(\vec{i} - \vec{d}_k)$ en fonction de m_k éléments du tableau A_{k-1}, qui sont produits respectivement par $\vec{i}_1$, $\vec{i}_2 \ldots \vec{i}_{m_k}$. L'élément $A_k(\vec{i} - \vec{d}_k)$ sera rangé dans le tampon B_k à la position $F_k(\vec{i})$. Les éléments du tableau A_{k-1} sont déjà rangés dans le tampon B_{k-1} par les itérations $\vec{i}_1$,$\vec{i}_2 \ldots \vec{i}_{m_k}$ aux positions $F_{k-1}(\vec{i}_1)$, $F_{k-1}(\vec{i}_2)$,..,$F_{k-1}(\vec{i}_{m_k})$ ($\vec{i}_q = \vec{i} - (\vec{d}_k - \vec{d}_{k-1} - \vec{v}_q^k)$). Par conséquent, l'instruction S_k sera remplacée par :

$$C_k(\vec{i}) :\ B_k(F_k(\vec{i})) = B_{k-1}(F_{k-1}(\vec{i}_1)) \otimes .. \otimes B_{k-1}(F_{k-1}(\vec{i}_{m_k})).$$

4.2. *Exemple*

Nous considérons la séquence de nid de boucles donnée en exemple dans la *section 2*. Pour pouvoir fusionner les deux nids de cette séquence, nous translatons le domaine du premier nid avec un délai $\vec{d} = (1,\ 0)^t$, ce qui va nous donner le code suivant :

```
do (i = 0, Ni − 1)
   do (j = 1, Nj)
      A1(i + 1, j) = ....
      if (i ≥ 2) and (2 ≤ j ≤ Nj − 1)
        A2(i, j) = A1(i − 1, j) + A1(i, j − 1)+
                   A1(i, j) + A1(i, j + 1) + A1(i + 1, j)
   enddo
enddo
```

FIG. 9. *Nid fusionné.*

Comme nous l'avons signalé précédemment, pour maintenir en mémoire les données vivantes et pour éviter de charger plusieurs fois les même données du tableau A_1, nous remplaçons ce dernier par un tampon circulaire B_1. La taille de ce tampon qui est notée par S, sera égale au volume de données vivantes du tableau A_1 . Pour calculer ce volume, nous considérons une itération $\vec{i}$ dans le nid fusionné (*Figure 9*). À cette itération, nous utilisons 5 données du tableau A_1 qui sont produites par $\vec{i}_1 = \vec{i} - (2,\ 0)^t$, $\vec{i}_2 = \vec{i} - (1,\ 1)^t$, $\vec{i}_3 = \vec{i} - (1,\ 0)^t$, $\vec{i}_4 = \vec{i} - (1,\ -1)^t$ et $\vec{i}_5 = \vec{i} - (0,\ 0)^t$.

La plus ancienne de ces productions est $\vec{i}_1 = \vec{i} - (2,\ 0)^t$. Par conséquent, le volume mémoire $M(\vec{i})$ de l'itération $\vec{i}$ est donné par le nombre d'itérations entre $\vec{i}_1$ et $\vec{i}$, c'est à dire $M(\vec{i}) = (N_j,\ 1)^t.(2,\ 0)^t + 1$.

Après le remplacement du tableau A_1 par le tampon B_1, nous obtenons le code suivant :

```
do (i = 0,  N_i - 1)
 do (j = 1,  N_j)
  B_1((i * N_j + j - 1)  mod  S) = ...
  if (i >= 2) and (2 <= j <= N_j - 1)
    A2(i,  j) =  B_1(( (i - 2) * N_j + j - 1) mod S)
               + B_1(((i - 1) * N_j + j - 2) mod S)
               + B_1(((i - 1) * N_j + j - 1) mod S)
               + B_1(((i - 1) * N_j + j) mod  S)
               + B_1((i * N_j + j - 1) mod S)
 enddo
enddo
```

FIG. 10. *Nid avec l'allocation du tampon.*

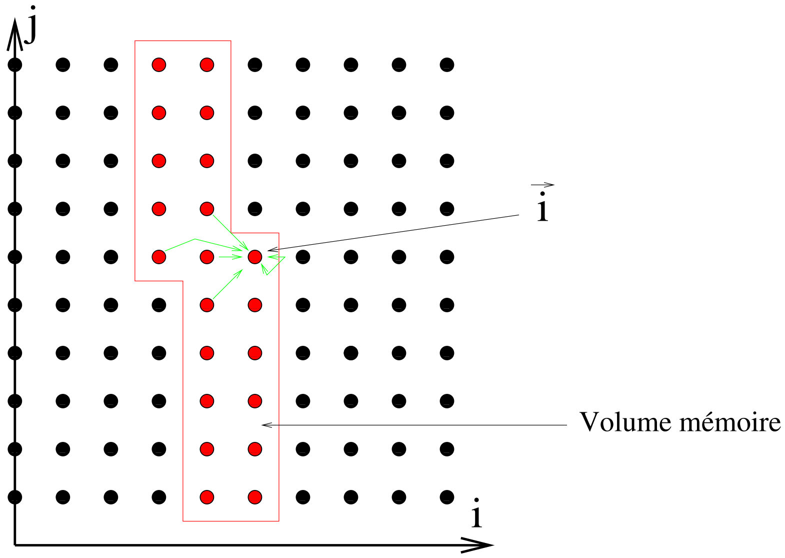


FIG. 11. *Volume mémoire pour le code de la $figure$ 9.*

L'introduction de ce tampon permet de gérer explicitement le cache de la machine. Si la taille de ce tampon s'avère très grand par rapport à la taille du cache, nous introduisons des tampons à plusieurs dimensions. Par exemple, chaque colonne du volume de la *figure 11* va correspondre à une dimension dans le tampon.

5. Fusion avec tiling

Dans la littérature, plusieurs auteurs se sont intéressés au tiling, mais la plupart de ces travaux sont consacrés uniquement aux codes constitués d'un seul nid de boucles. Dans cet article, nous présentons une méthode simple et efficace qui permet d'appliquer un tiling à deux niveaux (un niveau pour le cache et un autre pour les registres) pour une séquence de nids de boucles ($Figure$ 6). Notre tiling est représenté par deux matrices : 1) une matrice de tiling hiérarchique $\mathcal{A}$, qui donne la taille des différentes tuiles et 2) une matrice de permutation $\mathcal{P}$, qui spécifie l'ordonnancement des différentes tuiles.

De la même manière que la fusion, pour pouvoir appliquer le tiling sur le code initial (*Figure 6*), nous translatons le domaine de chaque nid avec une délai $\vec{d_k}$. Ce qui donne le code suivant :

do $\vec{i} \in D'_1$
  $A_1(\vec{i} - \vec{d_1}) = A_0(\vec{i} - \vec{d_1} + \vec{v}_1^1) \otimes .... \otimes A_0(\vec{i} - \vec{d_1} + \vec{v}_{m_1}^1)$
enddo
.
.
do $\vec{i} \in D'_k$
  $A_k(\vec{i} - \vec{d_k}) = A_{k-1}(\vec{i} - \vec{d_k} + \vec{v}_1^k) \otimes .... \otimes A_{k-1}(\vec{i} - \vec{d_k} + \vec{v}_{m_k}^k)$
enddo
.
.
do $\vec{i} \in D'_n$
  $A_n(\vec{i}_n - \vec{d_n}) = A_{n-1}(\vec{i}_n - \vec{d_n} + \vec{v}_1^n) \otimes .... \otimes A_{n-1}(\vec{i}_n - \vec{d_n} + \vec{v}_{m_n}^n)$
enddo

FIG. 12. *Séquence de nids de boucles après la translation des différents nids.*

Nous n'avons pas fusionné ces différents nids, car nous n'avons pas posé de conditions sur les délais ajoutés. Par conséquent, la condition de légalité de la fusion peut être fausse. Dans la $section$ 5.3, nous donnerons les relations que doivent vérifier ces délais pour que l'application du tiling et fusion simultanément soit valide.

Le domaine D'_k est la translation du domaine D_k par le vecteur $\vec{d_k}$. Pour simplifier la génération du code, nous choisissons le délai $\vec{d_1}$ d'une manière que l'origine du domaine D'_1 soit égal à $(0,\ 0)^t$.

5.1. *Tiling hiérarchique*

La matrice de tiling hiérarchique $\mathcal{A}$, permet de transformer le code en *Figure 12* en un nid intermédiaire donné en *Figure 13*. Cette matrice donne les différents facteurs du tiling et elle doit avoir la forme suivante :

$$\begin{pmatrix} a_{1,1} & a_{1,2} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{2,4} & a_{2,5} & 1 \end{pmatrix}$$

– $a_{1,1} \times a_{2,4}$ est le tuilage le plus externe (premier niveau du tiling) ;
– $a_{1,2} \times a_{2,5}$ est le tuilage le plus interne (deuxième niveau du tiling).

```
do (i1 = ...)
 do (i2 = ..)
  do (i3 = ...)
   do (j1 = ...)
    do (j2 = ...)
     do (j3 = ...)
     S1 : C1(i') A1(Ai' - d1) = A0(Ai' - d1 + v1^1) ⊗ ... ⊗ A0(Ai' - d1 + v^1_m1)
     .
     .
     Sk : Ck(i') Ak(Ai' - dk) = Ak-1(Ai' - dk + v1^k) ⊗ ... ⊗ Ak-1(Ai' - dk + v^k_mk)
     .
     .
     Sn : Cn(i') An(Ai' - dn) = An-1(Ai' - dn + v1^n) ⊗ ... ⊗ An-1(Ai' - dn + v^n_mn)
     enddo
    enddo
   enddo
  enddo
 enddo
enddo
```

FIG. 13. *Code après l'application de la matrice $\mathcal{A}$.*

De plus, les éléments de cette matrice doivent satisfaire les conditions suivantes :

$\forall\, i, j :\ \ a_{i,j} > 0$, $a_{1,2}$ divise $a_{1,1}$ et $a_{2,5}$ divise $a_{2,4}$.

Soient $\vec{i} = (i,\ j)^t$ le vecteur d'itération des domaines des différents nids du code de la *Figure 12* et $\vec{i'} = (i_1,\ i_2,\ i_3,\ j_1,\ j_2,\ j_3)^t$) le vecteur d'itération du code après l'application de la matrice $\mathcal{A}$ (*Figure 13*). Les relations entre ces deux vecteurs sont données par :

– $\vec{i} = \mathcal{A}\,\vec{i'}$

– $\vec{i'} = (\lfloor \frac{i}{a_{1,1}} \rfloor,\ \lfloor \frac{i\ mod\ a_{1,1}}{a_{1,2}} \rfloor,\ i\ mod\ a_{1,2},\ \lfloor \frac{j}{a_{2,4}} \rfloor,\ \lfloor \frac{j\ mod\ a_{2,4}}{a_{2,5}} \rfloor,\ j\ mod\ a_{2,5})^t$

Le schéma de la *Figure 14*, explique comment nous calculons les trois premières composantes ($i_1,\ i_2,\ et\ i_3$) du vecteur $\vec{i'}$ en fonction de i (la première composante du vecteur $\vec{i}$). L'explication reste la même pour le calcul des trois dernières composantes ($j_1,\ j_2,\ et\ j_3$) en fonction de j. Par exemple, si nous considérons une itération $\vec{i} = (i,\ j)^t$ tel que sa première composante $i = 10$, après l'application d'une matrice $\mathcal{A}$ tel que $a_{1,1} = 4$ et $a_{1,2} = 2$, l'itération $\vec{i}$ sera transformée à une itération $\vec{i'}$, avec $i_1 = 2$, $i_2 = 1$ et $i_3 = 0$.

De la même manière que la fusion, dans le code après l'application de la matrice $\mathcal{A}$ (*Figure 12*) l'instruction S_k doit être gardée par la condition $C_k(\vec{i'})$ donnée par "if ($\mathcal{A}\ \vec{i'}\ \in\ D'_k$)", car cette instruction n'est pas exécutée pour toute itération du domaine du nid de la *Figure 13*.

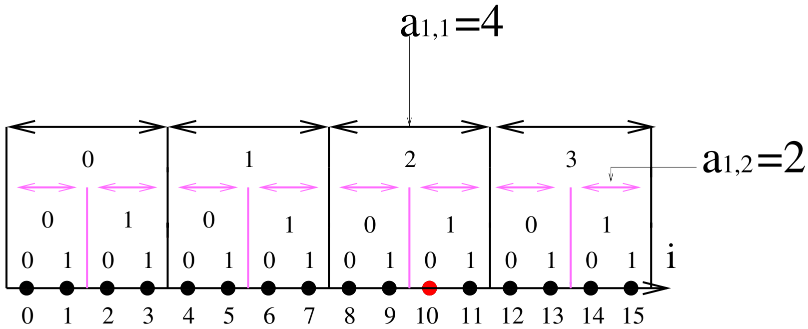


FIG. 14. *Exemple de l'application de $\mathcal{A}$ sur l'axe i*

5.2. *Échange de boucles*

La matrice de tiling hiérarchique $\mathcal{A}$ permet uniquement de spécifier la taille des tuiles. Elle n'a par contre aucune influence sur l'ordre d'exécution des itérations par rapport au code initial. Grâce à la matrice de permutation $\mathcal{P}$, qui est une matrice de dimension 6 et qui permet d'échanger plusieurs boucles en même temps, nous pouvons spécifier l'ordre dans lequel les tuiles seront exécutées. Cette matrice permet de transformer le code de la *Figure 13* vers le code donné en *Figure 15*.

`do` $(l_1 = \ldots)$
 `do` $(l_2 = ..)$
  `do` $(l_3 = \ldots)$
   `do` $(l_4 = \ldots)$
    `do` $(l_5 = \ldots)$
     `do` $(l_6 = \ldots)$
      $S_1 : C_1(\vec{l}) \;\; A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d_1}) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d_1} + \vec{v}_1^1) \otimes \ldots . \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d_1} + \vec{v}_{m_1}^1)$
      .
      .
      $S_k : C_k(\vec{l}) \;\; A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d_k}) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d_k} + \vec{v}_1^k) \otimes \ldots . \otimes A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d_k} + \vec{v}_{m_k}^k)$
      .
      .
      $S_n : C_n(\vec{l}) \;\; A_1(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d_n}) = A_0(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d_n} + \vec{v}_1^n) \otimes \ldots . \otimes A_0(AP^{-1}\vec{l} - \vec{d_n} + \vec{v}_{m_n}^n)$
     `enddo`
    `enddo`
   `enddo`
  `enddo`
 `enddo`
`enddo`

FIG. 15. *Code après l'application de la matrice $\mathcal{P}$.*

La condition $C_k(\vec{l})$ qui est la garde de l'instruction S_k, est définie par :

if $(\mathcal{A}\mathcal{P}^{-1}\, \vec{l} \in D'_k)$.

Soient $\vec{i'} = (i_1,\ i_2,\ i_3,\ j_1,\ j_2,\ j_3)^t$ le vecteur d'itération du code après l'application de la matrice $\mathcal{A}$ et $\vec{l} = (l_1,\ l_2,\ l_3,\ l_4,\ l_5,\ l_6)^t$ le vecteur d'itération du code après l'application de la matrice $\mathcal{P}$, les relations entre ces deux vecteurs sont données par :

– $\vec{l} = \mathcal{P}\vec{i'}$
– $\vec{i'} = \mathcal{P}^{-1}\vec{l}$

5.2.1. *Tiling contigu*

Un tiling est dit contigu, si nous exécutons entièrement une tuile avant de passer à une autre. Pour que notre tiling soit contigu, il est nécessaire que la matrice $\mathcal{P}$ vérifie ces deux conditions de contiguïté de parcours :

$$\forall a, b, c : (\mathcal{P}_{a,1} = \mathcal{P}_{b,2} = \mathcal{P}_{c,3} = 1) \Rightarrow a < b < c \qquad [3]$$

$$\forall a, b, c \ : (\mathcal{P}_{a,4} = \mathcal{P}_{b,5} = \mathcal{P}_{c,6} = 1) \Rightarrow a < b < c \qquad [4]$$

La condition 3 (resp 4) signifie qu'après l'application de la matrice $\mathcal{P}$, la composante i_1 doit être avant i_2 et i_2 avant i_3 (resp la composante j_1 doit être avant j_2 et j_2 avant j_3).

Exemple

La matrice $\mathcal{P} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ nous donne un tiling contigu, car le vecteur d'itération $\vec{l}$ sera égal à $(i_1,\ j_1,\ i_2,\ j_2, i_3,\ j_3)^t$. Nous pouvons remarquer que les deux conditions [3] et [4] sont bien vérifiées.

Par contre le tiling résultant de la matrice $\mathcal{P} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ n'est pas contigu, car le vecteur $\vec{l}$ sera égal à $(i_2,\ j_1,\ i_1,\ j_2, i_3,\ j_3)^t$. On voit bien que i_2 est avant i_1.

5.3. *Modélisation du tiling*

Contrairement à la fusion sans tiling, le calcul des différents délais $\vec{d_k}$ pour la fusion avec tiling du code de la *Figure 12*, nécessite plus d'efforts. Comme nous l'avons signalé précédemment, parmi les auteurs qui se sont intéressés au tiling, le seul qui a modélisé le tiling comme une transformation de programmes est Xue [XUE 97]. Mais ce dernier s'est intéressé à un seul niveau de tiling et les codes qu'il traite sont constitués d'un seul nid de boucles. Notre tiling défini par les matrices $\mathcal{A}$ et $\mathcal{P}$, sera représenté par une transformation ω.

$\omega : Z^2 \rightarrow Z^6$

$\vec{i} \longmapsto \vec{l} = \mathcal{P}.(\lfloor \frac{i}{a_{1,1}} \rfloor,\ \lfloor \frac{i \bmod a_{1,1}}{a_{1,2}} \rfloor,\ i \bmod a_{1,2},\ \lfloor \frac{j}{a_{2,4}} \rfloor,\ \lfloor \frac{j \bmod a_{2,4}}{a_{2,5}} \rfloor,\ j \bmod a_{2,5})^t.$

À l'itération $\vec{i_0} = (i_0,\ j_0)^t$ du nid $k+1$ $(1 \leq k \leq n)$ de la *Figure 12*, nous utilisons les éléments $A_k(\vec{i_0} - \vec{d}_{k+1} + \vec{v}_q^{k+1})$ $(q = 1, m_{k+1})$.

L'élément $A_k(\vec{i_0} - \vec{d}_{k+1} + \vec{v}_q^{k+1})$ est calculé dans le nid k à l'itération

$\vec{i_0} + \vec{v}_q^{k+1} - \vec{d}_{k+1} + \vec{d}_k$

Théorème 1 *Le tiling défini par les matrices $\mathcal{A}$ et $\mathcal{P}$ est valide si et seulement si :*

$$\forall\ k, \forall \vec{i}, \forall q : \omega(\vec{i} + \vec{v}_q^{k+1} - \vec{d}_{k+1} + \vec{d}_k) \preceq \omega(\vec{i}) \qquad [5]$$

En remplaçant ω par la transformation donné précédemment, le *Théorème 1* sera équivalent à :

$$\forall \vec{i},\ k,\ q : \mathcal{P} \begin{pmatrix} \lfloor \frac{i+v_{q,1}^{k+1}-d_{k+1,1}+d_{k,1}}{a_{1,1}} \rfloor \\ \lfloor \frac{(i+v_{q,1}^{k+1}-d_{k+1,1}+d_{k,1}) \bmod a_{1,1}}{a_{1,2}} \rfloor \\ (i+v_{q,1}^{k+1}-d_{k+1,1}+d_{k,1}) \bmod a_{1,2} \\ \lfloor \frac{j+v_{q,2}^{k+1}-d_{k+1,2}+d_{k,2}}{a_{2,4}} \rfloor \\ \lfloor \frac{(j+v_{q,2}^{k+1}-d_{k+1,2}+d_{k,2}) \bmod a_{2,4}}{a_{2,5}} \rfloor \\ (j+v_{q,2}^{k+1}-d_{k+1,2}+d_{k,2}) \bmod a_{2,5} \end{pmatrix} \preceq \mathcal{P} \begin{pmatrix} \lfloor \frac{i_0}{a_{1,1}} \rfloor \\ \lfloor \frac{i_0 \bmod a_{1,1}}{a_{1,2}} \rfloor \\ i_0 \bmod a_{1,2} \\ \lfloor \frac{j_0}{a_{2,4}} \rfloor \\ \lfloor \frac{j_0 \bmod a_{2,4}}{a_{2,5}} \rfloor \\ j_0 \bmod a_{2,5} \end{pmatrix}$$

Nous pouvons remarquer que le calcul de $\vec{d_1}$ dépend de $\vec{d_2}$, $\vec{d_2}$ dépend de $\vec{d_3} \ldots \vec{d}_{n-1}$ dépend de $\vec{d_n}$. Pour calculer les différents $\vec{d_k}$ nous commençons par calculer $\vec{d_n}$, ensuite $\vec{d}_{n-1}$.... et enfin nous calculons $\vec{d_1}$. Après le calcul de ces différents délais, nous devons assurer que l'origine du domaine D_1' est $(0,\ 0)^t$.

Il existe une solution indépendante de la matrice $\mathcal{A}$, pour une matrice $\mathcal{P}$ qui vérifie les conditions de contiguïté de parcours ([3] *et* [4]). Cette solution est donnée par :

$-\vec{d_k} = -\vec{d}_{k+1} + (\max_l\ v_{l,1}^{k+1},\ \max_l\ v_{l,2}^{k+1})^t.$

Le choix de cette solution rend la fusion des différents nids possibles et les boucles i et j du code fusionné complètement permutables. Nous savons que si un nid de boucles est complètement permutable, nous pouvons lui appliquer n'importe quel tiling parallèle aux axes [WOL 92] ($\mathcal{P}$ doit satisfaire les conditions [3] et [4]). Nous pouvons penser à une autre solution, qui consiste à choisir des délais rendant la deuxième composante des vecteurs de dépendances supérieure à la taille des tuiles sur l'axe i $(a_{1,1})$. Cette solution n'est pas intéressante lorsque les tailles des tuiles sont très grandes, car le volume mémoire des différents tableaux devient très important.

5.4. *Volume mémoire*

Dans cette section, nous nous intéresserons à un seul niveau de tiling, la matrice de tiling hiérarchique $\mathcal{A} = \begin{pmatrix} a_{1,1} & 1 & 0 & 0 \\ 0 & 0 & a_{2,3} & 1 \end{pmatrix}$. Pour simplifier cette présentation, nous avons supposé la matrice de permutation $\mathcal{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$. La méthode reste la même pour les autres matrices $\mathcal{P}$, seules les fonctions d'accès F_k et les fonctions du volume mémoire $M_{k,q}$ changent.

5.5. *Données vivantes*

À une itération $\vec{l}$, dans le nid de la *Figure 15*, pour calculer l'élément $A_k(\mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d}_k)$, nous utilisons m_k données du tableau A_{k-1}, qui sont produites respectivement par les itérations $\vec{l}_1$, $\vec{l}_2$,.....et $\vec{l}_{m_k}$ tels que :

$$\mathcal{A}\mathcal{P}^{-1}\vec{l}_q - \vec{d}_{k-1} = \mathcal{A}\mathcal{P}^{-1}\vec{l} - \vec{d}_k + \vec{v}_q^{\,k} \ (q = 1, m_k) \qquad [6]$$

L'ensemble des données vivantes $M_k(\vec{l})$ est borné par le nombre d'itérations dans $\omega(D'_{k-1})$, entre la plus ancienne de ces productions et $\vec{l}$. La plus ancienne de ces productions varie en fonction de la position de l'itération $\vec{l}$ dans la tuile (*Figure 16*). Soit $M_{k,q}(\vec{l})$ $(q = 1, m_k)$ le nombre d'itérations dans $\omega(D'_{k-1})$ entre $\vec{l}_q$ et $\vec{l}$.

$$M_{k,q}(\vec{l}) = (N_{k-1,j}.a_{1,1},\ a_{1,1}.a_{2,3},\ a_{2,3},\ 1)^t.(\vec{l} - \vec{l}_q) + 1 \qquad [7]$$

Dans le code généré, nous remplaçons chaque tableau A_k par un tampon B_k. À chaque itération $\vec{l}$, le tampon B_k doit contenir toutes les données vivantes du tableau A_k. Par conséquent, la taille de ce tampon sera égale au maximum des $M_{k,q}(\vec{l})$. Pour calculer $M_{k,q}(\vec{l})$, nous devons en premier lieu calculer $\vec{l}_q$ en fonction de $\vec{l}$, mais ceci n'est pas possible en utilisant uniquement l'équation donnée en [6]. Nous savons que

– $\vec{i} = AP^{-1}\vec{l}$
– $\forall\, \vec{l}_q,\ \exists\, \vec{i}_q :\ \vec{i}_q = AP^{-1}\vec{l}_q$

Par conséquent, l'équation [6] sera équivalente à :

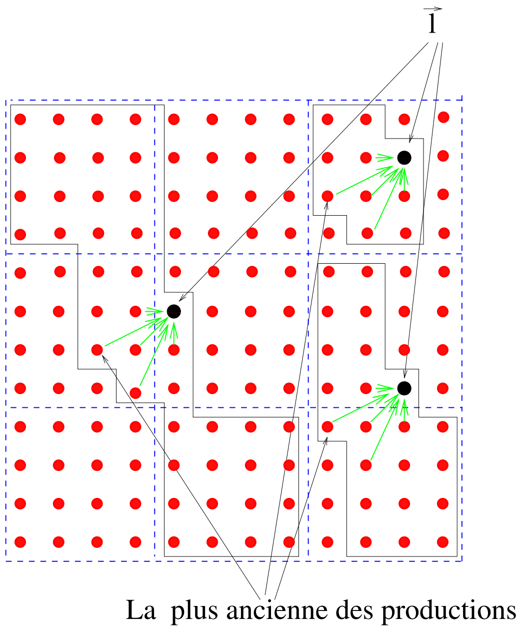


FIG. 16. *Volume mémoire pour le code de la figure 8.*

$\vec{i}_q - \vec{d}_{k-1} = \vec{i} - \vec{d}_k + \vec{v}_q^k \ (q = 1, m_k)$

Le maximum des $M_{k,q}(\vec{l})$ sera donné par :

$\mathrm{Sup}_{k+1} = \max(\ M_{k+1,q=1,m_{k+1}}(\vec{l}))$

$$\left\{ \begin{array}{l} \vec{i} = \mathcal{AP}^{-1}\vec{l} \\ \vec{i} = \mathcal{AP}^{-1}\vec{l}_q + \vec{d}_{k+1} - \vec{d}_k - \vec{v}_q^{k+1} \ (q = 1, m_{k+1}) \\ 0 \preceq l_{q,3} \preceq a_{1,1} - 1 \ (q = 1, m_{k+1}) \\ 0 \preceq l_{q,4} \preceq a_{2,3} - 1 \ (q = 1, m_{k+1}) \\ \vec{i} \in D'_{k+1} \end{array} \right.$$

5.6. *Génération du code*

Pour chaque domaine D_k, nous définissons une fonction d'accès F_k de la manière suivante :

$F_k : \omega(D'_k) \rightarrow N$ tel que $\vec{l} \longmapsto F_k(\vec{l})$ avec

1) $F_k(\omega(\vec{O}_k)) = 0$

2) $F_k(succ(\vec{l})) = \left\{ \begin{array}{ll} F_k(\vec{l}) + 1 & \text{if } (F_k(\vec{l}) \neq \mathrm{Sup}_{k+1} - 1) \\ 0 & \text{sinon} \end{array} \right.$

Pour satisfaire ces deux conditions, il suffit de choisir

$$F_k(\vec{l}) = [(N_{k,j}.a_{1,1},\ a_{1,1}.a_{2,3},\ a_{2,3},\ 1)^t.(\vec{l} - \omega(\vec{O}_k))] \bmod \mathrm{Sup}_{k+1}$$

6. Résultats expérimentaux

Pour effectuer nos expériences, nous avons utilisé une machine *UltraSparc10* qui possède 512 *Mo* de mémoire centrale, 2 *Mo* de cache externe L_2 et 16 *Ko* de cache interne L_1. Nous avons utilisé les fonctions de comptage d'événements matériels, issues de la librairie *libcpc.h*, pour mesurer les défauts de cache externe engendrés par les différentes transformations du code donné en exemple de la *section* 2 :

– code initial,

– code fusionné,

– code fusionné avec allocation de tampon,

– code tuilé,

– code tuilé avec allocation de tampon.

La *Figure 17* donne les défauts de cache externe causés par ces différents codes et montre que le tiling avec allocation de tampon, donne le meilleur résultat (le minimum de défauts de cache). Comme nous pouvons le remarquer, l'allocation de tampons dans le code fusionné et dans le code tuilé, permet de réduire de manière considérable le nombre de défauts de cache.

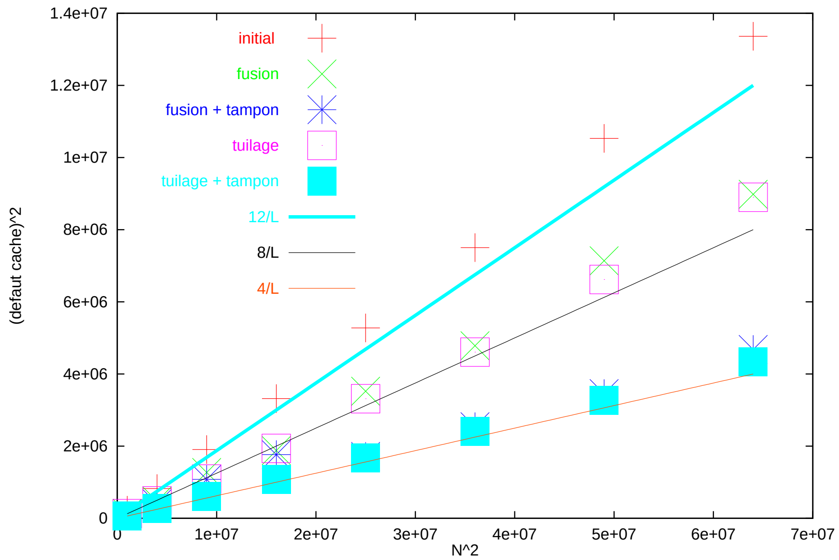


FIG. 17. *Défauts de cache externe (L2).*

N^2 dans cette figure, représente la taille des matrices utilisées dans ces différents programmes (N lignes et N colonnes).

6.1. *Résultats attendus*

La taille du cache externe est 2 *Mo* divisé en 31250 lignes, la taille de chacune de ces lignes étant $L = 64$ o. Les deux matrices utilisées dans ces programmes ont la taille N^2 $(1000 \leq N \leq 8000)$, chaque élément de ces matrices est stocké sur 4 octets. Ces deux matrices ne peuvent pas tenir en même temps dans le cache, car la taille en octet de ces deux matrices est $2 \times 4 \times N^2$ $o \geq 8$ *Mo*. Par contre, le cache peut contenir plusieurs lignes des deux matrices (la taille d'une ligne est $4 \times N$ $o \leq 32$ *Ko* $<< 2$ *Mo*).

Le parcourt contigu des éléments d'une matrice en écriture ou en lecture, engendre naturellement $\frac{4 \times N^2}{L}$ défauts de cache.

6.1.1. *Code initial*

Le code initial (*Figure 1*) est constitué de deux nids. Les écritures des deux nids engendrent $\frac{4 \times N^2}{L}$ défauts de cache chacune. Dans le second nid, la lecture de la matrice A_1, engendre également $\frac{4 \times N^2}{L}$ défauts de cache. En effet, en régime permanent, pour exécuter toutes les itérations j d'un i donné, nous avons besoin des lignes $i-1$, i et $i+1$ de la matrice A_1 (*Figure 18*). Les lignes $i-1$ et i sont déjà chargées précédemment dans le cache. Il n' y a que la ligne $i+1$ qui n'est pas disponible, le chargement de cette ligne, engendre $\frac{4 \times N}{L}$ défauts de cache. Par conséquent, l'exécution de toutes les itérations i engendre $\frac{4 \times N^2}{L}$ défauts de cache.

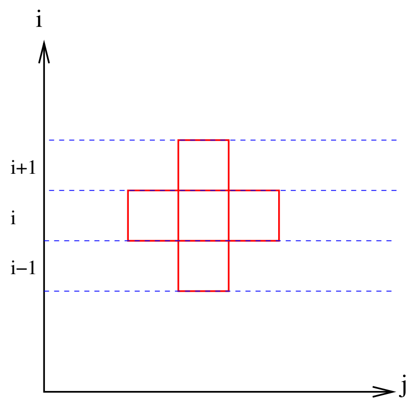


FIG. 18. *Patron de données*

Le nombre total de défauts de cache engendré par le code initial est alors :

$\frac{3 \times 4 \times N^2}{L} = \frac{12 N^2}{L} = 0.1875 \times N^2$.

6.1.2. *Code fusionné*

Dans le code fusionné (*Figure 9*), l'écriture dans la matrice A_1 dans la première instruction, engendre $\frac{4 \times N^2}{L}$ défauts de cache. Dans la deuxième instruction, seul l'écriture dans la matrice A_2 engendre $\frac{4 \times N^2}{L}$ défauts de cache, la matrice A_1 étant déjà chargée par la première instruction. Par conséquent, le nombre total de défauts de cache engendré par le nid fusionné est $\frac{8 \times N^2}{L} = 0.125 \times N^2$.

6.1.3. *Code fusionné avec allocation de tampon*

Dans le code fusionné avec allocation de tampon (*Figure 10*), le tampon S engendre $\frac{2 \times 4 \times N+1}{L}$ défauts de cache, lors du premier parcourt. Par contre, l'écriture dans la matrice A_2 engendre $\frac{4 \times N^2}{L}$ défauts de cache.
REMARQUE. — Dans cet exemple, le modèle théorique pour le tiling est presque le même que celui de la fusion. Par contre, si nous choisissons un autre exemple nous obtenons de meilleurs résultats .

6.2. *Comparaison entre les résultats théoriques et expérimentaux*

La *Figure 17* montre aussi la comparaison entre les résultats théoriques et les résultats expérimentaux. Nous pouvons remarquer que nos résultats expérimentaux sont légèrement supérieurs aux résultats théoriques. Cette petite différence peut s'expliquer : 1) par la non prise en compte du régime d'amorçage dans le modèle théorique, comme nous ne nous sommes intéressés qu'au régime permanent et 2) dans nos expériences, l'appel des fonctions de la librairie *libcpc* engendre des défauts de cache supplémentaires.

7. Conclusion

À notre connaissance, l'application du tiling et fusion simultanément sur une séquence de nids de boucles n'a été traitée par aucun auteur. Par contre de nombreux travaux se sont intéressés à ces deux problèmes isolément. Dans notre approche, ces deux problèmes sont traités simultanément en combinant les travaux de Xue [XUE 97] qui a modélisé le tiling (un seul nid de boucles et seul niveau de tiling) par une transformation de boucles et les autres travaux sur la fusion de plusieurs nids de boucles [ZIM 90, MCK 96, PUG 99, WOL 96, MAN 97].

Notre approche consiste à translater les calculs de chaque nid de telle façon qu'après le tiling toutes les données nécessaires à une itération ont déjà été produites par les itérations précédentes. Notre approche nous donne un système d'inéquations qui met en relation les vecteurs des translations associés aux différents nids, les vecteurs des différents *patrons*, la matrice de tiling hiérarchique $\mathcal{A}$ et la matrice de permutation

$\mathcal{P}$. Pour ce système d'inéquations nous avons donné une solution pour une classe de tiling (tiling contigu).

Pour maintenir en mémoire les données vivantes et pour éviter de charger plusieurs fois les même données, nous avons remplacé les différents tableaux par des tampons circulaires, dont la taille est égale au nombre de données vivantes du tableau correspondant. Les résultats expérimentaux montrent que l'allocation de ces tampons, permet de réduire d'une manière importante le nombre de défauts de cache.

Dans nos prochains travaux, nous étudierons d'autres techniques pour l'allocation de tampons, par exemple allouer des tampons multidimensionnels pour la fusion et plusieurs tampons pour le tiling. Nous étudierons aussi l'allocation et la réutilisation des registres pour le deuxième niveau du tiling.

8. Bibliographie

[BAN 93] BANERJEE U., *Loop transformations for restructuring compilers*, Kluwer Academic Publishers, 93.

[CAT 98] CATHOOR F., AL, *Custom memory management methodology-Exploration of memory organisation for embedded multimedia system design*, Kluwer Academic Publishers,ISBN 3-540-64105-X, 98.

[CHE 94] CHESNEY D., CHENG B., « Generalising the unimodular approach program code transformation », *Proceedings 1994 International Conference on Parallel and Distributed Systems*, , 1994, p. 398-404.

[IRI 88] IRIGOIN F., TRIOLET R., « Super-node partitioning », *In Proc. 15th Annual ACM Symp. Principles of Programming Languages*, , 1988, p. 319-329.

[KAN 99] KANDEMIR M., CHOUDHARY A., RAMANUJAM J., « I/O-Conscious Tiling for Disk-Resident Data Sets », *Euro-par*, , 1999.

[MAN 97] MANJIKIAN M., ABDELRAHMAN T. S., « Fusion of loops for parallelism and locality », *IEEE trans. on paral*, , 97.

[MCK 96] MCKINLEY K., CARR S., TSENG C., « Improving data locality with loop transformations », *ACM Trans. on Programming Languages and Systems*, vol. 18(4), 96, p. 424-453.

[PUG 99] PUGH W., ROSSER E., « Iteration space slicing for locality », *LCPC99*, , 1999, p. 165-184.

[SAR 97] SARKAR V., « Automatic selection of high order transformations in the IBM XL FORTRAN compilers », rapport, 1997.

[WOL 92] WOLF M. E., « Improving locality and parallelism in nested loops », PhD thesis, University of stanford, 1992.

[WOL 96] WOLFE M., *High performance Compilers for parallel Computing*, Addison-Wesley, 1996.

[WOL 98] WOLF M., MAYDAN D., DING-KAI-CHEN, « Combining loop transformations considering caches and scheduling », *International-Journal-of-Parallel-Programming*, vol. 26, 1998, p. 479-503.

[WOL 91] WOLF M. E., LAM M. S., « A loop transformation theory and an algorithm to maximize parallelism. », *IEEE trans. on paral*, vol. 2, 91.

[XUE 97] XUE J., « On tiling as a loop Transformation », *Parallel Processing Letters*, , 1997.

[XUE 98] XUE J., HUANG C., « Reuse-driven tiling for improiving data locality », *Parallel programming*, , 98.

[ZIM 90] ZIMA H. P., CHAPMAN B. M., *Supercompilers for parallel and vector computers*, vol. 1, Addison-Wesley, 1990.